le cnam
Paris

Conservatoire National des Arts et Métiers (CNAM)

Habilitation thesis

From Heuristics to Column Generation and Semidefinite Optimization by Following an Idea of Projection

Daniel Porumbel

Thesis for the degree of "Habilitation á diriger des recherches" defended in front of the jury below
Friday, March 27, 2026

Referees	
Sophie Demassey	Prof., Institut Mines-Télécom
Christoph Dürr	Senior Researcher (DR), CNRS - LIP6, Sorbonne Université
Renata Sotirov	Prof., Tilburg University, The Netherlands
Examiners	
Claudia D'Ambrosio	Senior Researcher (DR), CNRS - LIX, École Polytechnique (committee chair)
Amélie Lambert	Prof., CNAM
Viet Hung Nguyen	Prof., Unversité Clermont Auvergne
Christophe Picouleau	Prof., CNAM (garant)

Acknowledgements

This semidefinite thesis would not have appeared online without the help or approval of certain people to whom I am indebted. They do have a place in my heart and I can only hope it's not very important for them to see it in print. Cause I published an essay on my web page that starts as follows:

> *Take the acknowledgements of any PhD thesis (including mine) or even of any modern book. You will often find out that the author is happy to share with the whole world how many encouragements he has had.*

This is why I want to acknowledge something else, something that eludes our main reasoning powers. I don't know if anybody could ever manage to explain it in clear words, but I surely can't do it now. I'll only cite these inconclusive lyrics from an 1986 song by Stan Ridgway.

And it was near the riverbank when the ambush came on top of us

My weapon jammed and I got stuck way out and all alone

And I could hear the enemy moving in close outside

Just then I heard a twig snap, and I grabbed my empty gun

Then a bullet with my name on it came buzzing through a bush

And that big marine, he just swat it, with his hand

I said: "Well, thanks a lot". I told him my name and asked him his

And he said "The boys just call me Camouflage"

Contents

Editorial note *This document includes a number of corrections or style improvements compared to the official thesis submitted to the three referees, Sophie Demassey, Christoph Dürr and Renata Sotirov. Some modifications represent suggestions from the reports. Others represent expository refinements introduced on my own initiative, including a new figure and a few references. I deliberately refrained from changing the table of contents, except for Appendix A which provides an additional experiment. The goal is not to alter the general flow of ideas, but to improve readability and complement a few notions with clearer insights.*

1 The overall scientific trajectory

1.1 Chronological curriculum elements

1.1.1 On my modest contact with science up to my PhD

The spark of my interest in science did not come with my PhD or my undergraduate studies; even since I was a high-school student, I enjoyed solving math problems and participated in many contests (including the International Mathematical Olympiad in 2000). To some extent, this remains a characteristic of my scientific identity even today, because I approach many scientific questions like a problem-solver. I call problem-solver any person who can take a reasonably-difficult (sub-)problem in some arbitrary field and, as long as the solution does not require a very deep technical apparatus, may hope to solve it with elementary methods. This may seem unimportant, but without such skills I could have had a harder time to evolve outside the topic of my PhD subject without constant guidance from a mentor (a luxury I could not afford).

In college I started to split from mathematics (which I regret) because I enrolled in a Computing Engineering School majoring in Computer Science. I graduated in 2005 having acquired a lot programming skills in various languages. I never had any undergraduate training in optimization or Operations Research; I had to learn everything I know in optimization by myself starting with my PhD.

My PhD began in 2006 and was devoted to meta-heuristic algorithms for graph coloring. A novelty we could introduce was the heavy use of the notion of distances between candidate solutions; an initial goal was to apply learning techniques in optimization, an idea that was not so widespread in 2006 as it is today. While most of this meta-heuristic work was devoted to solving graph coloring using Tabu Search and Memetic Algorithms, we also published an article with no implementation on a linear Las-Vegas algorithm to compute distances between colorings. These distances can be useful for heuristic understanding or to classify the candidate solutions in clusters. Mostly based on my PhD work, I received the Simon Régnier prize, awarded by the "Société Francophone de Classification" (French-speaking Classification Society, at sfc-classification.net) each year to a researcher under 35 years old with a PhD related to the field of data classification (or clustering).

1.1.2 After my PhD

I had my first job of Assistant Professor at Artois University (Béthune campus in northern France) in 2010. In 2014, I moved to the Conservatoire National des Arts et Métiers (Paris, France) where I still work as an Assistant Professor in the team OC (Combinatorial Optimization) of the Computer Science laboratory CEDRIC.

The scientific areas and papers written over this period are listed in Section 1.2. I provide hereafter some curriculum elements over these years.

I was (co-)adviser for a defended PhD thesis: Nathalie Helal, An evidential answer for the capacitated vehicle routing problem with uncertain demands, defended in 2017 (main adviser: Eric Lefèvre, second adviser Frédéric Pichon, collaboration with David Mercier). I have just started (2025) co-advising the PhD thesis of Pierre Arvy about Flow-Based domain Optimization under Uncertainty in Energy Markets; main adviser: Amélier Lambert, collaboration with company Artelys.

I advised some master theses along the years: Thomas Savin (Observer optimization for battery system monitoring) with Amélie Lambert, Thach Dinh and Mathieu Moze, to be finished in September 2025; Luthfi Bin Fauzi Bafana (Sparse Dictionary Learning by Quadratic Programming), with Amélie Lambert, 2022; Thomas Ridremont (Robust optimisation of wiring in renewable-energy installations, with Marie-Christine Costa and Cédric Bentz, 2015); Hubert Arnoux (A tool of visualisation techniques for local search methods, with J-K Hao, P. Kuntz, 2012); Junlin Zhu (Mathematical Programming Models for Graph Isomorphism, with Amélie Lambert, 2015); Ben Ibrahima Badji (Meta-heuristics for the Arc-Routing Problem, with T. Hsu, H. Allaoui, G. Goncalves, 2014); Fahrur Rozi and Mochammad Luthfi (Airport passenger flow simulation in Quest and Arena and resource allocation optimization, with G. Gonvalves, 2011).

I accepted review invitations for different journals since 2010, in alphabetical order: ACM Transactions on Evolutionary Learning and Optimization, AI communications, Annals of Operations Research, Canadian Mathematical Bulletin, Computational Optimization and Applications, Computers & Mathematics with Applications, Computers and Operations Research, Computing Discrete Applied Mathematics, Engineering Applications of Artificial Intelligence, Engineering Optimization, European Journal of Operational

Research, Evolutionary Computation, Expert Systems and Applications, IEEE Transactions on Cybernetics, IEEE Transactions on Evolutionary Computation, INFORMS Journal of Computing, Information Processing Letters, Journal of Artificial Intelligence Research, Journal of Computational Science, Operational Research, The Computer Journal, Theoretical Computer Science, Transportation Research part E. I was in the program committee of different conferences (Evocop 2012-2024, Gecco Ecom 2014-2024). I acted as subreviewer for SEA 2023.

I organize since 2020 a session "On best coding practices and their link with theory" as part of the French Operations Research congress (ROADEF). The section received between three and eight contributions per year, one of which was always mine. I've collected my ideas on a web page[1] which describes my overall scientific positioning on the artificial and injurious gap between theory and computing. One starting point was given by C. Strachey (Oxford 50 years ago): *"Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of [our] central aims has been to set up an atmosphere in which this separation cannot happen. "*

I also do like coding. My web-page does contain a section "Software" where you can find the `C++` code source behind six of my publications. You can see my coding guidelines at `http://cedric.cnam.fr/~porumbed/CODE_GUIDELINES`. I copy-paste the beginning.

> Performance is a priority in optimization. You may have a brilliant pseudo-code, but if your implementation is not very elaborately tuned, you won't achieve too much numerical success against your competition. Unless you have very particular time-consuming mathematical calculations that take all the time, your (C++) code needs to be WYSIWYG: What You See Is What You Get. One of the advantages of `C` over `C++` is that it gives better control over what happens when your code is executed. We can keep this advantage by using a rather `C-like C++` code. I must confess my `C++` code might fall under what M. Torvalds said: "the only way to do good, efficient, and system-level and portable `C++` end up to limit yourself to all the things that are basically available in `C`.". But at least for simple convenience, I do prefer to use only some `C++` features, e.g., every now and then I do use templates. However, it is essential to see what is really going on when you run a piece of code, avoiding implicit or hidden calls (e.g., like infinite inheritance rules or overly-complex templates). Any external function can bring confusion to your code when you do not exactly know what's inside. Did you know that calling `size()` on a `std::list` can take linear time? Did you know that changing an objective coefficient with `setLinearCoef` (in `C++` with `cplex`) can take more than constant time? I once experienced a 3-fold increase of the running time only because I forgot to call `cplex.end()` at the end of a function (see `https://youtu.be/YuKnE6zH-J8`).

1.2 List of contributions or visited areas since 2010

To give a sketch of my scientific positioning, I present below in pseudo-random order a few areas I worked in. If I can't describe them in greater detail, its not because they are unimportant, but I prefer to keep this thesis to a modest size (50 pages is the minimum required by Cnam) and to make it follow a unique research thread.

quadratic programming I worked in this area by collaborating and under the guidance of my team colleague Amélie Lambert. I needed a few months to understand her field; it seemed quite deep for me, as I was working for the first time to publish in a non-linear context (without mentioning the time I needed to learn semidefinite optimization). In a conference paper [27], we generalize the notion of cutting planes to cutting hypersurfaces (also called quadrics). We underestimate a quadratic non-convex function by a piecewise quadratic convex function composed of many hypersurfaces. It's a generalization of piecewise linear convex function. The full paper [28] was accepted by the *Journal Of Global Optimization* in June 2025. It was presented at ISMP 2024 (Montreal) and EURO 2025 (Leeds).

polynomial optimization I only have a working paper in this area submitted with Amélie Lambert in December, 2025. It relies on a convexification approach as above, also building on [10].

submodular function minimization This is a stand-alone and well-established theoretical field. It motivated certain researchers to invest all their career on it; the 2003 Fulkerson prize was accorded for

[1] `http://cedric.cnam.fr/~porumbed/soft/`

showing that submodular minimization is strongly polynomial. As for me, I only have a modest theoretical contribution [41] extending to a more general setting an algorithm called `Greedy` for optimizing over the submodular polytope [33, Sec. 2.1]. Given a linear order of the variables, *Greedy* progresses towards the optimum by increasing one by one each variable to its maximum value. Each such maximum value can be seen as a "maximum step length" that one can advance in the direction of the variable until intersecting a polytope facet. We will see this idea has some similarities to the projection subproblem I'll use in the rest of the thesis.

vehicle routing under uncertainty This work was carried out in the PhD thesis of my former student Nathalie Helal [19]. I was collaborating with a team working on belief functions and we modelled the uncertainties of a vehicle routing problem using such functions. The resulting optimization problem was solved using heuristics.

heuristics for Arc-Routing and Graph Isomorphism For the isomorphism problem, I first proposed a polynomial graph extension procedure that provides a graph coloring (labeling) capable of rapidly guiding a simple-but-effective heuristic toward the solution [48]. This was later used with several collaborators working in Constraint-Programming to publish an exact algorithm in a conference from their field [2]. Arc-Routing is a problem of a different nature, but I proposed a decoder that can cast it as a permutation problem, *i.e.*, a problem for which a candidate solution can be encoded as a permutation, exactly as for graph isomorphism [46, 44].

heuristic understanding using distances or spacing This is my only post-PhD work that does extend and builds upon my PhD thesis (2006-2009). We published in 2011 a paper called "Spacing Memetic Algorithms" in the most competitive track (GA – Genetic Algorithms) of GECCO 2011 [49]. The main ideas is to design all genetic operators (e.g., replacement and mating selection) by taking into account the distances between the individuals that constitute the current population. I remember this was the most difficult and essential primordial task: ***achieve a high diversity without sacrificing the quality of the population***. We continued working on such ideas to later design an algorithm called "Distance Guided Local Search" published by the *Journal of Heuristics* in 2020 [47].

To (try to) to follow a common thread of research thought in the rest of the manuscript, from now on I will focus on the following two areas of work.

1.2.1 Exact algorithms for (integer) linear programs with prohibitively-many constraints

I started to work in these areas by studying the dual polytopes arising in Column Generation models, mostly on set-covering problems like Cutting-Stock, Arc-Routing or Graph Coloring. I was first motivated to study such polytopes by working in (and learning) the field of Dual Feasible Functions with François Clautiaux between 2010 and 2013 [6]. I wrote an article extending the use of such functions to routing and location problems [45]; I also provided a little bit of help for writing a book on the subject [1].

Yet my most important work in Column Generation [39] introduces the following idea (inspired by the above dual feasible functions). Given the dual polytope $\mathscr{P}$ (of the Column Generation model), generate feasible dual solutions by calling the intersection (or projection) subproblem : given feasible $\mathbf{y}$ in some polytope $\mathscr{P}$ and direction $\mathbf{d}$, what is the maximum t such that $\mathbf{y} + t\mathbf{d} \in \mathscr{P}$. This subproblem generalizes the widely used separation subproblem and will be integrated into the Cutting-Planes framework.

In the above work, I could solve the projection subproblem only for $\mathbf{y} = \mathbf{0}$ and for integer directions $\mathbf{d}$. This subproblem will be referred to as the $\mathbf{0}$-*origin integer projection*. From around 2015, I realized it is not always so difficult to extend this setting to arbitrary origins $\mathbf{y} \in \mathscr{P}$ and to arbitrary non-integer directions $\mathbf{d}$. I thus introduced the "Projective-Cutting Planes" algorithm [42] to optimize over a primal polytope that arise in the Benders reformulation (for a network design problem) or a dual polytope for the Column-Generation graph coloring model [42]. A bit later, I applied the same idea for robust linear programming [43]. The most challenging aspect lies in designing a algorithm for the projection sub-problem that is fast-enough, as such an algorithm has to be tailored to the context under consideration.

1.2.2 Semidefinite optimization

Transitioning to the field of Semidefinite Programming (SDP) from a background in mixed-integer linear programming can be difficult. To understand the SDP area, I decided to write myself an introduction of 100

pages[2] called *Demystifying the characterizations of SDP matrices in mathematical programming* [38]. I wrote it because I found no other introduction to SDP programming that targets the same audience. It is intended to be accessible to anybody who does not hate maths, who knows what a derivative is and accepts (or has a proof of) results like $\det(AB) = \det(A)\det(B)$. This will cover most if not all people working in fields like Column Generation, Integer Programming, Benders reformulation, (meta-)heuristics, robust optimization, etc. The goal is to provide such people with all the tools needed to carry out SDP research work.

It is enough to see how other introductions to SDP programming present the eigen-decomposition (4.B) to see they have a different target audience. The eigen-decomposition is often listed without proof, while I give two proofs to really familiarize the reader with it. For example, In "Handbook of Semidefinite Programming Theory, Algorithms, and Applications" by H. Wolkowicz, R. Saigal and L. Vandenberghe, the eigen-decomposition (called spectral theorem) is listed with no proof in Chapter 2 "Convex Analysis on Symmetric Matrices". The introduction of the "Handbook on Semidefinite, Conic and Polynomial Optimization" by M. Anjost and J.B. Lasserre refers the reader to the (700 pages long) book "Matrix analysis" by Horn and Johnson. The book "Convex Optimization" by S. Boyd and L. Vandenberghe starts using SDP matrices from the beginning (*e.g.*, to define ellipsoids in Section 2.2.2) without defining the concept of SDP matrix, not even in appendix. The slides of the course "Programmation linéaire et optimisation combinatoire" of Frédéric Roupin for the *Master Parisien de Recherche Opérationnelle* (MPRO) at `lipn.univ-paris13.fr/~roupin/docs/MPROSDPRoupin2018-partie1.pdf` provide results that cover half of my introduction to SDP, but my manuscript also seeks to prove them.

The only unproven facts from the above manuscript are the fundamental theorem of algebra and two results from Section 5.3.2.3. But it does provide complete proofs, for instance, for *the Cholesky decomposition of SDP matrices, the hyperplane separation theorem, the strong duality theorem for linear conic programming (including SDP programming), six equivalent formulations of the Lovász theta number, the copositive formulation of the maximum stable, a few convexification results for quadratic programs and many others*. I tried to prove everything by myself, so that certain proofs are original although this introduction was not intended to be research work. I did receive feedback on this article by people who found certain things useful to make things clear to them. I'm still looking to ideas on how this work could be submitted for publication (as a book chapter), but this was not my original intent.

I needed this work to hope conducting research in SDP optimization. Chapter 4 of the current thesis constitutes a journal paper I plan to send for publication. It remains connected to my work on polytopes with unmangeably-many constraints from Chapters 2–3. This comes from the fact that the SDP cone can be seen as a polytope with infinitely-many constraints (semi-infinite programming). I'll still use the projection subproblem: given an SDP matrix X and an arbitrary direction matrix D, what is the maximum t such that $X + tD$ is an SDP matrix ? This subproblem is (theoretically) more difficult than in a linear context.

2 Column Generation by 0-Integer Projection in a Dual Polytope

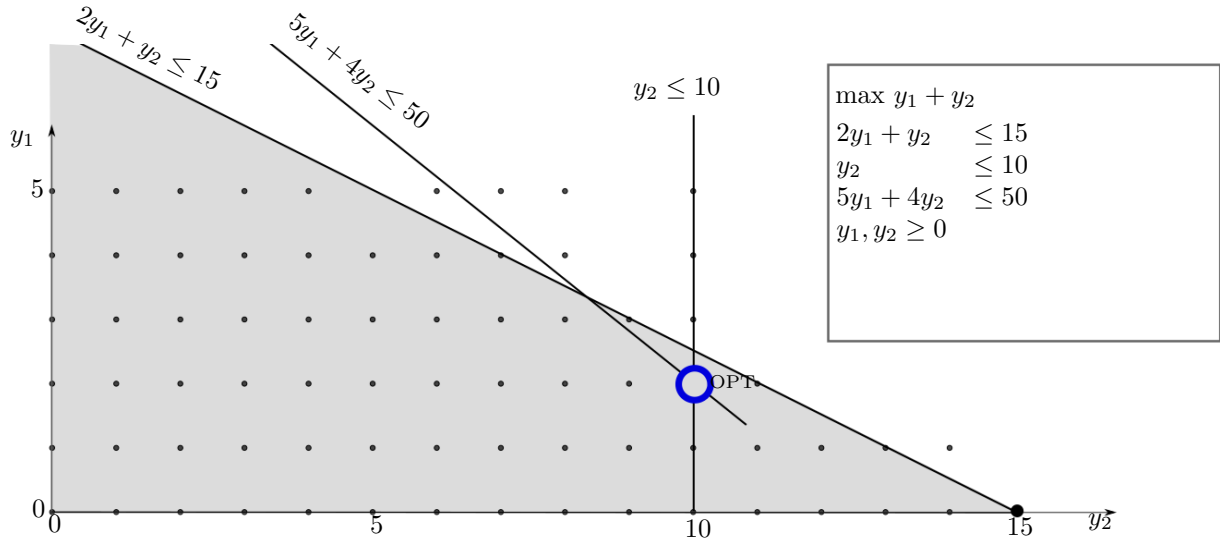
This chapter corresponds to article [Ray Projection for Optimizing Polytopes with Prohibitively Many Constraints in Set-Covering Column Generation, Mathematical Programming 155(1): 147-197, 2016]. A light introduction is available in these slides: `http://cedric.cnam.fr/~porumbed/papers/irmslides.pdf`. The `C++` code is available here: `http://cedric.cnam.fr/~porumbed/irm/`.

Yet I wont simply copy-paste any material from above papers. I first start below by presenting the main ideas in the simplest possible terms, using figures never published before.

2.1 A first intuitive sketch

Let's consider the Linear Program (LP) below with only three constraints (besides $\mathbf{y} \geq \mathbf{0}$). Keep in mind that the goal is to optimize over a polytope $\mathscr{P}$ with prohibitively many constraints. The depicted integer points do not represent integer solutions, but integer projection directions $\mathbf{d}$. We assume that for each such integer $\mathbf{d}$, it's possible to solve in reasonable time the following $\mathbf{0}$-*origin integer projection* subproblem : find the maximum t^* so that $t^*\mathbf{d} \in \mathscr{P}$.

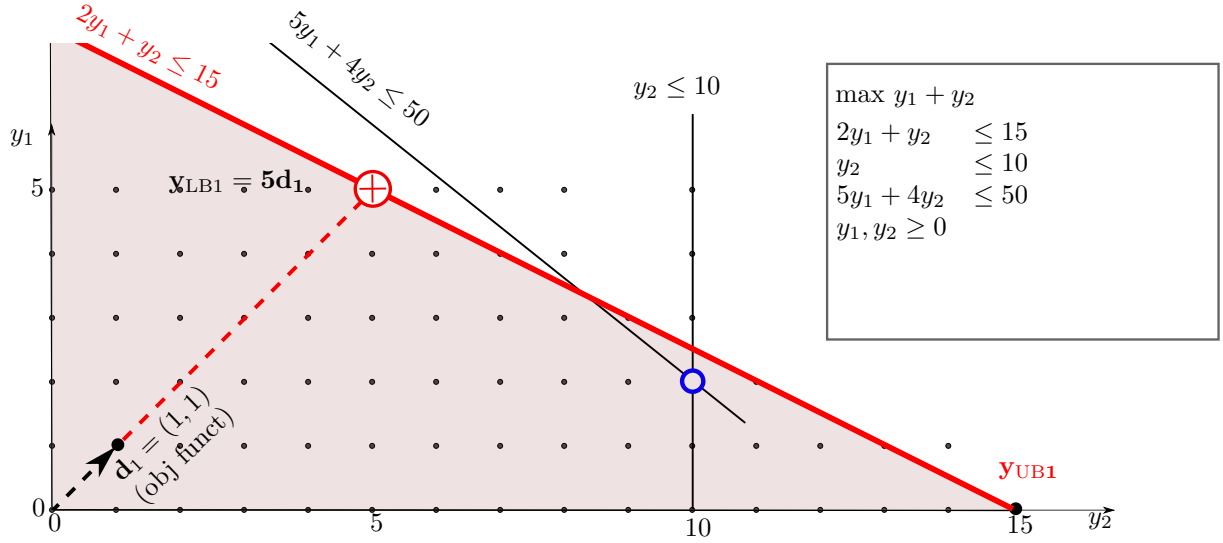
[2] `http://cedric.cnam.fr/~porumbed/papers/sdp.pdf`



Consider the above program and a standard `Cutting-Planes` that at the very first iteration only integrates constraint $2y_1 + y_2 \leq 15$. The feasible area at the first iteration is the shaded area whose optimal solution is $(0, 15)$. At the next iteration, a call to the standard separation subproblem would have to choose between $y_2 \leq 10$ and $5y_1 + 4y_2 \leq 50$ to separate $(0, 15)$. By substituting $y_1 = 0$ and $y_2 = 15$, the separation logic would evaluate the violation level of $y_2 \leq 10$ at $15 - 10 = 5$ and the violation level of $5y_1 + 4y_2 \leq 50$ at $4 \cdot 15 - 50 = 10$, hence choosing the latter. But you can simply notice in this figure that this is not the case. We will see below in sub-section "Iteration 2" that the **0**-*origin integer projection* subproblem correctly identifies $y_2 \leq 10$ as the constraint that separates $(0, 15)$ in a deeper manner.

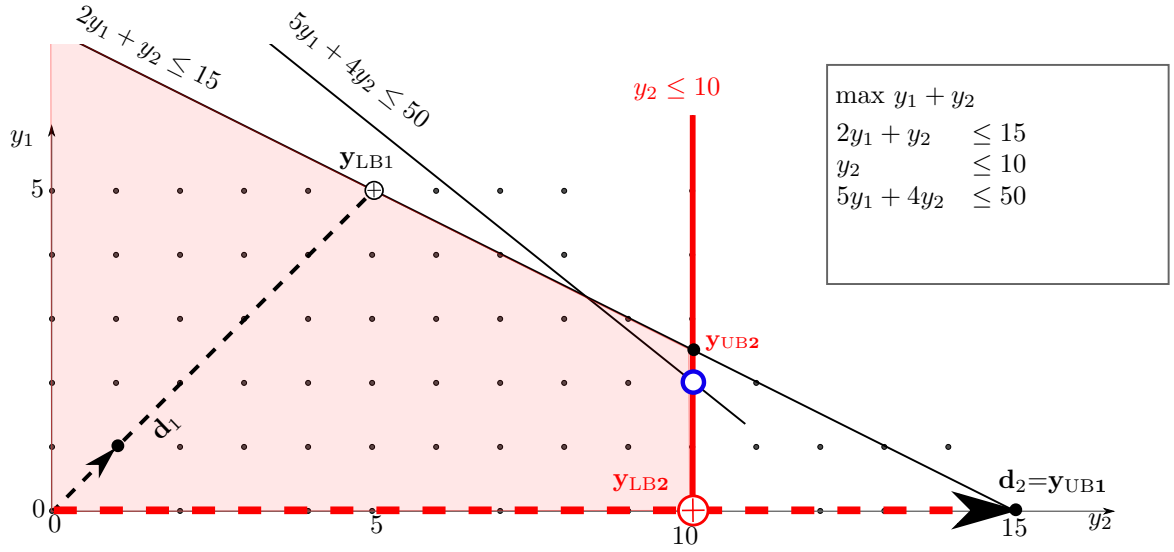
2.1.1 Iteration 1

The first chosen projection direction is the objective function, because it's the direction with the fastest rate of objective improvement; we write $\mathbf{d}_1 = [1\ 1]$. Note in figure below how the **0**-integer projection subproblem for $\mathbf{d}_1$ returns $t^* = 5$, which gives a first inner (feasible solution) $\mathbf{y}_{\text{LB1}} = \mathbf{5d_1} = (\mathbf{5}, \mathbf{5})$. This pierce point is depicted by the cross inside the red circle. The subproblem will also return the constraint $2y_1 + y_2 \leq 15$, also called the first-hit constraint emphasized in red in figure below. We also use the Cutting-Planes logic: considering only this constraint, we obtain a first outer point $\mathbf{y}_{\text{UB1}} = (\mathbf{0}, \mathbf{15})$, which is the optimal (outer) solution over the light-red area below.



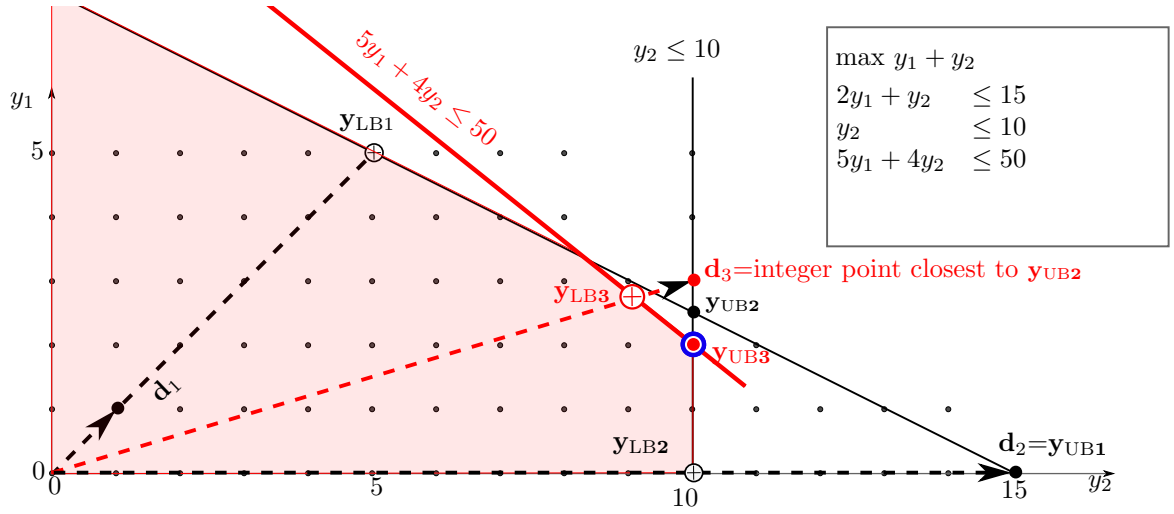
2.1.2 Iteration 2

Choosing the next projection direction in a thoughtful manner is important for numerical success. A Cutting-Planes would proceed by separating $\mathbf{y}_{\mathrm{UB1}} = (\mathbf{0}, \mathbf{15})$. We proceed by applying the **0**-*origin integer projection* subproblem on the same point, *i.e.*, we choose $\mathbf{d}_2 = \mathbf{y}_{\mathrm{UB1}} = (\mathbf{0}, \mathbf{15})$, see the big horizontal arrow at the bottom of next figure. The subproblem will return $t^* = \frac{2}{3}$, which gives the feasible solution $\mathbf{y}_{\mathrm{LB2}} = \frac{\mathbf{2}}{\mathbf{3}}\mathbf{d_2} = (\mathbf{0}, \mathbf{10})$; this first-hit point is depicted by the cross inside the circle. The subproblem also returns the first-hit facet $y_2 \leq 10$, see the vertical constraint in red. Using the Cutting-Planes logic, the outer approximation of the feasible polytope is the area depicted in light-red below, whose optimal solution is point $\mathbf{y}_{\mathrm{UB2}} = (\mathbf{2.5}, \mathbf{10})$ that still violates the third constraint.



2.1.3 Iterations 3-4

Notice now that the sought optimal solution (blue empty circle) is now on the segment between $\mathbf{y}_{\mathrm{LB2}}$ and $\mathbf{y}_{\mathrm{UB2}}$. Yet in the context of a dual polytope in a Column Generation model, it may be computationally too difficult to project from $\mathbf{y}_{\mathrm{LB2}}$ and $\mathbf{y}_{\mathrm{UB2}}$. We will not address such projections in this chapter, but we'll stick to a simpler **0**-*origin integer projection* that projects from **0** towards integer directions only. We can *not* take $\mathbf{d}_3 = \mathbf{y}_{\mathrm{UB2}}$, because this is not an integer point. The algorithm from this chapter will scan all integer points close to the segment $[\mathbf{y}_{\mathrm{LB2}}, \mathbf{y}_{\mathrm{UB2}}]$ and solve the projection subproblem on each of them. For this, we will generate points $\mathbf{y}_{\mathrm{LB2}} + \beta(\mathbf{y}_{\mathrm{UB2}} - \mathbf{y}_{\mathrm{LB2}})$, where β is decreased step by step from 1 to 0. In this example, we would have to generate four values of β like 1, $\frac{4}{5}$, $\frac{2}{5}$, and, resp., 0, so that $\lfloor \mathbf{y}_{\mathrm{LB2}} + \beta\,(\mathbf{y}_{\mathrm{UB2}} - \mathbf{y}_{\mathrm{LB2}}) + [\mathbf{0.5}\ \mathbf{0.5}] \rfloor$ becomes $(3, 10)$, $(2, 10)$, $(1, 10)$, and resp., $(0, 10)$.



The first such choice is $\mathbf{d}_3 = (3, 10)$. By solving the $\mathbf{0}$-*origin integer projection* subproblem on it, we obtain the inner solution $\mathbf{y}_{\mathrm{LB}\mathbf{3}}$ and we discover the last constraint. The optimal solution of the outer approximation becomes $\mathbf{y}_{\mathrm{UB}\mathbf{3}} = (\mathbf{2}, \mathbf{10})$, which is the sought optimal solution of the original problem. A final (non-depicted) iteration with $\mathbf{d}_4 = (2, 10)$ will return $t^* = 1$, certifying that this solution can not be separated, which means it is optimal.

2.2 Context and related ideas

As presented up to now, the above projection idea is relatively simple as a generic building block. It has certainly nothing revolutionary and other researchers might have had similar thoughts. Yet the main difficulty is to act on this idea long enough to turn it into a very practical algorithm. The key to numerical success lies in addressing all details and particular cases. The final pseudo-code in Algorithm 1 has 26 lines; Algorithm 3 (also built upon a projection idea in Section 4) has 23 lines. To the best of my knowledge, there is no related work that involves very similar pseudo-codes.

2.2.1 Converging through outer or/and inner solutions

The optimization of Linear Programs (LPs) with unmanageably-many constraints has a rich and long history in mathematical programming, *e.g.*, consider the primal LP in cutting-planes methods or the dual LP in Column Generation (CG). A highly successful approach to solve such LPs rely on dual (outer) methods: converge to the optimal solution through a series of outer (infeasible solutions). A Cutting-Planes algorithm uses a separation subproblem to progressively remove infeasibility; using such a separation oracle approach one can generate at each iteration a new constraint that separates the current infeasible (outer) solution. This infeasible solution is the optimum of some "outer polytope" defined only by the constraints generated up to now.

While the algorithms described in this thesis do also follow a *dual method* logic (*i.e.*, to get upper bounds using "outer polytopes" as above), they were first designed to combine the outer logic with an inner (or primal) one. An inner method proceeds by converging a sequence of interior (feasible) solutions rather than exterior solutions. Such methods were first studied to solve Integer Linear Programs (ILPs); some related references in [30, §1] or in [56, § 4.1] date to the 1960s. They rely on repeating the following operation: start from the current integer feasible solution and find a cut on the simplex tableau such that the resulting simplex pivot leads to a new feasible solution of higher quality that is still integer. These cuts can be seen as a tool for driving the current primal feasible solution towards the optimum integer solution.

It may be difficult to transpose all related ideas above to Column Generation (CG), because this field has some particularities. In CG, the inner and outer methods behave as described above with respect to a dual polytope. Its prohibitively-many constraints come from an exponential set of primal columns. These columns represent patterns of a combinatorial optimization problems, for instance clients serviced by a route. Generating one such primal column (or dual constraint) may require solving a combinatorial NP-hard problem. I refer the reader to [39, § 1.1.2] for more ideas related to CG.

2.2.2 The idea of 0-integer projection, in-out separation, line-search or general projection

As hinted above, certain related ideas arise in primal algorithms for ILPs. Such algorithms are often described in terms of pivot operations in the Simplex tableau and of (Chvátal-Gomory) cuts. However, their dynamics can also be presented as a process that moves from one integer feasible solution to another by *integer augmentation*, *e.g.*, see Step 4 in the algorithm from [30, §3.1] or Step (2.b) in Algorithm 1.1 of [56]. In this latter example, the next *integer* feasible solution is determined by advancing an *integer* length *from the current solution along an integer augmenting direction*. These integer lengths are essential in an Integer LP context, because the goal is to produce integer solutions.

In the primal cutting plane algorithms from [30, § 3.1], it is sometimes useful to solve the separation problem by looking only for constraints that are saturated by "a boundary point". In the context of convex optimization, the method of Veinott [59] determines this boundary point as the intersection between the feasible area and the line joining the interior and exterior solution.

Certain standard `Cutting-Planes` algorithms do use interior points to ensure a stable behavior of the convergent process. For instance, instead of solving the separation problem on the optimal solution $\mathbf{y}_{\mathrm{out}}$ of the current outer (dual) polytope, one can replace $\mathbf{y}_{\mathrm{out}}$ with a solution situated on the segment joining $\mathbf{y}_{\mathrm{out}}$

to an interior (feasible) solution of $\mathbf{y}_{\text{in}}$. This is called the *in-out separation* in [3, p. 4]. Algorithm 1 from [3, p. 6] defines $\mathbf{y}_{\text{sep}} = \alpha\mathbf{y}_{\text{in}} + (1-\alpha)\mathbf{y}_{\text{out}}$ for some $\alpha \in (0, 1]$. If $\mathbf{y}_{\text{sep}} \notin \mathscr{P}$, the constraint returned by separating $\mathbf{y}_{\text{sep}}$ may be more efficient. Otherwise, if $\mathbf{y}_{\text{sep}} \in \mathscr{P}$, this algorithm considers a kind of second inner point $\mathbf{y}'_{\text{in}} = \mathbf{y}_{\text{sep}}$ and finds a second separation point $\mathbf{y}_{\text{sep}} = \alpha\mathbf{y}'_{\text{in}} + (1-\alpha)\mathbf{y}_{\text{out}}$. It thus generates a recursion defined by an α-strategy, so that this second separation point is actually $\mathbf{y}_{\text{sep}} = \alpha\left(\alpha\mathbf{y}_{\text{in}} + (1-\alpha)\,\mathbf{y}_{\text{out}}\right) + (1-\alpha)\mathbf{y}_{\text{out}}$. I found no claim that any such α-strategy ends up in determining the exact pierce point.

A form of projection arises in the field of Submodular Function Minimization (SFM). One of the simplest approaches for optimizing the submodular polytope is the *Greedy* algorithm [33, Sec. 2.1]. Given a linear order of the variables, *Greedy* progresses towards the optimum by increasing one by one each variable to its maximum value. Each such maximum value can be seen as a "maximum step length" that one can advance in the direction of the variable until intersecting a polytope facet. Generalizing this idea, Schrijver's algorithm advances on more complex directions, by simultaneously increasing one variable while decreasing some other. In certain cases, one can exactly determine the maximum feasible step length on such directions [33, Lemma 2.5], but Schrijver's algorithm is making use of lower bounds for it. The problem of finding the intersection of a line with a sub-modular polytope (or polymatroid) is referred to as the "line search problem" [34] or as the "intersection problem" [12]. The intersection algorithm from [34] consists of solving a minimum ratio problem. Our projection subproblem will also reduce to a ratio minimization problem – see (2.D) a few pages below or (3.B) next chapter.

The above literature review reflects my knowledge on the topic around 2014 when [39] was published. The situation has evolved since then. The study [7] of 2019 uses a similar **0**-origin intersection concept (see their Figure 1), although not with integer directions. Their study is also written in a different spirit: while my goal has always been numerical success first, they develop and provide more theory than experimental results. Yet their idea was followed by [4] and probably by (many) others and implemented in the Benders reformulation module of `Cplex`. I did write to the authors of [4] in June 2020, signaling that the program (20) they used to solve the subproblem is the dual of my (3.2.6) from [42], modulo notational translations or reformulations. I would be happy if these kind of ideas ever become more successful in `Cplex` or other software; who gets most credit for it is a different story, for whatever that means.

Yet I apologise for other ideas out there that escaped my attention. I must confess I did not spend very long hours to find them all. Such work could provide a listing of related concepts that may cover many pages, but I'm a bit skeptical this would shed too much light into the question. If I found a form of projection even in the field of submodular function minimization (as above), I imagine I could find others in many other fields. For instance, there are similar line search problems in the field of continuous optimization. Section 4.5 is devoted to related literature in SDP optimization. Covering all such fields can lead to an itemized list of disparate ideas that could constitute together a fine collection, but they would not form the basis of a firmly coherent solution method. I prefer to generally stick to pieces of knowledge that I'm able to place in the overall algorithmic framework like stones that find their place into a coherent mosaic. We will see that the proposed algorithm integrates many other building blocks besides the question of projection.

I repeat that the subproblem discussed here is only one building-block; it is called twice in the pseudo-code of 26 lines (Algorithm 1, p. 14). Even today I find no idea in the literature that gets very close to these 26 lines.

2.3 General Model and Algorithmic Overview

Given a set of (unmanageably-many) constraints C, we aim at solving:

$$\max\left\{\mathbf{b}^\top\mathbf{y} : \mathbf{a}^\top\mathbf{y} \leq c_a,\ \forall(\mathbf{a}, c_a) \in C\right\} = \max\left\{\mathbf{b}^\top\mathbf{y} : \mathbf{y} \in \mathscr{P}\right\}, \tag{2.A}$$

where all vectors have size k. Throughout this thesis, we will also address a few variations of (2.A), *e.g.*, we actually $\mathbf{y} \geq \mathbf{0}$ in Column Generation (CG). In Chapter 4, the set C will become infinite because it will characterize semidefinite-positive matrices. In the current chapter, the above program is actually the dual of *a fractional relaxation* of an *integer* CG program and we actually seek the best *rounded-up* objective value, replacing "$\max \mathbf{b}^\top\mathbf{y}$" with "$\max\left\lceil\mathbf{b}^\top\mathbf{y}\right\rceil$"; the origin $\mathbf{0}_k$ is usually feasible because we often have $c_a \geq 0\ \forall(\mathbf{a}, c_a) \in C$.

A well-known example of a dual Column Generation program fitting this setting is the dual LP of the Gilmore-Gomory model for `Cutting-Stock`; each $(\mathbf{a}, c_a) \in C$ represents an integer solution (pattern) of a knapsack sub-problem: $c_a = 1$ is a constant pattern cost, a_i is the number of copies of item i included

(packaged) in the pattern. In vehicle or arc routing problems, $\mathbf{a}$ represents a feasible route in some graph, c_a is usually a route cost depending on certain traversed distances and $\mathbf{b}$ is traditionally $\mathbf{1}_k$ (each service is required only once). In location or p-median problems, $\mathbf{a}$ can represent a cluster of customers and c_a the cost of reaching them. Implicitly or explicitly, (2.A) arises in many other `Set-Covering` problems; I designed projection algorithms only for `Elastic Cutting-Stock` and `Capacitated Arc Routing` in [39].

The standard `Cutting-Planes` for solving this LP maintains at each iteration `it` an outer approximation $\mathscr{P}_{\mathtt{it}} \supset \mathscr{P}$ of $\mathscr{P}$ obtained by restricting the constraint set C to some subset $C_{\mathtt{it}}$. This $\mathscr{P}_{\mathtt{it}}$ corresponds to the dual of a restricted master problem in CG. To (try to) separate the current optimal solution $\mathbf{y}_{\text{out}} = \mathtt{opt}(\mathscr{P}_{\mathtt{it}})$ of $\mathscr{P}_{\mathtt{it}}$, the most standard Cutting-Planes usually solves the separation sub-problem $\min_{(\mathbf{a},c_a)\in C} c_a - \mathbf{a}^\top \mathbf{y}_{\text{out}}$. If the optimum value of this sub-problem is less than 0 for some $(\overline{\mathbf{a}}, \overline{c_a}) \in C$, then $\mathbf{y}_{\text{out}}$ is infeasible. In this case, the `Cutting-Planes` method inserts $\overline{\mathbf{a}}^\top \mathbf{y} \leq \overline{c_a}$ into the current constraint set (*i.e.*, it performs $C_{\mathtt{it}+1} = C_{\mathtt{it}} \cup \{(\overline{\mathbf{a}}, \overline{c_a})\}$), so as to construct a new more refined outer approximation $\mathscr{P}_{\mathtt{it}+1}$ and to separate $\mathbf{y}_{\text{out}} \notin \mathscr{P}_{\mathtt{it}+1}$. The process is repeated by (re-)optimizing over $\mathscr{P}_{\mathtt{it}+1}$ at the next iteration, until the current optimal outer solution $\mathbf{y}_{\text{out}}$ becomes optimal (non-separable). The proposed **0**-Integer-Projection approach replaces the well-known separation sub-problem with the following sub-problem.

Definition 2.A. *(**0**-origin integer projection sub-problem) Given an integer projection direction $\mathbf{d} \in \mathbb{Z}^k$, determine: (1) the maximum step length t^* such that $t^*\mathbf{d}$ is feasible inside $\mathscr{P}$, i.e., $t^* = \max\{t \geq 0 : t\mathbf{d} \in \mathscr{P}\}$ and (2) a first-hit constraint $(\mathbf{a}, c_a) \in C$ satisfied with equality by the $t^*\mathbf{d}$. The solution $t^*\mathbf{d}$ is referred to as the pierce (or first-hit) point. If $\mathbf{d}$ is an unbounded ray of $\mathscr{P}$, the sub-problem returns $t^* = \infty$.*

If we only use integer directions in this chapter, it is because otherwise the new sub-problem may become too difficult to solve in practice. Since it generalizes the separation subproblem, it may even seem computationally far more expensive, but we will see this is not always the case. The use of integer directions will render the sub-problem tractable using a Dynamic Programming scheme based on states indexed by integer direction values. We will see that under such conditions, the projection sub-problem can be even easier than the CG sub-problem, especially when no other integer data is available to index states in Dynamic Programming, *i.e.*, if the CG sub-problem input consists of fractional (or large-range) values.

The proposed **0**-integer-direction approach offers a relatively-high flexibility in choosing the directions: any point in the space of $\mathscr{P}$ can be potentially used as a direction. This flexibility can be useful for simplifying the projection sub-problems: it is generally easier to solve a (sub-)problem on some controllable input data $\mathbf{d}$ than on a rather unpredictable $\mathbf{y}_{\text{out}} = \mathtt{opt}(\mathscr{P}_{\mathtt{it}})$ determined by optimizing the outer polytope $\mathscr{P}_{\mathtt{it}} \supset \mathscr{P}$ constructed at some iteration `it`. Yet, this idea could be used in standard Column Generation, by rounding $\mathbf{y}_{\text{out}}$.

2.4 Pseudo-code and formalization of the 0→Integer Projection Approach

The very first direction is the objective function vector $\mathbf{d} = \mathbf{b}$ (or a rounding of $\mathbf{b}$ when necessary). By solving the projection sub-problem for $\mathbf{d}$, we find an inner point $t^*\mathbf{d}$ and a $(t^*\mathbf{d})$-tight constraint. We then optimize the current outer polytope, *i.e.*, the polytope delimited by the first-hit facets discovered while computing projections. This generates an outer solution $\mathbf{y}_{\text{out}}$ so that $\mathbf{b}^\top \mathbf{y}_{\text{out}}$ is an upper bound. We then have to determine the next direction and repeat. Determining this next direction is a key step. We do it by seeking integer solutions in the proximity of the segment joining $\mathbf{y}_{\text{in}}$ and $\mathbf{y}_{\text{out}}$, We formalize this approach into four main steps.

1. *Solve the **0**-origin integer projection sub-problem to determine a lower bound solution* Advance (or shoot) from $\mathbf{0}$ towards a given direction $\mathbf{d}$ until intersecting a first constraint of $\mathscr{P}$ at $t^*\mathbf{d} \in \mathscr{P}$. This leads to a lower bound solution $\mathbf{y}_{\text{in}} = t^*\mathbf{d}$, and to a (first-hit) constraint $\mathbf{a}^\top \mathbf{y} \leq c_a$ that is $\mathbf{y}_{\text{in}}$-tight (*i.e.*, $\mathbf{a}^\top \mathbf{y}_{\text{in}} = c_a$). This first-hit constraint is added to the set C of currently available constraints; we write $C \leftarrow C \cup \{(\mathbf{a}, c_a)\}$, which is a simplification of $C_{\mathtt{it}+1} = C_{\mathtt{it}} \cup \{(\mathbf{a}, c_a)\}$ obtained by dropping the index `it`.

2. *Calculate upper bound solution* $\mathbf{y}_{\text{out}}$ This $\mathbf{y}_{\text{out}}$ is simply determined by optimizing $\mathbf{b}^\top \mathbf{y}$ over $\mathbf{y} \in \mathscr{P}_C$, where $\mathscr{P}_C \supset \mathscr{P}$ is the polytope obtained from $\mathscr{P}$ by considering the set of manageably-many constraints C available up to now. We can write $\mathbf{y}_{\text{out}} = \mathtt{OPT}(\mathscr{P}_C)$, and so, $\mathbf{b}^\top \mathbf{y}_{\text{out}}$ is an upper bound for (2.A).

3. *Generate Next Integer Direction* Given $\mathbf{d}, \mathbf{y}_{\text{in}}$ and $\mathbf{y}_{\text{out}}$, search for new rays among the closest integer points to $\mathbf{d} + \beta(\mathbf{y}_{\text{out}} - \mathbf{y}_{\text{in}})$, with $\beta > 0$. For a fixed β, the closest integer point is determined by applying a simple rounding on each of the k coordinates of $\mathbf{d}+\beta(\mathbf{y}_{\text{out}}-\mathbf{y}_{\text{in}})$. New potential better rays $\mathbf{d}_{\text{new}}$ are iteratively generated by gradually increasing β until one of them leads to some lower or upper bound update (using Step 4 below). In fact, if none of the proposed rays can update either bound, we consider that the current ray coefficients are too small (imprecise) to reduce the gap. In this case, the proposed method calls a "discretization refining" routine that increases the ray coefficients: multiply $\mathbf{d}$ by 2 and divide t^* by 2; as such, $\mathbf{y}_{\text{in}} = t^*\mathbf{d}$ and $\mathbf{y}_{\text{out}}$ stay unchanged. This allows our ray generators to find new larger rays (*i.e.*, from updated $\mathbf{d}, \mathbf{y}_{\text{out}}$ and $\mathbf{y}_{\text{out}}$) with higher chances of improving the bounds (see below), at the cost of a potential slowdown of the projection calculations. The idea is that larger rays lead to a higher resolution in constructing high-quality projection directions. For example, if $n = 2$ and all ray coefficients are binary, we can only project towards $(1, 0)$, $(1, 1)$ or $(0, 1)$. If all coefficients are positive integers up to 4, we can generate a dozen of directions; if we go up to 15, we can generate hundreds of directions.

4. *Complete a major iteration: update* $\mathbf{y}_{\text{in}}$ *or* $\mathbf{y}_{\text{out}}$ The new directions $\mathbf{d}_{\text{new}}$ generated by Step 3 are iteratively provided as input to the $\mathbf{0}$-*origin integer projection* sub-problem. Each such $\mathbf{d}_{\text{new}}$ can either lead to a lower bound improvement (if the first-hit solution $t^*_{\text{new}}\mathbf{d}_{\text{new}} \in \mathscr{P}$ dominates the best inner solution known up to now) or to an upper bound update (if the first-hit constraint separates $\mathbf{y}_{\text{out}}$). As soon as $\mathbf{y}_{\text{in}}$ or $\mathbf{y}_{\text{out}}$ is updated, the ray generation from Step 3 is *restarted* with new input data (*i.e.*, with updated $\mathbf{d}, \mathbf{y}_{\text{out}}$ or $\mathbf{y}_{\text{in}}$). If neither bound can be improved as above, we continue generating new rays as described in Step 3; as such, Step 3 and Step 4 are actually iteratively intertwined until one of the new rays triggers an update of $\mathbf{y}_{\text{in}}$ or $\mathbf{y}_{\text{out}}$. In the worst case, Step 3 uses discretization refining to construct rays with increasingly larger coefficients. This can eventually make the new rays pass arbitrarily close to the segment joining $\mathbf{y}_{\text{in}}$ and $\mathbf{y}_{\text{out}}$, which is guaranteed to eventually trigger some bound update. I needed a proof over three pages to show this process is finitely-convergent, see Theorems 1-2 in Section 2.4.2 (worst case scenario) of [39].

Algorithm 1 above provides the general pseudocode. The outer `repeat-until` loop (Lines 5-26) performs a *major iteration*, intertwining the ray generation routines (Step 3 above, roughly corresponding to Lines 10-15) and the mechanism for updating $\mathbf{y}_{\text{in}}$ or $\mathbf{y}_{\text{out}}$ (Step 4 above, implemented in Lines 17-25). The inner `repeat-until` loop (Lines 9-19) performs a minor iteration, in which new rays are iteratively generated until one of the bounds can be updated. Except `zeroIntProjSubprob` (see Section 2.4.1), all remaining routines are generic with respect to (2.A):

`optimize`$(\mathscr{P}_C, \mathbf{b})$ stands for any LP algorithm that can return an optimal vertex $\mathbf{y}_{\text{out}}$ that maximizes $\mathbf{b}^\top\mathbf{y}$ over $\mathbf{y} \in \mathscr{P}_C$.

`nextDirect` $(\mathbf{d}, \mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}, \mathbf{d}_{\mathbf{prev}})$ returns the next integer ray after $\mathbf{d}_{\text{prev}}$ in a sequence of new rays constructed from $\mathbf{d}$, $\mathbf{y}_{\text{in}}$ and $\mathbf{y}_{\text{out}}$. This sequence is actually generated by the first `nextDirect` call at Line 10 when $\mathbf{d}_{\text{prev}} = \mathbf{0}_k$ (see Section 2.4.3).

2.4.1 How to solve 0→integer projections in Column Generation models

We propose solving the $\mathbf{0}$-*origin integer projection* sub-problem from Definition 2.A (p. 12) using a Dynamic Programming (DP) scheme relying on proposition below.

Proposition 2.B. *Given a dual polytope* $\mathscr{P}$ *defined by a constraint set* C *such that* $c_a \geq 0$ *for any* $(\mathbf{a}, c_a) \in C$*, the min-max equivalence* $t^* = t^\#$ *holds for any* $\mathbf{d} \in \mathbb{R}^k$*:*

$$t^* = \max\{t \in \mathbb{R}_+ \cup \{\infty\} : \ t\mathbf{d} \in \mathscr{P}\}$$

$$t^\# = \min\left\{ \min_{\substack{(\mathbf{a},c_a)\in C \\ \mathbf{a}^\top\mathbf{d}>0}} \frac{c_a}{\mathbf{a}^\top\mathbf{d}}, \infty \right\}$$

Proof. The equivalence may seem straightforward, but some exceptional cases have to be carefully addressed. Notice that $t^\#$ in the minimization problem can be ∞ only if all $(\mathbf{a}, c_a) \in C$ satisfy $\mathbf{a}^\top\mathbf{d} \leq 0$. Indeed, only

Algorithm 1 **0**-Integer Direction Method for optimizing $\mathbf{b}^\top\mathbf{y}$ over large-scale polytope $\mathscr{P}$ in (2.A)

1: $C \leftarrow \emptyset$ ▷ abusing notation a bit, C is an ever increasing set not meant to cover all constrains of $\mathscr{P}$
2: $\mathbf{d} \leftarrow \mathbf{b}$ ▷ scale $\mathbf{b}$ down and make it integer if all b_i are too large or fractional
3: $t^*, \mathbf{a}, c_a \leftarrow$`zeroIntProjSubprob`$(\mathbf{d})$
4: $C \leftarrow C \cup \{(\mathbf{a}, c_a)\}$
5: **repeat**
6: $\quad\mathbf{y}_{\text{in}} \leftarrow t^*\mathbf{d}$ ▷ if $t^* = \infty$, return ∞
7: $\quad\mathbf{y}_{\text{out}} \leftarrow$`optimize`$(\mathscr{P}_C, \mathbf{b})$ ▷ optimal solution (or extremal ray) of $\mathscr{P}_C$, *i.e.*, of $\mathscr{P}$ restricted to C
8: $\quad\mathbf{d}_{\text{prev}} \leftarrow \mathbf{0}_k$ ▷ $\mathbf{d}_{\text{prev}}$ stands for previous direction new ray ($\mathbf{0}_k$ means none yet)
9: $\quad$**repeat**
10: $\quad\quad\mathbf{d}_{\text{new}} \leftarrow$`nextDirect`$(\mathbf{d}, \mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}, \mathbf{d}_{\text{prev}})$ ▷ return $\mathbf{0}_k$ if no more rays can be found after $\mathbf{d}_{\text{prev}}$
11: $\quad\quad$**while** $(\mathbf{d}_{\text{new}} = \mathbf{0}_k)$ ▷ while no more rays:
12: $\quad\quad\quad\mathbf{d} \leftarrow 2\mathbf{d}$, $t^* \leftarrow \frac{t^*}{2}$ ▷ i) discretization refining
13: $\quad\quad\quad\mathbf{d}_{\text{new}} \leftarrow$`nextDirect` $(\mathbf{d}, \mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}, \mathbf{0}_k)$ ▷ ii) re-try ray generation
14: $\quad\quad$**end while**
15: $\quad\quad\mathbf{d}_{\text{prev}} \leftarrow \mathbf{d}_{\text{new}}$
16: $\quad\quad t^*_{\text{new}}, \mathbf{a}, c_a \leftarrow$`zeroIntProjSubprob`$(\mathbf{d}_{\text{new}})$
17: $\quad\quad newLb \leftarrow \left(\mathbf{b}^\top (t^*_{\text{new}}\mathbf{d}_{\text{new}}) > \mathbf{b}^\top\mathbf{y}_{\text{in}}\right)$ ▷ better lower bound if expression is true
18: $\quad\quad newUb \leftarrow \left(\mathbf{a}^\top\mathbf{y}_{\text{out}} > c_a\right)$ ▷ new constraint violated by current $\mathbf{y}_{\text{out}}$
19: $\quad$**until** ($newLb$ or $newUb$)
20: $\quad$**if** ($newLb$)
21: $\quad\quad\mathbf{d} \leftarrow \mathbf{d}_{\text{new}}$, $t^* \leftarrow t^*_{\text{new}}$ ▷ if $\mathbf{d}_{\text{new}} \gg \mathbf{d}$, scale $\mathbf{d}_{\text{new}}$ down to avoid implicit discretization refining
22: $\quad$**end if**
23: $\quad$**if** ($newUb$)
24: $\quad\quad C \leftarrow C \cup \{(\mathbf{a}, c_a)\}$ ▷ $\mathbf{y}_{\text{out}}$ will be updated at the iteration at Line 7
25: $\quad$**end if**
26: **until** $\left(\mathbf{b}^\top\mathbf{y}_{\text{out}} - \mathbf{b}^\top\mathbf{y}_{\text{in}} \leq \epsilon\right)$ ▷ [39, Theorem 2] shows the convergence is finite for any $\epsilon > 0$

such a situation would allow $\mathbf{a}^\top(t\mathbf{d}) \leq c_a$ to be true for any $(\mathbf{a}, c_a) \in C$ *and for any indefinitely large* t. In this case, $\mathbf{d}$ is an extremal ray in $\mathscr{P}$ and the intersection sub-problem (Definition 2.A) also returns $t^* = \infty$. If the minimization problem returns $t^\# \neq \infty$, then any $t > t^\#$ would lead to $t\mathbf{d} \notin \mathscr{P}$: such $t\mathbf{d}$ would violate a constraint $\mathbf{a}^\top\mathbf{y} \leq c_a$ that minimizes the above ratio. This simply follows from: $\mathbf{a}^\top(t\mathbf{d}) = t(\mathbf{a}^\top\mathbf{d}) > t^\#(\mathbf{a}^\top\mathbf{d}) = c_a$. Similarly, we observe that any $t \leq t^\#$ does lead to $t\mathbf{d} \in \mathscr{P}$. As such, the maximum step length t^* returned by Definition 2.A is $t^* = t^\#$. □

Observation 2.C. *If $c_a = 1 \forall (\mathbf{a}, c_a) \in C$, then the above ratio is no longer a proper ratio and the projection reduces to maximizing $\mathbf{a}^\top\mathbf{d}$, i.e., to a simple separation. A* **0***-origin projection (with integer or non-integer* $\mathbf{d}$*) is equivalent to normalizing all constraints (to make them all have the same right-hand side value) and then choosing one by separating* $\mathbf{d}$*. Consider choosing between $2y_1 + 3y_2 \leq 4$ and $200y_1 + 300y_2 \leq 450$. The standard separation logic chooses the later constraint because $200+300-450 > 2+3-4$, while the projection logic chooses the former, which is the stronger constraint as you can check by normalizing them.*

2.4.2 Projection by Dynamic Programming in CG models (for Elastic Cutting-Stock)

We give an example of applying above Proposition 2.B to calculate projections on a particular CG problem using Dynamic Programming (DP). In [39, § 3.2] you can find a more general discussion, pointing towards generalizations of the idea for other problems like `Arc-Routing`.

Using the models from Section 2.3, recall that the Gilmore-Gomory model for `Cutting-Stock` is a direct particularization of the dual of (2.A). Each constraint $\mathbf{a}^\top\mathbf{y} \leq c_a$ is associated to a pattern $\mathbf{a} \in \mathbb{Z}^k$ which indicates for each article $i \in [1..k]$ the number of times item i has to be cut (from a single log). The pattern cost $c_a = 1$ and the feasibility of $\mathbf{a}$ only relies on knapsack condition $\mathbf{a}^\top\mathbf{w} \leq Q$, where $\mathbf{w} \in \mathbb{R}^k$ are the lengths of the items. In the elastic version of `Cutting-Stock`, feasible patterns are allowed to exceed a *base capacity* Q by paying a penalty cost. The larger the excess over Q, the higher the elastic pattern penalty. Let us give a natural bin-packing illustration: a knapsack (vehicle) constructed to hold $Q = 15$ kilograms (tonnes) would be very often able to hold 16 kilograms (tonnes), but this would bear some additional cost

(*e.g.*, for unwanted damage risk). We consider however a maximum extended capacity $Q_{ext} > Q$ that can *never* be exceeded, *e.g.*, we will use $Q_{\text{ext}} = 2Q$. Formally, the elastic cost of pattern $\mathbf{a} \in \mathbb{Z}_+^k$ is:

$$c_a = f\left(\frac{\mathbf{a}^\top \mathbf{w}}{Q}\right), \tag{2.B}$$

where f is a non-decreasing `Elastic` function $f : [0, \frac{Q_{\text{ext}}}{Q}] \to \mathbb{R}_+$ with a fixed value of 1 over $[0, 1]$.

While the elastic terminology is not widespread in the `Cutting-Stock` literature, this problem is not new. First, a stair-case function f can lead to `Variable-Size Bin Packing` or `Multiple-Length` Cutting-Stock. More generally, the new problem can be interpreted as `Residual Cutting Stock` in the typology of [60] and reformulated as `Bin-Packing with Usable Leftovers`. However, (2.A) becomes:

$$\left.\begin{array}{ll} \max \mathbf{b}^\top \mathbf{y} & \\ \mathbf{a}^\top \mathbf{y} \leq c_a = f\left(\frac{\mathbf{w}^\top \mathbf{a}}{Q}\right), & \forall (\mathbf{a}, c_a) \in C \\ y_i \geq 0 & i \in [1..k] \end{array}\right\} \mathscr{P}, \tag{2.C}$$

where $C = \left\{(\mathbf{a}, c_a) \in \mathbb{Z}_+^k \times [1, \infty) : \ \mathbf{a}^\top \mathbf{w} \leq Q_{\text{ext}} = 2Q, \ c_a = f\left(\frac{\mathbf{a}^\top \mathbf{w}}{Q}\right)\right\}$. The objective function comes from the demands $\mathbf{b} \in \mathbb{Z}_+^k$ in the dual of (2.C).

We formulate the **0**-origin projection sub-problem using Proposition 2.B. Given a base capacity $Q \in \mathbb{R}$, weights $\mathbf{w} \in \mathbb{R}^k$, and non-decreasing function $f : [0, 2] \to \mathbb{R}$ (with $f(x) = 1, \forall x \in [0, 1]$), and direction $\mathbf{d} \in \mathbb{Z}_+^k$, find a feasible pattern $\mathbf{a} \in \mathbb{Z}_+^k$ (with $\mathbf{a}^\top \mathbf{d} > 0$) that minimizes:

$$\min \frac{f\left(\frac{\mathbf{w}^\top \mathbf{a}}{Q}\right)}{\mathbf{d}^\top \mathbf{a}}. \tag{2.D}$$

Notice the separation sub-problem would replace the above ratio with a difference $f\left(\frac{\mathbf{w}^\top \mathbf{a}}{Q}\right) - \mathbf{d}^\top \mathbf{a}$. This difference can be minimized by extending the Dynamic Programming (DP) scheme for the standard knapsack-problem. Instead of generating a state for each realizable total weight $\mathbf{w}^\top \mathbf{a}$ in $[1..Q]$, we simply replace this interval with $[1..Q_{\text{ext}}] = [1..2Q]$. As such, we generate a state for each realizable weight $\mathbf{w}^\top \mathbf{a} \in [1..2Q]$ and we evaluate the cost of the state by the above difference.

In all problems studied in [39], we calculate the **0**-origin projections by generalizing the Dynamic Programming for the separation sub-problem. For `Arc-Routing`, this may take a few pages, but for `Elastic Cutting-Stock` a few paragraphs will suffice.

Let us interpret (2.D) as a cost/profit ratio minimization objective: $c_a = f\left(\frac{\mathbf{w}^\top \mathbf{a}}{Q}\right)$ is a cost and $\mathbf{d}^\top \mathbf{a}$ is a (feasible) integer profit. The cost/profit ratio is the inverse of the notion of return per investment (profit/cost). The state associated to any profit $p \in \mathbb{Z}_+$ only records the minimum total weight $W(p) = \mathbf{w}^\top \mathbf{a}_p^*$ required to realize a profit $p = \mathbf{d}^\top \mathbf{a}_p^*$. It is clear that it is not necessary to explicitly record the cost $f(\frac{W(p)}{Q})$. Given some other pattern $\mathbf{a}_p \neq \mathbf{a}_p^*$ such that $\mathbf{d}^\top \mathbf{a}_p = \mathbf{d}^\top \mathbf{a}_p^* = p$ and $\mathbf{w}^\top \mathbf{a}_p > W(p)$, the cost of $\mathbf{a}_p$ is dominated by that of $\mathbf{a}_p^*$, *i.e.*, $f(\frac{\mathbf{w}^\top \mathbf{a}_p}{Q}) \geq f(\frac{W(p)}{Q})$, owing to the fact that f is non-decreasing. Thus, the state of pattern $\mathbf{a}_p^*$ is only determined by its profit $p = \mathbf{d}^\top \mathbf{a}_p^*$ and by the weight $W(p) = \mathbf{w}^\top \mathbf{a}_p^*$; we ignore other patterns $\mathbf{a}_p$ such that $\mathbf{d}^\top \mathbf{a}_p = p$ and $\mathbf{w}^\top \mathbf{a}_p > \mathbf{w}^\top \mathbf{a}_p^* = W(p)$.

The DP pseudo-code is provided in Algorithm 2. To fully ensure its technical correctness, we need to investigate more closely how it discovers and updates the states. We need to verify that the states generated from $\mathbf{a}_p^*$ do dominate all states that could have been generated from $\mathbf{a}_p$. Recall $\mathbf{a}_p^*$ is the pattern that minimizes $\mathbf{w}^\top \mathbf{a}_p^*$ among all patterns that realize a profit of p. In other words, we need to check whether the state update condition from Line 12 is correct: can the new state replace the old one only because the new weight *newW* is lower? This is true, because the updated state accepts all the transitions that the old state could have generated from the current point on (even if the older state was discovered at some lower value of j, this does not influence on the transitions that can be generated from the current point on).

The states of Algorithm 2 are indexed by profit values of the form $\mathbf{d}^\top \mathbf{a}$. We say this DP scheme is profit-indexed or **d**-indexed. As for classical weight-indexed DP, the asymptotic running time of profit-indexed DP depends on the number of elements k and on the number of states $|\texttt{states}|$. In profit-indexed DP, $|\texttt{states}|$ depends closely on the number of realizable *profit* values. In weight-indexed DP, the number of states depends closely on the number of realizable *weight* values.

Algorithm 2 Integer **0**-origin projection for `Elastic Cutting-Stock`

Require: direction $\mathbf{d} \in \mathbb{Z}^k$, capacity $Q \in \mathbb{R}$, weights $\mathbf{w} \in \mathbb{R}^k$, demands $\mathbf{b} \in \mathbb{Z}_+^k$ and function f
Ensure: minimum cost/profit t^* ratio

1: $t^* \leftarrow \infty$ ▷ this is an upper bound that will converge to the real t^*
2: `states`$\leftarrow \{0\}$ ▷ a linked list of realizable profits p
3: $W(0) \leftarrow 0$ ▷ a linked list records the minimum weight $W(p), \forall p \in$ `states`
4: **for** i **in** $[1..k]$ ▷ scan each new article
5: **for** j **in** $[1..b_i]$ ▷ at maximum b_i times (no need for more).
6: **for** p **in** `states` ▷ scan only the current states (not updated by the current loop itself)
7: **if** $p + d_i \notin$ `states`
8: `states` $\leftarrow$ `states` $\cup \{p + d_i\}$ ▷ $p + d_i$ is a new realizable profit
9: $W(p + d_i) \leftarrow \infty$ ▷ $W(p + d_i)$ will be updated below
10: **end if**
11: $newW = W(p) + w_i$
12: **if** $newW < 2Q$ **and** $newW < W(p + d_i)$ ▷ state update
13: $W(p + d_i) \leftarrow newW$ ▷ the lost state is not completely deleted
14: $cost = f(\frac{newW}{Q})$ ▷ so as to keep track of precedence relations.
15: $t^* \leftarrow \min(t^*, \frac{cost}{p+d_i})$
16: **end if**
17: **end for**
18: **end for**
19: **end for**
20: **return** t^* ▷ An associated pattern $\mathbf{a}$ can be built from precedence relations between states

Observation 2.D. *The resulting* $\mathbf{d}$*-indexed DP will be faster when the directions* $\mathbf{d} \in \mathbb{Z}_+^k$*, contain low values, for instance if* $\mathbf{d} = \mathbf{b} = \mathbf{1}_k$ *at the first iteration. The profits* $\mathbf{d}$ *are usually much smaller than the weights* $\mathbf{w}$ *of typical instances, at least in the beginning of the overall method. In such cases, the* $\mathbf{w}$*-indexed separation DP can often require substantially more states than Algorithm 2. As long as the values of* $\mathbf{d}$ *are close to* $\mathbf{1}$*, we noticed the number of* $\mathbf{d}$*-indexed states can often stay in the order of tens or hundreds. The number of realizable total weight values* $\mathbf{w}^\top \mathbf{a}$ *is generally much larger, because most instances are defined using weights in the hundreds or the thousands. Thus, as long as* $\mathbf{d}$ *does not become too large by discretization refining (an operation that doubles the magnitude of* $\mathbf{d}$*), the* $\mathbf{d}$*-indexed DP is naturally faster than the classical* $\mathbf{w}$*-indexed DP used for separating. This means the integer* $\mathbf{0}$*-origin projection can be faster than the separation as long as* $\mathbf{d}$ *contains elements of smaller magnitude than* $\mathbf{w}$*.*

2.4.3 Choosing the next direction

The overall approach may actually allow some flexibility in choosing the next direction $\mathbf{d}_{\text{new}}$ when calling `nextDirect` in Algorithm 1. The convergence of the inner and outer solutions towards `opt`$(\mathscr{P})$ is guided by the way each next projection direction is chosen. This practically-important question might be addressed in multiple manners and I here provide one possible way to do it. A reader interested more in the overall aspects may skip this (more technical) subsection, as it has no impact on Chapters 3-4.

The ray generation relies on a sequence or list *dLst* of new direction proposals that are provided *one by one* to Algorithm 1 at each call of `nextDirect`$(\mathbf{d}, \mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}, \mathbf{d}_{\texttt{prev}})$. Each `nextDirect` call returns the new direction situated right after $\mathbf{d}_{\text{prev}}$ in *dLst*, or the first new ray in *dLst* if $\mathbf{d}_{\text{prev}} = \mathbf{0}_k$. After trying all rays in *dLst*, if none of them leads to updating $\mathbf{y}_{\text{in}}$ or $\mathbf{y}_{\text{out}}$ (*i.e.*, *newLb* and *newUb* remain `false` in Lines 17-18), then `nextDirect`$(\mathbf{d}, \mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}, \mathbf{d}_{\texttt{prev}})$ eventually returns $\mathbf{0}_k$. The condition in Line 11 thus leads to *discretization refining*.

Intuitively, the discretization refining implements the idea that if all components of a direction $\mathbf{d} \in \mathbb{R}^k$ are multiples of $\frac{1}{2}$, then it can be considered an integer direction. We can force the algorithm into such behaviour by multiplying by 2 the current direction. The factor 2 can later be replaced with 4, 8, 16, etc. Thus, this procedure simply means that the next directions with have larger coefficients, making the sub-problem computationally more difficult. I think this intuitive view may be enough for now, but the interested reader may see all technical detail in [39, § 2.3.3].

We hereafter describe the construction of the list *dLst* of future directions in the standard case, *i.e.*, when $\mathbf{y}_{\text{in}}$ and $\mathbf{y}_{\text{out}}$ are proper non-null solutions and $t^* \neq 0$; see [39, § 2.3.2] for the degenerate cases. We pick the remaining key points in italic.

Rationale for locating promising new directions Let us focus on the line segment $[\mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}] = \{\mathbf{y}^\alpha = \mathbf{y}_{\text{in}} + \mathbf{y}_{\text{out}}(1-\alpha) : \alpha \in [0,1]\}$ and let $\Delta = \mathbf{y}_{\text{out}} - \mathbf{y}_{\text{in}}$. Consider now shrinking $[\mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}]$ by a factor of $\frac{1}{t^*}$, obtaining the *t^*-scaled-down segment* $\left[\mathbf{d}, \mathbf{d} + \frac{1}{t^*}\Delta\right]$. Let us write $\mathbf{d}^\beta := \mathbf{d} + \beta\Delta$, with $\beta \in [0, \frac{1}{t^*}]$. Since $\mathbf{d} = \frac{1}{t^*}\mathbf{y}_{\text{in}}$, we can say $\mathbf{d}^0 = \frac{1}{t^*}\mathbf{y}_{\text{in}}$ and $\mathbf{d}^{\frac{1}{t^*}} = \mathbf{d} + \frac{1}{t^*}\Delta = \frac{1}{t^*}\mathbf{y}_{\text{out}}$.

We can prove that if a new direction $\mathbf{d}^\beta = \mathbf{d} + \beta\Delta$ is integer for some $\beta \in [0, \frac{1}{t^*}]$, projecting towards $\mathbf{d}^\beta$ can surely lead to some update of $\mathbf{y}_{\text{in}}$ or $\mathbf{y}_{\text{out}}$. Indeed, such an integer new direction $\mathbf{d}^\beta$ intersects the segment $[\mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}]$ at some point $\mathbf{y}^\alpha = t^*(\mathbf{d} + \beta\Delta) = \mathbf{y}_{\text{in}} + t^*\beta\Delta$. By solving the $\mathbf{0} \to \mathbf{d}^\beta$ projection sub-problem, one implicitly solves the $\mathbf{0} \to \mathbf{y}^\alpha$ sub-problem. This also determines the feasibility status of $\mathbf{y}^\alpha$: (i) if $\mathbf{y}^\alpha \in \mathscr{P}$, the lower bound can be improved because a call `zeroIntProjSubprob`($\mathbf{d}^\beta$) would lead (see Lines 16-17) to a feasible solution of higher quality than $\mathbf{y}_{\text{in}}$; (ii) if $\mathbf{y}^\alpha \notin \mathscr{P}$, then the $\mathbf{y}_{\text{out}}$ can be updated, because both $\mathbf{y}^\alpha$ and $\mathbf{y}_{\text{out}}$ can be separated from $\mathscr{P}$. Intuitively, the closer the point $\mathbf{d} + \beta\Delta$ lies to an integer lattice point, the higher the chances of improving the optimality gap.

Locating the closest integer points to $\mathbf{d}^\beta = \mathbf{d} + \beta\Delta$. Given a fixed β_ℓ, the closest integer point to $\mathbf{d} + \beta_\ell\Delta$ is $\lfloor \mathbf{d}^{\beta_\ell} + \frac{1}{2}\mathbf{1}_k \rfloor$. To advance from a fixed β_ℓ to the next one, one starts with $\beta = \beta_\ell$ and increases β until $\lfloor \mathbf{d}^\beta + \frac{1}{2}\mathbf{1}_k \rfloor \neq \lfloor \mathbf{d}^{\beta_\ell} + \frac{1}{2}\mathbf{1}_k \rfloor$, *i.e.*, until the following holds $d_i^\beta + \frac{1}{2} = \lfloor d_i^{\beta_\ell} + \frac{1}{2} \rfloor \pm 1$ for some coordinate $i \in [1..k]$, where $\pm$ depends on the sign of Δ_i (use "+" if $\Delta_i > 0$ or "−" if $\Delta_i < 0$). Algorithm 2 from [39] exploits this idea to generate a sequence of increasing values $\beta_1 < \beta_2 < \beta_3, \ldots$, each one associated to a different integer lattice point. We thus obtain a sequence *dLst* of projection directions. This sequence is actually constructed only after each update of $\mathbf{y}_{\text{in}}$ or $\mathbf{y}_{\text{out}}$ when Algorithm 1 needs to generate new directions, *i.e.*, when it calls `nextDirect`($\mathbf{d}, \mathbf{y}_{\text{in}}, \mathbf{y}_{\text{out}}, \mathbf{0}_k$) with a last argument of $\mathbf{d}_{\text{prev}} = \mathbf{0}_k$, see Line 8 of Algorithm 1. In fact, $\mathbf{d}_{\text{prev}} = \mathbf{0}_k$ indicates that we ask for the *very first* new direction for current $\mathbf{d}, \mathbf{y}_{\text{in}}$ and $\mathbf{y}_{\text{out}}$, triggering the execution of Algorithm 2 from [39]. Afterwards, `nextDirect` returns (one by one) the elements from *dLst*, *i.e.*, observe the assignment $\mathbf{d}_{\text{prev}} \leftarrow \mathbf{d}_{\text{new}}$ (Line 15). When all elements from *dLst* are finished, `nextDirect` returns $\mathbf{0}_k$.

Sequence Reshuffling The order of directions in *dLst* has a strong influence on the general evolution of the overall method. A very practical enhancement of the initial order simply consists of moving the last element to the front. The last discovered direction (close to the direction $\mathbf{0}_k \to \mathbf{y}_{\text{out}}$) can be more useful for updating the upper bound, while the first directions are rather useful for updating the inner solution $\mathbf{y}_{\text{in}}$. The outer solutions are somehow more critical, because they can be absolutely *anywhere* in the search space, which can imply some dangerous lack of stability (too much fluctuation from one iteration to another) that can slow down the convergence.

2.5 Numerical results

Section 5 of [39] provides several pages of numerical results, for `Elastic Cutting-Stock` and `Arc-Routing`, using both standard literature instances and their scaled variants. These scaled variants increase the magnitude of the weights (or route demands in `Arc-Routing`), imposing a 1000-fold increase. This operations can seriously slow-down the DP for the separation sub-problem, but it has a limited impact on the integer projection. Recall from Observation 2.D (p. 16) that when the projection coefficients $\mathbf{d}$ are smaller than the weights, the projection calculations can be even faster than the separation ones.

In this thesis, we only present numerical results for three elastic variants of `Cutting-Stock`. We identify an `Elastic` variant by the elastic penalty function from (2.B), *i.e.*, the cost of a pattern $\mathbf{a}$ becomes $f(\frac{\mathbf{a}^\top \mathbf{w}}{Q})$.

f_{CS} This is the pure `Cutting-Stock` problem defined by $f_{\text{CS}}(x) = \infty \; \forall x > 1$, *i.e.*, any excess of the capacity Q leads to a prohibitive cost. The separation sub-problem is the knapsack problem that can be solved by any knapsack algorithm.

$f_{\text{11-bins}}$ This represents a staircase function that is constant over intervals $\left(\frac{Q_{\ell-1}}{Q}, \frac{Q_\ell}{Q}\right]$, with $\ell \in [1..10]$, where $Q = Q_0 < Q_1 < Q_2 < \cdots < Q_{10}$ represent 11 different bin sizes, each with its own cost [60,

§ 7.13].

$f_2(x) = x^2, \forall x > 1$ The separation sub-problem for this `Elastic` variant is quadratic and we could solve it with the quadratic programming library of `Cplex`.

The `Cutting-Stock` benchmark sets are described in Appendix 4 (with references provided therein) of [39]. Most instances have at most $n = 100$ items and the capacity Q can be 100, 10.000, 30.000 and rarely 100.000 (only for `Hard` instances, the most difficult ones).

For a fair comparison, we did our best to use the fastest knapsack algorithms for the separation sub-problem. The following *three* approaches have been considered:

`Minknap` This is one of the most competitive knapsack algorithms from the literature [36].

`Cplex` This is the ILP solver provided by `Cplex`.

`Std-DP` Standard Dynamic Programming with weight-indexed states (ad-hoc `C++` implementation with linked lists).

Table 1 presents the results, using the following format: the first two columns indicate the instance set (the `Elastic` function in Column 1 and the benchmark set in Column 2), Column 3 reports the average *computing effort* of the new method to reach a gap of 10% between the lower and the upper bound (*i.e.*, $\mathbf{b}^\top \mathbf{y}_{\text{in}} \geq \frac{9}{10}\mathbf{b}^\top \mathbf{y}_{\text{out}}$), Column 4 provides the average *computing effort* of the new method for complete convergence (we stop when $\lceil \mathbf{b}^\top \mathbf{y}_{\text{in}} \rceil = \lceil \mathbf{b}^\top \mathbf{y}_{\text{out}} \rceil$). The last three columns indicate the CG *computing effort*, *i.e.*, there is one column for each of the three CG pricing algorithms above.

Table 1 of [39] provides results on the scaled (large-range) instances. Confirming theoretical arguments, the 1000-fold weight increase (the scaling) has a limited impact on time running time of the new method (*i.e.*, a CPU time increase in $[-10\%, 30\%]$). For standard CG using DP pricing (weight-indexed!), this 1000-fold increase can make the running time explode from a few seconds (*e.g.*, 3.6s for `m20` or 2.7s for `m35` in Table 2, section $f_2(x) = x^2$) to dozens of seconds (*i.e.*, 63s for `m20` or 22s for `m35`, in Table 1 of [39], section $f_2(x) = x^2$).

Table 1 shows suggests that the new method is very fast in finding two in-out solutions $\mathbf{y}_{\text{in}}$ and $\mathbf{y}_{\text{out}}$ that yield a gap of 10%. There is no built-in feature in standard Column Generation to generate converging sequences of both inner and outer solutions, hence the Columns "gap 10%" are absent in the CG results of Table 1. At best, the lower bound $\mathbf{b}^\top \mathbf{y}_{\text{in}} = \mathbf{b}^\top (t^* \mathbf{d})$ can even be nearly optimal *from the first iterations*. Indeed, certain `Bin-Packing` instances (*e.g.*, the triplets) have an optimal dual solution of the form $\mathbf{y}^* = t^* \mathbf{b}$ ($\mathbf{b}$ is $\mathbf{1}_k$ in `Bin-Packing`) that is "hit" at the very first iteration with $\mathbf{d} = \mathbf{b}$.

The only disadvantage of the new method is a certain lack of robustness. It can fail in solving 100% of all instances in certain sets. This is due to some *early* excessive discretization refining that slows down the $\mathbf{d}$-indexed DP.

2.6 Further Reading and Prospects

The overall study [39] behind this chapter contains 50 pages in the *Springer* format used by *Mathematical Programming* ten years ago, which is roughly the length of this thesis. We can not cover all this material, but let us say that it addressed Column Generation problems with rather diverse features, *e.g.*, there is no restriction on the column costs (they have large variations in `Arc-Routing`) or on the demands $\mathbf{b}$ (they vary in `Cutting-Stock`). The cited paper used the term "ray intersection" instead of "$\mathbf{0}$-*origin integer projection*", but one can almost consider the word "ray" synonymous with direction in Column Generation.

This is one main conclusion of this chapter: if one can design a fast projection algorithm, the new method can be more useful than a standard Column Generation: besides the final convergence (generally needing less iterations), it naturally generates at each iteration a feasible (inner) dual solution which constitute a valid lower bound for the original integer problem. If we stop a standard Column Generation before fully converging, the reported upper bound has no relation whatsoever with the original integer problem because it's an upper bound of a lower bound (of the lower bound resulting from the linear relaxation).

The potential of the new method can not be evaluated only in terms of the CPU time needed to fully solve different problems. We hope this work may shed useful light beyond the Column Generation field. This was only a dream when I published [39], but this thesis aims at showing how some of the ideas above may work for other polytopes with prohibitively-many constraints, on more different models constructed using the Benders reformulation, or arising in robust linear programming or semidefinite programming.

Elast. Ver.	Instance Set	*Computing Effort* of new method (`iters (discr)/time`$^{-\texttt{fails}}$)		*Computing Effort* of std. Column Generation (`iters/time`$^{-\texttt{fails}}$)		
		Gap 10%	Full convergence	`Minknap`	`Cplex`	`Std. DP`
$f_2(x) = x^2$	`vb10`	5(0.0) / 0.01	24(2.1) / 0.04	—	23 / 1.3	22 / 19.1
	`vb20`	9(0.0) / 0.03	41(2.2) / 4.2	—	56 / 14.2^{-1}	`tm. out`
	`vb50-c1`	51(0.0) / 0.2	156(4.0) / 1.6	—	140 / 28.9	`tm. out`
	`vb50-c2`	26(0.0) / 0.2	153(3.5) / 23.9^{-3}	—	`tm. out`	`tm. out`
	`vb50-c3`	12(0.0) / 0.3	83(2.8) / 20.6^{-2}	—	129 / 43.7	`tm. out`
	`vb50-c4`	29(0.0) / 0.1	160(3.6) / 18.2^{-2}	—	`tm. out`	`tm. out`
	`vb50-c5`	13(0.0) / 0.1	109(3.2) / 34.6^{-4}	—	`tm. out`	`tm. out`
	`vb50-b100`	29(0.0) / 0.1	175(3.5) / 33.7^{-5}	—	`tm. out`	`tm. out`
	`m01`	141(2.2) / 0.4	255(5.7) / 1.1	—	204 / 13.6	218 / 4.0
	`m20`	143(2.1) / 0.4	223(5.9) / 0.8	—	190 / 11.6	292 / 3.6
	`m35`	88(2.5) / 0.3	150(6.1) / 0.5	—	199 / 7.7	289 / 2.7
	`Hard`	153(2.2) / 1.1	526(6.3) / 17.5^{-1}	—	`tm. out`	`tm. out`
	`Triplets`	42(1.0) / 0.1	71(1.0) / 0.1	—	`tm. out`	`tm. out`
Variable Sized `Cut. Stock`	`vb10`	5(0.0) / 0.01	21(2.2) / 0.7	$32_{\times 11}$/ 0.9	$32_{\times 11}$/ 60.20^{-2}	`tm. out`
	`vb20`	8(0.0) / 0.04	37(2.5) / 2.6	$55_{\times 11}$/ 3.5	`tm. out`	`tm. out`
	`vb50-c1`	33(0.0) / 0.1	136(4.0) / 15.3^{-4}	$185_{\times 11}$/ 27.6^{-1}	`tm. out`	`tm. out`
	`vb50-c2`	23(0.0) / 0.4	`tm. out`	$132_{\times 11}$/ 16.9	`tm. out`	`tm. out`
	`vb50-c3`	11(0.0) / 0.7	66(2.9) / 15.4^{-2}	$98_{\times 11}$/ 8.0	`tm. out`	`tm. out`
	`vb50-c4`	25(0.0) / 0.1	111(3.5) / 20.5^{-5}	$134_{\times 11}$/ 19.5	`tm. out`	`tm. out`
	`vb50-c5`	13(0.0) / 0.1	82(3.4) / 25.5^{-3}	$101_{\times 11}$/ 9.8	`tm. out`	`tm. out`
	`vb50-b100`	24(0.0) / 0.1	`tm. out`	$132_{\times 11}$/ 37.2	`tm. out`	`tm. out`
	`m01`	70(1.8) / 0.2	232(5.1) / 0.9^{-6}	$148_{\times 11}$/ 0.7	$140_{\times 11}$/ 73.35	$138_{\times 11}$/ 57.6
	`m20`	94(2.4) / 0.3	197(5.6) / 1.1^{-2}	$175_{\times 11}$/ 0.9	`tm. out`	$145_{\times 11}$/ 42.7
	`m35`	99(2.6) / 0.3	230(6.2) / 1.3^{-3}	$206_{\times 11}$/ 1.0	`tm. out`	$157_{\times 11}$/ 36.0
	`Hard`	165(2.0) / 1.4	442(5.3) / 8.0^{-1}	$715_{\times 11}$/ 37.3	`tm. out`	`tm. out`
	`Triplets`	44(1.0) / 0.2	73(1.0) / 0.5	$398_{\times 11}$/ 13.7	`tm. out`	`tm. out`
$f_{\text{cs}}(x) = 1$, pure `Cut. Stock`	`vb10`	5(0.0) / 0.01	17(0.9) / 0.03	14 / 0.03	16 / 1.12	15 / 0.6
	`vb20`	10(0.0) / 0.02	42(1.9) / 0.6	35 / 0.1	49 / 11.06^{-1}	51 / 61.5
	`vb50-c1`	55(0.0) / 0.1	115(2.0) / 0.3	105 / 0.4	101 / 12.3	`tm. out`
	`vb50-c2`	28(0.0) / 0.1	169(3.5) / 5.7^{-1}	154 / 1.3	`tm. out`	`tm. out`
	`vb50-c3`	12(0.0) / 0.1	91(2.8) / 12.2	82 / 0.5	124 / 32.3	`tm. out`
	`vb50-c4`	30(0.0) / 0.1	186(3.4) / 6.6	153 / 1.2	`tm. out`	`tm. out`
	`vb50-c5`	12(0.0) / 0.0	131(3.5) / 12.8^{-3}	101 / 0.8	130 / 61.3	`tm. out`
	`vb50-b100`	30(0.0) / 0.1	199(4.8) / 6.6^{-2}	171 / 3.2	`tm. out`	`tm. out`
	`m01`	109(0.9) / 0.3	300(3.5) / 1.1	122 / 0.4	128 / 6.7	130 / 0.9
	`m20`	97(0.9) / 0.3	302(1.9) / 1.1^{-1}	199 / 0.5	141 / 6.7	248 / 0.9
	`m35`	199(0.2) / 0.6	199(0.4) / 0.6	166 / 0.4	89 / 2.0	199 / 0.6
	`Hard`	254(2.0) / 1.9	872(6.1) / 14.8^{-1}	454 / 4.5	`tm. out`	`tm. out`
	`Triplets`	317(1.9) / 2.5	1001(2.7) / 15.8	546 / 4.9	`tm. out`	551 / 23.7

Table 1: Results for `Elastic Cutting Stock` with a time limit of 100 seconds.

$^{-\texttt{f}}$ *$\texttt{f}$ is the number of failures, instances not solved in 100 seconds. If $\texttt{f}$ represent $\geq 25\%$ of instances, we mark* "`tm. out`".

A limitation of this Column Generation work was the use of the **0**-origin when projecting. This limitation is lifted in the subsequent chapters where we'll project using arbitrary origins.

3 Random Origin Projection in Robust Linear Programming

The material from this section is mainly extracted from [Projective Cutting-Planes, *SIAM Journal on Optimization*, 30(1): 1007-1032, 2020] and [Projective Cutting-Planes for robust linear programming and cutting stock problems, *INFORMS Journal of Computing*, 34:2736-2753, 2022]. A light introduction is available in these slides: https://cedric.cnam.fr/~porumbed/papers/slidesProjCutPlanes.pdf. The full C++ code is available here: https://cedric.cnam.fr/~porumbed/projcutplanes/. The code associated to the second paper was also integrated in the IJOC GitHub software repository at http://dx.doi.org/10.5281/zenodo.5745335.

Given a polytope $\mathscr{P}$, an interior point $\mathbf{y} \in \mathscr{P}$ and a direction $\mathbf{d} \in \mathbb{R}^k$, the most general projection sub-problem asks to find the maximum step length t^* such that $\mathbf{y} + t^*\mathbf{d} \in \mathscr{P}$; we say $\mathbf{y} + t^*\mathbf{d}$ is the pierce point. The previous chapter only considered integer directions $\mathbf{d}$ in dual polytopes $\mathscr{P}$ of Column Generation models, always with a $\mathbf{0}_k$ as origin. These limitations come from the numerical difficulty of solving the projection in the considered polytopes; Algorithm 1 had to address some intricate questions to be sure to use only integer directions. I needed a series of papers ([40] in 2018, [42] in 2020, [43] in 2022) to generalize the idea to more varied settings, like the Benders reformulation or robust optimization. I currently work on the semidefinite optimization version, which gives Chapter 4 of this thesis.

The `Proj-Cut-Pl` (or `Projective-Cutting-Planes`) algorithm developed in this chapter optimizes over polytopes $\mathscr{P}$ with prohibitively-many constraints by repeating the following type of operations. First, it requires an initial feasible point $\mathbf{y}$ to start, since it was not designed to detect infeasibilities. The first iteration projects this $\mathbf{y}$ along the objective function $\mathbf{d} = \mathbf{b}$, generating a pierce point $\mathbf{y} + t^*\mathbf{d}$. At each next iteration, it selects a point $\mathbf{y}_{\texttt{new}}$ on the segment joining the points $\mathbf{y}$ and $\mathbf{y} + t^*\mathbf{d}$; then, it projects $\mathbf{y}_{\texttt{new}}$ along the direction $\mathbf{d}_{\texttt{new}}$ pointing towards the current optimal (outer) solution of the current outer approximation of $\mathscr{P}$, so as to generate a new pierce point $\mathbf{y}_{\texttt{new}} + t^*_{\texttt{new}}\mathbf{d}_{\texttt{new}}$ and a new constraint of $\mathscr{P}$.

Figure 3.A illustrates this process iteration by iteration. At each iteration $\texttt{it}$, an inner solution $\mathbf{y}_{\texttt{it}} \in \mathscr{P}$ is projected towards the direction $\mathbf{d}_{\texttt{it}}$ of the current optimal outer solution $\texttt{opt}(\mathscr{P}_{\texttt{it}-1})$, *i.e.*, we take $\mathbf{d}_{\texttt{it}} = \texttt{opt}(\mathscr{P}_{\texttt{it}-1}) - \mathbf{y}_{\texttt{it}}$. The projection sub-problem asks to determine $t^*_{\texttt{it}} = \max\{t : \mathbf{y}_{\texttt{it}} + t\mathbf{d}_{\texttt{it}} \in \mathscr{P}\}$. 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 $\mathscr{P}$, which is added to the constraints of $\mathscr{P}_{\texttt{it}-1}$ to construct $\mathscr{P}_{\texttt{it}}$. 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 $\mathbf{y}_{\texttt{it}} + t^*_{\texttt{it}}\mathbf{d}_{\texttt{it}} \in \mathscr{P}\ \forall t \in [0, t^*_{\texttt{it}}]$. At next iteration $\texttt{it}+1$, `Proj-Cut-Pl` 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 $\mathbf{d}_{\texttt{it}+1} = \texttt{opt}(\mathscr{P}_{\texttt{it}}) - \mathbf{y}_{\texttt{it}+1}$.

Let us present some conclusions in advance to motivate the interest. On the experimental side, `Proj-Cut-Pl` was able to reach for the (well–studied) graph coloring problem a few lower bounds that had never been reported before when it was published (see Remark 6.2, p. 1028). More generally, the underlying work [42] was not (only) meant to be a competition paper. It is more important that the new approach has certain features that do not exist in `Cutting-Planes`, *e.g.*, it generates feasible solutions along the iterations, it can eliminate the "yo-yo" effects arising very often (if not always) in many `Cutting-Planes` especially in Column Generation (see Figure 3 of [42]), the degeneracy risk is seriously reduced (see Remark 2 of [43]).

The proposed algorithm is (loosely) reminiscent of an Interior Point Method (IPM) in that it systematically generates a sequence of interior points that converge towards the optimal solution. 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 [16]. Advancing along a Newton direction is not really equivalent to performing a projection, because 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 – since these conditions correspond to a primal objective function penalized by a barrier term that only vanishes at the last iteration. Certain IPMs for (dual) LPs with prohibitively-many constraints (in Column Generation) generate well-centered dual solutions along the iterations by keeping them in the proximity of a central path [17, § 3.3]. This shares certain goals with the construction of the feasible solutions $\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{y}_3, \ldots$ in `Proj-Cut-Pl`.

3.1 Formalizing `Proj-Cut-Pl`

We now delve into more technical details, addressing notations more carefully. Given unmanageably-many constraints C, we repeat the goal (2.A):

$$\max\left\{\mathbf{b}^\top\mathbf{y} : \mathbf{a}^\top\mathbf{y} \leq c_a,\ \forall(\mathbf{a}, c_a) \in C\right\} = \max\left\{\mathbf{b}^\top\mathbf{y} : \mathbf{y} \in \mathscr{P}\right\} \tag{3.A}$$

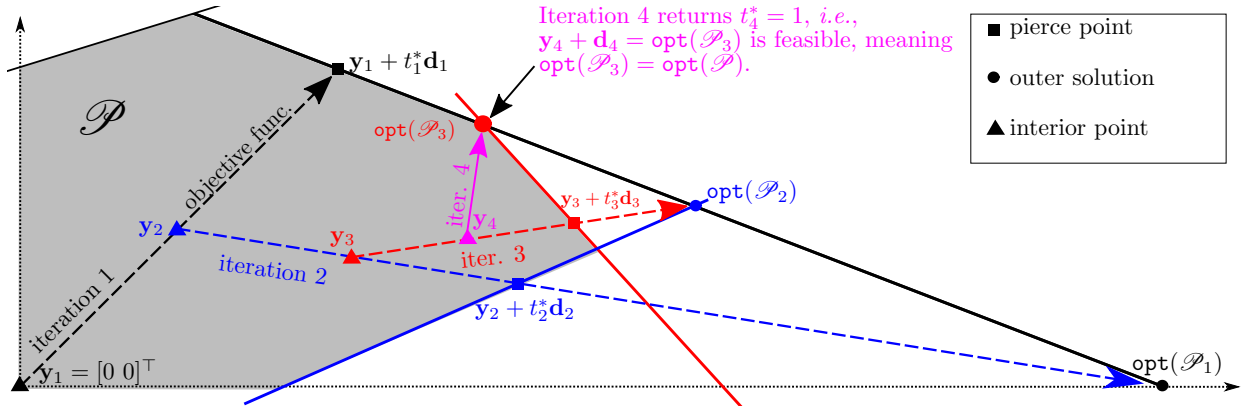


Figure 3.A: `Proj-Cut-Pl` on an LP with $k = 2$. The first iteration projects $\mathbf{y}_1 = \mathbf{0} = [0\ 0]^\top$ along the objective function, as depicted by the black dashed arrow. At iteration $\texttt{it} = 2$, the midpoint $\mathbf{y}_2$ of this black arrow is projected towards the optimal outer solution $\texttt{opt}(\mathscr{P}_1)$ — at iteration 1, the outer approximation $\mathscr{P}_1 \supset \mathscr{P}$ only contains the largest triangle. This generates a second facet (blue solid line) that is added to the facets of $\mathscr{P}_1$ to construct $\mathscr{P}_2$. The process is repeated until finding the optimal solution at iteration 4.

Recall `Cutting-Planes` maintains at each iteration $\texttt{it}$ an outer approximation $\mathscr{P}_{\texttt{it}} \supset \mathscr{P}$ of $\mathscr{P}$ obtained by restricting the constraint set C to a subset $C_{\texttt{it}}$. To (try to) separate the current optimal solution $\mathbf{y}_{\text{out}} = \texttt{opt}(\mathscr{P}_{\texttt{it}})$ of $\mathscr{P}_{\texttt{it}}$, one usually solves the separation sub-problem $\min_{(\mathbf{a},c_a)\in C} c_a - \mathbf{a}^\top \mathbf{y}_{\text{out}}$. If this is less than zero, the constraint that separates $\mathbf{y}_{\text{out}}$ is added $C_{\texttt{it}}$ to construct $C_{\texttt{it}+1}$, which corresponds to a new more refined outer approximation $\mathscr{P}_{\texttt{it}+1}$. The process is repeated by (re-)optimizing over $\mathscr{P}_{\texttt{it}+1}$ at the next iteration, until the current optimal outer solution $\mathbf{y}_{\text{out}}$ becomes optimal (non-separable).

Definition 3.A. *(Projection sub-problem) Given an interior point $\mathbf{y} \in \mathscr{P}$ and a direction $\mathbf{d} \in \mathbb{R}^k$, the projection sub-problem $\texttt{proj-subp}(\mathbf{y} \to \mathbf{d})$ asks to find:*

1) *the maximum step length t^* such that $\mathbf{y} + t^*\mathbf{d}$ is feasible in $\mathscr{P}$, i.e., $t^* = \max\{t \geq 0 : \mathbf{y} + t\mathbf{d} \in \mathscr{P}\}$. The solution $\mathbf{y} + t^*\mathbf{d}$ is referred to as the pierce point. If $\mathbf{y} + t\mathbf{d}$ is a ray of $\mathscr{P}$, the sub-problem returns $t^* = \infty$.*
2) *a first-hit constraint $(\mathbf{a}, c_a) \in C$ 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 $t^* \neq \infty$.*

The first projection is $\mathbf{y}_1 \to \mathbf{d}_1$ where $\mathbf{d}_1 = \mathbf{b}$ and $\mathbf{y}_1$ is an initial feasible solution. If it is particularly difficult to find such $\mathbf{y}_1$, then the underlying problem is outside the scope of this work. Solving $\texttt{proj-subp}(\mathbf{y}_1 \to \mathbf{d}_1)$ leads to a first pierce point $\mathbf{y}_1 + t_1^*\mathbf{d}_1$ and a first-hit constraint $(\mathbf{a}, c_a) \in C$. After updating $C_1 = C_0 \cup \{(\mathbf{a}, c_a)\}$, the first outer approximation $\mathscr{P}_1$ is constructed. Notice C_0 represents any (simple) initial constraints like $\mathbf{y} \geq \mathbf{0}_k$. We then execute the following steps at each iteration $\texttt{it} \geq 2$:

1. Select an inner solution $\mathbf{y}_{\texttt{it}}$ on the segment joining $\mathbf{y}_{\texttt{it}-1}$ and $\mathbf{y}_{\texttt{it}-1} + t_{\texttt{it}-1}^*\mathbf{d}_{\texttt{it}-1}$, *i.e.*, between the previous inner solution and the last pierce point.
2. Take the direction $\mathbf{d}_{\texttt{it}} = \texttt{opt}(\mathscr{P}_{\texttt{it}-1}) - \mathbf{y}_{\texttt{it}}$, so as to advance towards the current optimal (outer) solution $\texttt{opt}(\mathscr{P}_{\texttt{it}-1})$. Given that $\mathscr{P}_{\texttt{it}-1} \supseteq \mathscr{P} \ni \mathbf{y}_{\texttt{it}}$, we obtain that if $\mathbf{y}_{\texttt{it}}$ is strictly interior, then the objective value can *only strictly improve* by advancing along $\mathbf{y}_{\texttt{it}} \to \mathbf{d}_{\texttt{it}}$; under these conditions, it is impossible to execute degenerate iterations that keep the objective value constant like in standard `Cutting-Planes`. We mention a particular case: if $\texttt{opt}(\mathscr{P}_{\texttt{it}-1})$ is an extreme ray $\mathbf{d}_\infty$ of $\mathscr{P}_{\texttt{it}-1}$ such that $\{\mathbf{y} + \lambda\mathbf{d}_\infty : \lambda \geq 0\} \subset \mathscr{P}_{\texttt{it}-1}\ \forall \mathbf{y} \in \mathscr{P}_{\texttt{it}-1}$, we take $\mathbf{d}_{\texttt{it}} = \mathbf{d}_\infty$.
3. Solve the projection sub-problem $\texttt{proj-subp}(\mathbf{y}_{\texttt{it}} \to \mathbf{d}_{\texttt{it}})$ to determine the maximum step length $t_{\texttt{it}}^*$, the pierce point $\mathbf{y}_{\texttt{it}} + t_{\texttt{it}}^*\mathbf{d}_{\texttt{it}}$, and a first-hit constraint $(\overline{\mathbf{a}}, \overline{c_a}) \in C$. If t^* is ∞, we conclude (3.A) is unbounded.
4. If $t_{\texttt{it}}^* \geq 1$, return $\texttt{opt}(\mathscr{P}_{\texttt{it}-1})$ as an optimal solution of (3.A) over $\mathscr{P}$.
   If $t_{\texttt{it}}^* < 1$, then current optimal solution $\texttt{opt}(\mathscr{P}_{\texttt{it}-1})$ can be separated and we perform the following:
   - set $C_{\texttt{it}} = C_{\texttt{it}-1} \cup \{(\overline{\mathbf{a}}, \overline{c_a})\}$ to obtain a new enlarged constraint set, producing to a more refined outer approximation $\mathscr{P}_{\texttt{it}} \supseteq \mathscr{P}$.
   - calculate a new current optimal outer solution $\texttt{opt}(\mathscr{P}_{\texttt{it}})$ by (re-)optimizing over $\mathscr{P}_{\texttt{it}}$.
   - if $\mathbf{y}_{\texttt{it}} + t_{\texttt{it}}^*\mathbf{d}_{\texttt{it}}$ and $\texttt{opt}(\mathscr{P}_{\texttt{it}})$ are close enough (in terms of objective value), stop and return $\texttt{opt}(\mathscr{P}_{\texttt{it}})$.

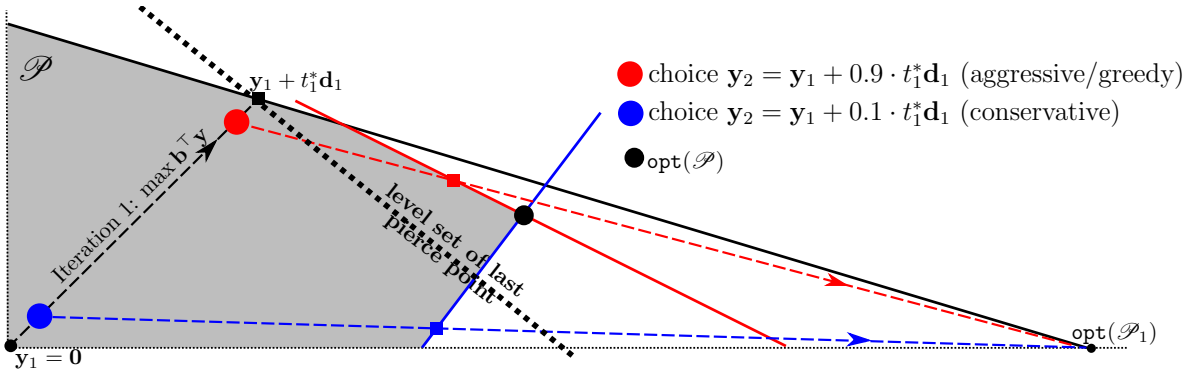


Figure 3.B: Intuitive illustration of two different choices of the interior point $\mathbf{y}_2$ at iteration 2. The red choice (originating in the red disk) is more aggressive while the blue one (originating in the blue disk) is more cautious.

– repeat from Step 1 after updating $\texttt{it} \leftarrow \texttt{it} + 1$.

The above steps are finitely convergent because we implicitly solve a separation sub-problem on $\texttt{opt}(\mathscr{P}_{\texttt{it}-1})$ at each iteration $\texttt{it}$, generalizing the standard Cutting-Planes. As hinted at Step 4, if the projection sub-problem returns $t^*_{\texttt{it}} < 1$, the solution $\texttt{opt}(\mathscr{P}_{\texttt{it}-1})$ is certainly separated by the first-hit constraint $(\overline{\mathbf{a}}, \overline{c_a})$. In pure theory, in the worst case, the proposed method ends up enumerating all constraints of $\mathscr{P}$ and it then eventually returns $\texttt{opt}(\mathscr{P})$. The fact that this convergence proof is very short is not really fortuitous. In the previous Chapter 2 we needed longer proofs because it was more complicated to handle integer-only directions. The new `Proj-Cut-Pl` has been deliberately designed to simplify all proofs as much as possible. But in some sense, the difficulty was shifted to the projection algorithm that has to address a more general sub-problem.

Just like the standard `Cutting-Planes`, the `Proj-Cut-Pl` is a rather generic framework that allows a number of problem–specific adaptations. Unlike in Chapter 2, we do not provide a pseudo-code, which allows a higher degree of flexibility.

3.1.1 Choosing the interior point $\mathbf{y}_{\texttt{it}}$ at each iteration $\texttt{it}$

A key question is the choice of the interior point $\mathbf{y}_{\texttt{it}}$ at (Step 1 of) each iteration $\texttt{it}$. One might attempt to choose the best feasible solution found up to iteration $\texttt{it}$ (the last pierce point) via $\mathbf{y}_{\texttt{it}} = \mathbf{y}_{\texttt{it}-1} + t^*_{\texttt{it}-1}\mathbf{d}_{\texttt{it}-1}$. This may seem to offer an interesting advantage: each new iteration pushes the next inner solution $\mathbf{y}_{\texttt{it}}$ to the maximum extent possible, generating monotonically increasing lower bounds, like in [43, Fig. 3 (§ 3.2.2)]. However, this aggressive greedy idea may lead to poor results in the long run, at least for robust linear programming and semidefinite programming. This is because $\mathbf{y}_{\texttt{it}}$ can fluctuate too much from iteration to iteration, leading to a yo-yo (heavy oscillation) effect.[3]

Since we restrict this thesis to the study of `Proj-Cut-Pl` in robust linear programming and semidefinite optimization, we prefer to define $\mathbf{y}_{\texttt{it}} = \mathbf{y}_{\texttt{it}-1} + \alpha t^*_{\texttt{it}-1}\mathbf{d}_{\texttt{it}-1}$ with α in a *safe interval* of $[0.1, 0.7]$; we used the lower bound 0.1 only for the robust case. In a loose sense, this is reminiscent of interior point methods that avoid touching the boundary of the polytope before fully converging [16]. In our case, even when pushing α to 1 is useful, we noticed it is better to use $\alpha = 0.9999$ in practice; this recalls the "fraction-to-the-boundary stepsize factor" used in (some) interior point algorithms to prevent them from touching the boundary — see the parameter $\alpha_0 = 0.99$ in the pseudo-code above Section 3 in [16, p. 590].

Figure 3.B illustrates the difference between an aggressive choice (large α) and a "cautious" or well-centered choice (small α). The red disk represents an aggressive definition of $\mathbf{y}_2$ associated to a large α, so that $\mathbf{y}_2$ is very close to the last pierce point $\mathbf{y}_1 + t_1^*\mathbf{d}_1$. Such a choice enables the projection sub-problem at iteration 2 to easily exceed the objective value of the last pierce point $\mathbf{y}_1 + t_1^*\mathbf{d}_1$ by only advancing a little from $\mathbf{y}_2$ towards $\texttt{opt}(\mathscr{P}_1)$ – see how rapidly the red dashed arrow crosses the black dotted line, *i.e.*, the level set of the last pierce point $\left\{\mathbf{y} \in \mathbb{R}^2_+ : \mathbf{b}^\top \mathbf{y} = \mathbf{b}^\top (\mathbf{y}_1 + t_1^*\mathbf{d}_1)\right\}$. The blue circle represents a choice of a point $\mathbf{y}_2$ closer to $\mathbf{0}_k$: it is more difficult to reach the level set of $\mathbf{y}_1 + t_1^*\mathbf{d}_1$ by advancing from this $\mathbf{y}_2$ towards

[3] As analyzed in [43, § 3.3], this aggressive variant is not recommended for the Benders decomposition from [42, § 4.1] either. It proved useful for the graph coloring CG model from [42, § 4.2] and (to some extent) for the Multiple-Length Cutting-Stock CG model from [43, § 3.2.2].

$\texttt{opt}(\mathscr{P}_1)$, but this blue projection can lead to a stronger (blue) constraint, *i.e.*, the blue solid line cuts off a larger area of $\mathscr{P}_1$ (*i.e.*, of the largest triangle) than the red solid line.

3.1.2 Techniques for designing a fast projection algorithm

The most challenging part is to design a fast projection algorithm, because the iterations of a successful `Proj-Cut-Pl` should not be significantly slower than the iterations of the standard `Cutting-Planes`. For instance, if the projection iterations were two–three times slower than the separation iterations, the `Proj-Cut-Pl` could be too slow, *i.e.*, it could remain slower than the standard `Cutting-Planes` even if it converged in half iterations. Although the projection sub-problem generalizes the separation one, we present below several techniques that lead to designing a projection algorithm that competes (very) tightly with the separation algorithm in terms of computational speed. However, it was hard to find polytopes $\mathscr{P}$ for which this was possible.

Let us first describe how the projection sub-problem $\texttt{proj-subp}(\mathbf{y} \rightarrow \mathbf{d})$ reduces to minimizing the following fractional program (for any $\mathbf{y} \in \mathscr{P}$ and any $\mathbf{d} \in \mathbb{R}^k$):

$$t^* = \min \left\{ \frac{c_a - \mathbf{a}^\top \mathbf{y}}{\mathbf{a}^\top \mathbf{d}} : \ (\mathbf{a}, c_a) \in C, \ \mathbf{d}^\top \mathbf{a} > 0 \right\}. \tag{3.B}$$

We have to show $\mathbf{y} + t\mathbf{d} \in \mathscr{P} \ \forall t \in [0, t^*]$, *i.e.*, $\mathbf{a}^\top(\mathbf{y} + t\mathbf{d}) \leq c_a \ \forall (\mathbf{a}, c_a) \in C \ \forall t \in [0, t^*]$. If $\mathbf{a}^\top \mathbf{d} \leq 0$, then $\mathbf{a}^\top(\mathbf{y}+t\mathbf{d}) \leq \mathbf{a}^\top \mathbf{y} \leq c_a \ \forall (\mathbf{a}, c_a) \in C$ actually holds for all $t \in [0, \infty]$, because $\mathbf{y} \in \mathscr{P} \implies \mathbf{a}^\top \mathbf{y} \leq c_a \ \forall (\mathbf{a}, c_a) \in C$. Otherwise, if $\mathbf{a}^\top \mathbf{d} > 0$, then $\mathbf{a}^\top(\mathbf{y} + t\mathbf{d}) \leq c_a \ \forall (\mathbf{a}, c_a) \in C$ is equivalent to $t \leq \frac{c_a - \mathbf{a}^\top \mathbf{y}}{\mathbf{a}^\top \mathbf{d}} \ \forall (\mathbf{a}, c_a) \in C$ which is true for any $t \leq t^*$, because t^* minimizes the above ratio in (3.B). As such, we will only focus on $(\mathbf{a}, c_a) \in C$ such that $\mathbf{a}^\top \mathbf{d} > 0$ when designing the projection algorithm. Finally, $\mathbf{y} + t^*\mathbf{d}$ belongs to the boundary of $\mathscr{P}$ because it satisfies with equality the constraint associated to the minimizer of (3.B).

3.1.2.1 Generalizing the separation (for robust programming) without repeated separation

A first approach to efficiently solve the projection sub-problem consists of generalizing (the main ideas of) the separation algorithm without greatly increasing its computation time. This can *not* be simply achieved by repeated separation: such projection method would call the separation algorithm at least twice, or usually 3 or 4 times, making the projection 3 or 4 times slower than the separation. A first call to the separation sub-problem is needed to find a constraint satisfied with equality by some $\mathbf{y} + t_1\mathbf{d}$, followed by *at least* a second call to check if $\mathbf{y} + t_1\mathbf{d}$ can be separated. If $\mathbf{y} + t_1\mathbf{d}$ can be separated by a new constraint, we obtain $t_2 < t_1$; the process could be repeated, leading to some $\mathbf{y} + t_3\mathbf{d}$ with $t_3 < t_2$, etc. Experiments on several problems suggested that a 3^{rd} or a 4^{th} call may sometimes be needed. However, it is more fruitful (on the long-term) to explore techniques that can bring us close to designing a projection algorithm as fast as the separation one.

We will give in Section 3.2.2 an example of how a more intricate generalization of the separation algorithm (without simple repetitions) induces almost no slowdown in robust linear optimization. For a (very) classical robust LP [11], the separation of a given $\mathbf{y} \in \mathbb{R}^k$ reduces to minimizing a difference $c_a - (\mathbf{a} + \widehat{\mathbf{a}})^\top \mathbf{y}$ over a set of nominal constraints $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$ and over all possible deviations $\widehat{\mathbf{a}}$ of the nominal coefficients $\mathbf{a}$. The projection sub-problem (3.B) reduces to minimizing $\frac{c_a - (\mathbf{a}+\widehat{\mathbf{a}})^\top \mathbf{y}}{(\mathbf{a}+\widehat{\mathbf{a}})^\top \mathbf{d}}$ over the same $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$ and over the same $\widehat{\mathbf{a}}$. Both sub-problems can be solved by iterating over the nominal constraints $C_{\texttt{nom}}$; for each $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$, the separation sub-problem tries to minimize the above difference while the projection sub-problem tries to minimize the above ratio. This objective function change (minimize a ratio instead of a difference) does not change the nature of the subproblem algorithm: the main computational bottleneck consists of iterating over $C_{\texttt{nom}}$. And if the above ratio is reduced to zero (meaning $t^* = 0$), the projection algorithm may even stop without scanning all $C_{\texttt{nom}}$; unexpectedly, this can make the projection even faster than the separation (at certain iterations).

3.1.2.2 The case of constraints C of $\mathscr{P}$ given by the solutions of an auxiliary LP

Consider the (numerous) problems in which the constraints of $\mathscr{P}$ are associated to the feasible solutions of an auxiliary LP. This is the case for most Benders decomposition models in which the separation sub-problem is often formulated as an LP over a Benders sub-problem polytope P_B. In this case, (3.B) reduces to a

linear-fractional program: minimize a ratio of two linear functions subject to linear constraints. Using the Charnes–Cooper transformation [5], it is possible to translate this linear-fractional program to a similar size LP. This leads to a projections algorithm having the same complexity as the separation one (see [42, § 3.1.3]), *i.e.*, they both have the complexity of solving an LP over P_B.

3.1.2.3 The case of constraints C of $\mathscr{P}$ given by the solutions of an auxiliary ILP

The above technique can be generalized to the (numerous) problems in which the constraints of $\mathscr{P}$ are given by the feasible solutions of an Integer LP (ILP). This idea was developed in [42, § 3.2.3], where $\mathscr{P}$ is the dual polytope of the Column Generation model for graph coloring. In this model, each constraint $(\mathbf{a}, c_a) = (\mathbf{a}, 1) \in C$ of $\mathscr{P}$ is associated to a primal column, which, in turn, is given by the incidence vector $\mathbf{a} \in \{0,1\}^k$ of a stable in the considered graph. Such Column Generation problem was not considered in Chapter 2, because all constraints have a right-hand side coefficient of $c_a = 1$, which would reduce the **0**-*origin integer projection* from Chapter 2 to a simple separation (see also Observation 2.C). However, the most general projection never reduces to a separation.

The constraints of $\mathscr{P}$ that come from graph stables can be seen as the integer solutions of the well-studied stable set polytope defined by edge inequalities. For this case, we proposed a discrete Charnes-Cooper transformation [5] that turns (3.B) into a *Disjunctive LP* and the integrality constraints $\mathbf{a}_i \in \{0,1\}$ into disjunctive constraints of the form $\overline{\mathbf{a}}_i \in \{0, \overline{\alpha}\}$, where $\overline{\alpha}$ is an additional decision variable. This *Disjunctive LP* has a discrete feasible area and can be solved with the same techniques as the separation ILP, *i.e.*, using a `Branch and Bound` with bounds determined from continuous relaxations.

This discrete Charnes-Cooper transformation can apply in the same manner to any problem in which the constraints C of $\mathscr{P}$ are given by the solutions of an ILP; as a matter of fact, we do consider in [42, § 2.4.1] the case of a different (although related) ILP that defines C using other inequalities. However, we think there is no deep reason why such *Disjunctive LP* should be much harder in absolute terms than the ILP solved by the separation sub-problem. The two programs have rather similar continuity-breaking constraints ($\overline{\mathbf{a}}_i \in \{0, \overline{\alpha}\}$ resp. $\mathbf{a}_i \in \{0,1\}$) and they are typically solved with similar `Branch and Bound` methods that calculate bounds by lifting these continuity-breaking constraints, see also Remark 2 (p .1019) of [42].

3.2 `Proj-Cut-`*`Pl`* for robust linear programming

Let us first consider a set $C_{\texttt{nom}}$ of nominal constraints that is small enough to be enumerated in practice, *i.e.*, there is no need of `Cutting-Planes` to solve the nominal version of (3.A). We then associate to each $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$ a prohibitively-large set $\texttt{Dev}(\mathbf{a})$ of *deviation vectors* $\widehat{\mathbf{a}}$, *i.e.*, vectors $\widehat{\mathbf{a}} \in \mathbb{R}^k$ that have at maximum Γ non-zero components and that satisfy $\widehat{\mathbf{a}}_i \in \{-\delta\mathbf{a}_i,\ 0,\ \delta\mathbf{a}_i\}\ \forall i \in [1..n]$, using $\delta = 0.01$ in practice [11]. The limit Γ of the number of coefficients that can vary formalizes an important idea in robust optimization: the nominal coefficients can not change all at the same time, always in an unfavorable manner. Each deviation vector $\widehat{\mathbf{a}}$ yields a robust cut $(\mathbf{a} + \widehat{\mathbf{a}})^\top \mathbf{y} \leq c_a$, so that we can state $(\mathbf{a} + \widehat{\mathbf{a}}, c_a) \in C$. In theory, each $\widehat{\mathbf{a}}_i$ $(\forall i \in [1..n])$ might be allowed to take a fractional value in the interval $[-\delta\mathbf{a}_i, \delta\mathbf{a}_i]$, thus leading to infinitely-many robust cuts (semi-infinite programming); however, the strongest robust cuts are always obtained when each non-zero $\widehat{\mathbf{a}}_i$ is either $\delta\mathbf{a}_i$ or $-\delta\mathbf{a}_i$. There are at most $\binom{n}{\Gamma}2^\Gamma$ deviation vectors for each nominal constraint $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$, because there are $\binom{n}{\Gamma}$ ways to choose the non-zero components of $\widehat{\mathbf{a}}$ and each one of them can be either positive or negative, hence the 2^Γ factor.

3.2.1 The separation in robust linear programming

We implemented a standard `Cutting-Planes` for the resulting problem based on the following separation sub-problem: given any $\mathbf{y} \in \mathbb{R}^k$, minimize $c_a - (\mathbf{a} + \widehat{\mathbf{a}})^\top \mathbf{y}$ over all $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$ and over all $\widehat{\mathbf{a}} \in \texttt{Dev}(\mathbf{a})$. For a fixed nominal constraint $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$, the strongest possible deviation $\widehat{\mathbf{a}}_\mathbf{y}^\top \mathbf{y}$ of $(\mathbf{a}, c_a)$ with respect to $\mathbf{y}$ is determined by maximizing $\widehat{\mathbf{a}}_\mathbf{y} = arg\ \max\left\{\widehat{\mathbf{a}}^\top \mathbf{y} : \widehat{\mathbf{a}} \in \texttt{Dev}(\mathbf{a})\right\}$. To find this $\widehat{\mathbf{a}}_\mathbf{y}$, one needs to determine the largest Γ absolute values in the terms of the sum $\mathbf{a}^\top \mathbf{y} = \sum_{i=1}^k \mathbf{a}_i \mathbf{y}_i$; this way, $\widehat{\mathbf{a}}_\mathbf{y}^\top \mathbf{y}$ can be written as a sum of Γ terms of the form $\delta|\mathbf{a}_i y_i|$. These largest Γ values are determined by a partial-sorting algorithm of linear complexity described in [43, Remark 1 (§ 2.1.1)].

3.2.2 The projection in robust linear programming

Based on (3.B), the projection sub-problem reduces to minimizing $\frac{c_a - (\mathbf{a}+\widehat{\mathbf{a}})^\top \mathbf{y}}{(\mathbf{a}+\widehat{\mathbf{a}})^\top \mathbf{d}}$ over all nominal constraints $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$ and over all deviation vectors $\widehat{\mathbf{a}} \in \texttt{Dev}(\mathbf{a})$ such that $(\mathbf{a}+\widehat{\mathbf{a}})^\top \mathbf{d} > 0$. Just as the separation algorithm, the projection algorithm iterates over all nominal constraints $C_{\texttt{nom}}$, in an attempt to reduce the above ratio – *i.e.*, the step length – at each $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$. Let t_i^* denote the optimal step length obtained after considering the robust cuts associated to the first i constraints from $C_{\texttt{nom}}$. It is clear that t_i^* can only decrease as i grows. Starting with $t_0 = 1$, the projection algorithm determines t_i^* from t_{i-1}^* by applying the following five steps:

1. Set $t = t_{i-1}^*$ and let $(\mathbf{a}, c_a)$ denote the i^{th} constraint from $C_{\texttt{nom}}$.

2. Determine the strongest deviation vector $\widehat{\mathbf{a}}_t$ with respect to $\mathbf{y} + t\mathbf{d}$ by maximizing:
$$\widehat{\mathbf{a}}_t = arg\ \max \left\{ \widehat{\mathbf{a}}^\top (\mathbf{y} + t\mathbf{d}) : \widehat{\mathbf{a}} \in \texttt{Dev}(\mathbf{a}) \right\}. \tag{3.C}$$
For this, one has to extract the Γ largest absolute values from the terms of the sum $\mathbf{a}^\top (\mathbf{y} + t\mathbf{d})$; we apply the same partial-sorting algorithm used for the separating described in [43, Remark 1 (§ 2.1.1)].

3. If $(\mathbf{a} + \widehat{\mathbf{a}}_t)^\top (\mathbf{y} + t\mathbf{d}) \leq c_a$, then $\mathbf{y} + t\mathbf{d}$ is feasible with regards to the first i constraints from $\mathcal{A}_{\texttt{nom}}$ and the associated robust cuts, because any deviation vector $\widehat{\mathbf{a}} \in \texttt{Dev}(\mathbf{a})$ satisfies $\widehat{\mathbf{a}}^\top (\mathbf{y} + t\mathbf{d}) \leq \widehat{\mathbf{a}}_t^\top (\mathbf{y} + t\mathbf{d})$. In this case, the final value $t_i^* = t$ has been obtained and the algorithm terminates for this value of i. Otherwise, the robust cut $(\mathbf{a} + \widehat{\mathbf{a}}_t, c_a)$ leads to a smaller feasible step length:
$$t' = \frac{c_a - (\mathbf{a} + \widehat{\mathbf{a}}_t)^\top \mathbf{y}}{(\mathbf{a} + \widehat{\mathbf{a}}_t)^\top \mathbf{d}} < t. \tag{3.D}$$

4. If $t' = 0$, then the overall projection algorithm returns $t^* = 0$ without checking the remaining nominal constraints, because it is not possible to return a step length below 0 since $\mathbf{y}$ is feasible.

5. Set $t = t'$ and repeat from Step 2 (without incrementing i). The underlying idea is that the deviation vector $\widehat{\mathbf{a}}_t$ determined via (3.C) is not the strongest one with regards to $\mathbf{y} + t'\mathbf{d}$, because $\widehat{\mathbf{a}}_t$ yields the highest deviation in (3.C) with regards to a different point (*i.e.*, $\mathbf{y} + t\mathbf{d}$). But there might exist a different robust cut $(\mathbf{a} + \widehat{\mathbf{a}}_{t'}, c_a)$ such that $\widehat{\mathbf{a}}_{t'}^\top (\mathbf{y} + t'\mathbf{d}) > \widehat{\mathbf{a}}_t^\top (\mathbf{y} + t'\mathbf{d})$. This could further reduce the step length below t', proving that $\mathbf{y} + t'\mathbf{d}$ is infeasible.

By sequentially applying the above steps to all constraints $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$ one by one, the step length returned at the last constraint of $C_{\texttt{nom}}$ provides the sought t^* value.

In theory, the above projection algorithm could repeat Steps 2-5 many times for each i, iteratively decreasing t in a long loop. However, experiments suggest that long loops arise only rarely in practice; the value of t is typically decreased via (3.D) only a dozen of times at most *for all* (thousands of) nominal constraints, *i.e.*, for *all* i. For many nominal constraints $(\mathbf{a}, c_a) \in C_{\texttt{nom}}$, the above algorithm only concludes at Step 3 that $\mathbf{y} + t\mathbf{d}$ satisfies all robust cuts associated to $(\mathbf{a}, c_a)$, and, in such cases, the most computationally expensive task is the partial-sorting algorithm (called once at Step 2). The overall projection algorithm can even stop earlier without scanning all nominal constraints, by returning $t^* = 0$ at Step 4. An exact separation algorithm could never stop earlier, because $c_a - (\mathbf{a} + \widehat{\mathbf{a}}_{\mathbf{y}})^\top \mathbf{y}$ can certainly decrease up to the last nominal constraint $(\mathbf{a}, c_a)$. In a few cases, the projection algorithm can become even faster than the separation one. In the best case, a projection iteration takes 20% less time than a separation one. In the worst case, a projection iteration takes about 30% more time than a separation one.

3.3 Numerical results on robust LP instances

We use the `Netlib` instances from [11], considering $\Gamma \in \{1, 10, 50\}$. In fact, we discarded all instances that are infeasible for $\Gamma = 50$, since our methods are not designed to detect infeasibilities. We also ignored all instances solved by the algorithm from [11] in less than 5 iterations (*i.e.*, `seba`, `shell` and `woodw`) because they are too small to produce meaningful comparisons; they are also the only instances that `Proj-Cut-Pl` can solve in very few iterations. We thus remain with a test bed of 21 instances with between $k = 1000$ and $k = 15695$ variables. We refer to [11, Table 1] for the nominal objective value of each instance. We mention that `stocfor3` is an exceptionally large instance with $k = 15695$ and more than 15000 constraints.

All instances are formulated with a minimization objective, so that the inner solutions $\mathbf{y}_{\mathtt{it}}$ determined by `Proj-Cut-Pl` along the iterations `it` generate *upper* bounds $\mathbf{b}^\top \mathbf{y}_{\mathtt{it}}$.

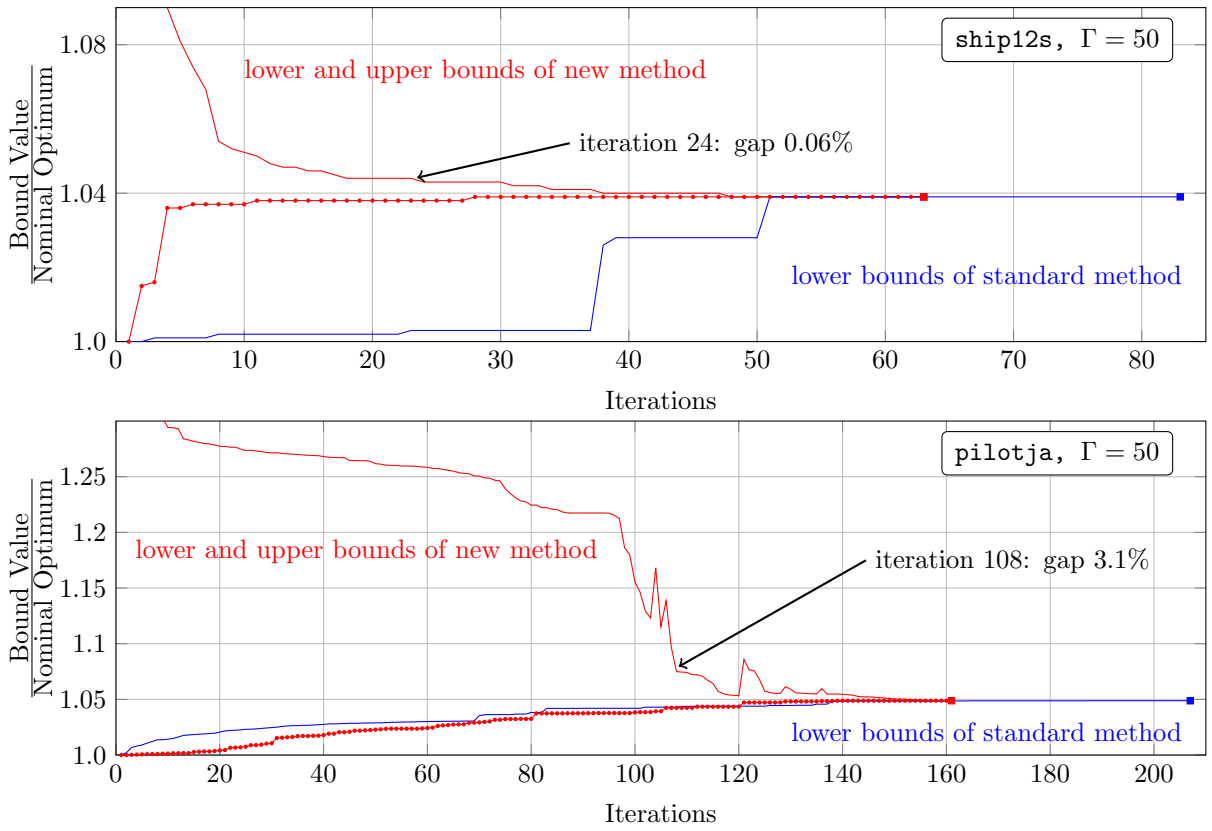


Figure 3.C: The progress over the iterations of the lower and upper bounds reported by the `Proj-Cut-Pl` (in red), compared to those of the standard `Cutting-Planes` (lower bounds only, in blue).

Figure 3.C plots the running profile of the standard `Cutting-Planes` compared to that of the `Proj-Cut-Pl` on two instances. The standard `Cutting-Planes` needed 83 iterations to fully converge on the first instance, while `Proj-Cut-Pl` reported a feasible solution with a proven low gap of 0.06% after only 24 iterations. On the second instance, `Cutting-Planes` needed 207 iterations to fully converge, while `Proj-Cut-Pl` reported a feasible solution with a proven low gap of 3.1% after 108 iterations, as indicated by the arrows in the figure.

Table 2 compares the total computing effort (iterations and CPU time) needed to fully solve each instance using the new and the standard method. For `Proj-Cut-Pl`, we also provide the computing effort needed to reach a gap of 1% between the lower and the upper bounds; this may often require a very short time. For example, the standard `Cutting-Planes` needed between 45 minutes and one hour (depending on Γ) to determine the optimal solution for the last instance `stocfor3`, while the `Proj-Cut-Pl` reported in less than 3 seconds a feasible solution with a proven gap below 1% (see columns "gap 1%" in bold in the last row).

This table suggest that another advantage of `Proj-Cut-Pl` (for this problem) lies in its immunity to degeneracy-related issues. For `sctap2` and `sctap3` with $\Gamma = 50$, the standard `Cutting-Planes` is seriously slowed down by degeneracy issues, *i.e.*, it performs too many Simplex pivots that only change the basis without improving the objective value. It thus needs significantly more iterations than normally expected — see the figures in bold in the rows of `sctap2` and `sctap3`. We suppose that such degeneracy phenomena are also visible for `czprob` with $\Gamma = 50$ in Table 1 of [11], because their algorithm takes 100 times more time for $\Gamma = 50$ than for $\Gamma = 10$, which is unusual.

Observation 3.B. *Except for the above experiments, the degeneracy issues of the standard* `Cutting-Planes` *are not very visible in other* `Cutting-Planes` *implementations from this thesis. Yet such problems are well acknowledged in the* `Cutting-Planes` *literature, especially in Column Generation. As [31, §4.2.2] put it, "When the master problem is a set partitioning problem, large instances are difficult to solve due to massive degeneracy [...] Then, the value of the dual variables are no meaningful measure for which column to adjoin to the Reduced Master Problem". In* `Proj-Cut-Pl`*, the inner-outer solutions* $\mathbf{y}_{it}$ *and* $\mathtt{opt}(\mathscr{P}_{it-1})$

	Γ = 50							Γ = 10							Γ = 1						
	OPT	new method				std. method		OPT	new method				std. method		OPT	new method				std. method	
	(+%)	gap 1%		full converg.		full converg.		(+%)	gap 1%		full converg.		full converg.		(+%)	gap 1%		full converg.		full converg.	
Instance		iters	time	iters	time	iters	time		iters	time	iters	time	iters	time		iters	time	iters	time	iters	time
25fv47	2.548	146	1.168	149	1.191	199	1.265	2.541	158	1.188	169	1.273	204	1.455	1.457	135	1.023	147	1.11	149	1.12
bnl2	1.847	486	13.66	491	13.81	1927	48.13	1.84	703	20.92	708	21.08	1295	31.31	0.7903	501	14.47	504	14.56	552	12.66
czprob	0.6401	61	0.485	734	32.3	1293	58.52	0.3749	32	0.181	125	0.899	170	1.302	0.1223	14	0.0935	25	0.196	25	0.124
ganges	0.4736	1	<0.001	25	0.059	25	0.059	0.4302	1	<0.001	31	0.085	33	0.074	0.0531	1	<0.001	25	0.063	25	0.049
gfrd-pnc	0.0649	64	0.1404	64	0.141	64	0.092	0.0649	64	0.1086	64	0.109	64	0.090	0.0592	67	0.1415	67	0.142	67	0.100
maros	12.12	272	2.402	278	2.458	379	3.518	12.11	281	2.508	300	2.683	395	3.486	5.76	200	1.812	219	1.983	227	1.714
nesm	0.8752	56	0.579	80	0.798	80	0.659	0.8752	56	0.604	80	0.824	80	0.658	0.4515	58	0.647	82	0.880	82	0.639
pilotja	4.877	121	2.418	161	3.042	207	3.701	4.815	125	2.316	172	3.074	179	3.418	2.344	110	1.522	135	1.908	143	1.693
pilotnov	8.51	96	4.227	120	4.615	119	3.714	8.51	103	6.12	139	6.912	141	4.915	4.402	94	3.355	120	3.805	119	1.776
pilotwe	6.109	102	0.97	118	1.102	143	1.204	6.108	102	1.045	119	1.19	144	1.308	3.193	98	0.853	115	1.005	124	1.066
scfxm2	2.114	93	0.387	139	0.584	146	0.537	2.113	101	0.401	152	0.603	150	0.498	0.9889	88	0.357	131	0.536	142	0.486
scfxm3	2.142	139	0.957	196	1.353	215	1.27	2.141	142	0.955	227	1.575	222	1.309	0.977	91	0.605	197	1.352	213	1.216
sctap2	2.844	185	1.946	242	2.567	**6545**	147.3	2.814	332	3.685	696	8.4	954	10.62	1.533	191	2.035	353	3.88	302	2.644
sctap3	3.04	145	2.649	239	4.55	**9463**	366.1	2.995	180	3.45	773	15.49	1168	20.22	1.602	213	3.785	406	7.394	347	4.799
ship08l	0.1244	1	0.002	20	0.111	29	0.171	0.1157	1	0.002	19	0.128	23	0.134	0.0300	1	0.002	19	0.127	24	0.147
ship08s	0.1396	2	0.006	32	0.122	42	0.139	0.129	2	0.006	34	0.134	35	0.123	0.0317	1	0.001	32	0.123	38	0.127
ship12l	0.3528	1	<0.001	48	0.442	65	0.576	0.3462	1	<0.001	48	0.418	65	0.555	0.0600	1	0.004	45	0.451	56	0.483
ship12s	0.3898	4	0.015	63	0.377	83	0.376	0.3857	5	0.019	64	0.387	86	0.398	0.0617	4	0.015	58	0.305	63	0.281
sierra	0.0239	1	0.001	54	0.414	61	0.567	0.0239	1	0.001	54	0.412	61	0.569	0.0223	1	0.004	51	0.538	51	0.483
stocfor2	1.522	6	0.022	437	5.047	484	6.67	1.522	7	0.025	438	5.387	486	6.562	0.7588	3	0.054	438	7.573	712	10.3
stocfor3	1.482	29	**2.192**	3777	2125	4329	2701	1.482	32	**1.862**	3781	2029	4330	2851	0.7327	1	**0.99**	3720	3023	6069	3482

Table 2: Results of `Proj-Cut-`*`Pl`* and standard `Cutting-Planes` on the robust optimization instances. The figures in bold attract your attention to instances on which the new approach is particularly successful. The columns OPT indicate the increase in percentage of the robust objective value with respect to the nominal one (with no robustness). Columns "gap 1%" indicate the computing effort needed to reach the iteration `it` when the gap between the upper bound $\mathbf{b}^\top \mathbf{y}_{\mathtt{it}}$ and the lower bound $\mathtt{optVal}(\mathscr{P}_{\mathtt{it}})$ is below 1%, *i.e.*, either $0 < \mathtt{optVal}(\mathscr{P}_{\mathtt{it}}) \leq \mathbf{b}^\top \mathbf{y}_{\mathtt{it}} \leq 1.01\mathtt{optVal}(\mathscr{P}_{\mathtt{it}})$ or $\mathtt{optVal}(\mathscr{P}_{\mathtt{it}}) \leq \mathbf{b}^\top \mathbf{y}_{\mathtt{it}} \leq 0.99\mathtt{optVal}(\mathscr{P}_{\mathtt{it}}) < 0$.

represent together a more "meaningful measure" for selecting a new constraint, avoiding iterations that keep the objective value constant.

Table 2 from [43] provides a similar evaluation of a `Proj-Cut-Pl` variant that generates multiples cuts per iteration as described in [43, § 3.1.3]. With few exceptions, this multi-cut variant requires (far) less iterations than the multi-cut `Cutting-Planes`. On roughly half of the instances, it needs less than 10 iterations, which is surprisingly low. In the best case (instance `stocfor3` for $\Gamma = 1$), the multi-cuts `Proj-Cut-Pl` fully converged in less than 10 seconds (and 3 iterations), while all other studied algorithms needed thousands of iterations and thousands of seconds to fully converge for the same problem.

3.4 Conclusions on `Proj-Cut-Pl`

An advantage of `Proj-Cut-Pl` lies in its *built-in mechanism* to generate feasible inner solutions along the iterations; these inner solutions converge towards $\texttt{opt}(\mathscr{P})$, somehow similarly to the solutions of the central path in interior point algorithms. The standard `Cutting-Planes` does not contain such a built-in mechanism. In fact, even if some ad-hoc methods may be used in `Cutting-Planes` to construct feasible solutions along the iterations, these inner solutions represent merely a by-product of the `Cutting-Planes` algorithm, not meant to influence its evolution in any setting beyond the considered problem.

Implementing `Proj-Cut-Pl` can be difficult (compared to a standard `Cutting-Planes`), primarily because designing a projection algorithm may remain rather challenging for general polytopes $\mathscr{P}$ and different problems. In some sense, one of the hardest tasks was to find well-known problems for which such a general projection sub-problem is not prohibitively-hard, at least not substantially harder than the separation one. I must confess one of my concerns in the beginning was that it might be necessary to invent myself new problems for which these ideas lead to numerical success. Finally, such artificial problems were not needed: there do exist well-established problems for which `Proj-Cut-Pl` works very well.

4 Solving (Easily-Feasible) Semidefinite Programs via `Proj-Cut-`*Pl*

Semidefinite programming (SDP) represents a significant generalization of linear programming that is still polynomially solvable and that may offer stronger relaxations of certain combinatorial problems. This is often the case for combinatorial problems with a quadratic or non-linear component; if the underlying problem is completely linear, the potential of an SDP relaxation may be very limited. This area of study is also rich in theoretical development, at least because certain SDP relaxations (like the one in Section 4.6.6) may offer provable guarantees on the proximity of the relaxed solution to the optimal one. We will still use variables $\mathbf{y} \in \mathbb{R}^k$ as in the previous chapters, but the constraints will involve SDP matrices of size $n \times n$.

4.1 SDP optimization as an LP with infinitely-many linear constraints

A semi-positive (SDP) matrix serves as a generalization of the concept non-negativity in the space of real symmetric matrices. This extension allows for a more comprehensive framework in optimization problems, where certain positivity properties will be applied to matrices rather than merely to scalar values, *e.g.*, an SDP matrix can be written as a kind of sum of squares (technically, outer products $\mathbf{v}\mathbf{v}^\top$ with $\mathbf{v} \in \mathbb{R}^n$), all its eigenvalues are non-negative, the dot product of two SDP matrices is non-negative, etc. We try to make this chapter as self-contained as possible, without extending its length to any inappropriate degree. In a few cases, I will refer to my personal expository work [38] of roughly 100 pages that is enough to cover all background necessary for this chapter.

4.1.1 Notations and preliminaries

We will use multiple times the following technical notations and facts (see [38, § 1] for proofs).

Definition 4.A. *A symmetric matrix $X \in \mathbb{R}^{n\times n}$ is positive semidefinite (SDP) if the following quadratic form is non-negative for any non-zero $\mathbf{d} \in \mathbb{R}^n$*

$$\mathbf{d}^\top X \mathbf{d} = X \bullet \mathbf{d}\mathbf{d}^\top = \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} d_i d_j \geq 0, \tag{4.A}$$

where $\bullet$ stands for the scalar product. If the above inequality is always strict, the matrix is positive definite.

Equivalently, a real symmetric matrix $X \in \mathbb{R}^{n\times n}$ is SDP if and only if all its eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ are non-negative. In such case, we write $X \succeq \mathbf{0}$ and say that X accepts the following *eigendecomposition*.

$$\begin{aligned} X &= \underbrace{\begin{bmatrix} v_{11} & v_{21} & \ldots & v_{n,1} \\ v_{21} & v_{22} & \ldots & v_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n,1} & v_{n,2} & \ldots & v_{n,n} \end{bmatrix}}_{\text{unitary eigenvectors of } X} \underbrace{\begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix}}_{\text{eigenvalues of } X} \underbrace{\begin{bmatrix} v_{11} & v_{12} & \ldots & v_{1,n} \\ v_{12} & v_{22} & \ldots & v_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,n} & v_{2,n} & \ldots & v_{n,n} \end{bmatrix}}_{\text{transposed eigenvectors}} \\ &= \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \ldots \mathbf{v}_n \end{bmatrix} \mathtt{diag}(\lambda_1, \ \lambda_2, \ \ldots \lambda_n) \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_n^\top \end{bmatrix} \\ &= \sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top, \end{aligned} \tag{4.B}$$

where $\mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_n$ are the unitary (column) eigenvectors of X associated to (some repeated) eigenvalues $\lambda_1, \ \lambda_2, \ldots \lambda_n$ and $\mathtt{diag}(\lambda_1, \ \lambda_2 \ \ldots, \lambda_n)$ is the diagonal matrix with $\lambda_1, \ \lambda_2 \ \ldots, \lambda_n$ on the diagonal. The eigenvectors are unitary and orthogonal, meaning that $\mathbf{v}_i^\top \mathbf{v}_j = 0, \ \forall i, j \in [1..n], \ i \neq j$ and $\mathbf{v}_i^\top \mathbf{v}_i = 1 \ \forall i \in [1..n]$. Perhaps a simpler way to visualise the form of an SDP matrix consists of rewriting it as a $\lambda-$weighted

sum of outer products $X = \sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top$. Using subscript i as a superscript, we can develop this sum as follows.

$$X = \sum_{i=1}^{n} \lambda_i \mathbf{v}^i \mathbf{v}^{i\top} = \sum_{i=1}^{n} \lambda_i \begin{bmatrix} v_1^i \cdot v_1^i & v_1^i \cdot v_2^i & \dots & v_1^i \cdot v_n^i \\ v_2^i \cdot v_1^i & v_2^i \cdot v_2^i & \dots & v_2^i \cdot v_n^i \\ \vdots & \vdots & \ddots & \vdots \\ v_n^i \cdot v_1^i & v_n^i \cdot v_2^i & \dots & v_n^i \cdot v_n^i \end{bmatrix}$$

In a loose sense, this is a way of writing X as a kind of sum of squares. If X satisfies above property, it is not hard to check that $\mathbf{d}^\top X \mathbf{d} \geq 0 \ \forall \mathbf{d} \in \mathbb{R}^n$, because $\mathbf{d}^\top \left(\sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top \right) \mathbf{d} = \sum_{i=1}^{n} \lambda_i \mathbf{d}^\top \mathbf{v}_i \mathbf{v}_i^\top \mathbf{d} = \lambda_i (\mathbf{v}_i^\top \mathbf{d})^2$. If all eigenvalues are strictly positive (*i.e.*, $\lambda_i > 0 \ \forall i \in [1..n]$), X is positive definite. One can generate an SDP matrix by taking a (Gram) product $VV^\top$ for any matrix with n rows, see [38, § 1.4]. Such a matrix has to be SDP because $\mathbf{d}^\top VV^\top \mathbf{d} = u_1^2 + u_2^2 + \dots u_n^2 \geq 0$, where $[u_1 \ u_2 \dots u_n] = \mathbf{d}^\top V$.

We can cast any semidefinite program as an LP with infinitely many cuts. The set of SDP matrices is not a polyhedron but a cone. Yet we will apply `Proj-Cut-Pl` as if this cone were a polyhedron, in the sense that we only need the following linear characterization that derives from (4.A):

$$X \text{ is SDP if and only if } X \bullet \mathbf{d}\mathbf{d}^\top \geq 0 \text{ for any vector } \mathbf{d} \in \mathbb{R}^n. \tag{4.C}$$

4.1.2 The `Cutting-Planes` LP models

The general idea of solving SDP programming by `Cutting-Planes` is not new [26, 51, 52, 53, 54, 25], but we hope the projection logic can push its potential to a higher level of (practical) performance. We consider the following semidefinite optimization problem that we will reformulate to make it similar to the LP models (2.A) or (3.A); we will still use a `Cutting-Planes` framework like in Chapters 2 and 3.

$$(SDP) \begin{cases} \max\limits_{\mathbf{y} \in \mathbb{R}^k} & \mathbf{b}^\top \mathbf{y} & (4.D.1) \\ s.t & \mathcal{A}^\top \mathbf{y} \preceq A_0 & (4.D.2) \\ & \mathbf{a}^\top \mathbf{y} \leq c_a \ \forall (\mathbf{a}, c_a) \in \mathscr{C} & (4.D.3) \end{cases}$$

where $\mathcal{A}^\top \mathbf{y} = \sum_{i=1}^{k} A_i y_i$; $A_1, \ A_2, \dots, A_k$ and A_0 are symmetric $n \times n$ matrices. The set $\mathscr{C}$ in (4.D.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 $c_a = 0$). This set $\mathscr{C}$ usually contains a number of initial linear constraints. But in Section 4.6.5.1, $\mathscr{C}$ will evolve to contain prohibitively many robust constraints derived using the robust logic from Chapter 3.

It may be useful to follow the evolution of certain algorithmic steps with regards to the SDP dual of above program. All SDP optimization algorithms return a primal-dual pair of optimal solutions.

"$= b_i$" becomes "$\geq b_i$" if (4.D.3) contains $y_i \geq 0$

$$(DSDP_{\mathscr{C}}) \begin{cases} \min & C \bullet S + \sum_{(\mathbf{a}, c_a) \in \mathscr{C}} c_a x_a & (4.E.1) \\ s.t. & A_i \bullet S + \sum_{(\mathbf{a}, c_a) \in \mathscr{C}} a_i x_a = b_i \ \forall i \in [1..k] & (4.E.2) \\ & S \succeq \mathbf{0}, \mathbf{x} \geq \mathbf{0} & (4.E.3) \end{cases}$$

To reach a form better suited to `Cutting-Planes`, (4.D) is equivalent to the program below when $\mathscr{D}$ is equal to $\mathbb{R}^n$ in (4.F.4); in such case, (4.F.4) actually reduces to $X \succeq \mathbf{0}$ for $X = A_0 - \mathcal{A}^\top \mathbf{y}$. Recall from (4.C) that $X \succeq \mathbf{0} \iff X \bullet \mathbf{d}\mathbf{d}^\top \geq 0 \ \forall \mathbf{d} \in \mathbb{R}^n$, where $X \bullet \mathbf{d}\mathbf{d}^\top = \mathbf{d}^\top X \mathbf{d}$.

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

In a `Cutting-Planes` scheme, both sets $\mathscr{C}$ and $\mathscr{D}$ could be generated on the fly; we can say $\mathscr{C} \cup \mathscr{D} = C$ stand for the constraints C from (2.A) or (3.A) in the previous chapters. The constraints $\mathscr{D}$ are variously termed eigenvalue-cuts [8, eq. (26)], eigenvector-cuts, or simply eigen-cuts [32, § 1.1]. Since we can not enumerate them all and reach $\mathscr{D} = \mathbb{R}^n$, one difficulty will be to control the growth of the set $\mathscr{D}$ along the iterations. The goal is to find the optimum solution of the SDP program (4.D.1)-(4.D.3) using reasonably-sized sets $\mathscr{C}$ and $\mathscr{D}$ in the LP (4.F.1)-(4.F.4). Using the *LP duality*, we write below the dual of above *(SDP−LP$_{\mathscr{C},\mathscr{D}}$)*, using a variable $\lambda_{\mathbf{d}}$ for each $\mathbf{d} \in \mathscr{D}$. The proposed method will use this dual LP to construct at each iteration a dual SDP solution alongside the primal one.

"$= b_i$" becomes "$\geq b_i$" if (4.F.3) contains $y_i \geq 0$

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

Notice the variables $\mathbf{x}$ are the duals of (4.F.3) and $\mathbf{d}$ are the duals of (4.F.4). Given a fixed $\mathscr{D}$ obtained at some `Cutting-Planes` iteration working on *(SDP−LP$_{\mathscr{C},\mathscr{D}}$)*, we can use the dual multipliers λ_d (with $d \in \mathscr{D}$) and generate SDP matrix $S = \sum_{\mathbf{d}\in\mathscr{D}} \lambda_{\mathbf{d}} \mathbf{d}\mathbf{d}^\top \succeq \mathbf{0}$. The set of matrices that can be generated this way for a *fixed* $\mathscr{D}$ may cover only a subset of the SDP cone. Any such matrix S is a feasible dual SDP solution in (4.E.1)-(4.E.3). Thus, we can construct both a primal and dual solution of (4.D.1)-(4.D.3) along the iterations, see also [52, Theorem 9].[4]

This work is devoted to solving SDP programs with a large n, of the order of thousands; k is usually much smaller. But the projection sub-problem can also be used to solve by `Proj-Cut-Pl` the SDP dual (4.E), which can be useful when n is very small and k very large, *i.e.*, when k is larger than the number $\frac{n\cdot(n+1)}{2}$ of free elements of a symmetric matrix of size $n \times n$. In such case, to ease the work of the LP solver, it may be more practical to make the `Cutting-Planes` work with (4.E) considering (the lower triangular part of) S as linear variables. We also have to replace $S \succeq \mathbf{0}$ with the infinitely-many cuts $S \bullet \mathbf{d}\mathbf{d}^\top \geq 0\ \forall \mathbf{d} \in \mathbb{R}^n$. A `Cutting-Planes` algorithm for such (quite rare) instances was developed in [26].

4.2 Adapting `Proj-Cut-Pl` to (robust) SDP programming

The number $|\mathscr{C}|$ of linear constraints in (4.D.3) or (4.F.3) is usually finite in semidefinite optimization. But what happens if the coefficients of these linear constraints are subject to change and vary according to robust programming principles? We would obtain a robust SDP program version with prohibitively-many linear constraints. I think it is not very hard to prove almost theoretically that `Proj-Cut-Pl` may solve such a robust SDP program more rapidly than many other methods not (yet) adapted to such a case. Or, at least, `Proj-Cut-Pl` can easily be adapted to an infinite set $\mathscr{C}$, by solving two projection sub-problems at each general iteration, one for $\mathscr{C}$ using the algorithm from Section 3.2.2 and one for the SDP constraints $\mathscr{D}$ (as developed in this chapter). We will discuss this case in Section 4.6.5.1, but for now we present the new approach considering a set $\mathscr{C}$ that does not require `Cutting-Planes`.

We did our best to design a very lightweight approach for SDP optimization, mostly using linear tools. It is enough to take the `Proj-Cut-Pl` description from Section 3.1 and to apply it to (4.F). A step-length of $\alpha = 0.3$ is a safe choice. Figure 4.A is the non-linear counterpart of Figure 3.A from Section 3.1, presenting a more clear image of the evolution of `Proj-Cut-Pl` when optimizing over a non-linear feasible area.

The projection sub-problem from Definition 3.A reduces to finding the maximum t^* so that $X + t^* D \succeq \mathbf{0}$. In our context, X is the SDP matrix $X = A_0 - \mathcal{A}^\top \mathbf{y}_{in} \succeq \mathbf{0}$ associated to the current inner point $\mathbf{y}_{in}$ of (4.F) and $D = -\mathcal{A}^\top(\mathbf{y}_{out} - \mathbf{y}_{in})$, where $\mathbf{y}_{out}$ is the current outer point. We also have to determine a first-hit vector $\mathbf{v} \in \mathbb{R}^n$ such that the following condition proves $X + tD \not\succeq \mathbf{0}\ \forall t > t^*$:

$$(X + t^* D) \bullet \mathbf{v}\mathbf{v}^\top = 0 \text{ and } D \bullet \mathbf{v}\mathbf{v}^\top < 0. \tag{4.H}$$

[4] Program (4.G) could also be useful in a different way: there may be other sets $\hat{\mathscr{D}}$ so that $S = \sum_{\hat{\mathbf{d}}\in\hat{\mathscr{D}}} \lambda_{\hat{\mathbf{d}}} \hat{\mathbf{d}}\hat{\mathbf{d}}^\top$. For instance, one could compute the eigendecomposition of S and take the vectors $\mathbf{v}^i$ with $i \in [1..n]$ from (4.B) to constitute new vectors $\hat{\mathbf{d}}$. These vectors could generate a different set of constraints (4.F.4) that are tight for X.

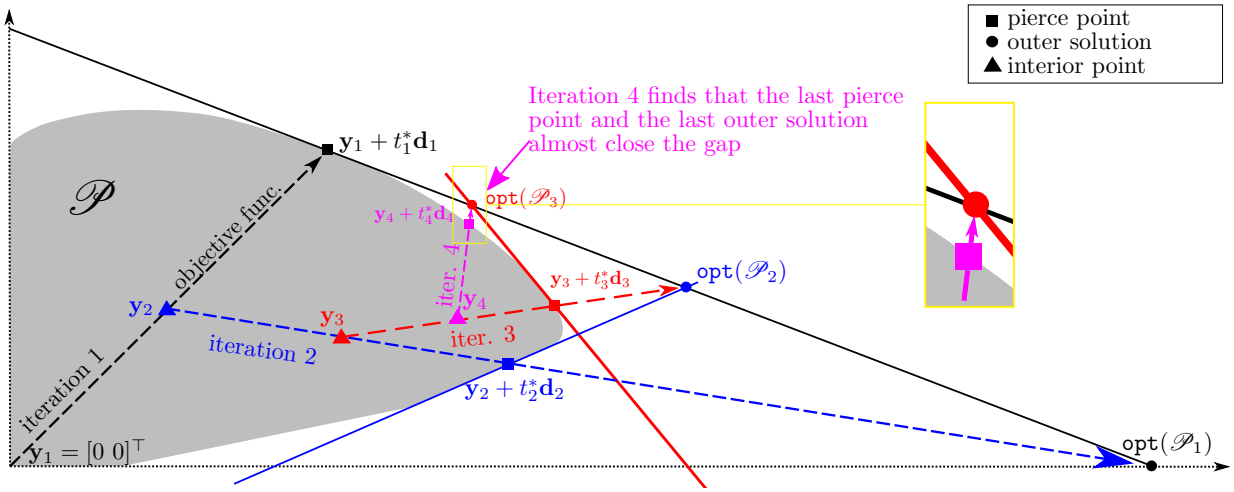


Figure 4.A: SDP `Proj-Cut-Pl` illustration for $k = 2$. The red and black constraints are tangents to the non-linear feasible area determined by $\mathcal{A}^\top\mathbf{y} \preceq A_0$. The blue constraint belongs to set $\mathscr{C}$. The first iteration projects $\mathbf{y}_1 = \mathbf{0} = [0\ 0]^\top$ along the objective function, as depicted by the black dashed arrow. At iteration $\texttt{it} = 2$, the midpoint $\mathbf{y}_2$ of this black arrow is projected towards the optimal outer solution $\texttt{opt}(\mathscr{P}_1)$ — at iteration 1, the outer approximation $\mathscr{P}_1 \supset \mathscr{P}$ only contains the largest triangle. This generates a second facet (solid blue line) that is added to the facets of $\mathscr{P}_1$ to construct $\mathscr{P}_2$. 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 on this figure.

The situation is a bit more complex than for a polyhedral feasible area, because (4.H) may not be satisfied strictly. 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 $\mathbf{v}$ that satisfies (4.H) exactly. Unlike in the rest of this thesis, addressing prosaic questions about infinitesimal tolerances will become a necessary evil for multiple (other) reasons. For example, if the equality from (4.H) is true only within a small tolerance range, we need to find a vector $\mathbf{v}$ like $[\epsilon\ \ 1]^\top$ with an infinitesimal ϵ such that $D \bullet \mathbf{v}\mathbf{v}^\top$ is significantly below zero while $X \bullet \mathbf{v}\mathbf{v}^\top$ is almost zero. In such a case, advancing a sensible short step t_ϵ beyond t^* will push $(X + (t^* + t_\epsilon)D) \bullet \mathbf{v}\mathbf{v}^\top$ seriously below zero, out of the equality tolerance range.

4.2.1 Finding a first feasible solution

For many instances from the literature it is not (so) easy to find a single feasible solution. While `Proj-Cut-Pl` 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.

For most considered instances, the first stage below was sufficient. It relies on using 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 can be seen as an iterative heuristic seeking to advance along exterior-to-interior projections to solve the SDP feasibility problem, *i.e.*, the considered SDP program (4.D.1)-(4.D.3) with a null objective function $\mathbf{b} = \mathbf{0}_k$. To solve this variant, some Interior Point Methods (IPMs) may need up to one-third of the computational time they require for an arbitrary $\mathbf{b}$.

Stage 1 A first inner point using Weyl's inequalities
To (try to) get a first feasible solution $\mathbf{y}_{\text{in}}$, we will construct an LP that is included in the feasible area of (4.D). For this, we seek some linear constraints that imply (4.D.2), *i.e.*, so that $\mathcal{A}^\top\mathbf{y} \preceq A_0$. To ensure this relation, we need to ensure $\lambda_{\max}(\mathcal{A}^\top\mathbf{y} - A_0) \leq 0$, where $\lambda_{\max}(\cdot)$ and $\lambda_{\max}(\cdot)$ is the maximum, resp., minimum eigenvalue function. Weyl's inequalities 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 $\mathcal{A}^\top\mathbf{y} \preceq A_0$:

$$\sum_{i=1}^{k} \lambda_{\max}(A_i) \cdot y_i \leq \underbrace{\lambda_{\min}(A_0)}_{-\lambda_{\max}(-A_0)} . \tag{4.I}$$

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, `Proj-Cut-Pl` does not necessarily start with $\mathbf{y}_{\text{in}} = \mathbf{0}_k$ when $\mathbf{0}_k$ is feasible. It constructs an LP by replacing (4.D.2) with (4.I) in (4.D). 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 variable $\overline{\mathbf{y}}$ so that $\overline{y}_i \geq \lambda_{\max}(A_i) y_i$ and $\overline{y}_i \geq \lambda_{\min}(A_i) y_i$. These two conditions will suffice to ensure $\overline{y}_i \geq \lambda_{\max}(A_i y_i)$. The function $y_i \to \overline{y}_i$ is piecewise convex, like the absolute value function. And we construct a first $\mathbf{y}_{\text{in}}$ by maximizing a similar program but extended with k variables $\overline{\mathbf{y}}$ and replacing (4.I) with $\sum \overline{y}_i \leq \lambda_{\min}(A_0)$.

If (4.I) does not lead to a feasible solution as described above, we must have $\lambda_{\min}(A_0) < 0$. This is the only case in which $\mathbf{y} = \mathbf{0}_k$ can be infeasible in (4.I), letting aside the initial linear constraints $\mathscr{C}$ from (4.F.3). It is even impossible to find other feasible solution in (4.I) if we have $\lambda_{\max}(A_i) \geq 0$ for all $i \in [1..k]$ when $\mathbf{y}$ is non-negative, or $\lambda_{\max}(A_i) = 0$ if $\mathbf{y}$ is free.

Stage 2: A heuristic for solving the feasibility SDP problem (the initial SDP model for $\mathbf{b} = \mathbf{0}_k$)
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 from $\mathscr{C}$, meaning that X has a set of $\overline{j}$ negative eigenvalues $\lambda_1 \leq \lambda_2 \leq \lambda_3 \cdots \leq \lambda_{\overline{j}} < 0$ associated to eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots \mathbf{v}_{\overline{j}}$ such that

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

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 $j \in [1..\overline{j}]$ 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 $j \in [1..\overline{j}]$ 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 means that when advancing along the direction $\mathbf{y}_{\text{out}} \to \mathbf{y}_{\text{in}}$, each function f_j is strictly increasing. The main idea of the proposed heuristic is to iteratively determine and follow this kind of directions until getting inside the SDP cone. To determine the direction we solve the following LP for some reasonable $\Delta = 100$ that simply puts a box around $\mathbf{y}_{\text{out}}$:

$$\min_{t,\mathbf{y}} \left\{ t :\ t \geq f_j(\mathbf{y}) - f_j(\mathbf{y}_{\text{out}})\ \forall j \in [1..\overline{j}],\ \mathbf{y}_{\text{out}} - \Delta \leq \mathbf{y} \leq \mathbf{y}_{\text{out}} + \Delta \right\}.$$

If the optimal value of this LP is not strictly positive, one can stop by reporting the instance is infeasible, because of this: any feasible solution $\mathbf{y}_{\text{in}}$ of the initial SDP model yields $f_j(\mathbf{y}_{\text{in}}) \geq 0\ \forall j \in [1..\overline{j}]$ so that $\mathbf{y}_{\text{out}} + \tau(\mathbf{y}_{\text{in}} - \mathbf{y}_{\text{out}})$ remains feasible and attains a positive objective value in the above LP for any $\tau > 0$. We hereafter consider the optimum value of this LP is strictly positive; let $\mathbf{y}^*$ be the associated optimal solution. The goal is to make the minimum eigenvalue of X increase when 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. However, given the exterior-to-interior projection direction $\mathbf{y}_{\text{out}} \to \mathbf{y}^*$, the next question is to determine an appropriate step length τ.

The ideal τ should maximize $g(\tau) := \lambda_{\min}(X + \tau X^*)$ where $X^* = -\sum_{i=1}^{k} A_i\, (y_i^* - y_i^{\text{out}})$. The sub-differential (the set of all subderivatives) of g at $\tau = 0$ is the set $-\sum_{i=1}^{k} A_i \bullet \mathbf{v}\mathbf{v}^\top\, (y_i^* - y_i^{\text{out}})$ where $\mathbf{v}$ belongs to the convex hull of the eigenvectors of X associated to the minimum eigenvalue λ_1. These eigenvectors surely contain the above $\mathbf{v}_1$ and possibly others vectors in subsequent positions among $\mathbf{v}_2, \mathbf{v}_3, \dots \mathbf{v}_{\overline{j}}$. But all these eigenvectors are considered in the above LP, 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 advancing from $\mathbf{y}_{\text{out}}$ to $\mathbf{y}_{\text{out}} + \tau(\mathbf{y}^* - \mathbf{y}_{\text{out}})$. This is a kind of projection from the exterior of the SDP cone towards its interior.

In practice, the final τ is determined by evaluating the piecewise concave function g over multiple positive values $\tau_1, \tau_2, \tau_3, \dots$ in 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 interval (τ_2, τ_4).

These ideas have been implemented into a practical algorithm that also integrates a few simple engineering customizations that may be needed to reduce the number of iterations. For instance, it is often useful to consider less than $\bar{j}$ eigenvectors, *i.e.*, only those associated to 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 to a positive close-to-zero eigenvalue. More customizations are naturally needed when implementing this dichotomic search and such practical aspects could constitute alone the subject of a short paper on a simple SDP heuristic. Yet, since this is not directly related to the main core engine of `Proj-Cut-Pl` and since this stage was only used in Section 4.6.2, it may be superfluous to provide here a more detailed description of the implementation.

4.2.2 The initial outer approximation

In the very beginning there is no default constraint that `Proj-Cut-Pl` may use to construct a very first outer approximation of the feasible area $\mathscr{P}$ of (4.F). This may be useful to generate quality outer solution in the beginning. For this purpose, we add some initial constraints to (try to) produce a reasonable outer approximation $\mathscr{P}_{\texttt{out}}$ of $\mathscr{P}$. This is the simplest constraint to be added for each $i \in [1..n]$: we insert $X_{ii} \geq 0$ into the definition of $\mathscr{P}_{\texttt{out}}$ since any SDP matrix including X has only non-negative elements on the diagonal.

We now define a second class of initial cuts that work only if $\mathbf{y} \geq \mathbf{0}$. A different form of the Weyl's inequalities state $\lambda_{\max}(A+B) \geq \lambda_{\min}(A) + \lambda_{\max}(B)$ for any two symmetric matrices A and B. This ensures the second inequality in formula below.

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

But when $\mathbf{y} \geq \mathbf{0}$, we can also apply yet another form of the Weyl's inequality, 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 $\mathscr{P}_{\texttt{out}}$ if we have non-negative variables $\mathbf{y} \geq \mathbf{0}_k$:

$$\underbrace{\lambda_{\min}(A_0)}_{-\lambda_{\max}(-A_0)} \geq \sum \lambda_{\min}(A_i) y_i \tag{4.J}$$

The use of the above constraints make the considered problem easier when it is defined in the non-negative orthant ($\mathbf{y} \geq \mathbf{0}$). Moreover, (4.J) can be applied on any principal minor of the considered matrices, to obtain a larger family of constraints. Preliminary tests show this is too expensive and does not lead to a huge speed-up.

4.2.3 Pseudo-code and computational aspects

The pseudo-code is provided by Algorithm 3, dropping the index $_{\texttt{out}}$ from $\mathscr{P}_{\texttt{out}}$. There are four important departs from the most canonical `Proj-Cut-Pl` described in Section 3.1:

1. It is often useful to add 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 $\mathbf{v}_1$ (such that $\mathbf{v}^\top X \mathbf{v}_1 = 0$). Thus, $\mathbf{v}_1$ and $\mathbf{v}_2$ shall satisfy the following

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

   The step-length t_2^* associated to $\mathbf{v}_2$ satisfies $t_2^* \geq t^*$. We thus seek a form of second projection restricted to hit-vectors in a space of size $n-1$, *i.e.*, in the null space of $X\mathbf{v}_1$. This may make sense because the above space of size $n-1$ may be 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 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 strength of the cut $(A_0 - \mathcal{A}^\top \mathbf{y}) \bullet \mathbf{v}_0\mathbf{v}_0^\top \geq 0$. Similarly, the strength of the cut returned by projection is $(A_0 - \mathcal{A}^\top \mathbf{y}_{\text{out}}) \bullet \mathbf{v}_1\mathbf{v}_1^\top < 0$; using the compact rewriting from Line 16, this strength becomes $c_a - \mathbf{a}^\top \mathbf{y}_{\text{out}} = (A_0 - \mathcal{A}^\top \mathbf{y}_{\text{out}}) \bullet \mathbf{v}_1\mathbf{v}_1^\top$. If this is 100 times weaker than the strength $\lambda_{\min}$ 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, like the practical projection with tolerance windows from Section 4.4 or (optionally) to stop earlier if $\lambda_{\min}$ is too close to zero even when $t^* \ll 1$.[5]

3. Like in other `Cutting-Planes` approaches, some 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 perform 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 4.3 and one in Section 4.4. The first one is exact, while the second one may return (sometimes) an underestimated step-length $t^*_{\texttt{safe}} \leq t^*$ and an inexact hit vector $\widehat{\mathbf{v}}_1$ that may be (a bit) distant from the real pierce point. Moreover, each of the two algorithms may be executed in practice with different options, and so, choosing the best code to calculate the projection under the current running conditions is quite challenging in practice. The decision may depend on questions like: is $\mathbf{y}_{\text{in}}$ close to zero? Is the artificial box over $\mathscr{P}$ from Obs. 4.B (Section 4.2.3.1) still in place? Does $\mathbf{y}_{\text{in}}$ include an artificial term that makes it feasible at the cost of penalizing the objective? An interesting option is to launch the slower exact algorithm in background[6] and wait for it (or not!) depending on the running time and the results of the fastest inexact algorithm. The overall pseudo-code works in the same manner whatever algorithm is called at Line 9, even if it returns an underestimated step-length $t^*_{\texttt{safe}}$.

4.2.3.1 Engineering customizations and design choices

Like for many projects that imply software development, the overall pseudo-code provides the main guidelines, but certain adaptations are necessary to make `Proj-Cut-Pl` reach its full potential. We here describe only a few (but not all) design choices that may be important for a reader interested in implementing the new approach; other readers may safely skip this part.

Lines 6-7 involve computing two 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 my first implementation I was surprised to find that the calculation of these two sums was the main computational bottleneck for $n \geq 1000$ and $k \geq 500$, being slower than all 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 each matrix A_i as an column vector of size $\frac{n(n+1)}{2}$ and to collect the k vectors into a matrix $\widetilde{A} \in \mathbb{R}^{k \times \frac{n(n+1)}{2}}$. This $\widetilde{A}$ is calculated only once, before starting the iterations. Any product $\mathcal{A}^\top \mathbf{y}$ is then computed as $\mathbf{y}^\top \widetilde{A}$, using the very optimized `Matlab` routines for matrix multiplication. Many other calculations can be accelerated using this flat (vectorized) version of the A_i matrices.

The likelihood 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 density is below 3%. We simply use the `Matlab` features of 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).

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 `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

[5] If $\mathbf{y}_{\text{in}} = \mathbf{0}$, projecting $\mathbf{0} \to D$ reduces to returning $t^* = \infty$ if $D \succeq \mathbf{0}$ of 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.

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

Algorithm 3 `Proj-Cut-Pl` for maximizing $\mathbf{b}^\top \mathbf{y}$ in (4.F)

1: Construct a first inner solution $\mathbf{y}_{\text{in}}$ using Section 4.2.1.
2: Build an initial outer approximation $\mathscr{P}$ of the feasible area of (4.F) using Section 4.2.2.
3: (optional presolve) Use standard `Cutting-Planes` to improve the above outer approximation, trying to remove from $\mathscr{P}$ the extreme rays that do not appear in (4.F) – see Obs. 4.B, Section 4.2.3.1.
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: $\quad \mathbf{v}_1 \leftarrow \frac{\mathbf{v}_1}{L^2\texttt{-norm}(\mathbf{v}_1)}$ ▷ Be sure the vector is normalized
11: $\quad \mathbf{y}_{\text{hit}} = \mathbf{y}_{\text{in}} + t^*(\mathbf{y}_{\text{out}} - \mathbf{y}_{\text{in}})$
12: $\quad \mathbf{y}_{\text{in}} = \mathbf{y}_{\text{in}} + \alpha t^*(\mathbf{y}_{\text{out}} - \mathbf{y}_{\text{in}})$ ▷ 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** $\mathbf{b}^\top \mathbf{y}_{\text{out}} - \mathbf{b}^\top \mathbf{y}_{\text{hit}} < 10^{-\texttt{precision}}$ ▷ or equality on `precision` most significant digits

$i \in [1..k]$ contains too many huge values compared to the rest (meaning the program is ill-conditioned). A full solver needs to address many prosaic aspects of this nature and we can not even list them all in a paper of appropriate length.

Observation 4.B. *(optional presolve) In the very beginning, we may apply a standard* `Cutting-Planes` *and use the separation to obtain a more refined (and bounded) outer approximation. The polyhedron* $\mathscr{P}_0 \supsetneq \mathscr{P}$ *constructed in Line 2 according to Section 4.2.2 may lead to an unbounded* $\max\{\mathbf{b}^\top \mathbf{y} : \mathbf{y} \in \mathscr{P}_0\}$. *To overcome this issue, we place an artificial box over During this phase, we put an artificial box over all variables, limiting each* y_i *with* $i \in [1..k]$ *to the interval [-10000,10000]. By putting this box, the resulting outer LP will not contain unbounded rays. Yet the outer solution* $\mathbf{y}_{\text{out}}$ *can touch the artificial box. Each* `Cutting-Planes` *iteration may reduce the number of components of* $\mathbf{y}_{\text{out}}$ *that touch the box. This presolve is stopped if 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 keep 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 can not be separated, it is clear that we can not declare it optimal. In such case, we increase the box side length by 10: 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, `Proj-Cut-Pl` solves 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, because this projection advances along an unbounded ray. We conclude the same if the objective value of the inner point reaches 10^{10}.

4.2.3.2 Theoretical computational comparisons with existing algorithms

We think its possible to anticipate a few speed conclusions without needing full numerical results. One ingredient in the overall approach is computing minimum eigenvalues. The software available during this decade (2020-2030) is very fast for this kind of matrix operations even for large values of $n > 2000$; this will also be useful inside the projection algorithm to be described next. After some preliminary tests, we concluded that `Matlab` is one of the best numeric computing environments for such tasks, and so, we used it to implement `Proj-Cut-Pl`. Perhaps the overall numerical results might have been (much) worse in previous decades.

We will often refer to the `Mosek` (Matlab) implementation of one of the most popular approaches for SDP optimization, namely Interior Point Methods (IPMs); many consider `Mosek` very fast and reliable. IPMs can offer everything one can desire in theory, but may be too slow for $n \geq 2000$, because they would have to solve huge Newton systems. Each iteration may easily end up in requiring computing a positive definite (Schur) matrix of size $k \times k$ in which the calculation of each element is rather long (it involves multiplying several matrices), introducing an important overall computational bottleneck. The whole operation may require $O(k^2n^2 + kn^3)$ operations, see, *e.g.*, [21, p. 66], [52, p. 26], or point 1 at the end of [54, §4]. To my knowledge, many man-hours have been devoted to accelerating such operations in IPM software.[7] One may keep that in mind when comparing to `Proj-Cut-Pl`, a software that was not (yet) so elaborately tuned and that does not exploit all possible engineering acceleration ideas.

We will also compare to `SeDuMi` using the Matlab code available at `github.com/sqlp/sedumi`. The first article to provide an elaborate discussion of this software [58] describes it as an IPM that relies on a self-dual embedding technique, which provides an elegant way for dealing with (in)feasibilities. The idea is to reformulate the initial conic problem (adding a slack variable) so as to transform its feasible area into a cone that is equal to its own dual.

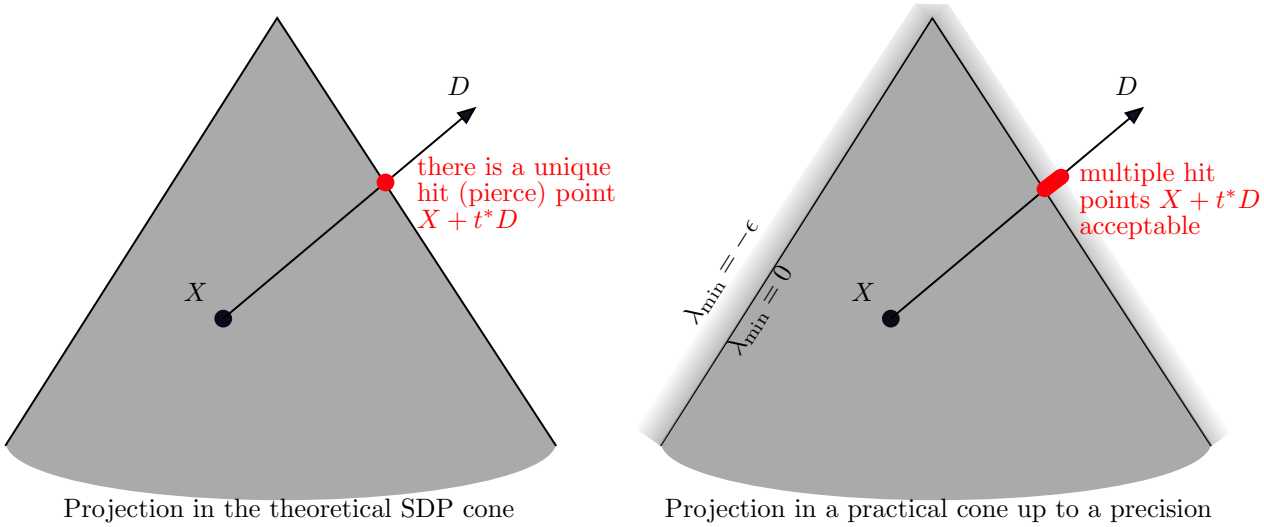
Another very successful approach is the `ConicBundle` method [22, 20] that reformulates the semidefinite program as an eigenvalue optimization problem, which is then solved by a subgradient method. This eigenvalue optimization problem arises as follows. The `ConicBundle` requires a constant trace constraint in the dual (4.E). Consider the expression $\mathcal{A}^\top \mathbf{y} = \sum_{i=1}^{k} A_i y_i$ from (4.F.2) and suppose we have $A_k = I_n$ and $b_k > 0$. The optimal way to enforce $A_0 - \mathcal{A}^\top \mathbf{y} \succeq \mathbf{0}$ and maximize the objective is to set $y_k = \lambda_{\min}\left(A_0 - \sum_{i=1}^{k-1} A_i y_i\right)$. Separating this variable from the other $k-1$ variables, one has to maximize this minimum eigenvalue function over the decision variables $y_1, y_2, \ldots, y_{k-1}$. Such methods have to maintain a cutting model that underestimates the concave non-smooth minimum eigenvalue function. The use of the term $A_k = I_n$ with $b_k > 0$ reduces to imposing $\texttt{trace}(S) = b_k$ in (4.E).

We will also refer to the following implementations, usually considered as black-box software packages: `CSDP` (an IPM; we used the JuMP interface `github.com/jump-dev/CSDP.jl` for Julia), `Clarabel`(an IPM; we used the JuMP interface `github.com/oxfordcontrol/Clarabel.jl`), `Loraine`(an IPM; we used the Matlab code `github.com/kocvara/Loraine.m`, but we also tried the JuMP package), `ClusteredLowRankSolver` (an IPM; we used the Julia library `github.com/nanleij/ClusteredLowRankSolver.jl`),

`Proj-Cut-Pl` was deliberately designed to use a less heavy (computing) machinery than IPMs; its theory is even more lightweight than that of the `ConicBundle` or of most other SDP algorithms and software. This makes it more naturally amenable to integrate the robust programming logic from Section 3. What happens if the coefficients of the given nominal linear inequalities (4.F.3) vary according to robust rules and produce prohibitively-many robust linear inequalities? The projection algorithm for the associated robust LP (ignoring the SDP part) was already presented in Section 3.2.2. To make `Proj-Cut-Pl` work for such SDP program with robust linear constraints, it is enough to call two projection algorithms, for two different sub-problems, one for (4.F.3) and one for (4.F.4), *i.e.*, one for the LP part and one for the SDP part.

More generally, it is rather easy to adapt `Proj-Cut-Pl` to re-optimization tasks like the following: after solving a (4.F.1)-(4.F.4) program, solve the same program again after adding a new linear constraint, as may needed be needed by a `Branch-and-bound`. We are not aware of other SDP algorithms that can easily adapt to address such questions, even if work on warm-starting IPM does exist.

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



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

Figure 4.B: Projection in an ideal world with a hard line of SDP separation (left) and a practical world with a blurred 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 will complicate the projection sub-problem compared to the polyhedral case, since it may allow multiple pierce (hit) points more-or-less acceptable in practice.

4.3 An SDP projection algorithm in ideal exact arithmetic

Most SDP algorithms need to take into account the following problem. It may be numerically very hard to decide if $\lambda_{\min}(S) > 0$ for some matrices S at the edge of feasibility. If $\lambda_{\min}(S) = 2^{-99}$, most eigenvalue algorithms available today will be unable to certify if S is SDP or not. 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 that 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 all 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 transition between the interior and the exterior of the SDP cone, as described in Figure 4.B. This phenomenon will complicate the projection algorithm. We first address this ideal case in exact arithmetics in the current Section 4.3. We will move to a very practical (and sometimes faster) algorithm using tolerances in Section 4.4. For now, we distinguish four cases in exact arithmetics addressed below by Sections 4.3.1, 4.3.2, 4.3.3 and 4.3.4, respectively.

4.3.1 Case (a): full-rank positive-definite $X \succ 0$

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` is able to compute it much more rapidly in practice because K is triangular. The first two equations below are simply the same by above substitution. The last two inequalities are equivalent because the involved matrices are congruent, see Prop 4.C below.

$$\max\{t : X + tD \succeq \mathbf{0}\} \tag{4.L.1}$$
$$\max\left\{t : KI_nK^\top + tKD'K^\top \succeq \mathbf{0}\right\} \tag{4.L.2}$$
$$\max\{t : I_n + tD' \succeq \mathbf{0}\}. \tag{4.L.3}$$

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 (4.L.3).

2. According to Lines 13-15 of Alg. 3, 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 to 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 the eigenvalue of $\mathbf{u}$ with regards to D' is $\lambda_{\min}(D')$, we can write $(I_n + t^* D')^\top \mathbf{u} = \mathbf{u} + t^* \lambda_{\min}(D')\mathbf{u}$, 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 we can not advance by some ϵ beyond t^*: since $\mathbf{u}^\top D' \mathbf{u} = \lambda_{\min}(D') < 0$, we have $\mathbf{v}^\top K D' K^\top \mathbf{v} < 0 \implies \mathbf{v}^\top D \mathbf{v} < 0$, which leads to $\mathbf{v}^\top (X + (t^* + \epsilon)D)\mathbf{v} < 0$ for any $\epsilon > 0$.

We used the following property to prove the equivalence of (4.L.2) and (4.L.3).

Property 4.C. *(congruent matrices) Two matrices X and X' are congruent if there is some non-singular M such that $X' = MXM^\top$. It is well known (see, for example, [38, Prop 1.2.3.]) that two congruent matrices have the same SDP status: $X \succeq \mathbf{0} \iff X' \succeq \mathbf{0}$.*

As mentioned in point 1 of Section 4.2.3, it may be practically useful to find a second-hit pierce point $X + t_2^* D$ with $t_2^* \geq t^*$ associated to a kind of second-hit vector $\mathbf{v}_2$ such that $\mathbf{v}_2^\top X \mathbf{v} = 0$. This $\mathbf{v}_2$ has to satisfy $(X + t_2^* D) \bullet \mathbf{v}_2 \mathbf{v}_2^\top = 0$ and $D \bullet \mathbf{v}_2 \mathbf{v}_2^\top < 0$. 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. First, $(X + t_2^* D) \bullet \mathbf{v}_2 \mathbf{v}_2^\top = \mathbf{v}_2^\top (KK^\top + t_2^* K D' K^\top)\mathbf{v}_2 = \mathbf{u}_2^\top \mathbf{u}_2 + t_2^* \cdot \mathbf{u}_2^\top D' \mathbf{u}_2 = 1 + t_2^* \lambda^2_{\min}(D') = 0$, which is true for the considered t_2^*. Secondly, we can calculate $\mathbf{v}_2^\top X \mathbf{v} = \mathbf{v}_2^\top K K^\top \mathbf{v} = \mathbf{u}_2^\top \mathbf{u} = 0$, since the eigenvectors of D' are orthonormal.

A final note: if K contain 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.

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

We move to a generalized version of above case, starting with a generalized version of Prop. 4.C.

Property 4.D. *(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 $\mathbf{v}'^\top M X M^\top \mathbf{v}' < 0$, 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 $\mathbf{v}'^\top M X M^\top \mathbf{v}' < 0$, equivalent to $\mathbf{v}'^\top X' \mathbf{v}' < 0$, contradiction. □

Property 4.E. *(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; thus, $X\mathbf{d} = 0 \implies D\mathbf{d} = 0 \; \forall \mathbf{d} \in \mathbb{R}^n$.*

Unlike in Section 4.3.1, we hereafter consider X to be 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.

Applying the LDL decomposition, we write $X = L\mathtt{diag}(\mathbf{p})L^\top$ with $\mathbf{p} \geq \mathbf{0}_n$, where $L \in \mathbb{R}^{n \times n}$ is lower triangular. All $i \in [1..n]$ such that $p_i > 0$ constitute together a number 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.

Observation 4.F. *(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'. 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 because 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 4.3.3. The curent case leads to a reduced-size version of (4.L.3) working in the space of $c \times c$ matrices:

$$\max \{t : I_c + tD' \succeq \mathbf{0}\}. \tag{4.M}$$

And the maximum value of t is here: $t^* = -\frac{1}{\lambda_{\min}(D')}$, or $t^* = \infty$ if $\lambda_{\min}(D') \geq 0$.

We finally determine a first-hit vector $\mathbf{v}_c \in \mathbb{R}^c$ over the core rows and columns exactly like in the case of non-singular matrices from Section 4.3.1. To lift $\mathbf{v}_c$ to a 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 like towards the end of Section 4.3.1.

4.3.3 Case (c): D has some components outside the image of X but with no impact on 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 named active (independent) columns. We first apply the QR decomposition on matrix $[K_{nc}\ D] \in \mathbb{R}^{n \times (c+n)}$ and write $[K_{nc}\ D] = QR$, 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. In fact, 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. Let us focus on the first c columns of R. Since K_{nc} is full rank, the matrix R restricted to the first c rows and columns is full rank and so are the first c columns of Q. Since this $c \times c$ top-left block of R is upper triangular, it 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 means 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. The QR decomposition algorithm might have very well ignored constructing non-active 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 non-active; otherwise, we say column $c+i$ is active.

Let N denote the matrix Q restricted to its $m > 0$ active columns detected above. If N were empty with $m = 0$, we would have been in the case of Section 4.3.2 – 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 $X = K_{nc} K_{nc}^\top$.

$$X = \underbrace{\begin{bmatrix} K_{nc} & N \end{bmatrix}}_{c\ +m} \begin{bmatrix} I_c & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} K_{nc}^\top \\ N^\top \end{bmatrix} \tag{4.N}$$

Next, we will show we can solve the system below in the variables F, G and E, to obtain a decomposition of D.

$$D = \underbrace{\begin{bmatrix} K_{nc} & N \end{bmatrix}}_{c\ +\ m} \underbrace{\begin{bmatrix} F & G^\top \\ G & E \end{bmatrix}}_{D_{c+m}} \begin{bmatrix} K_{nc}^\top \\ N^\top \end{bmatrix}. \tag{4.O}$$

The hardest computational task is computing D_{c+m}, 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 (4.O). 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 R_{cc} is the $c \times c$ top-left part 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 $[K_{nc}\ D] = [K_{nc}\ Q_{n,c+1..c+n}] \begin{bmatrix} R_{cc}^{-1} & \mathbf{0} \\ \mathbf{0} & I_n \end{bmatrix} R$. Let us restrict the right matrix multiplication to its last n columns and denote the result of this restricted multiplication by $T \in \mathbb{R}^{(n+c)\times n}$. We can write $T = \begin{bmatrix} R_{cc}^{-1} & \mathbf{0} \\ \mathbf{0} & I_n \end{bmatrix} R_{c+n,c+1..c+n}$. If we also restrict $[K_{nc}\ D]$ to its last n columns (*i.e.*, to D), the above QR factorization becomes:

$$D = [K_{nc}\ Q_{n,c+1..c+n}]T. \tag{4.P}$$

We can restrict $Q_{n,c+1..c+n}$ to its m active columns identified above, *i.e.*, to N, because recall that a non-active 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 (4.P) to $[K_{nc}\ N]$ by removing the non-active columns of Q. The associated T factor in (4.P) is also reduced to some $\overline{T}$ by removing its null rows that come from the null rows of R. We can re-write (4.P) as $D = [K_{nc}\ N] \cdot \overline{T}$. We finally determine the sought D_{c+m} from (4.O) by solving $D_{c+m} \begin{bmatrix} K_{nc}^\top \\ N^\top \end{bmatrix} = \overline{T}$.[8] We recall D_{c+m} has the form:

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

The case described by this sub-section is characterized by finding $G = \mathbf{0}$ when computing the above decomposition. Expanding (4.O) into $D = K_{nc}FK_{nc}^\top + NEN^\top + K_{nc}G^\top N^\top + NGK_{nc}^\top$ means that D has actually the form $D = K_{nc}FK_{nc}^\top + NEN^\top$ because the terms containing $G = \mathbf{0}$ vanish. Since N is orthogonal to K_{nc} by (the QR decomposition) construction, if we left-multiply this with $N^\top$ and right-multiply with N, we obtain $N^\top DN = E$. This may be useful later.

For now, applying the congruence expansion Property 4.D on (4.N) and (4.O), the SDP status of $X + t \cdot D$ is the same as that of

$$\begin{bmatrix} I_c & 0 \\ 0 & 0 \end{bmatrix} + t \cdot \begin{bmatrix} F & \mathbf{0} \\ \mathbf{0} & E \end{bmatrix}. \tag{4.Q}$$

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:

$E \succeq \mathbf{0}$ The above projection reduces to determining $\max\{t : I_c + tF \succeq \mathbf{0}\}$, for which it is enough to apply the approach for the simplest case of Section 4.3.1 (in a smaller world of $c \times c$ matrices). Since $N^\top DN = E \in \mathbb{R}^{m\times m}$ as developed above, $E \succeq \mathbf{0}$ means D is SDP over the image of N, namely over $\{\mathbf{d} = N\mathbf{d}_m : \mathbf{d}_m \in \mathbb{R}^m\}$. Vectors $\mathbf{d} \in \mathbb{R}^n$ that are outside the image of both K_{nc} and N have to yield $D \bullet \mathbf{d}\mathbf{d}^\top = 0$ using (4.O). This means D is actually SDP over the null space of X or K_{nc} *i.e.*, $D \bullet \mathbf{d}\mathbf{d}^\top \geq 0 \forall \mathbf{d} \in \texttt{null}(X) = \texttt{null}(K_{nc})$.

$E \not\succeq \mathbf{0}$ The sought t^* is 0. It is straightforward to see in (4.Q) 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. We can find some $\mathbf{d}$ in the image of N such that $D \bullet \mathbf{d}\mathbf{d}^\top < 0$.

[8] 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 if we apply (4.P) after replacing all non-active columns of Q with zeros, we obtain $\mathbf{u}^\top D = \mathbf{0}$, equivalent to $D\mathbf{u} = \mathbf{0}$. Multiplying (4.P) this time at right with $\mathbf{u}$, we obtain $D\mathbf{u} = \mathbf{0} = [K_{nc}\ Q_{n,c+1..c+n}]T\mathbf{u}$. The contradiction is that we can not have $T\mathbf{u} \neq \mathbf{0}$ in the last equality, because $[K_{nc}\ Q_{n,c+1..c+n}]$ was constructed to be full rank.

4.3.4 Case (d): the most general case

If all above cases fail, we still apply the logic of (4.Q). We assume $E \succeq \mathbf{0}$, because otherwise it is clear that t^* is zero. We also consider we determined above a non-zero G (otherwise we are in the previous case); thus, we have to find the maximum t for which the following generalization of (4.Q) remains in the SDP cone:

$$\underbrace{\begin{bmatrix} I_c & 0 \\ 0 & 0 \end{bmatrix}}_{X_{c+m} \in \mathbb{R}^{(c+m)\times(c+m)}} + t \cdot \underbrace{\begin{bmatrix} F & G \\ G & E \end{bmatrix}}_{D_{c+m} \in \mathbb{R}^{(c+m)\times(c+m)}} \succeq \mathbf{0}. \tag{4.R}$$

We first present a tricky case, in which we find no hard line of separation associated to 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_{c+m} is zero while its non-diagonal element (i,j) is non-zero, then any $t > 0$ leads to $X_{c+m} + tD_{c+m} \not\succeq \mathbf{0}$. We return $t^* = 0$, but there is no hit vector $\mathbf{v}$ such that $(X_{c+m} + 0D_{c+m}) \bullet \mathbf{v}\mathbf{v}^\top = 0$ and $(X_{c+m} + tD_{c+m}) \bullet \mathbf{v}\mathbf{v}^\top < 0$ 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 here fail, we solve the projection in two steps: (1) find a small t_1 such that $X_{c+m} + t_1 D_{c+m} \succeq \mathbf{0}$ and (2) solve the projection $(X_{c+m} + t_1 D_{c+m}) \to D_{c+m}$. In this second step, D_{c+m} belongs to the image of $X_{c+m} + t_1 D_{c+m}$ (Prop 4.E satisfied) and we can use the techniques from Sections 4.3.1 and 4.3.2. However, finding t_1 may require a limited number of (costly) repeated separations.

4.4 A practical projection algorithm with tolerance windows

Sections 4.3.1 to 4.3.4 presented a projection algorithm that may work in an ideal world in which all mathematical operations can be calculated in exact arithmetic. As described there, the difficult projections arise when $\lambda_{\min}(X) = 0$. Practice introduces further complication: for the matrices X generated by `Proj-Cut-Pl`, all equalities like $\lambda_{\min}(X) = 0$ are most frequently only satisfied within a certain numerical precision. Also referring to Figure 4.B, we introduce two prosaic but very practical tolerance parameters:

$\epsilon_{\texttt{SDP}}$ The "inner points" to be generated can be matrices X such that $\lambda_{\min}(X) > -\epsilon_{\texttt{SDP}}$. We often use $\epsilon_{\texttt{SDP}} = 10^{-6}$, like most 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 4.3.1. We usually set $\epsilon_{\texttt{proj}} = 0.1 \cdot \epsilon_{\texttt{SDP}} = 10^{-5}$.

The fastest approach for solving the projection from Section 4.3.1 relies on decomposing $X = KK^T$. It works only when X is strictly inside the SDP cone, *i.e.*, when $\lambda_{\min}(X) > 0$. However, in practice, if $\lambda_{\min}$ is very close to zero, this approach may be too error prone, because some prohibitively-large values may arise in K^{-1}; recall from Section 4.3.1 that we need to find the first-hit vector $\mathbf{v}$ by solving $K^\top \mathbf{v} = \mathbf{u}$. This is why in practice we apply this fast approach when $\lambda_{\min}(X) > \epsilon_{\texttt{proj}}$ instead of $\lambda_{\min}(X) > 0$, where $\epsilon_{\texttt{proj}}$ needs to be even large than $\epsilon_{\texttt{SDP}}$, hence the above $\epsilon_{\texttt{proj}}$ value.

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

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

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 blurred line of distinction between SDP and non-SDP matrices, see the phenomenon in the right part of Figure 4.B. The (unavoidable) infinitesimal inaccuracy of the LP solver certainly introduces some spurious data that complicates the projection sub-problem.

Figure 4.C illustrates the process for solving (4.S). Seeking to reduce the projection sub-problem to a form that can be solved with the fast algorithm from Section 4.3.1, we first raise the graph of the function $\lambda_{\min}(X + tD)$ 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 from the red one by adding $\epsilon_{\texttt{proj}} - \lambda_{\min}(X)$. We then apply the fast algorithm from Section 4.3.1 to project from $\widehat{X}$ towards D, resulting in an overestimated t^*_+. By subtracting $\epsilon_{\texttt{proj}} - \lambda_{\min}(X)$ to get back on the red curve,

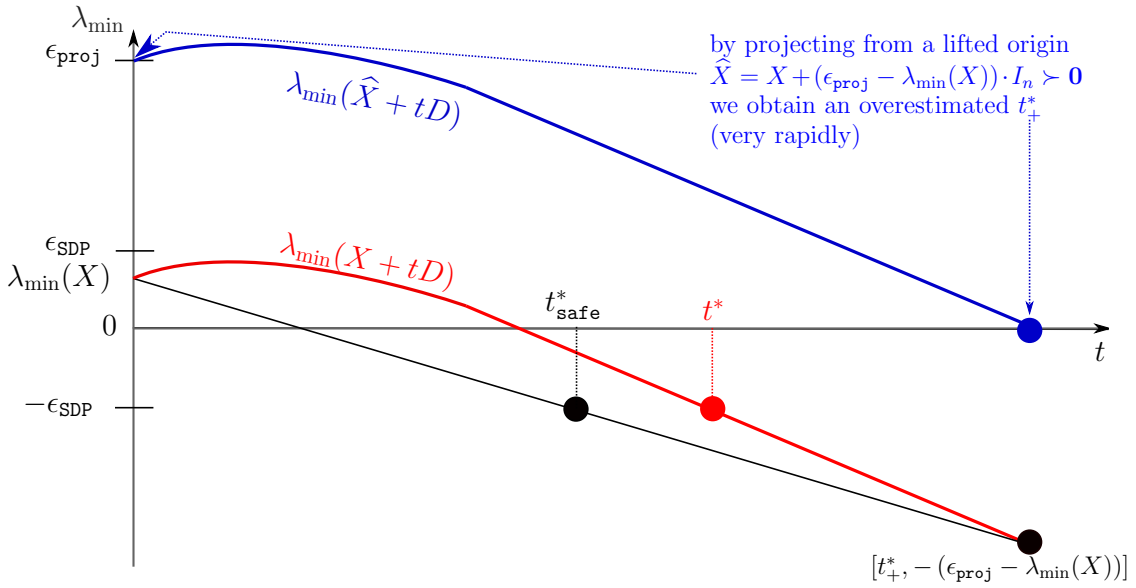


Figure 4.C: The sought t^* is marked in red, knowing that we solve the projection with the $\epsilon_{\texttt{SDP}}$ tolerance window from (4.S). Since $\widehat{X} = X + \big(\epsilon_{\texttt{proj}} - \lambda_{\min}(X)\big) \cdot I_n$, the curve in blue of the function $\lambda_{\min}(\widehat{X} +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

i.e., the point $[t^*_+, 0]$ is translated to $[t^*_+, -(\epsilon_{\texttt{proj}} - \lambda_{\min}(X))]$, see the black disk in the bottom-right part. This means that $\lambda_{\min}(X + t^*_+ D) = -(\epsilon_{\texttt{proj}} - \lambda_{\min}(X))$.

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

$$t^*_{\texttt{safe}} \leq t^* \leq t^*_+. \tag{4.T}$$

A second iteration can repeat the process after replacing $X \leftarrow X + t^*_{\texttt{safe}} D$, since the sought t^* must be at the right of $t^*_{\texttt{safe}}$. You notice in Figure 4.C an useful property: the red graph at right of $t^*_{\texttt{safe}}$ is actually a straight line. This means that the proposed approach will find at this second iteration a second $t^*_{\texttt{safe}}$ that is equal to the sought t^*. When this is not the case, the same process could be repeated multiple iterations, until (4.T) becomes $t^*_{\texttt{safe}} = t^* = t^*_+$. In fact, (4.T) could have been an equality from the very first iteration if $\lambda_{\min}(X + tD)$ were linear.

In practice, we stick to only one iteration and the projection algorithm returns $t^*_{\texttt{safe}}$ to `Proj-Cut-Pl`. The only disadvantage of this approach is that it is more difficult the find the appropriate hit-vector. Recall we actually applied the algorithm from Section 4.3.1 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.T) is an equality, we will have $(\widehat{X} + t^*_{\texttt{safe}} D) \bullet \mathbf{v}\mathbf{v}^\top > 0$. In short, we return to `Proj-Cut-Pl` a underestimated step length $t^*_{\texttt{safe}}$ and a first-hit constraint corresponding to an overestimated t^*_+ and a lifted X, generating imprecision: the first-hit constraint can be a bit distant from the real pierce point. As long as $t^*_+ - t^*_{\texttt{safe}}$ has the same magnitude as the considered tolerances, this is a good-enough compromise and `Proj-Cut-Pl` can proceed as usually. A very fast but not perfectly-exact projection algorithm may be better in practice than an exact-but-slow projection algorithm.

4.5 Computational aspects compared to the existing literature

Generalizing the separation, the SDP projection sub-problem is more difficult. The separation simply reduces to finding the minimum eigenvalue of a matrix $A_0 - \mathcal{A}^\top \mathbf{y}$ that is generally not SDP. This is a very well-studied question: many software libraries (refined over decades) can solve it very rapidly. Despite this, a canonical `Cutting-Planes` may need too many iterations and it may have difficulties in achieving an impressing success for large-scale SDP programs. Such algorithms based on repeated separations are a rare sight in the literature; they are often considered not very efficient. As [13, § 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". Yet a few implementations do exist; we aware of [26, 51] that use $n \leq 100$ and operate on the dual (4.E). The PhD thesis [52] and two associated articles [53, 54] established one of the first unifying frameworks offering multiple valuable insights into the overall question, including the idea to work on the primal (4.D) instead of (4.E).[9] The more recent study [9] provides interesting (theoretical approximation) results about different polyhedral approximations of the semidefinite cone.

Many ideas in the projection algorithms presented above may seem to arise out of the blue. But they were designed following a conscious effort to reduce the computational cost as much as possible, striving to solving projections $X \to D$ with $n \geq 1000$. Most of previous work on related topics was not so motivated (or directed) by such speed considerations. We describe below some alternative methods, often comparing their computational cost with the presented approach.

Restoring definiteness [23, 57] 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 it's indefinite in practice because of some noise, finding $t^* = \max\{t : X + t \cdot D \succeq \mathbf{0}\}$ may be seen as kind of repairing $X + D$. When seen through such lenses, the projection sub-problem may be useful beyond solving SDP programs. Citing [23, § 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]."

Three approaches have been described in the above study. The first one is a bisection method with similarities to the ideas from Section 4.4. Considering 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 $t_r = 0.5 \cdot (t_l + t_r)$. This reduces to a form of repeated separation or binary search that iteratively evaluates f in points like t_l or t_r that get closer and closer. For a faster convergence, a second approach consists of using a standard Newton's method to find the root of f. This leads to generating a sequence of overestimates of t^* of the form $1 = t_0 > t_1 > t_2 \ldots t_\iota = t^*$. In our case, we prefer to stick to one iteration and obtain an interval that cover t^*, since we aim at reducing the computational burden (almost) at all costs.

The third approach from the above study relies on the notion 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 $(X + \lambda D)\mathbf{v} = \mathbf{0}$, meaning that one of the eigenvalues of $X + tD$ needs to be zero. In this area dating back to the 1960s [14, 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 sought t^* thus needs to be a generalized eigenvalue of $(X, M) = (X, -D)$. But for finding t^*, generalized eigenvalues λ such that $\lambda_{\min}(X + \lambda D) < 0$ are of no interest. Many generalized eigen-pairs $(\lambda, \mathbf{v})$ with $\lambda > t^*$ may constitute generalized eigenvalues such that $(X + \lambda D)\mathbf{v} = \mathbf{0}$. We will see (Example 4.G below) that we actually seek a generalized eigenvalue t^* that satisfies a very particular condition: t^* is the smallest real non-negative generalized eigenvalue of $(X, M) = (X, -D)$ associated to an eigenvector $\mathbf{v}$ such that $D \bullet \mathbf{v}\mathbf{v}^\top < 0$. A similar but weaker condition (smallest non-zero eigenvalue for a generalized eigenvalue problem) appeared in a question asked on *Matlab Answers*, but the replies are not satisfying for our above goal.[10]

Many programming languages (including `Matlab` or `Julia`) 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 `Proj-Cut-Pl` easier to implement, but it is by far too computationally expensive. This may even involve considering working with complex generalized eigenvalues even when X and D are symmetric (see example below). However, it's absolutely natural to expect that computing a generalized eigendecomposition takes more time than a standard eigendecomposition. The latter operation was dismissed from the very beginning of this work as it is far too computationally expensive.

Example 4.G. *Any complex number can be a generalized eigenvalue for X and D below.*

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

[9] The associated semi-infinite LP will thus have k decisions variables while working with the dual requires $\frac{n \cdot (n-1)}{2}$ variables. It's only in the rare cases when $k > \frac{n \cdot (n-1)}{2}$ that it may be useful to design a `Proj-Cut-Pl` on the dual.

[10] fr.mathworks.com/matlabcentral/answers/397765-smallest-non-zero-eigenvalue-for-a-generalized-eigenvalue-problem

It's easy to see we have $t^* = 0.5$ *and that* $(0.5, [1\ 1]^\top)$ *is a generalized eigen-pair for* $(X, -D)$*. Yet, if we take* $\mathbf{u} = [1\ -1]^\top$*, we obtain* $\mathbf{0} = X\mathbf{u} = \lambda \cdot -D\mathbf{u}$ *for absolutely any* λ*, complex or real. We thus see a phenomenon that does not exist for standard eigenvalues: any number* λ *can be a generalized eigenvalue associated to a non-zero eigenvector* $\mathbf{u}$*. This explains the condition mentioned two paragraphs above: we seek the smallest real non-negative eigenvalue associated to an eigenvector* $\mathbf{v}$ *such that* $D \bullet \mathbf{v}\mathbf{v}^\top < 0$*. The above* $\mathbf{u}$ *does not satisfy this last condition.*

Since we need such a 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 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. I must confess that deepest algebra notions behind such algorithms (a field in itself) is still beyond my understanding and outside the scope of this thesis. Yet it is clear that the underlying theory does overlap some ideas used in the design of the presented projection algorithms.

The idea of exploiting congruence relations to show that (4.L.1), (4.L.2) and (4.L.3) are equivalent (Section 4.3.1) was certainly considered before in the context of generalized eigenvalues. This technique is pretty straightforward as long as the input matrices are invertible and symmetric. Based on [35, §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 $M_2 = EM_1E^{-1}$. The eigenvalues of two equivalent pencils are the same and the eigenvectors are related according to a procedure similar to the one developed at point 2 from Section 4.3.1. Similar ideas were explored in Section 5.2 "Transformation to Standard Problem" of the book chapter [18].

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. I extract one idea from [15, § 7.2]. We construct the eigendecomposition $X = P\mathtt{diag}(\lambda_1,\ \lambda_2,\ \ldots\lambda_n)P^\top$, recalling from (4.B) that matrix P is an orthonormal factor and $\lambda_i \geq 0\ \forall i \in [1..n]$ since $X \succeq \mathbf{0}$. Writing $Q = P\mathtt{diag}(\sqrt{\lambda_1},\ \sqrt{\lambda_2},\ \ldots\sqrt{\lambda_n})$, if X happens to be full-rank such that $X \succ \mathbf{0}$, we obtain that $X = QQ^\top$ leads to $Q^{-1}X(Q^{-1})^\top = I_n$. We now eigendecompose $Q^{-1}D(Q^{-1})^\top = RD_{\mathtt{diag}}R^\top$, where $D_{\mathtt{diag}}$ is diagonal and R is an orthonormal factor. Writing $S = R^{-1}Q^{-1}$, we obtain $SXS^\top = R^{-1}(R^{-1})^\top = I_n$ and $SDS^\top = D_{\mathtt{diag}}$. According to Prop. 4.C, projecting $X \to D$ reduces to the trivial projection $I_n \to D_{\mathtt{diag}}$.

An approach that uses only one eigendecomposition for the same goal as above was described by [24, p. 550]. Let us introduce the idea for the (non-singular pencil) case where there is some $t > 0$ such that $X + tD \succ \mathbf{0}$: we apply the Cholesky decomposition $X + tD = KK^\top$ and write

$$K^{-1}\left(X + tD\right)K^{\top -1} = I_n. \tag{4.U}$$

Constructing the eigendecomposition (4.B) of $K^{-1}DK^{\top -1}$ with an orthonormal $P \in \mathbb{R}^{n\times n}$ factor, we obtain that $PK^{-1}DK^{\top -1}P^\top = D_{\mathtt{diag}}$ is diagonal. But by using (4.U), we conclude that $PK^{-1}XK^{\top -1}P^\top = X_{\mathtt{diag}}$ is also diagonal. Using the equivalence between pencils discussed above, the question reduces to projecting $X_{\mathtt{diag}} \to D_{\mathtt{diag}}$. The approach can be generalized to the case in which X and D have a common 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 $U_\perp^\top(X + tD)U_\perp \succ \mathbf{0}$. Similar to the non-singular pencil case above, we can compute non-singular $R \in \mathbb{R}^{(n-m)\times(n-m)}$ such that both $R^\top U_\perp^\top XU_\perp R = X_{\mathtt{diag}}$ and $R^\top U_\perp^\top DU_\perp R = D_{\mathtt{diag}}$ are diagonal. Exploiting $XU = DU = \mathbf{0}$, we can extend these expressions to obtain $\left[\begin{smallmatrix} R & \mathbf{0} \\ \mathbf{0} & I_m \end{smallmatrix}\right]^\top [U_\perp U]^\top X [U_\perp U] \left[\begin{smallmatrix} R & \mathbf{0} \\ \mathbf{0} & I_m \end{smallmatrix}\right] = \left[\begin{smallmatrix} X_{\mathtt{diag}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{smallmatrix}\right]$, and, similarly, $\left[\begin{smallmatrix} R & \mathbf{0} \\ \mathbf{0} & I_m \end{smallmatrix}\right]^\top [U_\perp U]^\top D [U_\perp U] \left[\begin{smallmatrix} R & \mathbf{0} \\ \mathbf{0} & I_m \end{smallmatrix}\right] = \left[\begin{smallmatrix} D_{\mathtt{diag}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{smallmatrix}\right]$. Using again the above equivalence, the sought projection reduces to to projecting $X_{\mathtt{diag}} \to D_{\mathtt{diag}}$.

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 $A - \lambda B \succeq \mathbf{0}$ [37, (2.2)], see also [24, (7)]. However, most of these studies are devoted to finding all eigenvalues and the computational speed is not their primary concern.

4.6 Numerical results

When comparing the performances of different software, one should test for oneself, according to his or her needs. Only an external independent review covering many classes of instances can be very conclusive in a

broad sense. I'll clarify from the outset the cases in which `Proj-Cut-Pl` is not competitive.

1. If it is very hard to find a single feasible solution for your instance, do not bother (yet) with `Proj-Cut-Pl`, since it was not designed to detect infeasibilities. For the sake of completeness, it does try to solve all kinds of instances. `Proj-Cut-Pl` first tries to find a feasible solution using the two stages from Section 4.2.1. If this fails, it adds an artificial $A_{k+1} = -I_n$ with a penalty in the objective, generating artificial feasibility. Future work may include other methods to recover feasibility, like Infeasible Interior Point Methods or self-dual embedding techniques.

2. When k is rather large (*e.g.*, in the thousands) and the optimal $\mathbf{y}$ has very few zero components, the outer approximation and the LP solver may induce a too high slowdown for `Proj-Cut-Pl`. But if most components of $\mathbf{y}$ are zero at optimality, `Proj-Cut-Pl` may remain competitive. A Column Generation approach may be used in the future to (try to) make each LP optimization step optimize over only a subset of the $\mathbf{y}$ variables.

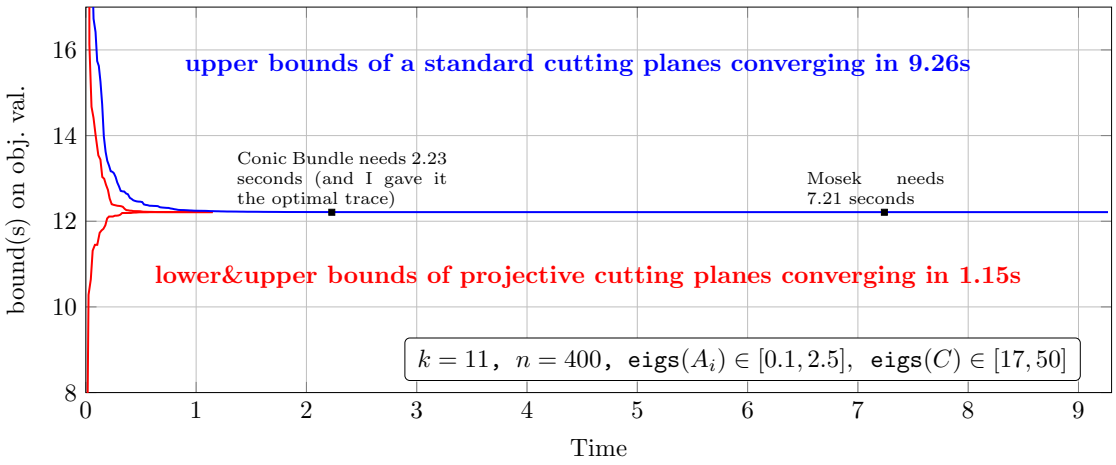
3. Many SDP solvers from the literature invested massive efforts to exploit sparsity, because sparsity may imply an important speed-up potential in their framework. This is not so much the case for `Proj-Cut-Pl` because many inner points of the form $A_0 - \sum_{i=1}^{k} A_i y_i$ may not be (so) sparse for a large k even if the input $A_0,\ A_1, \ldots A_k$ is sparse. While we did implement many sparsity features, this idea was not (yet) followed to an ultimate degree. Most public 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 this test bed is devoted to sparse instances.[11]

Most instances available on-line do satisfy one or two of the above criteria. Yet, my experience suggest that comparing SDP optimization algorithms may be tricky, even more so than comparing (integer) LP solvers. The performance of certain SDP software can change dramatically 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 `Proj-Cut-Pl`, because it is more difficult to find a good initial outer approximation (most of Section 4.2.2 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. I received by private communication a few atypical instances with a large initial LP, a very high $k > 20000$ and a very small n and the fastest solver is `Clarabel`, *i.e.*, one of the last in the ranking from Table 6. Thus, this section is not meant to show that `Proj-Cut-Pl` is superior to all other alternatives on the instances we present next. While we aim at being very competitive in speed, this work is not (only) devoted to competition.

4.6.1 A first run on an instance with strictly positive definite matrices

Let us present a first test on an instance with $A_0,\ A_1, \ldots, A_k \succ \mathbf{0}$, *i.e.*, all involved matrices have full rank (empty null space). To our knowledge, this case did not receive much attention in benchmarking SDP algorithms. But it may be interesting because such matrices are not a rare sight in many areas and various SDP programs may need to optimize over them. The sub-problem becomes easy to implement (Section 4.3.1 suffices) and very fast. Each input matrix was generated by applying the eigendecomposition (4.B) with a random orthogonal matrix $[\mathbf{v}_1\ \mathbf{v}_2\ \ldots \mathbf{v}_n]$; $\mathbf{b} = \mathbf{1}_k$. The non-negative eigenvalues are generated at random in the ranges mentioned in the legend of the figure. This first experiment confirms that `Proj-Cut-Pl` has a higher potential than a standard `Cutting-Planes`. 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.

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



4.6.2 Instances from existing benchmarks

We will address multiple solvers in the sequel, but for now it is enough to say that `Mosek` seems to be a very strong competitor on many comparisons we have carried out throughout this work. Since most instances from public benchmarks (see `plato.asu.edu/ftp/sparse_sdp.html` and references therein) are very sparse, `Proj-Cut-Pl` is not expected to compete with `Mosek` on such cases. We will see that the new method is more competitive on dense instances with a very large n and a low k; Interior Point Methods need to solve huge Newton systems in such cases.

Let us thus present a numerical test on benchmarks `buck` and `vibra`. While we can not compete with `Mosek` on the original very sparse format of these instances, let us introduce a procedure to make them dense. We first generate an ortho-normal matrix $M \in \mathbb{R}^{n\times n}$ and then transform the input matrices by replacing $A_i \leftarrow M^\top A_i M$ for all $i \in [0..k]$. Notice that the eigenvalues of $X = A_0 - \sum_{i=1}^k A_i y_i$ do not change after this substitution, because $X\mathbf{v} = \lambda \cdot \mathbf{v} \implies \left(M^\top X M\right)\left(M^\top \mathbf{v}\right) = M^\top X \mathbf{v} = \lambda \cdot \left(M^\top \mathbf{v}\right)$. In other words, the eigenvalues remain the same and each eigenvector $\mathbf{v}$ of the original optimal matrix X is translated into an eigenvector $M^\top \mathbf{v}$ of the new dense optimal matrix $MXM^\top$.[12] Thus, this transformation does not change the optimal solution (or the set of feasible solutions) $\mathbf{y}$ of the instance: it's an equivalent reformulation. The results are reported in Table 3.

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 3: Running times (seconds) of `Mosek` and `Proj-Cut-Pl` on eight instances from the literature, considering both their initial sparse expression and their densified variant. Column "LP cuts" reports the number of initial linear constraints of the instances, *i.e.*, the cardinal of $\mathscr{C}$ in (4.F.3).

If Table 3 does not provide the results of `Proj-Cut-Pl` on the original sparse instances, it is because it can not keep up the pace with `Mosek`: it needs hours for the biggest instances that `Mosek` solves in at most 3 minutes. Yet this table shows that the same instances become difficult for `Mosek` when they are densified.

[12] If the resulting instance still contains (too) many values close to zeros for whatever 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 to have 0.01 everywhere and 0.07 on the diagonal (so that its image is $\mathbb{R}^n$). This transformation does change the optimal solution of the instance.

In this latter case, `Proj-Cut-Pl` does represent a real competition for `Mosek`, and probably for many other solvers.

We next present results on the densified variants of instances `hand` and `foot`. To show a more general trend over multiple value of k, we also use procedure below to reduce the value of k using a compression factor $\texttt{k}_{\texttt{div}}$. The results are given in Table 4.

Observation 4.H. *(compressed instances) Given an initial instance and a compression factor* $\texttt{k}_{\texttt{div}}$*, we generate a new instance by replacing*

$$A_1 \leftarrow \sum_{i=1}^{\texttt{k}_{\texttt{div}}} A_i,$$

followed by $A_2 = \sum_{i=\texttt{k}_{\texttt{div}}+1}^{2\cdot\texttt{k}_{\texttt{div}}} A_i$*, or more generally* $A_{\ell+1} = \sum_{i=\ell\cdot\texttt{k}_{\texttt{div}}+1}^{(\ell+1)\cdot\texttt{k}_{\texttt{div}}} A_i$*. We also apply the corresponding transformation of* $\mathbf{b}$*, i.e.,* $b_{\ell+1} = \sum_{i=\ell\cdot\texttt{k}_{\texttt{div}}+1}^{(\ell+1)\cdot\texttt{k}_{\texttt{div}}} b_i$*. In fact, if* $(\ell+1)\cdot\texttt{k}_{\texttt{div}} > k$*, the range of* i *in above sums stops at* k*. Thus, the value of* k *evolves to* $\left\lceil \frac{k}{\texttt{k}_{\texttt{div}}} \right\rceil$.

Instance				Results		
	$\texttt{k}_{\texttt{div}}$	k	n	`Mosek` seconds	`Proj-Cut-Pl` seconds	`Proj-Cut-Pl` iterations
`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
`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

Table 4: Running times (seconds) of `Mosek` and `Proj-Cut-Pl` on the densified variant of instances `hand` and `foot` compressed as described by Obs. 4.H with the factor $\texttt{k}_{\texttt{div}}$ from Column 2 ($\texttt{k}_{\texttt{div}} = 1$ corresponds to the original instance).

Table 4 suggests that `Mosek` is faster than $\texttt{Proj} - \texttt{Cut} - Pl$ when both n and k are around (or bit larger) than 1000. This comes from the fact that modelling the outer approximation with linear constraints may require too many iterations. However, `Proj-Cut-Pl` becomes faster as we compress the instance to reduce k. The computational speed advantage of our algorithm is increasingly evident as the value of k decreases.

Corroborating runtime monitoring from other experiments, Table 4 also indicates that `Proj-Cut-Pl` requires much less memory than `Mosek`. A (generous) RAM amount of 128 Gigabytes seems insufficient for `Mosek` to carry out a single iteration for $n, k > 2000$, while `Proj-Cut-Pl` does report a duality gap under the same conditions. Namely, it can produce (even if after many hours) a primal and a dual solution with a gap below 1% for the densified uncompressed `foot` instance. To the best of my knowledge, a standard IPM can not report any intermediate partial information before fully converging; there is no `Mosek` feature in this sense, and so, `Mosek` can not report any primal or dual solution on this instance with the given resources.

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 above benchmark. We plan to extend Table 3 with instances `buck5` and `vibra5` as soon as we find a (super-)computer with at least 512 Gigabytes of RAM. More generally, `Proj-Cut-Pl` was implemented in `Matlab` version R2024b. All comparisons are carried out in mono-thread mode. The LP solver is Cplex version 12.10 (or gurobi for $k > 1000$).

	k				
n	10	100	500	1000	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
Opt 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	no mem
10000	0.4416	0.8831	no mem	no mem	no mem

Table 5: Time comparison between `Proj-Cut-Pl` and `Mosek` on very dense instances simply generated as in the first paragraph of Section 4.6.3. For larger n, `Mosek` can easily exceed a time cut-off limit of 10000 seconds; for $n = 10000$, it can not perform a single iteration within this time limit.

4.6.3 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 fix $\mathbf{b}_\kappa = \lfloor \sqrt[3]{\kappa} \rfloor$. It's more convenient to generate such big 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 5 compares `Proj-Cut-Pl` and `Mosek`: each cell (besides first two columns or first row) presents the running 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). The interior point method of `Mosek` may need $O(k^2n^2 + kn^3)$ operations at each iteration, as discussed in Section 4.2.3.2. Perhaps this is why this table suggest that `Mosek` can not compete very well for a value of n in the thousands.

Table 5 suggests that `Proj-Cut-Pl` may be a very good tool if you have a dense SDP program with a value of n in the order of thousands. Another characteristic of such instances that makes them well-suited to `Proj-Cut-Pl` is that few variables $\mathbf{y}$ are non-zero at optimality. This means that the needed outer approximation is really not expensive: a few generated linear constraints that involve only these few variables may be enough.

All results reported now and hereafter were obtained on a standard laptop with an Intel `i7-8665U` processor clocked at 1.90GHz with 16 Gigabytes of RAM. We used the Linux Mint operation system; the Linux kernel version is 4.15.0. There is only one exception to using this mainstream laptop: for $n \geq 5000$, we used the cluster from Section 4.6.2, which is necessary to record such big matrices. The running times reported in Table 5 for $n \geq 5000$ were actually obtained by scaling (a multiplication by $\frac{2}{3}$ seems fine) the running times obtained on the cluster.

Let us delve a bit into the details of the very low number of iterations needed by `Proj-Cut-Pl`. We noticed 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. In all cases, the initial outer approximation contains $n = 1000$ constraints from the first paragraph of Section 4.2.2 and from (4.J), hence a 1001 term in last row below. These 1001 constraints are not essential here: even without them, an outer approximation with a few constraints is still enough. This means that the dual solution matrix constructed as in (4.G.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
Total constraints $\mathscr{D}$ in final (4.F)	1001+2	1001+4	1001+3	1001+2	1001+3
Rank of the dual matrix in (4.G)	1	2	2	1	2

4.6.4 A non-negative dense SDP

Let us present a new way of generating random highly-feasible dense instances.[13] We first randomly generate $\frac{n}{5}$ vectors meant to cover the null space of all involved 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 the A_i with $i > 0$ with a probability of 80%. Once a null space of size n_z is generated for a matrix, we randomly generate $n - n_z$ orthonormal eigenvectors that constitute the image (kernel) of the matrix. We then construct the final matrix by applying the eigendecomposition (4.B), where the non-zero eigenvalues are generated 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 $A_i \succ \mathbf{0}$ for some i is almost zero. Each entry in the objective $\mathbf{b}$ is randomly chosen between 0.5 and 1.5. We thus obtain a kind of generalization of an LP with non-negative coefficients. `Proj-Cut-Pl` easily finds the starting point $\mathbf{0}_k$ because $\lambda_{\min}(A_0)$ is zero in (4.I).

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 precision` $= 5$	0.26	0.37	0.88	0.91	1.06	1.09	2.51
`Proj-Cut-Pl precision` $= 2$	0.23	0.27	0.58	0.64	0.67	0.76	1.41
`Proj-Cut-Pl precision` $= 6$	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	—	—
`CSDP`	8.2	49	441	2215	—	—	—
`Clarabel`	5.26	309	3346	—	—	—	—
`Loraine`	0.03	24	—	—	—	—	—
`ClusteredLowRankSolver`	8.97	79	—	—	—	—	—

Table 6: Wall time (seconds) needed by various solvers on different sizes for the instances from Section 4.6.4. The entries marked with a dash may indicate either various errors or time-out. The cut-off time limit is one hour.

Table 6 compares `Proj-Cut-Pl` with seven other solvers on seven instances generated as above, with increasing values of k and $n = 10 \cdot k$. The first data row represents the standard `Proj-Cut-Pl` version with `precision` $= 5$, *i.e.*, Alg. 3 (p. 36) 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 solve it in a matter of seconds. Only `Mosek` and `SeDuMi` seem to be able to keep pace on such instances. Preliminary experiments show that the `ConicBundle` can also be very fast if we keep $k \leq 20$ and let only n grow.

In Table 7 we use instances of a similar nature but with $n = 100$ and with the following difference. We fix $A_1 = I_n$, $A_2 = -I_n$, $b_1 = 1$ 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 check how the compared methods scale when increasing k, since large values of k induce a greater computational burden on the LP solver.

Table 7 shows that for large values of k, the LP solver step (Line 5 in Alg. 3, p. 36) becomes the main computational bottleneck. If it does not take more than 90% of the total running time, it is because of Line 13, *i.e.*, it may also be expensive to compute the k cut coefficients $\mathbf{a}$ 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, Table 7 suggest that Gurobi 8.11 is perfectly fine, *i.e.*, it is not necessarily slower than gurobi 12.1 for our needs. We use Cplex 12.10 because later versions do not (yet) support `Matlab`. Thus, the LP solver used in `Proj-Cut-Pl` is at least five years old at the time of the writing.

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

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
3500	234 (51%)	258 (67%)	236 (65%)	512	867	tm. out
4000	292 (55%)	329 (69%)	341 (71%)	592	1201	tm. out
4500	264 (68%)	554 (76%)	501 (74%)	782	1354	tm. out
5000	458 (68%)	796 (81%)	1209 (80%)	1005	2352	tm. out
5500	665 (71%)	984 (82%)	1072 (82%)	1149	2554	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 7: Comparing Proj-Cut-*Pl* with the three fastest algorithms from Table 6 considering a fixed $n = 100$ and increasing values of k. The information in parantheses is the proportion of CPU time spent on the LP solver. The time-out limit is the time from Column 2 multiplied by 100.

4.6.5 SDP programs with LP (in)equalities

We now consider the same instances from Table 6 by introducing $\frac{k}{5}$ linear inequalities and one equality. The new instance will capture the whole model (4.F), *i.e.*, including the initial constraints $\mathscr{C}$ in (4.F.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 8 presents the results on above instances, providing more detail on Proj-Cut-*Pl*. The first column provide the instance name under the format highFsb-k-lp, where $n = 10k$; Column 2 gives the optimal value. Columns 3-7 are devoted to Proj-Cut-*Pl*. The number of iterations in Column 3 may include a term $+iters_{box}$, where $iters_{box}$ is the number of iteration of the optional standard Cutting-Planes from Line 3 of Alg. 3 (presolve). Columns 5, 6 and, resp., 7 indicate what percentage of the total running time (Column 4) was spent on the projection sub-problem, the (optional) separation sub-problem and, resp, the LP solver. 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 both positive and negative.

Table 8 show that Proj-Cut-*Pl* can keep the pace with the fastest Interior Point Methods on (very) small instances, although it's slower. Yet, for a large $k = 50$ and $n = 10k = 500$, 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 4.2.2 is much weaker when $\mathbf{y}$ is free. This also leads to a need of using an artificial box meant to include the feasible area (Obs. 4.B, Section 4.2.3.1), hence a number of presolve iterations (the second term of the sums from Column 3).

Columns 5 and 6 of Table 8 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 current projection speed sounds very reasonable for the moment. In many cases, the projection time is less than half of the total Proj-Cut-*Pl* running time.

4.6.5.1 What if each linear constraint yields prohibitively-many robust cuts?

We now consider the given linear inequalities as nominal constraints in a robust optimization framework; to be specific, as in Chapter 3, each inequality can lead to prohibitively many robust cuts. We fixed $\Gamma = 3$ and $\delta = 0.1$: we allow at maximum 3 of the nominal coefficients to go up or down by 10% at maximum.

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 (4.F.3) and one for the SDP cone cuts $\mathscr{D}$ from (4.F.4). The robust

Instance	Opt obj val	Proj-Cut-*Pl* Iterations	Time[s]	Proj	Sep	LP solver	Mosek Time[s]	SeDuMi 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 both positive and 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 8: Comparing Proj-Cut-*Pl* with the two fastest algorithms identified previously on instances that include several initial (in)equalities. 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.

projection is solved using the algorithm from Section 3.2.2. The Cutting-Planes logic of these robust cuts is integrated very naturally into the SDP 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 robust cut the most violated by the current outer (non-robust) solution (iii) re-solve the problem enriched with the violated cut from point (ii) and generate a new current outer 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, the algorithm stops by reporting that the current solution is optimal.

Table 9 compares these algorithms and it confirms the above advantages of Proj-Cut-*Pl*. While the robust cuts are integrated seamlessly in the Proj-Cut-*Pl* framework, they lead to calling the SDP solver too many times if we use Mosek or SeDuMi. The number of solver calls from the last and the third-to-last column also corresponds to the number of robust cuts needed by the corresponding solver. The IPM does not have a reputation for being able to easily use warm-starting strategies, so as to start the process from some solution discovered before generating the last robust cut; unlike the Simplex algorithm, adding a new linear constraint to an SDP problem may require the IPM to start from scratch. Or, at least, we are not aware of any off-the-shelf feature in Mosek or SeDuMi that implements such re-optimization task.

More generally, the underlying idea from this section it that Proj-Cut-*Pl* is naturally capable of performing re-optimization, *i.e.*, it does not need to restart from scratch when new constraints are generated (*e.g.*, in branch-and-bound). The above approach can be used whenever one is faced prohibitively many linear constraints in the original model. If most of these constraints are not active (at optimality), the overall Cutting-Planes logic of the new algorithm integrates them on the fly. This is conditioned only on the solvability of the projection sub-problem over the non-SDP part.

The more general underlying idea from this section it that Proj-Cut-*Pl* is naturally capable of performing re-optimization, *i.e.*, it does not need to restart from scratch when new constraints are generated. If the original model (4.F.1) contains an excessive number of linear constraints $\mathscr{C}$ but most of them are not active at optimality, it may be enough to generate only a part of them on demand, *i.e.*, whenever they are needed (*e.g.*, by a branching rule in a Branch and Bound). If one cannot design a projection algorithm for $\mathscr{C}$, these constraints can still be dynamically generated by separation, albeit with some modifications of the Proj-Cut-*Pl* pseudo-code. Otherwise, they can be integrated into the Proj-Cut-*Pl* pseudo-code with almost no (major) modification.

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

4.6.6 Towards size ($n = 30000, k = 10$) via the Goemans-Williamson relaxation of Max-Cut

We here consider the following program with $n = k$, derived from the Goemans-Williamson relaxation of the Max-Cut problem, see, *e.g.*, [38, Prop. 7.1.1] or [21, § 6.1.1]. It was obtained by simple manipulations of the dual of $\max\left\{\frac{1}{4}A_0^L \bullet S : \mathtt{diag}(S) = I_n,\ S \succeq \mathbf{0}\right\}$.

$$\min \quad \sum_{i=1}^{k} y_i \tag{4.V.1}$$

$$s.t \quad \sum_{i=1}^{k} -A_i^L y_i \preceq -A_0^L \tag{4.V.2}$$

$$\mathbf{y} \geq 0. \tag{4.V.3}$$

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 (4.V.2) to the diagonal, you will notice we need to have $y_i \geq \widehat{D}_{ii} \geq 0$. Each feasible solution $\mathbf{y}_{\text{in}}$ of (4.V) generated by Proj-Cut-*Pl* at each iteration yields an upper bound $\mathbf{1}_k^\top \mathbf{y}_{\text{in}}$ for the Max-Cut problem. Each outer solution $\mathbf{y}_{\text{out}}$ produces the lower bound $0.8785 \cdot \mathbf{1}_k^\top \mathbf{y}_{\text{out}}$, 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 exploit the particular structure of this program: notice $\sum_{i=1}^{k} -A_i^L = -I_n \prec \mathbf{0}$ 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 competition. The sparse version simply encodes all involved matrices in a sparse format. Each instance is associated to 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. Recall the matrices $A_1^L, A_2^L, \ldots A_k^L$ contain only one non-zero element, so that all inner 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 is a negative case study for Proj-Cut-*Pl*. Given the large number of iterations needed to converge, Proj-Cut-*Pl* is no match for Mosek, SeDuMi, or probably other interior point methods

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 $A_1 \leftarrow A_1 + A_2 + \cdots + A_{\frac{k}{10}}$												
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.
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 (4.V.1)–(4.V.3) obtained via substitutions like $A_1 \leftarrow A_1 + A_2 + \cdots + A_{\frac{k}{10}}$, or $A_2 \leftarrow A_{\frac{k}{10}+1} + A_{\frac{k}{10}+2} + \cdots + A_{2\frac{k}{10}}$, *i.e.*, we compress the instances as in Obs 4.H with $\mathtt{k_{div}} = \frac{k}{10}$.

(IPMs). These methods need relatively few steps to converge, without having to construct an expensive outer approximation. It seems that the competition is very elaborately tuned to exploit all sparsity in the given program. May I argue that, in historic terms, the first SDP solvers perhaps not exploiting many (such) features needed almost one hour to solve this program for $n = k = 50$ [50, § 3].[14] In a similar context, [21, § 6.1] states that "before Interior Point Methods (IPMs) for semidefinite programming become available, problems on complete graphs with 40 nodes were considered unsolvable". However, compared to modern IPMs, preliminary experiments show `Proj-Cut-Pl` is not very competitive on other SDP relaxations of combinatorial optimization problems (like the Lovász theta number), perhaps for similar reasons.

The second half of Table 10 is a positive case study for `Proj-Cut-Pl`. Each considered instance is built from a `Max-Cut` instance with $n = k$ by applying the compression from Obs. 4.H 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 `Max-Cut`. Practice confirms the theoretical speed discussion from Section 4.2.3.2: for $n \geq 5000$ it is too hard for an interior point method to calculate the Schur matrix when that requires $O(k^2n^2 + kn^3)$ operations per step. For such instances (*low k, huge n and sparse matrices*), `Proj-Cut-Pl` is probably one of the best options. The sparse `Proj-Cut-Pl` variant is roughly 10 times faster 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.

The memory consumption of `Mosek` exceeds the available amount (recall we use a standard laptop with 16GB of RAM) for $n \geq 20000$ – or $n \geq 15000$ pour `SeDuMi` – even by massively exploiting the sparsity of the involved n-by-n matrices that have no more than a few dozens non-zero coefficients per row.

4.7 SDP conclusion and prospects

This chapter presented an SDP optimization algorithm powered by a different idea, namely the projection sub-problem in a `Cutting-Planes` framework. The project is still in a research phase: the current code (about 6000 lines) is still just a prototype or a proof of concept. Although we showed it can compete well

[14] This is how I understand the French text "Les premiers outils de résolution mettaient près d'une heure pour venir à bout d'une relaxation semidéfinie d'un maxcut d'un graphe de cinquante sommets" (Early solvers took almost an hour to solve a semidefinite maxcut relaxation of a fifty-vertex graph).

with well-established solvers developed over multiple decades, `Proj-Cut-Pl` is not (yet) a direct competitor to existing solvers in the view of the standard user.

The main goal is not numerical competition only. One advantage of `Proj-Cut-Pl` that goes beyond pure numerical speed is the fact that it generates both a primal-feasible and a dual-feasible SDP solution at each iteration, a feature that does not exist built-in in many SDP optimization methodologies. It also has an intrinsic ability to accommodate re-optimization, *i.e.*, it does not need to restart from scratch if the given SDP program is enriched with a new linear constraint (unlike most IPMs). We took advantage of this capacity when solving the robust SDP program with prohibitively-many linear constraints from Section 4.6.5.1. A similar phenomenon is illustrated in the binary SDP program from Appendix A, where each branching rule reduces to adding a linear constraint (4.F.3) to the SDP program.

On the practical side, a long-term goal is to publish a full Matlab (or even Julia) package addressing all particular cases imaginable (any ill-conditioned input), as well as exploiting any opportunity in terms of sparsity, parallelism, linear dependency checking (for the matrices A_i with $\in [0..k]$), more refined heuristics to find a starting feasible point, preprocessing, full parametric tests, etc. In short, such a package has to cover any idea that can simplify the problem, as well as all software aspects (*e.g.*, an on-line repository, documentation, pre-compiled executables, etc). Only time will tell if the proposal will have success with SDP practitioners whose needs correspond to the positive case studies of `Proj-Cut-Pl`, *e.g.*, dense instances with a large n and a limited k (or at least with an optimal $\mathbf{y} \in \mathbb{R}^k$ that has few non-zeros).

Research-wise, one of the greatest weaknesses is the initial outer approximation for the case when the variables $\mathbf{y}$ are free. If this initial envelope that relaxes the feasible area is very inaccurate, important time will be wasted in the beginning, because the outer solution $\mathbf{y}_{\text{out}}$ will fluctuate too much from iteration to iteration, far from the optimal solution (bang-bang oscillation effects).

More important for us, we hope the underlying idea may pave the way for solving more general conic problems, using many relatively-lightweight (`Cutting-Planes`) tools imported from linear optimization. For instance, the copositive cone is described by a (still infinite) subset of the SDP constraints $\mathscr{D}$ from (4.F.4), *i.e.*, it simply restricts $\mathscr{D}$ to the vectors $\mathbf{d} \in \mathscr{D}$ such that $\mathbf{d} \geq \mathbf{0}_n$. Identifying such non-negative $\mathbf{d}$ is NP-hard, even for simply separating copositive matrices. Still, should one succeed in designing a projection algorithm that accommodate this additional condition, the presented `Proj-Cut-Pl` could be turned into an algorithm that optimizes over the copositive program version of (4.F.4).

5 General conclusion

This Habilitation thesis presented a research thread built around a single idea, even if I also worked on other projects enumerated in Section 1.2. It's a theoretical-easy but practical idea of projection inside a convex body. It should not be intuitively seen as an orthogonal projection, but rather as a line-intersection with a convex body. We explored its development on three problems expressed as LPs with prohibitively-many constraints, *i.e.*, on a Column Generation (CG) model for (Elastic) Cutting-Stock in Chapter 2, on robust linear programming in Chapter 3 and for SDP optimization in Chapter 3. Other applications have been addressed in related publications, *e.g.*, a Benders reformulation model for network design in [40] or an `Arc-Routing` CG model in [39]. We hope this thesis can shed useful light in solving even other problems that reduce to optimizing over polytopes $\mathscr{P}$ with unmanageably-many constraints.

The experimental sections for the three addressed problems (Sections § 2.5, § 3.3, § 4.6) describe the numerical potential of `Proj-Cut-Pl` in terms of running time improvements, at least compared to a standard `Cutting-Planes`. To achieve numerical success it is essential to design a fast projection algorithm, not much slower than the separation one. This means more work needs to be invested to implement a successful `Proj-Cut-Pl` than for a standard `Cutting-Planes`. Designing a fast projection algorithm was particularly challenging in SDP optimization, because in this case $\mathscr{P}$ is actually the SDP cone. The resulting software seems to solve more rapidly certain SDP instances (with large n and small k) even when comparing to a very solid competition, *i.e.*, against more elaborately-tuned solvers refined over a few decades.

References

[1] Cláudio Alves, Francois Clautiaux, JM Valério de Carvalho, and Jürgen Rietz. *Dual-Feasible Functions for Integer Programming and Combinatorial Optimization.* Springer, 2016.

[2] Gilles Audemard, Christophe Lecoutre, Mouny Samy-Modeliar, Gilles Goncalves, and Daniel Porumbel. Scoring-based neighborhood dominance for the subgraph isomorphism problem. In *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*, pages 125–141. Springer, 2014.

[3] Walid Ben-Ameur and José Neto. Acceleration of cutting-plane and column generation algorithms: Applications to network design. *Networks*, 49(1):3–17, 2007.

[4] Pierre Bonami, Domenico Salvagnin, and Andrea Tramontani. Implementing automatic benders decomposition in a modern mip solver. In *Integer Programming and Combinatorial Optimization: 21st International Conference, IPCO 2020, London, UK, June 8–10, 2020, Proceedings*, pages 78–90. Springer, 2020.

[5] Abraham Charnes and William W. Cooper. Programming with linear fractional functionals. *Naval research logistics quarterly*, 9(3-4):181–186, 1962.

[6] François Clautiaux. *New collaborative approaches for bin-packing problems*. Habilitation à diriger des recherches, Université de Technologie de Compiègne, November 2010.

[7] Michele Conforti and Laurence A Wolsey. Facet separation with one linear program. *Mathematical Programming*, 178(1):361–380, 2019.

[8] 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.

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

[10] Sourour Elloumi, Amélie Lambert, and Arnaud Lazare. Solving unconstrained 0-1 polynomial programs through quadratic convex reformulation. *Journal of Global Optimization*, 80(2):231–248, 2021.

[11] Matteo Fischetti and Michele Monaci. Cutting plane versus compact formulations for uncertain (integer) linear programs. *Mathematical Programming Computation*, 4(3):239–273, 2012.

[12] Jean Fonlupt and Alexandre Skoda. Strongly polynomial algorithm for the intersection of a line with a polymatroid. In William Cook, László Lovász, and Jens Vygen, editors, *Research Trends in Combinatorial Optimization*, pages 69–85. Springer Berlin Heidelberg, 2009.

[13] 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.

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

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

[16] Jacek Gondzio. Interior point methods 25 years later. *European Journal of Operational Research*, 218(3):587–601, 2012.

[17] Jacek Gondzio, Pablo González-Brevis, and Pedro Munari. Large-scale optimization with the primal-dual column generation method. *Math. Prog. Comp.*, 8(1):47–82, 2016.

[18] M. Gu, A. Ruhe, G. Sleijpen, H. van der Vorst, Z. Bai, and R. Li. *Generalized Hermitian Eigenvalue Problems*, chapter 5 of book *Templates for the Solution of Algebraic Eigenvalue Problems*, pages 109–133. SIAM, 2000.

[19] Nathalie Helal, Frédéric Pichon, Daniel Porumbel, David Mercier, and Éric Lefèvre. The capacitated vehicle routing problem with evidential demands. *International Journal of Approximate Reasoning*, 95:124–151, 2018.

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

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

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

[23] 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.

[24] 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.

[25] Ken Kobayashi and Yuich Takano. A branch-and-cut algorithm for solving mixed-integer semidefinite optimization problems. *Computational Optimization and Applications*, 75(2):493–513, 2020.

[26] 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.

[27] Amélie Lambert and Daniel Porumbel. A piecewise-quadratic convexification for exactly solving box-constrained quadratic programs. In *The 2023 World Congress on Global Optimization*, 2023.

[28] Amélie Lambert and Daniel Porumbel. Using quadratic cuts to iteratively strengthen convexifications of box quadratic programs. In *Accepted by Journal of Global Optimization*, june 2025.

[29] 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.

[30] Adam N. Letchford and Andrea Lodi. Primal cutting plane algorithms revisited. *Mathematical Methods of Operations Research*, 56(1):67–81, 2002.

[31] Marco E. Lübbecke and Jacques Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023, 2005.

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

[33] S. Thomas McCormick. Submodular function minimization. In G.L. Nemhauser K. Aardal and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 321 – 391. Elsevier, 2005.

[34] Kiyohito Nagano. A strongly polynomial algorithm for line search in submodular polyhedra. *Discrete Optimization*, 4(3–4):349 – 359, 2007.

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

[36] David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271 – 2284, 2005.

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

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

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

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

[41] Daniel Porumbel. Prize-collecting set multicovering with submodular pricing. *International Transactions in Operational Research*, 25(4):1221–1239, 2018.

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

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

[44] Daniel Porumbel, Igor M Coelho, and El-Ghazali Talbi. Using an exact bi-objective decoder in a memetic algorithm for arc-routing (and other decoder-expressible) problems. *European Journal of Operational Research*, 313(1):25–43, 2024.

[45] Daniel Porumbel and Gilles Goncalves. Using dual feasible functions to construct fast lower bounds for routing and location problems. *Discrete Applied Mathematics*, 196:83–99, 2015.

[46] Daniel Porumbel, Gilles Goncalves, Hamid Allaoui, and Tienté Hsu. Iterated local search and column generation to solve arc-routing as a permutation set-covering problem. *European Journal of Operational Research*, 256(2):349–367, 2017.

[47] Daniel Porumbel and Jin-Kao Hao. Distance-guided local search. *Journal of Heuristics*, 26(5):711–741, 2020.

[48] Daniel Cosmin Porumbel. Isomorphism testing via polynomial-time graph extensions. *Journal of Mathematical Modelling and Algorithms*, 10:119–143, 2011.

[49] Daniel Cosmin Porumbel, Jin-Kao Hao, and Pascale Kuntz. Spacing memetic algorithms. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1061–1068, 2011.

[50] Frédéric Roupin. L'approche par programmation semidéfinie en optimisation combinatoire. *Article invité dans le Bulletin de la Société Française de Recherche Opérationnelle et d'aide à la décision*, (13):7–11, 2004.

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

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

[53] 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.

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

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

[56] Bianca Spille and Robert Weismantel. Primal integer programming. volume 12 of *Handbooks in Operations Research and Management Science*, pages 245–276. Elsevier, 2005.

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

[58] Jos F. Sturm. Implementation of interior point methods for mixed semidefinite and second order cone optimization problems. *Optimization Methods and Software*, 17(6):1105–1154, 2002.

[59] Arthur F. Veinott. The supporting hyperplane method for unimodal programming. *Operations Research*, 15(1):147–152, 1967.

[60] Gerhard Wäscher, Heike Haußner, and Holger Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109 – 1130, 2007.

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

Editorial note *This document includes a number of corrections or style improvements compared to the official thesis submitted to the three referees, Sophie Demassey, Christoph Dürr and Renata Sotirov. Some modifications were directly suggested by Christoph Dürr. Others represent expository refinements introduced on my own initiative, including a new figure and a few references. I deliberately refrained from changing the table of contents. The goal is not to alter the general flow of ideas, but to improve readability and complement a few notions with clearer insights.*

This appendix is actually the only change to the table of contents. It is devoted to an additional experiment on the binary version of the SDP program from Chapter 4.

I consider mixed-integer SDP programming important: many SDP formulations have received considerable attention because they provide tighter bounds than the LP relaxation to integer or combinatorial (quadratic) optimization problems, see, *e.g.*, overview papers [29, 55]. One may sometimes solve an 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 [8, 13, 32].

Let us thus solve the same model (4.F.1)-(4.F.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 2^k 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 variables $y_{\kappa+1}, \dots y_k$ free, where $\kappa \in [1..k]$. 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 feasible solution. A branch is pruned if the rounded-down optimal value of this SDP program is no larger than the best binary feasible solution found so far. We also naturally prune a branch 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 anticipate a few advantages of `Proj-Cut-Pl` without needing the numerical results.

- We do not solve each node-specific SDP program independently. The eigen-cuts (4.F.3) generated to solve a given SDP program are retained in a unique `Cutting-Planes` model that is used to solve all SDP programs needed during the whole `Branch and Bound`. Only the bounds of the fixed variables $y_1, \dots, y_\kappa$ have to be changed when switching from one branching node to another.

- Since `Proj-Cut-Pl` generates an upper bound at each iteration, we can stop it as soon as the rounded-down value of an intermediate upper bound is no larger than the best-known binary feasible solution. To the best of our knowledge, `Mosek` (and other IPMs) does not compute intermediate upper bounds. Even since the root node, this enables `Proj-Cut-Pl` to take advantage of a highly permissive stopping criterion, namely generating intermediate lower and upper bounds with the same rounded-down value.

- Since `Proj-Cut-Pl` generates a fractional feasible solution at each iteration, one can rounded it down an obtain a binary feasible solution; this increases the likelihood of finding heuristic solutions along the way, which is a feature we can not find in `Mosek` (and other IPMs).

Table 11 presents the `Branch and Bound` results. The first three columns provide the instance. Columns 4 and 5 report the number of branching nodes, and, resp., the CPU time of the `Mosek`-powered `Branch and Bound`. Columns 6 and 7 provide the same information for `Proj-Cut-Pl`. The last column reports 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 needed while exploring the branching tree.

These results confirm the above theoretical advantages of the `Proj-Cut-Pl` solution. If this solution is *an order of magnitude faster* than the `Mosek`-powered `Branch and Bound`, it is mainly because it does not solve each node-specific SDP program from scratch. The `Proj-Cut-Pl` approach maintains 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 node to node. For the first instance, it could solve the node-specific SDP programs corresponding to $y_1 = 0$ and $y_1 = 1$ without needing a single new eigen-cut: the outer

Instance	k	Opt obj val	Mosek		Proj-Cut-*Pl*		
			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 11: A Branch and Bound powered by Proj-Cut-*Pl* against the same Branch and Bound powered by Mosek, using the same instances as in Table 6 (Section 4.6.4), with a modified objective function $\mathbf{b} = \mathbf{1}_k$.

approximation computed at the root node was sufficiently tight and descriptive enough to close the gap of the child nodes.

The two methods do not necessarily generate the same number of nodes or the same branching tree because they differ in their approach to 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 optimal solution of each node-specific SDP program does offer an advantage to Mosek: by rounding down this optimal solution, it may find a higher quality binary-feasible 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 will 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 the Mosek approach needs 534 seconds, while the Proj-Cut-*Pl* one requires 14 seconds, using 11+492 iterations.

The more general underlying idea from this section and from Section 4.6.5.1 is that Proj-Cut-*Pl* has naturally a superior efficiency in performing re-optimization, since it does not need to restart from scratch when new constraints are generated. If the original model (4.F.1)-(4.F.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, *i.e.*, whenever they are needed, (*e.g.*, by a branching rule or by the robust paradigm).