Structure-Exploiting and Accelerated Methods for Convex Optimization

Govind Muralidharan Chari

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2026

Reading Committee:
Behçet Açıkmeşe, Chair
Mehran Mesbahi
Danylo Malyuta

Program Authorized to Offer Degree:
Aeronautics & Astronautics

University of Washington

Abstract

Structure-Exploiting and Accelerated Methods for Convex Optimization

Govind Muralidharan Chari

Chair of the Supervisory Committee:
Professor Behçet Açıkmeşe
Aeronautics & Astronautics

Convex optimization has matured into a reliable technology, with efficient algorithms and software available for most problem classes. Despite this maturity, developing solvers that are faster on small problems and capable of scaling to larger ones remains important. Latency-sensitive applications such as real-time trajectory optimization, real-time control, and portfolio backtesting demand fast solves on smaller problems, and applications in supply chain optimization, machine learning, and signal processing require solvers that scale to very large problems.

This dissertation presents three main contributions towards these goals, two practical and one theoretical. First, we present QOCOGEN, a custom solver generator for quadratic-objective second-order cone programs (SOCPs). QOCOGEN autocodes a primal-dual interior point method that exploits the sparsity structure of the problem at hand, achieving nearly a $4\times$ speedup over QOCO, a generic implementation of the same algorithm. Second, we develop a GPU-accelerated backend for QOCO to solve large quadratic-objective SOCPs, achieving nearly a $70\times$ speedup over QOCO's CPU backend on some large problems. Both QOCO (with its CPU and GPU backends) and QOCOGEN are open-source and can be called from modeling languages such as CVXPY and CVXPYGEN. Third, we derive Fast Douglas–Rachford splitting (FDR), an accelerated operator-splitting method for minimizing the sum of a convex and a strongly convex function. FDR achieves an accelerated $\mathcal{O}(1/N^2)$ convergence rate for the

squared distance to the optimal solution, with a leading constant that is a factor of four better than the fastest previously known methods. We also construct a complexity lower bound establishing the optimality of both FDR's $\mathcal{O}(1/N^2)$ convergence rate and the leading-order constant.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I have been unbelievably lucky in my life to come across so many fantastic people who have selflessly helped me and made me the person I am today. I would like to thank them all here.

I need to start with the most important people: my parents. Every positive trait I have is due to them, and this PhD is as much theirs as it is mine. I remember my dad would tell me to "go learn something new" every day before I went to school from elementary school through high school. He has always been curious and loved to learn new things for as long as I remember, whether it is physics or optimization from me, vehicle dynamics from my sister, or biology from my mom. I distinctly remember him explaining the Pythagorean theorem to me when I was very young. I cannot fathom why a business professor was so enamored by the Pythagorean theorem to teach it to a five-year-old, but that is exactly the kind of person my dad is. He would also insist that my own explanations of any concepts be clear and simple, which always solidified my understanding and is a skill I still use to this day to learn. I hope to share his fascination with knowledge for the rest of my life. My mom has also always nurtured my curiosity and has been patient with me (which is an incredible feat). I very distinctly remember going to her lab as a kid and she would pour liquid nitrogen on the floor to entertain me or stimulate my curiosity, I still don't know which. She also came to my elementary school's science day and showed my classmates and I how to extract DNA from strawberries. She has always patiently listened to every complaint I had had during college and my PhD and always gave very balanced, and frequently correct, advice (even if I was too stubborn to listen to her). Most importantly, my parents have always stressed intellectual honesty and having strong set of morals. I definitely have not made

their lives easy with all my career switches from propulsion to GNC to robotics to supply chain optimization to finance, but I am thankful they have always trusted me to make the right choices and believed in me every step of the way. Finally, I must mention my sister. Ever since we were both young, we have pushed each other and I am sure we are both better off for it. I cannot recall a single moment in her life where she has had a shred of self-doubt, which is something I have always admired, and I hope to be more like her in this way.

The most important teacher I have ever had was my high school physics teacher, Mr. Neff. I entered his class as an arrogant kid who thought that memorizing facts equaled having knowledge. As Richard Feynman said "I learned very early the difference between knowing the name of something and knowing something." Unfortunately for me, I am no Feynman so this was a lesson I learned in 11th grade due to Mr. Neff. I was always so eager to answer questions in his class to get my dopamine hit of giving a correct answer, but I always answered by recalling equations or giving some surface level explanation. These were never acceptable to him and he would always insist on first-principles reasoning, something that I am forever grateful for. He also has a genuine passion for physics and loved teaching students, especially if they share this enthusiasm. One day I told him some question I got wrong in science bowl related to Maxwell's equations (before we started the electricity and magnetism unit). He then told me to come to his classroom during my free period and he set up a demo of electromagnetic induction and described what was happening. By the time I left Mr. Neff's class, I had a love of first-principles reasoning, and I developed a very strong work ethic; I remember staying up until 2am reading our physics textbook and doing every problem in the book. Both traits have served me very well ever since then and are a large reason I am where I am today.

As soon as I got to Cornell, I quickly became friends with the only other freshman in my statics class, Shihao Cao. Shihao has some of the best engineering intuition I have seen and is the poster child of first-principles reasoning. The summer after my freshman year, Shihao

and I worked on Lodestar, an electric VTOL vehicle, which to this day was the most difficult project I have worked on. This project spurred my interest in GNC and rocket landing, as I designed and implemented the feedback controls for this vehicle and did all the flight testing until this project succeeded. It is no exaggeration to say that this project got me my first SpaceX internship, and every internship/job interview I had in undergrad. This project would not have succeeded without all of Shihao's help and advice, which he generously gave despite him working overtime in his SpaceX internship.

At some point in college, I came across Padraig Lysandrou's website. From an engineering perspective, Padraig's closest analogue is Tony Stark (after all he is building a rocket in his garage). What intrigued me was these rocket landing algorithms called *lossless convexification* and *successive convexification* which were plastered all over his website. Having just finished Lodestar, I wanted to understand how to land rockets properly and Padraig's website was the perfect resource. I even ended up recreating the lossless convexification algorithm as my final project in ECE 5555 as he did. Eventually, I met Padraig in person while first interning at SpaceX and he was always so generous with his time to explain GNC topics to me. I remember my biggest issue at the time was whether I would do a PhD or go to industry and work on rocket landing algorithms. This debate consumed me for weeks and I constantly went back and forth (and luckily I chose right in the end). I don't remember his advice (it was likely to do the PhD), but I clearly remember what he said after: "I am not worried about you". His belief that I would be fine regardless of how I chose immediately put me at ease. I first learned about Behçet's lab and his research through Padraig, so it is very largely due to him that I found myself at the University of Washington doing my PhD with Behçet.

During my first SpaceX internship, I had the great fortune of meeting Kevin Tracy, the most enthusiastic advocate of optimization, and the person who has shaped my PhD research the most. While at SpaceX, I remember mentioning to him that I wanted to write a convex

solver, but it seemed too hard since people write PhD theses on them. He told me to go for it and pointed me to a few resources to get started. Eventually, I implemented my first solver, a primal-dual interior point method for QPs (QOCO implements the same algorithm extended to SOCPs). During the next year, I sent him dozens of texts with questions about various optimization algorithms, and at some point, I was sure he would be fed up with answering all these texts. However, he always responded with very well thought out, detailed answers and pointed me to the right resources. Eventually, my research interests shifted from trajectory optimization to convex optimization algorithms, and everything I learned from Kevin gave me a very strong foundation.

I am incredibly grateful for my advisor, Professor Behçet Açıkmeşe. I find it unbelievable at times how much freedom Behçet has given me to explore whatever I found interesting, which is exactly the environment I needed to succeed. The first time I met him in person was when I was visiting the University of Washington before committing, and I told him I was no longer interested in rocket landing and trajectory optimization, but I wanted to do reinforcement learning research. I expected him to try to convince me otherwise or to be against the idea, but he just said that if that was what interested me, I would have his full support and he would help however he could. After listening to stories about PhD advisors from my parents and friends doing PhD programs, it amazes me that advisors like Behçet exist. It is clear that he has an incredible amount of trust in his students and believes that letting them pursue what excites them will produce the best work.

I feel very fortunate to have worked with Professor Ernest Ryu. I remember implementing the augmented Lagrangian method, ADMM, Douglas–Rachford splitting, yet I used to view all these methods as unrelated algorithms. One day I was lucky to pick up *Large-Scale Convex Optimization* by Professor Ryu and Wotao Yin. The unification of all these algorithms I have implemented and heard about through the lens of monotone operators was truly eye opening. I am a bit embarrassed to admit that I was too busy reading this book that I missed the

ball drop for the 2024 New Years'. I am very proud to tell anyone who will listen that his textbook is my favorite of any I have read. As I was wrapping up QOCO, I knew I wanted to work on a more theoretical project on optimization algorithms, so I cold emailed Professor Ryu to see if he had any ideas on good research directions. I did not expect that he would respond and suggest collaborating, but I am incredibly thankful that he did. Working on our accelerated Douglas–Rachford splitting algorithm greatly expanded my knowledge of operator-splitting methods, accelerated algorithms, algorithm design, and proof techniques.

I am extremely lucky to have met Danylo Malyuta, to have had him as a mentor at SpaceX, and now have had him as a committee member. The first time I met Danylo in person, we ended up spending hours discussing cache hits and misses, branch mispredictions, and x86 intrinsics. Ironically, for two people who did their PhD research in optimization, we have probably spent more time discussing computer performance than optimization. The most striking thing about Danylo is his discipline and consistency, which make him an amazing role model for all who know him. My favorite anecdote about Danylo is that he read *Convex Optimization* cover to cover by reading a few pages a day for months.

The first person to make me feel welcome in Seattle and in the lab was Abhinav Kamath. Abhi is extremely kind, a creative problem solver, and the clearest writer I have met. His "Batman mode", where he disappears for a few days only to emerge with a new paper written or massive code refactor is truly astonishing (I still don't know if he sleeps). I will cherish all of our engaging conversations at Sweetgreen, Madras Dosa Corner, or most importantly, Yezi. Thanks to him, I will never forget to TEX up my plot labels.

An amazing influence in the first few years of my PhD was Purnanand Elango. Purna is one of the sharpest thinkers, yet one of the most humble people I have met. He always seems to be able to produce questions to get to the heart of the matter, allowing him to instantly spot any shortcoming or flaw in an idea, which has saved me a lot of time early on in my PhD.

Samet Uzun is one of the most versatile and honest researchers I have had the pleasure to meet. His intuitive understanding of optimization from first-principles is inspiring and I am glad to have worked with him. Skye, thank you for all of our chats before I joined UW, which in part convinced me to do a PhD and join the lab. Dayou, thank you for all of our chats that taught me much about operator-splitting theory. Chris, thank you for all the help acquiring the necessary compute for all my numerical experiments. Jason, thank you for all the enjoyable moments we have had outside the lab and the much needed de-stressing. Carlos, thank you for all your great questions about optimization which has on many occasions solidified my own understanding.

Finally, I would like to thank everyone who has ever helped me along the way. I truly apologize if I have missed mentioning anyone by name.

DEDICATION

To my parents

"Experience is what you get when you didn't get what you wanted."

–Randy Pausch

Chapter 1
INTRODUCTION

This thesis develops structure-exploiting and accelerated methods for convex optimization. *Structure-exploiting* refers to methods that take advantage of specific properties of a problem class, such as sparsity structure, rather than applying a generic solution technique. In particular, we develop a code generator to exploit known, fixed sparsity patterns in problem data which allows the use of custom linear algebra rather than generic sparse linear algebra routines, leading to faster solves. *Accelerated* is used in two senses. The first refers to hardware acceleration, using specialized hardware (like GPUs), to speed up solving optimization problems. The second refers to algorithmic acceleration, as in Nesterov's accelerated gradient method, where a "base" algorithm is modified to achieve faster worst-case convergence rates without significantly increasing per-iteration cost.

1.1 Background

Optimization studies how to minimize some cost (or maximize some reward) subject to some set of constraints. This makes it a very general framework to model problems in fields such as logistics and supply chain planning, finance, energy distribution, structural design, machine learning, control, and estimation. Although optimization provides a natural framework for many real problems, solving them in general is intractable.

Convex optimization problems are an important exception, since all local minimizers are global minimizers. This property makes them tractable to solve to global optimality with mature, reliable algorithms.

1.1.1 Hierarchy of convex optimization problems

We will now focus on some specific examples of convex optimization problems.

Linear programs (LPs) are a subclass of convex problems with a linear objective function and linear (in)equality constraints as shown by Problem (1.1). These problems are common in logistics and supply chain planning, compressed sensing, and signal processing [38].

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^\top x \\ \text{subject to} \quad & Ax = b \\ & Gx \leq h \end{aligned} \tag{1.1}$$

Quadratic programs (QPs) are a subclass of convex problems with a convex quadratic objective function and linear (in)equality constraints as shown by Problem (1.2). These problems are common in control, machine learning, and finance [28, 29, 129].

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2} x^\top P x + c^\top x \\ \text{subject to} \quad & Ax = b \\ & Gx \leq h \end{aligned} \tag{1.2}$$

Second-order cone programs (SOCPs) are a subclass of convex problems with a linear (or quadratic) objective function, linear (in)equality constraints, and second-order cone constraints as shown by Problem (1.3). Note that although SOCPs with quadratic objectives can be rewritten as SOCPs with a linear objective by introducing an epigraph variable and an additional second-order cone constraint, it is computationally beneficial for solvers to handle this quadratic objective directly [46, 92]. These problems are also common in control, machine learning, finance, and robust optimization [3, 4, 20, 119, 198].

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^\top x \\ \text{subject to} \quad & Ax = b, \\ & Gx \leq h, \\ & \|F_i x + g_i\|_2 \leq d_i^\top x + f_i, \quad i = 1, \ldots, m \end{aligned} \tag{1.3}$$

Semidefinite programs (SDPs) are a subclass of convex problems with a linear objective function and linear (in)equality constraints over the cone of positive semidefinite matrices, as shown by Problem (1.4). SDPs are used in approximation algorithms to NP-hard problems, robust control, and machine learning [30, 39, 88]. Although they are convex, SDPs are quite difficult to solve as they have worse numerical conditioning and scale more poorly than LPs, QPs, and SOCPs [179].

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^\top x \\ \text{subject to} \quad & Ax = b \\ & F_0 + \sum_{i=1}^{n} x_i F_i \succeq 0 \end{aligned} \tag{1.4}$$

These problem classes form a nested hierarchy of convex programs,

$$\text{LP} \subset \text{QP} \subset \text{SOCP} \subset \text{SDP}.$$

Every LP can be cast as a QP, every (convex) QP as an SOCP, and every SOCP as an SDP, with each generalization providing additional modeling power at the expense of increased computational difficulty.

1.1.2 Historical background

Since the feasible set of linear programs is a polyhedron, and the objective function is linear, if the linear program is feasible and bounded, then there exists an optimal solution at a vertex. This observation was used by George Dantzig to develop the simplex algorithm in

1947, the first algorithm for linear programming. This algorithm starts at a vertex of the polyhedron and moves to an adjacent vertex with a lower cost until the optimal solution is found. The simplex method was known to perform very well in practice, but in 1972 Victor Klee and George Minty produced a linear program which showed that the simplex method had a worst case runtime that was exponential in the problem dimension [109]. Eventually, in 1979, the ellipsoid method was introduced by Leonid Khachiyan, the first polynomial-time algorithm for linear programming [105]. However, this algorithm performed much worse than simplex in practice [191].

Then, in 1984, Narendra Karmarkar developed a polynomial-time algorithm for linear programs which was a primal IPM, though not a path-following IPM, and it was reported to be competitive with simplex in practice [104]. Soon after, Masakazu Kojima, Shinji Mizuno, and Akiko Yoshise [110], as well as Renato Monteiro and Ilan Adler [137], developed primal-dual IPMs for linear programs which iterate on both the primal and dual variables while tracking the central path. These primal-dual IPMs have better practical performance than pure primal or pure dual methods [191, 192].

In 1992, Sanjay Mehrotra dramatically improved the practical performance of the primal-dual IPM with his predictor-corrector [134]. To this day, Mehrotra's predictor-corrector underpins many IPM codes such as ECOS [69], CLARABEL [92], and QOCO [46].

Finally, in 1994, Yurii Nesterov and Arkadii Nemirovski extended interior point methods from linear programming to general convex optimization [142]. They introduced the idea of a *self-concordant barrier function* and they proved that any convex optimization problem for which a self-concordant barrier function can be constructed can be solved in polynomial time with a path-following IPM. Notable problem classes that have such barriers include LPs, QPs, SOCPs, and SDPs. Shortly after this, the SDP solvers SeDuMi [178] and SDPT3 [184] were developed. By 2004, in their book titled *Convex Optimization*, Stephen Boyd and Lieven Vandenberghe argued that convex optimization was a technology in the sense that "there are very effective algorithms that can reliably and efficiently solve even large convex problems" [32].

However, the problem sizes and time budgets encountered in modern applications can exceed what traditional solvers can handle. Latency-sensitive applications such as model predictive control (MPC) and real-time trajectory optimization can require solve times below what general-purpose solvers can deliver. For example, MPC for power electronics requires QP solves in tens to hundreds of microseconds [103]. In finance, backtesting a strategy with daily rebalances over 20 years requires 5000 solves, and if a sweep is done over hyperparameters such as risk aversion factors and planning horizon, the total number of solves can easily exceed one million. In order for researchers to iterate on strategies quickly, it is important that each of these solves is as fast as possible [29].

Problems arising in signal processing, machine learning, logistics, and supply chain optimization can be too large for traditional solvers, motivating work on developing new algorithms and software for large-scale optimization [9, 48, 53, 121]. Thus, despite the maturity of convex optimization algorithms, there remains a need for solvers that are both faster on small problems and capable of scaling to larger ones, which is the focus of this thesis.

1.2 Contributions and thesis outline

Each of the three main contributions of this thesis advances one of the aforementioned themes. The first contribution is QOCOGEN, a custom solver generator for quadratic objective SOCPs that generates a structure-exploiting primal-dual IPM written in C and is around 4× faster on small problems than QOCO, an implementation of the same algorithm that does not exploit structure [46]. The second contribution is a GPU backend for QOCO that leverages hardware acceleration to solve substantially larger problems than the CPU implementation [48]. However, even with GPU acceleration, IPMs have a fundamental scalability limit due to their reliance on factorizing large KKT systems. The third contribution is the Fast Douglas–Rachford splitting method (FDR), an accelerated operator-splitting method for minimizing the sum of a convex and a strongly convex function [49]. As an operator-splitting method, it can avoid the linear system solves that limit the scalability of IPMs, provided the proximal operators of the splitting are inexpensive to evaluate. The acceleration improves the worst-

case convergence rate of the Douglas–Rachford splitting method without increasing per-iteration cost. The resulting rate is a factor of four better than the fastest previously known method for strongly-convex composite optimization. Additionally, the $\mathcal{O}(1/N^2)$ convergence rate and leading-order constant of FDR's rate are optimal.

This thesis also includes two chapters on earlier work from my PhD that does not directly align with the themes above. These chapters are included for completeness, and present a GPU-accelerated batch sequential convex programming framework [50] and a preconditioner for ill-conditioned constraints [51].

Chapter 2 discusses algorithms for constrained convex optimization, focusing on conic and quadratic programming, along with their tradeoffs and ideal use cases. The algorithms are not covered in great detail, but I provide references for further reading. Then I discuss several software implementations, the problem classes they solve, their strengths and weaknesses, and my opinions on each. This discussion of strengths and weaknesses reflects my own experience using and benchmarking these solvers.

Chapter 3 presents QOCO and QOCOGEN, the primal-dual interior point method they implement, and how QOCOGEN exploits the sparsity structure of problems to reduce solve time. This chapter is based on [46].

Chapter 4 discusses the GPU backend for QOCO and the implementation of a modular backend abstraction that allows the CPU and GPU backends to share a unified codebase. This chapter is based on [48].

Chapter 5 presents the Fast Douglas–Rachford splitting algorithm, its convergence proof, and the lower bound construction used to certify the optimality of FDR's $\mathcal{O}(1/N^2)$ convergence rate and leading constant. My collaborator, Uijeong Jang, was responsible for the lower bound construction and I developed the algorithm and its convergence proof. This chapter is based on [49].

Chapter 6 discusses the implementation of a GPU-accelerated batch implementation for sequential convex programming (SCP) that allows many nonconvex trajectory optimization problems to be solved in parallel. This is particularly useful for Monte Carlo analysis or for

multistart algorithms where multiple initial guesses are provided to a solver, a trajectory is computed for each, and the best solution is selected. The software implementation of this is in the SCVXGEN package [102]. This chapter is based on [50].

Chapter 7 discusses a constraint preconditioner called the *QR preconditioner*. It is well known that operator-splitting algorithms suffer when applied to ill-conditioned problems. Typically, the ill-conditioning comes from the Hessian of the objective function having a large condition number. This chapter discusses ill-conditioning that arises from nearly parallel equality constraints and presents a preconditioner to mitigate this type of ill-conditioning. This chapter is based on [51].

Chapter 8 provides an honest reflection on the work in each chapter and discusses future directions.

Chapter 2

ALGORITHMS AND SOFTWARE FOR CONVEX OPTIMIZATION

For most users, the right approach for solving convex optimization problems is to use a modeling language like CVXPY [5, 67] or JuMP [77, 122, 123] with its default solver. Aside from curiosity or pedagogical reasons, there are a few situations where implementing your own solver is justified: when problems need to be solved faster than standard solvers can manage or when problems are too large for standard solvers to handle. In either case, building an effective solver typically requires exploiting some problem structure that off-the-shelf solvers ignore. For example, when solving optimal control problems with an interior-point method, a Riccati-based factorization can exploit the block-banded structure of the KKT system to speed up the linear system solve at each iteration [158]. If your problem is too large for any available solver, either because the data does not fit in a single machine's memory, or because solving on one machine is too slow, you may need to write a distributed solver that runs across multiple machines. However, distributed algorithms and solvers are beyond the scope of this chapter.

Algorithms and software are closely connected for practitioners: you can have the best algorithm in the world from a theoretical standpoint, but without a good software implementation, it is of little use in practice. Conversely, the best software in the world is of little help if it implements an algorithm poorly suited to your application. This chapter aims to help readers understand both pieces.

2.1 Algorithms

2.1.1 Interior-point methods

Interior-point methods (IPMs) are second-order methods for convex optimization, meaning they use not only gradient information but Hessian information as well. Specifically, the Hessian is used to form the Newton system solved at each iteration to compute the step direction. At a high level, path-following IPMs approach the optimal solution by following a central path through the interior of the feasible region, traced out by a sequence of problems parameterized by a barrier parameter that is gradually driven to zero. The IPM algorithm is not presented here, but it will be covered in detail in Section 3.2.

Most of the benefits and drawbacks of IPMs stem from the use of Hessian information. IPMs are very robust to ill-conditioning in problem data, and tend to converge to a high-accuracy solution in few iterations (typically under 25). Additionally, they require very little or no parameter tuning to work effectively.

However, a linear system must be solved at each iteration to compute the step direction, which means IPMs do not scale particularly well as the problem dimension grows or if the problem data is dense. Although IPMs can effectively solve sparse problems with up to millions of variables using GPU-accelerated sparse direct solvers [48, 53], for much larger or dense problems, it may not be possible to store the matrix factors in memory, or the factorization time may become cumbersome.

Another drawback of IPMs is that they are difficult to warm-start. If a good guess for the optimal solution is known, IPMs typically cannot exploit this guess to converge faster than from a cold start. Warm-starting IPMs is an active area of research and several methods have been presented in [52, 91, 194], but I have not tried these myself and cannot comment on their maturity. For linear programs specifically, another drawback is that IPMs converge to the interior of the optimal face, rather than to a vertex (basic feasible solution). However, with a crossover procedure, an interior-point solution can be converted to a vertex solution [25].

Some good references on IPMs are [21, 90, 186, 192].

2.1.2 Operator-splitting methods

Operator-splitting schemes solve minimization problems involving the sum of multiple functions by using only each function's gradient (if differentiable) or proximal operator, along with applications of a linear operator and its transpose when one is present. Here, "operator" refers to a monotone operator associated with each function, typically its subdifferential or gradient. These methods are called operator-splitting because they use the proximal operator of each function separately to construct an algorithm for the overall problem. Recall that for a closed, convex, and proper function f, the proximal operator $\text{prox}_{\gamma f} : \mathbb{R}^d \to \mathbb{R}^d$ is

$$\text{prox}_{\gamma f}(x) = \underset{z \in \mathbb{R}^d}{\text{argmin}} \left\{ f(z) + \frac{1}{2\gamma} \|z - x\|^2 \right\}$$

where $\gamma > 0$ is the proximal step-size.

Operator-splitting methods are very general and encompass dual ascent, the method of multipliers (augmented Lagrangian method), the proximal method of multipliers (proximal augmented Lagrangian method), and the alternating direction method of multipliers (ADMM) [166].

Equation (2.1) gives an example of a problem that can be solved with an operator-splitting scheme, where f, g, and h are closed, convex, and proper. Although this problem appears unconstrained, constraints can be imposed by including an indicator function of a convex set in any of these functions. These methods are most useful when the proximal operators of f, g, and h can be computed efficiently.

$$\underset{x}{\text{minimize}} \quad f(x) + g(Ax) + h(x) \tag{2.1}$$

Depending on the properties of the functions and linear operator A, different operator-splitting schemes can be applied. If h is smooth, Condat–Vũ can be used [55, 188]. If $A = I$ and h is smooth, Davis–Yin splitting can be applied [63]. If $h = 0$, PDHG can be applied

[44]. If $h = 0$ and $A = I$, Douglas–Rachford splitting can be applied [70]. If $h = 0$, $A = I$, and either f or g is smooth, forward–backward splitting can be applied [54]. If $g = h = 0$, the proximal point method can be applied [163]. If $g = h = 0$ and additionally f is smooth, gradient descent can be applied. Strictly speaking, these last two algorithms are not operator-splitting methods because only one operator is involved. Although the specific algorithm depends on the problem structure, operator-splitting methods share common behaviors.

Operator-splitting methods are typically classified as first-order methods, though this is a slight oversimplification. Although they do not use Hessian information of the overall objective, evaluating a proximal operator involves solving a minimization problem, and for some functions, this implicitly uses second-order information. For example, the proximal operator of a quadratic function requires solving a linear system.

In practice, operator-splitting methods behave like first-order methods: they have a cheap per-iteration cost (assuming the proximal operators are easy to evaluate), are easy to warm-start, and when applied properly avoid forming and solving linear systems, allowing them to scale much better than IPMs. On the other hand, they are extremely sensitive to problem conditioning, require careful tuning of algorithm parameters (such as step-sizes) for best convergence, and struggle to converge to high-accuracy solutions. On ill-conditioned problems, it is not uncommon to see them fail to return a solution at all. Heuristics like adaptive step-size selection and Ruiz equilibration can mitigate some of these issues but do not eliminate them. These properties make operator-splitting methods best suited to problems too large for IPMs.

Operator-splitting methods can be viewed as meta-algorithms or algorithm templates: depending on how the original optimization problem is specified (i.e., how f, g, and h are chosen in Problem (2.1)), the same splitting method can yield different algorithms with different properties. To make this concrete, we will examine the algorithm underpinning OSQP [177], which is ADMM applied to a specific reformulation of a quadratic program. We first present the ADMM setup before specializing to the QP case.

The problem ADMM solves is

$$\begin{aligned} \underset{x,z}{\text{minimize}} \quad & f(x) + g(z) && \text{(2.2a)} \\ \text{subject to} \quad & Ax + Bz = c. && \text{(2.2b)} \end{aligned}$$

The ADMM algorithm is equivalent to applying Douglas–Rachford splitting to the dual of Problem (2.2) [166] and is

$$\begin{aligned} x_{k+1} &= \underset{x}{\text{argmin}}\ \mathcal{L}_\rho(x, z_k, u_k) && \text{(2.3a)} \\ z_{k+1} &= \underset{z}{\text{argmin}}\ \mathcal{L}_\rho(x_{k+1}, z, u_k) && \text{(2.3b)} \\ u_{k+1} &= u_k + \rho(Ax_{k+1} + Bz_{k+1} - c), && \text{(2.3c)} \end{aligned}$$

where $\mathcal{L}_\rho(x, z, u) = f(x) + g(z) + u^\top(Ax + Bz - c) + (\rho/2)\|Ax + Bz - c\|^2$ is the augmented Lagrangian of (2.2).

OSQP solves convex quadratic programs in the following standard form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^\top Px + q^\top x \\ \text{subject to} \quad & l \leq Ax \leq u. \end{aligned} \tag{2.4}$$

The obvious way to solve this problem is to apply ADMM to the following:

$$\begin{aligned} \underset{x,z}{\text{minimize}} \quad & \underbrace{\frac{1}{2}x^\top Px + q^\top x}_{f(x)} + \underbrace{\mathcal{I}_\mathcal{C}(z)}_{g(z)} \\ \text{subject to} \quad & Ax - z = 0 \end{aligned} \tag{2.5}$$

where $\mathcal{I}_\mathcal{C}(z)$ is the indicator function of $\mathcal{C} = [l, u]$ defined as

$$\mathcal{I}_{\mathcal{C}}(z) = \begin{cases} 0, & \text{if } z \in \mathcal{C} \\ \infty, & \text{otherwise.} \end{cases}$$

Applying ADMM to (2.5) yields the algorithm

$$x_{k+1} = (P + \rho A^\top A)^{-1} \left[\rho A^\top (z_k - \rho^{-1} y_k) - q\right] \tag{2.6a}$$

$$z_{k+1} = \Pi_{\mathcal{C}}(\alpha A x_{k+1} + (1-\alpha) z_k + \rho^{-1} y_k) \tag{2.6b}$$

$$y_{k+1} = y_k + \rho(\alpha A x_{k+1} + (1-\alpha) z_k - z_{k+1}) \tag{2.6c}$$

where $\alpha \in (0, 2)$ is an overrelaxation parameter.

However, this requires that $P + \rho A^\top A$ be invertible, which imposes additional assumptions on P and A. To avoid making such assumptions, the authors of OSQP introduce an equivalent reformulation of Problem (2.4)

$$\begin{aligned} \underset{\tilde{x}, \tilde{z}, x, z}{\text{minimize}} \quad & \underbrace{\frac{1}{2}\tilde{x}^\top P \tilde{x} + q^\top \tilde{x} + \mathcal{I}_{A\tilde{x}=\tilde{z}}(\tilde{x}, \tilde{z})}_{f(\tilde{x}, \tilde{z})} + \underbrace{\mathcal{I}_{\mathcal{C}}(z)}_{g(x,z)} \\ \text{subject to} \quad & (\tilde{x}, \tilde{z}) - (x, z) = 0. \end{aligned} \tag{2.7}$$

After some simplification, ADMM applied to Problem (2.7) is

$$(\tilde{x}_{k+1}, \nu_{k+1}) \leftarrow \text{Solve linear system } \begin{bmatrix} P + \sigma I & A^\top \\ A & -\rho^{-1} I \end{bmatrix} \begin{bmatrix} \tilde{x}_{k+1} \\ \nu_{k+1} \end{bmatrix} = \begin{bmatrix} \sigma x_k - q \\ z_k - \rho^{-1} y_k \end{bmatrix} \tag{2.8a}$$

$$\tilde{z}_{k+1} \leftarrow z_k + \rho^{-1}(\nu_{k+1} - y_k) \tag{2.8b}$$

$$x_{k+1} \leftarrow \alpha \tilde{x}_{k+1} + (1 - \alpha) x_k \tag{2.8c}$$

$$z_{k+1} \leftarrow \Pi_{\mathcal{C}} \left[\alpha \tilde{z}_{k+1} + (1 - \alpha) z_k + \rho^{-1} y_k \right] \tag{2.8d}$$

$$y_{k+1} \leftarrow y_k + \rho(\alpha \tilde{z}_{k+1} + (1 - \alpha) z_k - z_{k+1}), \tag{2.8e}$$

where $\rho, \sigma > 0$ are ADMM penalty parameters. Notice that this linear system is always invertible, since $\sigma, \rho > 0$. Although Algorithms (2.6) and (2.8) are both applications of ADMM, the latter is applied to a reformulation whose linear system is invertible without any assumptions on the problem data. This illustrates how the choice of problem formulation, not just the splitting scheme, leads to different algorithms with different properties.

Some good references on operator-splitting methods are [31, 149, 166].

2.1.3 Active set methods

Active set methods are a class of algorithms for solving quadratic programs of the form

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \frac{1}{2} x^\top P x + c^\top x \\ \text{subject to} & Ax \leq b. \end{array} \tag{2.9}$$

The simplex method for linear programming is a well-known example of an active set method. Active set methods take advantage of the fact that equality constrained problems are easier to solve than inequality constrained problems, since equality-constrained QPs reduce to a single linear system solve. A constraint $a_i^\top x \leq b_i$ is said to be active at x if $a_i^\top x = b_i$. Suppose we knew *a priori* the set of constraints $\mathcal{I}$ that will be active at the

optimal solution. We could then rewrite Problem (2.9) equivalently as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2} x^\top P x + c^\top x \\ \text{subject to} \quad & A_{\mathcal{I}} x = b_{\mathcal{I}}. \end{aligned} \tag{2.10}$$

where we discard all the inactive constraints. This problem can then be solved via the following linear system, obtained from the KKT conditions for Problem (2.10).

$$\begin{bmatrix} P & A_{\mathcal{I}}^\top \\ A_{\mathcal{I}} & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -c \\ b_{\mathcal{I}} \end{bmatrix}$$

However, the set of active constraints at the optimal solution is not known *a priori*. Active set methods therefore start with a guess for the set of active constraints, called the *working set*, and solve the equality constrained QP corresponding to this working set. Having solved the equality-constrained QP, the algorithm then takes a step from the current iterate toward the subproblem solution. If this step would violate a constraint not in the working set, the step is shortened so that the iterate reaches the boundary of the violated constraint, and that constraint is added to the working set. This ensures that all iterates remain primal feasible. If the full step can be taken without violating any constraints, the algorithm checks the dual variables of the subproblem solution. If all are nonnegative, the current iterate is optimal and the algorithm terminates. Otherwise, the constraint with the most negative multiplier is removed from the working set. Iterating in this way, the algorithm eventually arrives at the optimal active set and corresponding optimal solution. This is a description of a *primal* active set method. Dual active set methods also exist. They maintain dual feasibility at each iteration and terminate when the primal iterate becomes feasible.

These methods warm-start extremely well. Given a good guess for the optimal active set (from a previously solved problem), they can converge in a few iterations, which makes them very attractive for model-predictive control where the optimal active set changes very

little between two consecutive solves. For example, an active set method was used to solve the optimization problems arising in MPC for the Atlas humanoid robot [112]. Active set methods can also obtain high-accuracy solutions.

However, active set methods have several limitations. They are restricted to linear and quadratic programs with linear inequality constraints, and cannot handle conic constraints such as those in second-order cone programs. Without a good initial guess for the active set, they can require many iterations to converge, since each iteration only adds or removes a single constraint from the working set. They also scale poorly to large problems, as identifying the correct active set is a combinatorial problem. Finally, they struggle with degenerate constraints (when multiple constraints activate simultaneously) and are prone to numerical issues, since refactoring the KKT system from scratch at each iteration would be prohibitively expensive, so implementations instead apply rank-one updates to the existing factorization, which can accumulate numerical errors over many iterations.

Given their strong warm-starting capabilities and limited scalability, active set methods are best suited to small, sequential problems like MPC. A good reference on active set methods is [143, Chapter 16].

2.1.4 Algorithm recommendation

In my opinion, interior-point methods should be used by default due to their robustness and because they require little to no tuning. Operator-splitting methods should be considered only if problems are too large to solve with an IPM (i.e., when you encounter out-of-memory issues or factorizations are too slow). Active set methods should be considered only if you are solving a sequence of related small QPs, such as in MPC.

2.1.5 The role of theory and convergence rates

Before closing this section, it is worth making a few remarks regarding algorithm convergence rates. For optimization algorithms, the convergence statement is typically stated in terms of *worst-case* convergence rates of algorithms. For example, gradient descent (GD) on a

smooth convex function satisfies $f(x_k) - f(x_\star) = \mathcal{O}(1/k)$, meaning that on all smooth convex functions, the function value suboptimality converges at $\mathcal{O}(1/k)$. Nesterov's accelerated gradient (NAG) method achieves the faster rate $f(x_k) - f(x_\star) = \mathcal{O}(1/k^2)$ [141]. However, this faster worst-case rate does not mean that NAG outperforms GD on all smooth minimization problems, since the worst-case rate is an upper bound taken over an entire function class, and any particular problem may converge much faster than the worst-case suggests. More generally, an accelerated variant with a better worst-case rate will not necessarily outperform its base variant on every problem instance [166, Chapter 12].

This is a recurring theme in optimization: algorithms tend to perform better in practice than their worst-case rates suggest. This can be seen in the extreme case of Mehrotra's predictor-corrector [135], a heuristic for primal-dual interior-point methods that does not have a convergence guarantee. However, this scheme has extremely good practical performance and tends to outperform short-step and long-step IPMs which do have convergence guarantees [192]. Its strong practical performance has made it the basis for many IPM implementations such as ECOS [69], CLARABEL [92], and QOCO [46].

Just because an algorithm has the best worst-case theoretical convergence rate does not mean it will be the best algorithm for your specific problem. In my opinion, convergence rates are not particularly important when choosing an algorithm for a specific problem class. The best thing to do is try many algorithms and see what works best. This is not to say theory is useless. Convergence theory is interesting in its own right and can deepen our understanding of mechanisms like acceleration. But it should not be the deciding factor when you are looking for a good algorithm to solve your problem.

2.2 Software

Now I will discuss some solvers I have used. If I do not mention a particular solver, it means I do not have enough experience with it to make reliable comments.

2.2.1 Gurobi

GUROBI is a commercial solver that can solve LPs, QPs, SOCPs, mixed-integer LPs, mixed-integer QPs, and mixed-integer nonlinear programming problems. It also has infeasibility detection. They do provide a free license for academic use. I would say the specialty of GUROBI is in LPs and MILPs.

GUROBI has many different algorithms depending on the problem it solves. For linear programs, a primal simplex, dual simplex, and interior-point method are available. I have found GUROBI to be the best LP solver and mixed-integer solver I have used by a wide margin. However, it does struggle to solve extremely large LPs, and this can be observed in [9]. That said, in GUROBI 13.0, there are CPU and GPU primal-dual hybrid gradient (PDHG) implementations that allow it to solve large-scale linear programs.

I have also found GUROBI to be very robust on QPs although it can be much slower than other open-source solvers such as CLARABEL and QOCO on small- to medium-sized QPs. I have also observed that GUROBI can struggle on SOCPs. This can be seen in the group lasso and total-variation denoising problems in Table 4.1.

2.2.2 Mosek

MOSEK is a commercial solver that can solve LPs, QPs, SOCPs, conic programs with exponential and power cones, SDPs, mixed-integer LPs, mixed-integer QPs, and mixed-integer conic problems. MOSEK detects infeasibility by solving a homogeneous embedding of the original problem [156]. They also provide a free license for academic use. For continuous conic optimization, MOSEK implements a primal-dual interior-point method [60]. I would say the specialty of MOSEK is conic optimization (SOCPs, exponential and power cones, and SDPs). MOSEK is by far the best solver I have ever used for SDPs.

However, I have noticed MOSEK struggles on QPs and SOCPs with a quadratic objective function. Since MOSEK's standard form can only handle linear objectives, the quadratic objective must be replaced with an epigraph variable and a second-order cone constraint. It

appears that this reformulation results in some performance degradation as can be seen in the numerical results sections of [46] and [92].

2.2.3 ECOS

ECOS is an open-source solver written in C that can solve LPs, QPs, SOCPs, and conic programs with exponential cones and implements a primal-dual interior-point method [69, 173]. It also has infeasibility detection by using a homogeneous embedding. I have found it to be quite robust on linear programs and SOCPs with linear objectives.

However, like MOSEK, since it can only handle linear objectives, quadratic objectives have to be replaced with an epigraph variable and a second-order cone constraint. I have seen that ECOS struggles on quadratic programs and SOCPs with quadratic objectives (see numerical results of [46] and [92]). ECOS also does not appear to be maintained, so if bugs are found, they will likely not be patched.

2.2.4 Clarabel

CLARABEL is an open-source solver written in Rust that can solve LPs, QPs, SOCPs, conic programs with exponential and power cones, and SDPs [92]. Notably, it can directly handle quadratic objectives, instead of the epigraph reformulation needed by ECOS and MOSEK. It also detects infeasibility by using a homogeneous embedding. I have found CLARABEL to be very robust on LPs, QPs, and SOCPs. CLARABEL implements a primal-dual interior-point method. I have not tried CLARABEL on problems with exponential or power cones. CLARABEL is the default conic solver in CVXPY for good reason, as it is very robust and also does not suffer from the same performance degradation that MOSEK and ECOS experience with quadratic objectives. CLARABEL also has a GPU backend to solve large-scale optimization problems, but this is in a separate Julia repository [53].

I have seen CLARABEL struggle on SDPs arising from the performance estimation problem [182], which MOSEK is able to solve with no issue. Also, although CLARABEL's GPU backend can be called from CVXPY, it is very involved to set up the Python-Julia interface and due to

the just-in-time compilation of Julia, the first solve is extremely slow. I have also observed that the GPU backend can be significantly slower (up to 3× slower) than QOCO's GPU backend on some problems. This can be seen in the largest group lasso and Huber regression problems in Table 4.1.

2.2.5 QOCO

Disclaimer: Of course my take on QOCO *will be a bit biased since I developed it, but I will try to be objective.*

QOCO is an open-source solver written in C that implements a primal-dual interior-point method and can solve LPs, QPs, and SOCPs [46]. QOCO also has a custom solver generator called QOCOGEN which generates a custom solver written in C that leverages custom linear algebra to exploit the sparsity structure of the problem and results in a nearly 4× speedup over QOCO on small optimization problems (up to 10^4 nonzeros). QOCO also has a GPU-accelerated backend which allows it to solve large-scale optimization problems (with up to 10^8 nonzeros) [48]. QOCO is very robust and tends to be the fastest solver on small- to medium-sized problems (up to $\approx 10^5$ nonzeros) with its CPU backend. With its GPU backend it tends to be the fastest solver, or comparable with CLARABEL's GPU implementation, on large problems (up to $10^5 - 10^8$ nonzeros). Additionally, the GPU backend is also written in C, so it is very easy to switch between using the CPU and GPU backend from Python and CVXPY unlike CLARABEL. I have also observed that on some problems, QOCO's GPU backend significantly outperforms CLARABEL's GPU backend (up to 3× faster).

QOCO does not have infeasibility detection, since it does not solve a homogeneous embedding of the original problem, as ECOS, MOSEK, and CLARABEL do. However, if QOCO is given an infeasible or unbounded problem, it will return the status `QOCO_NUMERICAL_ERROR`, which can serve as an indication of infeasibility or unboundedness, though not a formal certificate.

2.2.6 PIQP

PIQP is an open-source solver written in C++ that can solve LPs and QPs and implements an algorithm that combines an IPM with the proximal method of multipliers [171]. This algorithm behaves like an IPM in the sense that it returns high-accuracy solutions and is robust to ill-conditioning. PIQP detects infeasibility by checking for divergence of either the primal or dual variables. PIQP also has a backend that exploits the sparsity structure of multistage problems to accelerate the linear system solve at each iteration [172]. PIQP is extremely robust, very fast, and my go-to solver for QPs. It is very clear that a lot of thought was put into the code to extract maximum performance from the CPU.

There are very few downsides to PIQP. The main one is that since it factors a linear system at every iteration, it will not scale well to large-scale optimization problems.

2.2.7 OSQP

OSQP is an open-source solver written in C that can solve LPs and QPs and implements ADMM [177]. OSQP detects infeasibility by checking certificates constructed from the differences of consecutive iterates [12]. OSQP is under very active development and well supported. New features are always being added, such as differentiating the solution with respect to problem parameters, and an FPGA implementation [189]. OSQP also has a GPU-accelerated backend [170]. As it implements an operator-splitting method, it can also effectively warm-start, making it attractive for applications such as MPC.

However, OSQP inherits the usual drawbacks of operator-splitting methods: it struggles to return high-accuracy solutions and is sensitive to problem conditioning, even with Ruiz equilibration enabled. I have had several poor experiences with the accuracy of OSQP and prefer to use PIQP.

It is also unclear whether OSQP can scale to solve optimization problems larger than what can be solved with GPU-accelerated IPMs such as CLARABEL and QOCO. In [170], optimization problems with up to 10^8 nonzeros are solved, however, GPU-accelerated IPMs

can also solve problems up to this size. Additionally, OSQP factors a linear system each time ρ is updated, but between updates, the factorization is reused. For large problems, block elimination is performed on the linear system to arrive at a condensed, positive-definite linear system which is then solved with the preconditioned conjugate gradient method (PCG). However, the performance of PCG is highly dependent on the conditioning of the condensed linear system and must be performed at every ADMM iteration (usually thousands are required), so it is unclear whether OSQP can outperform GPU-accelerated IPMs on large-scale optimization problems. Therefore, even if I had a large-scale QP, I would first try a GPU-accelerated IPM.

2.2.8 SCS

SCS is an open-source solver written in C that can solve LPs, QPs, SOCPs, conic programs with exponential and power cones and SDPs. It implements Douglas–Rachford splitting [144] and detects infeasibility by solving a homogeneous embedding of the original problem. SCS also has a GPU-accelerated backend.

Like OSQP, SCS inherits the drawbacks of operator-splitting methods described above, and I have had similarly poor experiences with its accuracy. It also requires linear system solves, which are handled with PCG for large problems, so the same scaling concerns apply. Therefore, even for a large-scale conic problem, I would first try a GPU-accelerated IPM.

2.2.9 PDLP

PDLP is an open-source solver written in C++ for massive-scale LPs (up to 10^9 nonzeros) that implements PDHG (an operator-splitting method) with enhancements [9]. PDLP detects infeasibility by checking for divergence of its iterates [10]. Unlike OSQP and SCS, PDLP does not require the solution of a linear system and only requires matrix-vector products. If you have extremely large linear programs that cannot be solved with an IPM (even with GPU acceleration), this is the only practical solver to use. There are also GPU-accelerated implementations of PDLP [121].

PDLP inherits the same operator-splitting drawbacks as OSQP and SCS, but its ability to avoid linear system solves entirely makes it the only practical option for extremely large LPs that cannot be solved with an IPM, even with GPU acceleration. I would not use PDLP outside these problems.

Chapter 3

QOCO: A QUADRATIC OBJECTIVE CONIC OPTIMIZER WITH CUSTOM SOLVER GENERATION

Second-order cone programs (SOCPs) with quadratic objective functions are common in optimal control and other fields. Most SOCP solvers which use interior-point methods are designed for linear objectives and convert quadratic objectives into linear ones via slack variables and extra constraints, despite the computational advantages of handling quadratic objectives directly. In applications like model-predictive control and online trajectory optimization, these SOCPs have known sparsity structures and require rapid solutions. When solving these problems, most solvers use sparse linear algebra routines, which introduce computational overhead and hinder performance. In contrast, custom linear algebra routines can exploit the known sparsity structure of problem data and be significantly faster. This work makes two key contributions: (1) the development of QOCO, an open-source C-based solver for quadratic objective SOCPs, and (2) the introduction of QOCOGEN, an open-source custom solver generator for quadratic objective SOCPs, which generates a solver written in C that leverages custom linear algebra. Both implement a primal-dual interior-point method with Mehrotra's predictor-corrector. On the benchmark problems we run, QOCO is more robust than many commonly used solvers and is faster on small- to medium-sized problems. Additionally, solvers generated by QOCOGEN are significantly faster than QOCO and are free of dynamic memory allocation making them amenable for use on embedded systems.

3.1 Introduction

We consider the quadratic objective second-order cone program (SOCP)

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \frac{1}{2}x^\top Px + c^\top x \\ \text{subject to} & Gx \preceq_{\mathcal{K}} h \\ & Ax = b, \end{array} \tag{3.1}$$

with optimization variable $x \in \mathbb{R}^n$. The cost is defined by the positive semidefinite matrix $P = P^\top \succeq 0$ and vector $c \in \mathbb{R}^n$. The constraints are defined by matrices $A \in \mathbb{R}^{p\times n}$ and $G \in \mathbb{R}^{m\times n}$ and vectors $b \in \mathbb{R}^p$ and $h \in \mathbb{R}^m$. The generalized inequality $\preceq_{\mathcal{K}}$ denotes membership in a closed, proper cone $\mathcal{K}$, i.e. $h - Gx \in \mathcal{K}$. We restrict $\mathcal{K}$ to be the Cartesian product

$$\mathcal{K} := \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_K,$$

where $\mathcal{C}_i$ is either the non-negative orthant or a second-order cone, which are self-dual. Each cone $\mathcal{C}_i$ corresponds to a subset of constraints, inducing a partition on G and h, i.e., $h_i - G_i x \in \mathcal{C}_i$ for $i = 1, \ldots, K$. Throughout this work, we assume that Problem 3.1 is **feasible** and has a **bounded** optimal objective.

Problem 3.1 appears in various applications, including model predictive control (MPC) [28], network flow optimization [82], portfolio optimization [129, 118], robust optimization [19, 20], and filter design [119], among others. Additionally, the solution of SOCPs is used as a subroutine in algorithms for nonconvex optimization such as sequential convex programming (SCP) [126, 74, 117, 199].

Problem 3.1 can be reformulated as a SOCP with a linear objective by introducing a slack variable t,

$$\begin{aligned} \underset{x,t}{\text{minimize}} \quad & t + c^\top x \\ \text{subject to} \quad & \left\| \begin{bmatrix} t - \frac{1}{2} \\ P^{1/2} x \end{bmatrix} \right\|_2 \leq t + \frac{1}{2} \\ & Gx \preceq_{\mathcal{K}} h \\ & Ax = b. \end{aligned}$$

If P is large (has many rows and columns), then computing $P^{1/2}$ can be prohibitively expensive, making this reformulation impractical. Solving this reformulation can also be slow if the matrix square root $P^{1/2}$ has significantly more nonzero elements than P. Thus, it is advantageous for a SOCP solver to natively handle quadratic objective functions without resorting to the linear objective reformulation.

Some SOCP solvers that can solve Problem 3.1 directly include CLARABEL [92], GUROBI [93], COSMO [84], and SCS [144]. However, COSMO and SCS implement operator-splitting methods. These methods are preferred for large-scale problems due to their low per-iteration cost but can struggle with problem scaling, conditioning, and achieving high-accuracy solutions [183]. For modestly sized SOCPs, an interior-point method (IPM) is preferred, due to its robustness to scaling and conditioning of the problem, and ability to converge to high accuracy solutions [166, Chapter 1]. One of the few open-source IPMs that can directly solve Problem 3.1 is CLARABEL. In addition to solving SOCPs, CLARABEL can also handle semidefinite programs, as well as problems involving exponential cones, power cones, and generalized power cones. Additionally, it supports infeasibility detection by solving a homogeneous embedding of Problem 3.1. However, CLARABEL is written in Rust. Although Rust is memory-safe due to its ownership system and can be used in embedded systems, its ecosystem is far less mature than that of C, and for legacy systems, such as aerospace software, C is still preferred.

Many engineering applications including linear MPC, nonlinear MPC [160], portfolio backtesting, sequential quadratic programming (SQP) [27, 143], and SCP, require solving

SOCPs with a fixed, known sparsity structures in P, A, G and a constant cone $\mathcal{K}$. These problems often arise in real-time settings, where computational efficiency is critical. For example, sequential convex programming (SCP) iteratively solves a sequence of SOCPs with quadratic objectives and identical sparsity patterns to find stationary points of nonconvex problems. SCP has been widely applied in aerospace trajectory optimization, where real-time performance is essential. Notable applications include powered-descent guidance [50, 101, 181], in-space rendezvous [23, 47], and hypersonic entry guidance [132, 133]. SCP is also the solution method used by NASA's SPLICE program for lunar landing guidance [41, 100, 136, 162].

Most general-purpose SOCP solvers, such as CLARABEL [92], COSMO [84], ECOS [69], GUROBI [93], MOSEK [11], and SCS rely on sparse linear algebra routines. Although sparse linear algebra allows the aforementioned solvers to solve SOCPs regardless of their sparsity structures, it can be quite slow due to the extra overhead of determining where the nonzero elements are before performing floating point operations, which leads to more CPU instructions being issued, more memory accesses, and more cache misses. However, when the sparsity structure of the problem matrices is known beforehand, it is possible to generate a custom solver that implements customized linear algebra routines tailored to the sparsity structure of the problem. Specifically, in sparse linear algebra matrices are stored as three arrays: a row/column index array, column/row pointer array, and data array. Before performing a floating point operation on a nonzero element in the matrix, sparse linear algebra routines must first index into the pointer and index arrays to determine the location of the element in the data array. In customized linear algebra, the index and pointer arrays do not exist, and we directly index into the data array, eliminating the overhead of sparse linear algebra, resulting in significantly faster operations. We discuss this further in Section 3.4.1.

A *custom solver generator* is a tool that takes the sparsity structure of an optimization problem as an input and outputs a solver (typically in C) optimized for that specific sparsity structure without relying on sparse linear algebra. Two notable custom solver generators are CVXGEN [131] and BSOCP [75, 76]. CVXGEN, which is academically licensed but not open-

source, solves quadratic programs (QPs) rather than SOCPs (i.e. $\mathcal{K}$ in Problem 3.1 is the non-negative orthant) and BSOCP, which is neither academically licensed nor open source, solves linear objective SOCPs (i.e. $P = 0$ in Problem 3.1). Both have demonstrated substantial speed improvements over general-purpose solvers and have had a significant impact on aerospace trajectory optimization. For instance, after the development of lossless convexification [3, 4], BSOCP was flight-tested on Masten's Xombie vehicle to validate G-FOLD, a powered-descent guidance algorithm based on lossless convexification [169]. Later, CVXGEN was used by SpaceX for landing their Falcon 9 boosters [26].

A seemingly related but distinct tool is CVXPYGEN [168]. It leverages CVXPY to parse the optimization problem to the standard form for solvers and generates C code for updating problem data and retrieving solutions. CVXPYGEN then wraps a backend solver without (or with very little) modification to its source code. Because it customizes only the parsing and solution retrieval while relying entirely on an existing solver, we do not classify CVXPYGEN as a custom solver generator.

3.1.1 Contribution

This work makes two key contributions: (1) QOCO, an open-source C-based solver for quadratic objective SOCPs, and (2) QOCOGEN, an open-source custom solver generator for quadratic objective SOCPs which generates a custom solver written in C [1]. QOCO and solvers generated by QOCOGEN implement the same algorithm as CLARABEL, a primal-dual interior point method with Mehrotra's predictor-corrector, but directly solve Problem 3.1 rather than a homogeneous embedding.

We demonstrate that QOCO is more robust than most commercial and open-source solvers, and is faster on small- to medium-sized problems. We also show that $\text{QOCO}_{\text{custom}}$ (the name of a solver generated by QOCOGEN) is significantly faster than QOCO. Both solvers are easy to use since QOCO and QOCOGEN can be called from CVXPY [5, 67] and CVXPYGEN [168]

[1] Both are available on GitHub at https://github.com/qoco-org

respectively, allowing users to formulate optimization problems in a natural way following from math rather than manually converting the problem to the solver-required standard form. Additionally, QOCO can be called from C/C++, Matlab, and Python, and QOCOGEN can be called from Python. Both implement a primal-dual interior point method with Mehrotra's predictor-corrector [135]. However, they both use a variety of numerical enhancements to quickly and robustly solve the linear system that arises in the step direction computation. Specifically, we use an $LDL^\top$ factorization along with the Approximate Minimum Degree (AMD) [7, 64] heuristic to permute the coefficient matrix, minimizing fill-in for the factor L. We also apply static and dynamic regularization to the coefficient matrix to ensure that the matrix is invertible and the factorization succeeds. After solving the linear system, we apply iterative refinement to ensure that the original, unregularized system was solved to high accuracy, further enhancing numerical stability and robustness.

3.1.2 Outline

Section 3.2 introduces the primal-dual interior point method implemented in QOCO and QOCOGEN. Section 3.3 explains the techniques used to ensure a stable factorization of the KKT matrix, which is essential to compute the step direction. Section 3.4 discusses the advantages of custom linear algebra over sparse linear algebra, and how QOCOGEN generates custom solvers which exploit the known sparsity structure of problem data. Finally, Section 3.5 presents extensive numerical results comparing QOCO and $\text{QOCO}_{\text{custom}}$ to existing solvers.

3.1.3 Notation

We use (a, b) to denote the concatenation of vectors $a \in \mathbb{R}^a$ and $b \in \mathbb{R}^b$, resulting in a vector in $\mathbb{R}^{a+b}$. The non-negative orthant in $\mathbb{R}^l$ is denoted by $\mathbb{R}^l_+ = \{u \in \mathbb{R}^l \mid u_i \geq 0\}$, and for any $u \in \mathbb{R}^l_+$, we use u_i to refer to its i^{th} element. The q-dimensional second-order cone is defined as $\mathcal{Q}^q = \{(u_0, u_1) \in \mathbb{R} \times \mathbb{R}^{q-1} \mid \|u_1\|_2 \leq u_0\}$, where any $u \in \mathcal{Q}^q$ is written as $u = (u_0, u_1)$ with $u_0 \in \mathbb{R}$ and $u_1 \in \mathbb{R}^{q-1}$. Finally, when a vector x lies in the Cartesian product of cones, i.e., $x \in \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_K$, we denote x_i as its i^{th} subvector belonging to cone $\mathcal{C}_i$.

3.2 Primal-dual interior point method

In this section, we describe the primal-dual interior point algorithm we implement in QOCO and QOCOGEN, as outlined in Algorithm 1. This algorithm is equivalent to the `coneqp` algorithm outlined in [186] and follows a derivation similar to those presented in [21, Chapter 6], [6, 192].

If strong duality holds for Problem 3.1, the optimal primal-dual solution (x^*, s^*, y^*, z^*) satisfies the Karush-Kuhn-Tucker (KKT) conditions [32, Chapter 5], which can be written as

$$Px + c + A^\top y + G^\top z = 0 \tag{3.2a}$$

$$Ax = b \tag{3.2b}$$

$$Gx + s = h \tag{3.2c}$$

$$s_i^\top z_i = 0 \text{ for } i = 1, \dots K \tag{3.2d}$$

$$(s, z) \in \mathcal{K} \times \mathcal{K}. \tag{3.2e}$$

The primal-dual interior point method applies a modified Newton's method to Equations (3.2a) - (3.2d), and a line search to satisfy Equation (3.2e). The modifications to Newton's method correct for linearization errors in the Newton step and bias the search directions towards the interior of $\mathcal{K}$. This prevents the iterates from prematurely approaching the boundary of the cone $\mathcal{K}$, enabling longer steps without violating Equation (3.2e). It also ensures iterates do not converge to spurious solutions that satisfy Equations (3.2a) - (3.2d), but not Equation (3.2e) [192].

3.2.1 Central path derivation

To understand how the algorithm biases the search directions towards the interior of $\mathcal{K}$, we first introduce the concept of the *central path*.

Problem 3.1 can be equivalently rewritten as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^\top Px + c^\top x + \mathcal{I}_\mathcal{K}(h - Gx) \\ \text{subject to} \quad & Ax = b, \end{aligned}$$

where $\mathcal{I}_\mathcal{K}$, the indicator function of the cone $\mathcal{K}$, is defined as

$$\mathcal{I}_\mathcal{K}(u) = \begin{cases} 0, & \text{for } u \in \mathcal{K} \\ \infty, & \text{otherwise.} \end{cases}$$

We then replace the indicator function, $\mathcal{I}_\mathcal{K}(u)$, with a smooth barrier function, $\phi_\mathcal{K}(u)$, which approaches ∞ as its argument approaches the boundary of the cone.

We use the barrier function

$$\phi_\mathcal{K}(u) = \sum_{i=1}^{K} \phi_i(u_i),$$

where ϕ_i is the barrier function for $\mathcal{C}_i$. The barrier function for the non-negative orthant and the second-order cone are

$$\phi_i(u) = \begin{cases} -\sum_{j=1}^{l} \log u_j, & \text{for } \mathcal{C}_i = \mathbb{R}^l_+ \\ -(1/2)\log(u_0^2 - u_1^\top u_1), & \text{for } \mathcal{C}_i = \mathcal{Q}^q, \end{cases}$$

where log is the natural logarithm.

Their gradients are

$$\nabla\phi_i(u) = \begin{cases} (-1/u_1, \ldots, -1/u_l), & \text{for } \mathcal{C}_i = \mathbb{R}^l_+ \\ -(u^\top Ju)^{-1}Ju, & \text{for } \mathcal{C}_i = \mathcal{Q}^q, \end{cases} \tag{3.3}$$

where

$$J = \begin{bmatrix} 1 & 0 \\ 0 & -I_{q-1} \end{bmatrix}.$$

After replacing the indicator function with the barrier function, we obtain

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^\top Px + c^\top x + \tau \sum_{i=1}^{K} \phi_i(h_i - G_i x) \\ \text{subject to} \quad & Ax = b, \end{aligned} \tag{3.4}$$

where $\tau > 0$ is a scalar. Denoting the optimal solution of Problem 3.4 as $x^*(\tau)$, it can be shown that as $\tau \to 0$, $x^*(\tau) \to x^*$ [21].

The KKT condition for Problem 3.4 are

$$\begin{aligned} Px + c + A^\top y - \tau \sum_{i=1}^{K} G_i^\top \nabla \phi_i(h_i - G_i x) = 0 \\ Ax = b \\ h - Gx \in \mathcal{K}. \end{aligned}$$

If we define $s = h - Gx$ and $z_i = -\tau \nabla \phi_i(h_i - G_i x)$ for $i = 1, \dots K$, we can rewrite the KKT conditions as

$$Px + c + A^\top y + G^\top z = 0 \tag{3.5a}$$

$$Ax = b \tag{3.5b}$$

$$Gx + s = h \tag{3.5c}$$

$$z_i = -\tau \nabla \phi_i(s_i) \text{ for } i = 1, \dots K \tag{3.5d}$$

$$(s, z) \in \mathcal{K} \times \mathcal{K}, \tag{3.5e}$$

where the condition $z \in \mathcal{K}$ arises because if $u \in \mathcal{K}$, then $-\nabla \phi(u) \in \mathcal{K}$ [186].

It is desirable to write Equation (3.5d), in a form where s_i and z_i appear symmetrically.

To this end, we define the *Jordan product* [6, 81, 186], a commutative and linear operation, for the non-negative orthant and second-order cone as

$$u_i \circ v_i = \begin{cases} (u_{i1}v_{i1}, \ldots, u_{il}v_{il}), & \text{for } \mathcal{C}_i = \mathbb{R}^l_+ \\ (u_i^\top v_i, u_{i0}v_{i1} + v_{i0}u_{i1}), & \text{for } \mathcal{C}_i = \mathcal{Q}^q, \end{cases}$$

and the Jordan product for $\mathcal{K}$ as

$$u \circ v = (u_1 \circ v_1, u_2 \circ v_2, \ldots, u_K \circ v_K), \tag{3.6}$$

where $u_i, v_i \in \mathcal{C}_i$.

The identity element $\boldsymbol{e}_i$ for cone $\mathcal{C}_i$ is defined as

$$\boldsymbol{e}_i = \begin{cases} (1, 1, \ldots, 1), & \text{for } \mathcal{C}_i = \mathbb{R}^l_+ \\ (1, 0, \ldots, 0), & \text{for } \mathcal{C}_i = \mathcal{Q}^q, \end{cases}$$

and the identity element for $\mathcal{K}$ is

$$\boldsymbol{e} = (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_K). \tag{3.7}$$

Taking the Jordan product with s_i on both sides of Equation (3.5d) and substituting Equation (3.3), we obtain

$$Px + c + A^\top y + G^\top z = 0 \tag{3.8a}$$

$$Ax = b \tag{3.8b}$$

$$Gx + s = h \tag{3.8c}$$

$$s_i \circ z_i = \tau \boldsymbol{e}_i \text{ for } i = 1, \ldots K \tag{3.8d}$$

$$(s, z) \in \mathcal{K} \times \mathcal{K}. \tag{3.8e}$$

We then rewrite Equation (3.8d) by stacking s_i and z_i and using Equations (3.6) and (3.7) to obtain

$$Px + c + A^\top y + G^\top z = 0 \tag{3.9a}$$
$$Ax = b \tag{3.9b}$$
$$Gx + s = h \tag{3.9c}$$
$$s \circ z = \tau \boldsymbol{e} \tag{3.9d}$$
$$(s, z) \in \mathcal{K} \times \mathcal{K}. \tag{3.9e}$$

The *central path* is defined as the trajectory of points parameterized by $\tau > 0$, satisfying Equations (3.9a) - (3.9e). We denote the central path with the tuple $(x^*(\tau), s^*(\tau), y^*(\tau), z^*(\tau))$. From the above analysis, we see that the central path equations given by Equations (3.9a) - (3.9e) are equivalent to the optimality conditions for the barrier formulation of Problem 3.1 given by Problem 3.4. It can be shown that as $\tau \to 0$, $(x^*(\tau), s^*(\tau), y^*(\tau), z^*(\tau)) \to (x^*, s^*, y^*, z^*)$ [21].

The primal-dual interior-point method applies Newton's method to move towards the central path rather than directly towards points satisfying Equation (3.2d). Since the central path lies within the cone, maintaining proximity to it allows the algorithm to take longer steps without violating Equation (3.2e) [192].

3.2.2 Nesterov-Todd scaling

If Newton's method were applied directly to Equations (3.5a) - (3.5d), the coefficient matrix of the resulting linear system would lack symmetry (see Appendix A.1). As a result, it would be necessary to store the entire matrix, rather than only the upper or lower triangular portion. Additionally, solving this system would require matrix factorizations for non-symmetric matrices, which are generally more computationally expensive than those for

symmetric matrices.

To derive a symmetric linear system, we introduce the following change of variables

$$\tilde{s} = W^{-\top} s, \ \tilde{z} = W z,$$

where we choose W such that the above transformation preserves cone membership and leaves the central path equations unchanged

$$s \in \mathcal{K} \iff \tilde{s} \in \mathcal{K}, \quad z \in \mathcal{K} \iff \tilde{z} \in \mathcal{K}, \quad s \circ z = \tau \boldsymbol{e} \iff \tilde{s} \circ \tilde{z} = \tau \boldsymbol{e}. \tag{3.10}$$

Using this transformation, the central path equations can be equivalently expressed as

$$Px + c + A^\top y + G^\top z = 0 \tag{3.11a}$$
$$Ax = b \tag{3.11b}$$
$$Gx + s = h \tag{3.11c}$$
$$(W^{-\top} s) \circ (W z) = \tau \boldsymbol{e} \tag{3.11d}$$
$$(s, z) \in \mathcal{K} \times \mathcal{K}. \tag{3.11e}$$

There are many matrices, W, which satisfy Equation (3.10). In particular, we use the *Nesterov-Todd scaling* matrix [139, 140]. This scaling matrix is determined based on the current iterates s_k and z_k and the unique point w that satisfies

$$\nabla^2 \phi_{\mathcal{K}}(w) s_k = z_k, \tag{3.12}$$

where $\nabla^2 \phi_{\mathcal{K}}(w)$ is the Hessian of the barrier function evaluated at w. Since the barrier function is strictly convex, $\nabla^2 \phi_{\mathcal{K}}(w)$ is positive definite.

From this, we compute W_k as

$$\nabla^2 \phi_{\mathcal{K}}(w)^{-1} = W_k^\top W_k. \tag{3.13}$$

A key property that follows from Equations (3.12) and (3.13) is

$$W_k^{-\top} s_k = W_k z_k. \tag{3.14}$$

This property allows us to define

$$\lambda_k = W_k^{-\top} s_k = W_k z_k. \tag{3.15}$$

For more details on how to compute the scaling point w and the scaling matrix W_k, see [186].

3.2.3 Computing search directions

Given the current iterate (x_k, s_k, y_k, z_k) we define the *duality measure* as

$$\mu_k = s_k^\top z_k / m, \tag{3.16}$$

which represents the average violation of the complementary slackness condition given by Equation (3.2d). To compute a search direction for updating the current iterate, we apply Mehrotra's predictor-corrector method [135]. This scheme computes a Newton step with three key objectives: reducing the duality measure, maintaining proximity to the central path, and correcting for the linearization error introduced during this step.

Applying Newton's method to Equation (3.11) first requires a linearization of the central path equations around the current iterate (x_k, s_k, y_k, z_k). This is done by substituting (x, s, y, z) with $(x_k + \Delta x, s_k + \Delta s, y_k + \Delta y, z_k + \Delta z)$ and ignoring the higher order term $(W^{-\top}\Delta s) \circ (W \Delta z)$. This results in a linear system of the form

$$P\Delta x + A^\top \Delta y + G^\top \Delta z = -r_x \tag{3.17a}$$
$$A\Delta x = -r_y \tag{3.17b}$$
$$G\Delta x + \Delta s = -r_z \tag{3.17c}$$
$$(W_k^{-\top} s_k) \circ (W_k \Delta z) + (W_k^{-\top} \Delta s) \circ (W_k z_k) = -r_s. \tag{3.17d}$$

Here, (r_x, r_y, r_z, r_s) are residual vectors that will be defined later.

Using Equation (3.15), we can rewrite the system as

$$P\Delta x + A^\top \Delta y + G^\top \Delta z = -r_x \tag{3.18a}$$
$$A\Delta x = -r_y \tag{3.18b}$$
$$G\Delta x + \Delta s = -r_z \tag{3.18c}$$
$$\lambda_k \circ (W_k \Delta z + W_k^{-\top} \Delta s) = -r_s. \tag{3.18d}$$

To obtain a symmetric system, we can eliminate Δs and rewrite the equations in matrix form as

$$\begin{bmatrix} P & A^\top & G^\top \\ A & 0 & 0 \\ G & 0 & -W_k^\top W_k \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} -r_x \\ -r_y \\ -r_z + W_k^\top (\lambda_k \backslash r_s) \end{bmatrix} \tag{3.19a}$$
$$\Delta s = -r_z - G\Delta x, \tag{3.19b}$$

where the operator $\backslash$ represents the inverse of the Jordan product $\circ$, meaning that $x \backslash (x \circ y) = y$.

To account for the higher order term we ignored when forming Equation (3.17), we apply

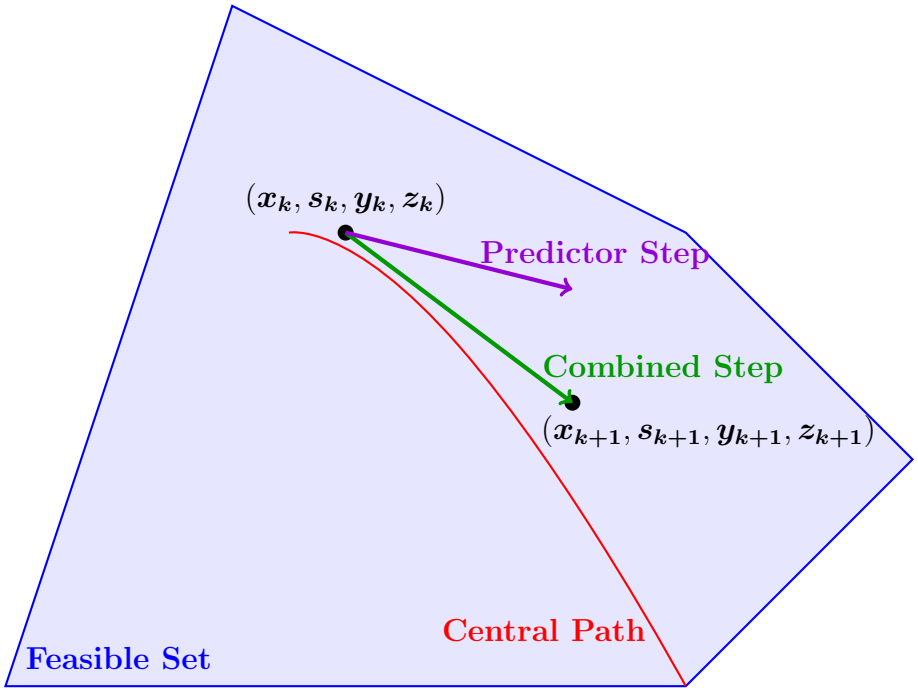


Figure 3.1: Mehrotra's predictor-corrector.

Mehrotra's predictor-corrector [135]. This method consists of a predictor step, where a search direction is computed by taking a Newton step on Equation (3.11) with $\tau = 0$, followed by a combined step. The combined step incorporates centering and correction terms that will help keep the next iterate close to the central path while compensating for the linearization error incurred by ignoring the $(W^{-\top}\Delta s) \circ (W\Delta z)$ term when linearizing Equation (3.11). Note that the predictor step is used to estimate this linearization error. Figure 3.1 illustrates Mehrotra's predictor-corrector.

To compute the predictor step, also called the *affine scaling direction*, $(\Delta x_a, \Delta s_a, \Delta y_a, \Delta z_a)$, we solve Equation (3.19) with the residual vectors

$$r_x = Px_k + c + A^\top y_k + G^\top z_k \tag{3.20a}$$
$$r_y = Ax_k - b \tag{3.20b}$$
$$r_z = Gx_k + s_k - h \tag{3.20c}$$
$$r_s = \lambda_k \circ \lambda_k. \tag{3.20d}$$

Since taking a full step along the affine scaling direction can violate the constraint $(s, z) \in \mathcal{K} \times \mathcal{K}$, we compute the centering parameter $\sigma \in [0, 1]$ as

$$\alpha = \sup\{\alpha \in [0, 1] \mid (s_k, z_k) + \alpha(\Delta s_a, \Delta z_a) \in \mathcal{K} \times \mathcal{K}\}$$
$$\rho = \frac{(s_k + \alpha \Delta s_a)^\top (z_k + \alpha \Delta z_a)}{s_k^\top z_k}$$
$$\sigma = \max\{0, \min\{1, \rho\}^3\}.$$

The centering parameter balances the need to reduce the duality measure while maintaining proximity to the central path, allowing for a longer step in the next iteration [192].

Finally, we compute the combined direction, which includes both predictor and corrector information. This direction, $(\Delta x, \Delta s, \Delta y, \Delta z)$ is obtained by solving Equation (3.19) with the residual vectors

$$r_x = Px_k + c + A^\top y_k + G^\top z_k \tag{3.21a}$$
$$r_y = Ax_k - b \tag{3.21b}$$
$$r_z = Gx_k + s_k - h \tag{3.21c}$$
$$r_s = \lambda_k \circ \lambda_k + (W_k^{-\top} \Delta s_a) \circ (W_k \Delta z_a) - \sigma \mu_k \boldsymbol{e} \tag{3.21d}$$

This direction moves the next iterate closer to $(x^*(\sigma\mu_k), s^*(\sigma\mu_k), y^*(\sigma\mu_k), z^*(\sigma\mu_k))$, a

point on the central path where the duality measure is reduced by a factor of σ. Note that the residual vector r_s has two additional terms when compared to Equation (3.20d): $(W_k^{-\top}\Delta s_a)\circ(W_k\Delta z_a)$ and $-\sigma\mu_k\boldsymbol{e}$. The former is to correct for linearization error in Newton's method, and the latter is to maintain proximity to the central path and arises when applying Newton's method to Equation (3.9d) with $\tau = \sigma\mu_k$.

We then compute the next iterate by performing a line search to ensure $(s_{k+1}, z_{k+1}) \in \mathcal{K} \times \mathcal{K}$. Specifically, we update the iterate as

$$(x_{k+1}, s_{k+1}, y_{k+1}, z_{k+1}) = (x_k, s_k, y_k, z_k) + \alpha(\Delta x, \Delta s, \Delta y, \Delta z)$$

where

$$\alpha = \sup\left\{\alpha \in [0,1] \mid (s_k, z_k) + \frac{\alpha}{0.99}(\Delta s, \Delta z) \in \mathcal{K} \times \mathcal{K}\right\}$$

The factor of 0.99 ensures that (s_{k+1}, z_{k+1}) remains strictly in the interior of $\mathcal{K} \times \mathcal{K}$.

3.2.4 Initialization

To initialize the primal-dual interior point method, we follow the approach outlined in [186]. The initialization procedure consists of two steps. First, we find (x, s, y, z) that satisfy Equations (3.9a) - (3.9c). If s and z do not satisfy Equation (3.9e), a correction is applied to ensure they are in $\mathcal{K}$.

We begin by solving the optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^\top P x + c^\top x + \|Gx - h\|_2^2 \\ \text{subject to} \quad & Ax = b. \end{aligned}$$

Since this is an equality-constrained quadratic program, the KKT conditions correspond to the linear system

$$\begin{bmatrix} P & A^\top & G^\top \\ A & 0 & 0 \\ G & 0 & -I \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -c \\ b \\ h \end{bmatrix}. \tag{3.22}$$

We then set $s = -z$. This results in the tuple (x, s, y, z) satisfying Equations (3.9a) - (3.9c), but s and z may not necessarily satisfy Equation (3.9e).

We then initialize x_0 and y_0 as $x_0 = x$, $y_0 = y$, and initialize s_0 as

$$s_0 = \begin{cases} s, & \text{if } s \in \mathcal{K} \\ s + (1 + \alpha_s)\boldsymbol{e} & \text{otherwise,} \end{cases}$$

where $\alpha_s = \inf\{\alpha \mid s + \alpha \boldsymbol{e} \in \mathcal{K}\}$ and $\boldsymbol{e}$ is defined in Equation (3.7). We initialize z_0 as

$$z_0 = \begin{cases} z, & \text{if } z \in \mathcal{K} \\ z + (1 + \alpha_z)\boldsymbol{e}, & \text{otherwise} \end{cases}$$

where $\alpha_z = \inf\{\alpha \mid z + \alpha \boldsymbol{e} \in \mathcal{K}\}$. Note that α_s and α_z are the minimum scalings of $\boldsymbol{e}$ that need to be added to s and z respectively to move them to the boundary of $\mathcal{K}$, but we scale $\boldsymbol{e}$ by $(1 + \alpha_s)$ and $(1 + \alpha_z)$ to ensure that s_0 and z_0 are strictly in the interior of $\mathcal{K}$.

Algorithm 1 Primal-dual interior point method

Ensure: Optimal solution (x^*, s^*, y^*, z^*)

Initialization:
  Solve KKT system (3.22) for x, y, z
  $x_0 \leftarrow x,\ y_0 \leftarrow y$
  $s \leftarrow -z$
if $s \in \mathcal{K}$ **then**
    $s_0 \leftarrow s$
else
    $\alpha_s \leftarrow \inf\{\alpha \mid s + \alpha \boldsymbol{e} \in \mathcal{K}\}$
    $s_0 \leftarrow s + (1 + \alpha_s)\boldsymbol{e}$
end if
if $z \in \mathcal{K}$ **then**
    $z_0 \leftarrow z$
else
    $\alpha_z \leftarrow \inf\{\alpha \mid z + \alpha \boldsymbol{e} \in \mathcal{K}\}$
    $s_0 \leftarrow z + (1 + \alpha_z)\boldsymbol{e}$
end if
repeat
    Compute duality measure:
      $\mu_k \leftarrow s_k^\top z_k / m$
    Compute Nesterov-Todd scaling:
      Construct W_k
      Calculate $\lambda_k = W_k^{-\top} s_k = W_k z_k$
    Compute affine direction $(\Delta x_a, \Delta s_a, \Delta y_a, \Delta z_a)$:
      Solve KKT system (3.19) with residuals (3.20)
    Calculate centering parameter:
      $\alpha \leftarrow \sup\{\alpha \in [0,1] \mid (s_k, z_k) + \alpha(\Delta s_a, \Delta z_a) \in \mathcal{K}\}$
      $\rho \leftarrow \frac{(s_k + \alpha \Delta s_a)^\top (z_k + \alpha \Delta z_a)}{s_k^\top z_k}$
      $\sigma \leftarrow \max\{0, \min\{1, \rho\}^3\}$
    Compute combined direction $(\Delta x, \Delta s, \Delta y, \Delta z)$:
      Solve (3.19) with residuals (3.21)
    Compute step-size:
      $\alpha \leftarrow \sup\left\{\alpha \in [0,1] \mid (s_k, z_k) + \frac{\alpha}{0.99}(\Delta s, \Delta z) \in \mathcal{K}\right\}$
    Update iterates:
      $x_{k+1} \leftarrow x_k + \alpha \Delta x$
      $s_{k+1} \leftarrow s_k + \alpha \Delta s$
      $y_{k+1} \leftarrow y_k + \alpha \Delta y$
      $z_{k+1} \leftarrow z_k + \alpha \Delta z$
until Stopping criteria (3.28) satisfied.

3.3 Algorithm implementation

This section describes the implementation of the primal-dual interior point method in QOCO and QOCOGEN. We focus on three key aspects: the matrix factorization used to solve the system in (3.19), numerical techniques for ensuring a robust factorization and solution of (3.19), and the stopping criteria. These aspects apply to both QOCO and QOCOGEN, but while both use the same underlying matrix factorization, their implementations are different. These differences will be discussed in Section 3.4.1.

3.3.1 Linear system solve

The coefficient matrix in (3.19), which we will refer to as the Karush-Kuhn-Tucker (KKT) matrix is given as

$$K = \begin{bmatrix} P & A^\top & G^\top \\ A & 0 & 0 \\ G & 0 & -W_k^\top W_k \end{bmatrix}. \tag{3.23}$$

Algorithm 1 requires solving the linear system $K\xi = r$ for two different residual vectors r. Since K remains constant for both solves, we factorize it once and then apply two backsolves.

The KKT matrix (3.23) is symmetric but indefinite, meaning it has both positive and negative eigenvalues. A common approach for factoring such matrices is the Bunch-Parlett factorization [37], which factors K as

$$\Pi K \Pi^\top = LDL^\top, \tag{3.24}$$

where Π is a permutation matrix, L is a lower triangular matrix, and D is a block diagonal matrix with 1×1 and 2×2 blocks. However, Π must be selected during runtime with knowledge of the data in K, as the factorization may not exist for all permutations. Additionally, K may not be invertible if, for example, A is not full row rank.

Static regularization To ensure a numerically stable factorization that exists for all permutations Π (allowing us to select Π based solely on the sparsity pattern of K), we apply *static regularization* [92, 131, 177] to the KKT matrix

$$\hat{K} = \left[\begin{array}{c|cc} P & A^\top & G^\top \\ \hline A & 0 & 0 \\ G & 0 & -W_k^\top W_k \end{array}\right] + \left[\begin{array}{c|cc} \epsilon_s I & 0 & 0 \\ \hline 0 & -\epsilon_s I & 0 \\ 0 & 0 & -\epsilon_s I \end{array}\right]. \tag{3.25}$$

In QOCO and QOCOGEN, we use $\epsilon_s = 10^{-8}$.

Now $\hat{K}$ is a *quasidefinite* matrix, which is a special case of a symmetric indefinite matrix where the $(1,1)$ block is positive definite and the $(2,2)$ block is negative definite [187].

For such matrices, the $LDL^\top$ factorization

$$\Pi K \Pi^\top = LDL^\top, \tag{3.26}$$

exists for any permutation Π, and D will be a diagonal matrix with known signs for its diagonal elements [187].

A judicious choice of Π can minimize the *fill-in* for the factor L. Fill-in is the introduction of non-zero elements in positions of a matrix where they previously did not exist. Excessive fill-in can lead to several negative consequences, including increased storage requirements (since more non-zero elements must be stored in memory) and increased computational cost (due to more floating-point operations). Finding an optimal Π to minimize fill-in is NP-complete [193], but heuristic methods such as the Approximate Minimum Degree (AMD) ordering can effectively compute near-optimal permutations [7]. In QOCO and QOCOGEN, we use Tim Davis' implementation of the AMD ordering [64] to determine Π.

After computing Π, the $LDL^\top$ factorization consists of two phases: a symbolic and numeric factorization. The symbolic factorization determines the elimination tree, a data structure which encodes dependencies between elements in the matrix during the factorization process, and the sparsity pattern of the factor L. It only requires the sparsity pattern of

the regularized KKT matrix $\hat{K}$, which remains unchanged throughout Algorithm 1. Therefore, the symbolic factorization needs to be computed only once. The numeric factorization calculates the numerical values of L and D, which depend on the values of the nonzero elements of $\hat{K}$. Since these values change at each iteration of Algorithm 1, the numeric factorization must be computed at every step.

To compute the $LDL^\top$ factorization, we use a modified version of QDLDL [177] with *dynamic regularization* for QOCO, and a custom $LDL^\top$ routine for QOCOGEN which we will discuss in Section 3.4.1.

Dynamic regularization Although the $LDL^\top$ factorization must theoretically exist for $\hat{K}$, due to the nature of floating-point arithmetic it is possible for diagonal elements D_{ii} to be rounded to zero or be very close to zero leading to divide-by-zero errors and the factorization failing. To alleviate this and factor $\hat{K}$ in a stable manner, we apply *dynamic regularization* as given by Algorithm 2, which bounds the magnitude of D_{ii} away from zero, by some $\epsilon_d > 0$, and corrects for its sign if necessary [69, 92, 131]. In QOCO and QOCOGEN, we use $\epsilon_d = 10^{-8}$.

Algorithm 2 Dynamic regularization

Given: $\epsilon_d > 0$
if D_{ii} should be positive **then**
    if $D_{ii} < 0$ **then**
        $D_{ii} = \epsilon_d$
    else
        $D_{ii} = D_{ii} + \epsilon_d$
    end if
else if D_{ii} should be negative **then**
    if $D_{ii} > 0$ **then**
        $D_{ii} = -\epsilon_d$
    else
        $D_{ii} = D_{ii} - \epsilon_d$
    end if
end if

Iterative refinement After static and dynamic regularization we end up solving the linear system $\hat{K}\xi = r$ rather than system $K\xi = r$. To get a solution to the latter system, we apply *iterative refinement* [157, Section 2.5.1]. This process corrects the solution by iteratively reducing the residual error, $\|r - K\xi\|$, ensuring higher numerical accuracy without requiring an additional factorization. It involves iteratively solving the system

$$\hat{K}\Delta\xi^k = r - K\xi^k, \tag{3.27}$$

and updating $\xi^{k+1} = \xi^k + \Delta\xi^k$, where ξ^0 solves the first linear system (i.e. $\hat{K}\xi^0 = r$). Note that since we have already computed the factorization of $\hat{K}$, one iteration of Equation (3.27) only requires backsolves and no matrix factorization.

3.3.2 Stopping criterion

Algorithm 1 terminates when three conditions are met: primal feasibility (3.28a), stationarity (3.28b), and complementary slackness (3.28c). We use both absolute and relative tolerances to ensure robustness across different problem scales. Our stopping criteria is met if the following conditions hold:

$$\left\| \begin{bmatrix} Ax_k - b \\ Gx_k + s_k - h \end{bmatrix} \right\|_\infty \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \max\left\{ \|Ax_k\|_\infty, \|b\|_\infty, \|Gx_k\|_\infty, \|h\|_\infty, \|s_k\|_\infty \right\} \tag{3.28a}$$

$$\|Px_k + A^\top y_k + G^\top z_k + c\|_\infty \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \max\left\{ \|Px_k\|_\infty, \|A^\top y_k\|_\infty, \|G^\top z_k\|_\infty, \|c\|_\infty \right\} \tag{3.28b}$$

$$|s_k^\top z_k| \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \max\left\{1, |p_k|, |d_k|\right\}, \tag{3.28c}$$

where the primal and dual objectives are

$$p_k := \frac{1}{2}x_k^\top P x_k + c^\top x_k, \quad d_k := -\frac{1}{2}x_k^\top P x_k - b^\top y_k - h^\top z_k.$$

3.4 Sparsity exploiting custom solver

This section discusses QOCOGEN, which generates an implementation of Algorithm 1 called $\text{QOCO}_{\text{custom}}$. Unlike generic solvers which use sparse linear algebra, $\text{QOCO}_{\text{custom}}$ uses a custom $LDL^\top$ factorization and other tailored linear algebra routines, which exploit the known sparsity structure of problem data, to solve Problem 3.1 significantly faster.

While QOCO can solve instances of Problem 3.1 with any sparsity pattern for P, A, and G and cone $\mathcal{K}$, the use of custom linear algebra in $\text{QOCO}_{\text{custom}}$ restricts it to instances with the same sparsity structure for P, A, and G and optimizes over the same cone $\mathcal{K}$. This makes $\text{QOCO}_{\text{custom}}$ useful for applications that repeatedly solve optimization problems with identical sparsity structures, such as sequential convex programming [126].

Figure 3.2 illustrates the usage of QOCO, QOCOGEN, and $\text{QOCO}_{\text{custom}}$. In the figure, the matrices (P, A, G) contain two pieces of information: their sparsity patterns $\text{str}(P, A, G)$ (the location of nonzero elements) and the values of the nonzero elements $\text{data}(P, A, G)$. We see that QOCO takes in (P, A, G), which contains both the sparsity patterns and nonzero elements for the matrices, and the rest of the problem data and returns the optimal solution (x^*, s^*, y^*, z^*). In contrast, QOCOGEN only takes in the sparsity patterns of (P, A, G) and cone $\mathcal{K}$, and then generates $\text{QOCO}_{\text{custom}}$. This instance of $\text{QOCO}_{\text{custom}}$ can then solve optimization problems where the sparsity patterns of (P, A, G) do not change, but the nonzero values for (P, A, G) and the vector data (c, b, h) can change.

Although QOCO avoids dynamic memory allocation during the solution process, it still relies on `setup` and `cleanup` functions for dynamic memory allocation and deallocation before and after solving the problem. In contrast, $\text{QOCO}_{\text{custom}}$ exclusively uses static memory allocation, making it well-suited for resource-constrained embedded systems, where dynamic memory allocation can lead to non-deterministic behavior and memory fragmentation.

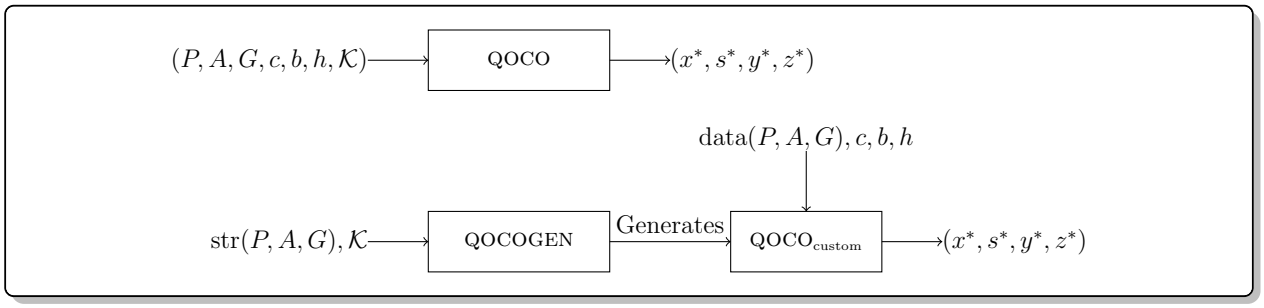


Figure 3.2: Usage of QOCO, QOCOGEN, and $\text{QOCO}_{\text{custom}}$.

3.4.1 Custom linear algebra

A key difference between $\text{QOCO}_{\text{custom}}$ and QOCO is that $\text{QOCO}_{\text{custom}}$ uses custom linear algebra routines rather than sparse linear algebra. The main motivation for custom routines is to speed up the $LDL^\top$ factorization of the regularized KKT matrix, which is the most computationally expensive step of Algorithm 1. Typically, the KKT matrix is factored using a sparse linear algebra routine that stores the matrix in either *compressed sparse column* (CSC) or *compressed sparse row* (CSR) format. However, these sparse routines involve more than just the floating-point operations required for factorization. They also incur additional overhead to locate nonzero elements and their positions in the matrix. For example, in a CSC matrix, accessing data first requires indexing into the column pointer and row index arrays. This results in more CPU instructions, increased memory accesses, and a higher likelihood of cache misses. We provide empirical evidence supporting these claims in Appendix A.2.

When the sparsity structure of the KKT matrix is known *a priori* it is possible to write a custom $LDL^\top$ factorization routine which only contains code to perform the necessary floating-point operations and data accesses, eliminating the additional overhead that typical sparse linear algebra routines have. This makes the custom $LDL^\top$ factorization significantly faster than a sparse $LDL^\top$ factorization. We also note that since the AMD ordering we employ to compute the permutation matrix Π only depends on the sparsity pattern of the KKT matrix, we can determine Π at the time of code generation, so the generated code does

```
    # Adata stores the nonzeros of A in column-major format, i.e.
    # Adata[0] = a00
    # Adata[1] = a20
    # Adata[2] = a02
    def custom_matvec(y, Adata, x):
        y[0] = Adata[0]*x[0] + Adata[2]*x[2]
        y[1] = 0
        y[2] = Adata[1]*x[0]
```

Listing 3.1: Custom matrix-vector multiplication.

not include the code to compute the AMD ordering.

A concrete example of a custom linear algebra routine for matrix-vector multiplication is given by Listing 3.1, where the sparsity pattern of A is fixed and defined in Equation (3.29). Note that the custom routine only includes the necessary floating-point operations. Although Listing 3.1 depicts Python code, the custom linear algebra routines in QOCO$_{\text{custom}}$ are in C.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \underbrace{\begin{bmatrix} a_{00} & 0 & a_{02} \\ 0 & 0 & 0 \\ a_{20} & 0 & 0 \end{bmatrix}}_{A} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \tag{3.29}$$

A drawback of these custom routines is that the amount of code generated increases with problem size. Consequently, both the time required to generate and compile QOCO$_{\text{custom}}$ and the resulting binary size increase for larger problems. On processors with small caches, such as the Raspberry Pi Compute Module 4, these large binaries can degrade performance due to increased instruction cache misses. This effect is discussed further in Section 3.5.

3.4.2 Code generation

QOCOGEN is written in Python and example code for generating QOCO$_{\text{custom}}$ for a problem family defined by Equation (3.30) is given by Listing 3.2. The generated solver QOCO$_{\text{custom}}$ is customized to the sparsity pattern of the matrices P, A, G which are passed to the function

```
import qocogen
import numpy as np
from scipy import sparse

# Define problem data
P = sparse.diags([2, 2, 2, 0], 0).tocsc()

c = np.array([0, 0, 0, 1])
G = -sparse.identity(4).tocsc()
h = np.zeros(4)
A = sparse.csc_matrix([[1, 1, 0, 0], [0, 1, 1, 0]])
b = np.array([1, 1])

l = 1
n = 4
m = 4
p = 2
nsoc = 1
q = np.array([3])

# Generate custom solver.
qocogen.generate_solver(n, m, p, P, c, A, b, G, h, l, nsoc, q)
```

Listing 3.2: Generating a custom solver with QOCOGEN.

`qocogen.generate_solver()` as sparse `scipy` matrices.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x_1^2 + x_2^2 + x_3^2 + x_4 \\ \text{subject to} \quad & x_1 + x_2 = 1 \\ & x_2 + x_3 = 1 \\ & x_1 \geq 0 \\ & \sqrt{x_3^2 + x_4^2} \leq x_2 \end{aligned} \tag{3.30}$$

The file tree for $\text{QOCO}_{\text{custom}}$ is given by Figure 3.3. The file `qoco_custom.c` contains the `qoco_custom_solve()` function which implements Algorithm 1, `ldl.c` contains the custom $LDL^\top$ factorization of the regularized KKT matrix, `CMakeLists.txt` is the CMake configuration file which automates the build process [108], `runtest.c` gives an example of calling $\text{QOCO}_{\text{custom}}$ with default data and times the resulting solve, and the remaining files contain various helper functions needed to implement Algorithm 1.

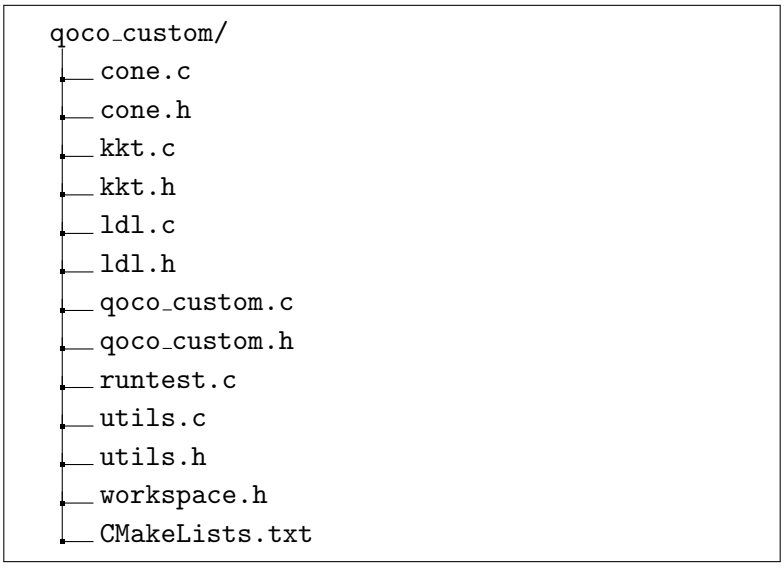


Figure 3.3: QOCO$_{\text{custom}}$ file structure.

Listing 3.3 provides an example of calling QOCO$_{\text{custom}}$ from C, updating the problem data, and solving it again.

```c
#include "qoco_custom.h"

int main() {
    // Instantiate workspace.
    Workspace work;

    // Set default settings, but settings can be modified with
    // work.settings.setting_name = ...
    set_default_settings(&work);

    // Loads the default data for the problem
    // (i.e. the data passed to the qocogen.generate() call).
    load_data(&work);

    // Solve original problem.
    qoco_custom_solve(&work);
    printf("\nobj: %f", work.sol.obj);

    // Can modify non-zero elements of P,A,G and c, b, h.
    update_P(&work, Pnew);
    update_A(&work, Anew);
    update_G(&work, Gnew);
    update_c(&work, cnew);
    update_b(&work, bnew);
    update_h(&work, hnew);

    // Solve updated problem.
    qoco_custom_solve(&work);
    printf("\nobj: %f", work.sol.obj);
}
```

Listing 3.3: Calling $\text{QOCO}_{\text{custom}}$ from C/C++.

3.5 *Numerical results*

We benchmark QOCO and $\text{QOCO}_{\text{custom}}$ against several conic interior-point solvers [2]: the open-source solvers ECOS [69] and CLARABEL [92], the commercial solvers GUROBI [93] and MOSEK [11], and against the academically licensed custom solver generator CVXGEN which generates a primal-dual interior point method for quadratic programs. We do not test against the custom solver generator BSOCP since it is not academically licensed. For all solvers, we use their default settings but set the tolerances $\epsilon_{abs} = \epsilon_{rel} = 10^{-7}$.

All numerical results were generated on a desktop computer with an AMD Ryzen 9 7950X3D processor running at 4.2 GHz and with 128 GB of RAM. Additionally, the model-predictive control problems in Section 3.5.2 were also solved on the Raspberry Pi Compute Module 4 (CM4), a resource-constrained, embedded Linux system with 4GB of RAM. Our benchmarks are written in Python and we use CVXPY [5, 67] to interface with the solvers.

In our experiments, we define problem size as the total number of nonzero elements in A, G, and in the upper half of P. We evaluate our solvers on a set of benchmark problems we developed, a set of model-predictive control problems from [111], the Maros–Mészáros problems [130], and least-squares problems derived from the SuiteSparse repository [65]. To evaluate the solvers' performance, we use performance profiles [68] and the shifted geometric mean, both of which are frequently used to assess solver performance [69, 84, 92, 171, 177].

When reporting solve times for $\text{QOCO}_{\text{custom}}$, we exclude code generation and compilation times and report them separately. We do this because the primary use cases for QOCOGEN and $\text{QOCO}_{\text{custom}}$ prioritize minimizing solve time, making solve time alone the most relevant performance metric, and the one-time cost of generating and compiling the solver is less critical. Typical examples include MPC and real-time trajectory optimization, where code generation and compilation can be done offline before deploying the solvers online.

When minimizing setup time (code generation and compilation) is important, such as iterating problem formulations or one-off solves, QOCOGEN and $\text{QOCO}_{\text{custom}}$ may be less at-

[2] Our benchmarks are publicly available at https://github.com/qoco-org/qoco-benchmarks

tractive than general-purpose solvers like QOCO. Additionally, as problem sizes become larger, code generation and compile times become longer and binary sizes become larger due to the explicit coding style used by custom linear algebra and depicted in Listing 3.1. Nevertheless, the combined code generation and compilation times remain reasonable (under ten minutes) for problems with sizes under 10,000.

We also expect $\text{QOCO}_{\text{custom}}$ performance to degrade once binaries become large compared to the cache size of the processor, due to increased instruction cache misses. On desktop computers, which have larger caches, excessive code generation time and compile times are typically experienced prior to degraded performance due to instruction cache misses. On systems with comparatively smaller caches, such as the Raspberry Pi CM4, instruction cache misses start to hinder performance for problem sizes over approximately 4,000 in our benchmarks, before code generation and compilation time becomes significant.

Shifted geometric mean We use the normalized shifted geometric mean to assign a scalar value to the performance of a solver on a test set of problems. The shifted geometric mean of N runtimes for solver s is computed as

$$g_s = \left[\prod_{p=1}^{N}(t_{s,p} + k)\right]^{1/N} - k,$$

where $t_{s,p}$ is the runtime of solver s on problem p and $k = 1$ is the shift. If the solver does not find an optimal solution, then the runtime is set to 10 seconds for benchmark problems, 0.5 seconds for the MPC problems, 1200 seconds for the Maros–Mészáros problems, and 600 seconds for the SuiteSparse least-squares problems. These times were chosen to be roughly an order of magnitude higher than the slowest successful solve on the respective problem set.

We then define the normalized shifted geometric mean for solver s as

$$r_s = g_s / \min_s g_s,$$

so the solver with the lowest shifted geometric mean will have a normalized shifted geo-

metric mean of 1.00.

Performance profiles We also plot the relative and absolute performance profiles to compare solvers. We first define the relative performance ratio of solver s on problem p as

$$u_{s,p} = t_{s,p} / \min_s t_{s,p}.$$

The relative performance profile then plots $f_s^r(\tau)$ where

$$f_s^r(\tau) = \frac{1}{N} \sum_{p=1}^{N} \mathcal{I}_{\leq \tau}(u_{s,p}),$$

and $\mathcal{I}_{\leq \tau}(z) = 1$ if $z \leq \tau$ and $\mathcal{I}_{\leq \tau}(z) = 0$ otherwise. If the relative performance profile passes through the point (x, y) for solver s, this means that solver s solves a fraction y of the problems within a factor of x of the fastest solver.

The absolute performance profile plots $f_s^a(\tau)$ where

$$f_s^a(\tau) = \frac{1}{N} \sum_{p=1}^{N} \mathcal{I}_{\leq \tau}(t_{s,p}).$$

If the absolute performance profile passes through the point (x, y) for solver s, this means that solver s solves a fraction y of the problems within x time. For both relative and absolute profiles, the profile of the highest-performing solver will lie above the profiles of the other solvers.

3.5.1 Benchmark problems

We solve optimization problems from five problem classes: robust Kalman filtering, group lasso regression [198], the losslessly convexified powered-descent guidance problem [4], Markowitz portfolio optimization [129], and the oscillating masses control problem [190]. The first three of these are SOCPs and the last two are QPs. For each of the five classes, we consider 10 different problem sizes, generating 20 unique problem instances per size, for a total of 1000

distinct optimization problems. Details on the problems we solve can be found in Appendix A.3. For these problems, we plot runtime as a function of problem size in addition to reporting performance profiles and shifted geometric means. We test QOCO on all problems, but only test $\text{QOCO}_{\text{custom}}$ on the smaller problems, as code generation and compile times become prohibitively long for the larger instances.

For each solver, we solve each optimization problem 100 times and record the minimum runtime, which includes both setup and solve time. Since the computer runs various background processes that can artificially inflate execution time, we conduct 100 runs and report the minimum execution time, thereby providing a more accurate measure of the solver's performance.

Although the oscillating masses and portfolio optimization problems are QPs and can be solved with CVXGEN, we limit testing to the two smaller oscillating masses problems. For larger instances, CVXGEN was unable to generate code due to the size of the problems. For the portfolio optimization problems, we do not test CVXGEN due to the sparsity pattern of the factor matrix F in Problem A.3. To specify the sparsity pattern of F in CVXGEN, we must manually provide the locations of nonzero elements into the web interface, which would have been prohibitively expensive, since F has hundreds of nonzero elements. In contrast, to generate code with QOCOGEN, we pass in problem data as sparse `scipy` matrices rather than manually specifying sparsity patterns. Although we can specify F as a dense matrix and only populate the nonzero elements for CVXGEN, this would severely hinder its performance and result in an unfair comparison.

When computing normalized shifted geometric mean, relative performance profiles, and absolute performance profiles, all solvers must be tested on the same set of problems. Since we test $\text{QOCO}_{\text{custom}}$ on only half of the problems we provide two versions of the aforementioned metrics. The first version, given by Figure 3.5, includes CLARABEL, ECOS, GUROBI, MOSEK, and QOCO evaluated on all problems. The second version, given by Figure 3.6, also includes $\text{QOCO}_{\text{custom}}$, but the solvers are evaluated on the half of the benchmark problems which $\text{QOCO}_{\text{custom}}$ is tested on.

Figures 3.4, 3.5, and 3.6 show that QOCO$_{\text{custom}}$ is the fastest solver on our benchmark problems by a significant margin and is as performant as CVXGEN on the 40 smallest oscillating mass problems. This latter result is unsurprising, as QOCO$_{\text{custom}}$ uses an identical algorithm to CVXGEN when solving QPs and both utilize custom linear algebra routines. Additionally, QOCO is the fastest open-source generic solver on all of the problems, although it is slower than Mosek on the group lasso problems and slower than Gurobi on some of the largest portfolio optimization problems. However, we notice that if we disable presolve for Mosek and Gurobi, QOCO is faster on these problems as well.

Table 3.1 reports the code generation time, compilation time, code size, and binary size of the generated solvers. We see that all problems have sizes under 10,000 and as the problem size increases, code generation time, compilation time, code size, and binary size all increase, but all solvers still take less than eight minutes to generate and compile.

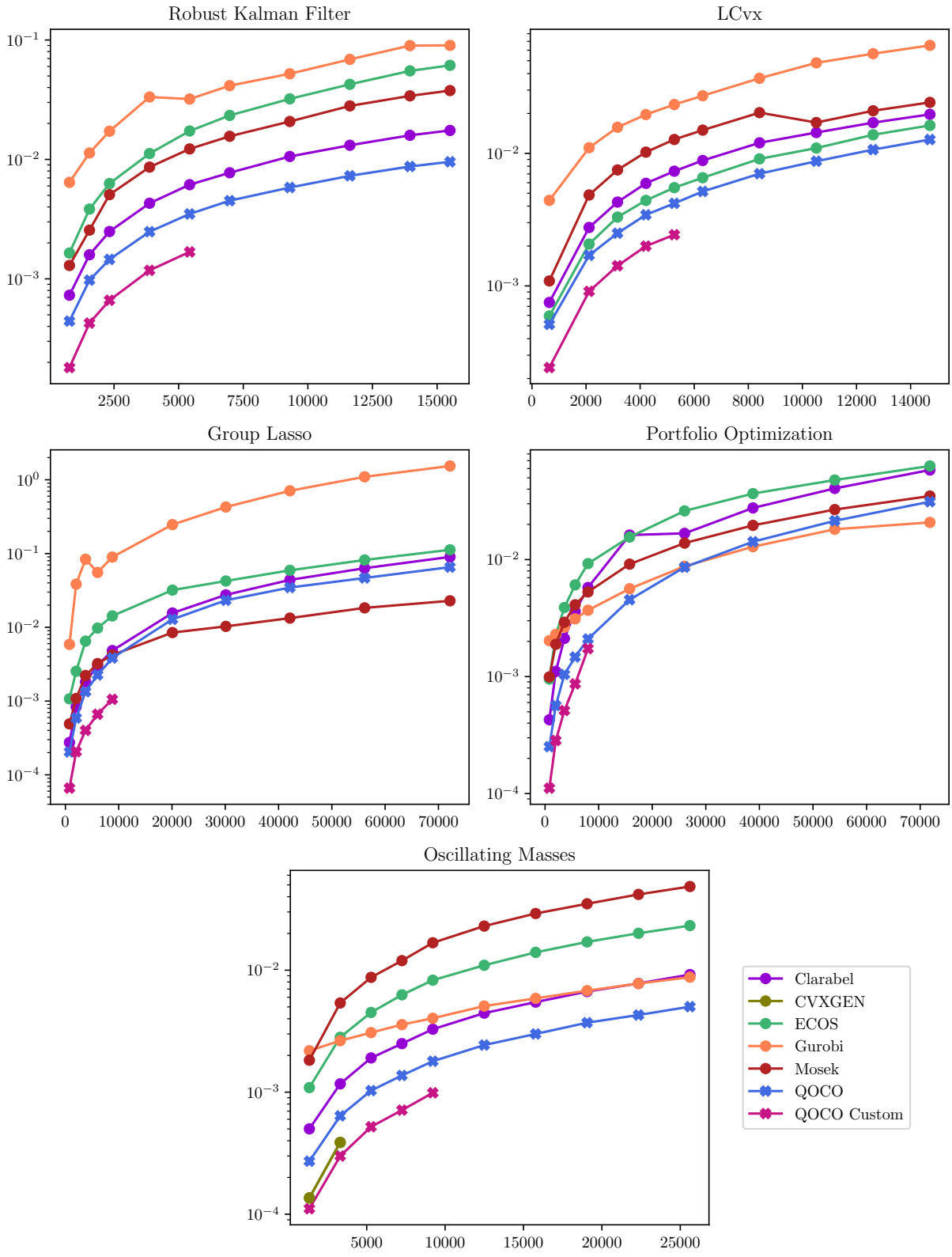
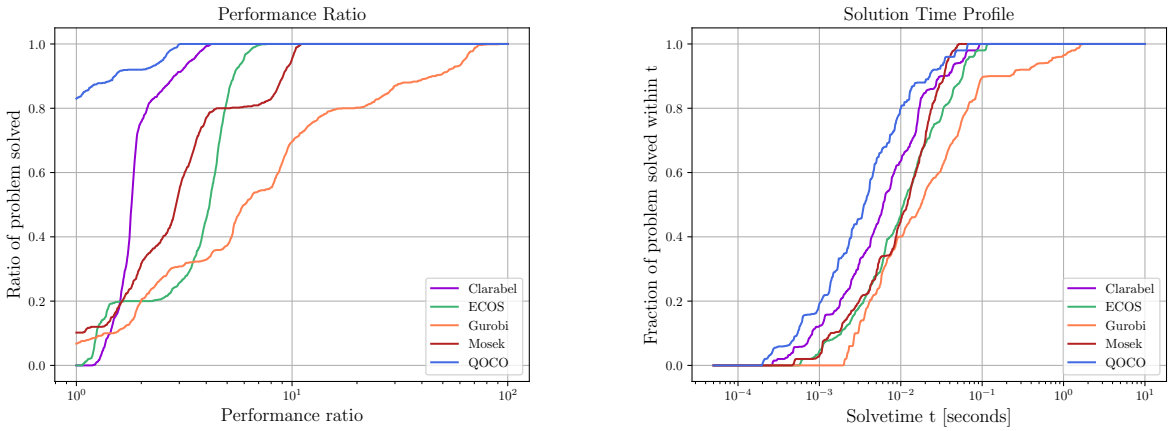


Figure 3.4: Solvetime in seconds vs problem size for benchmark problems



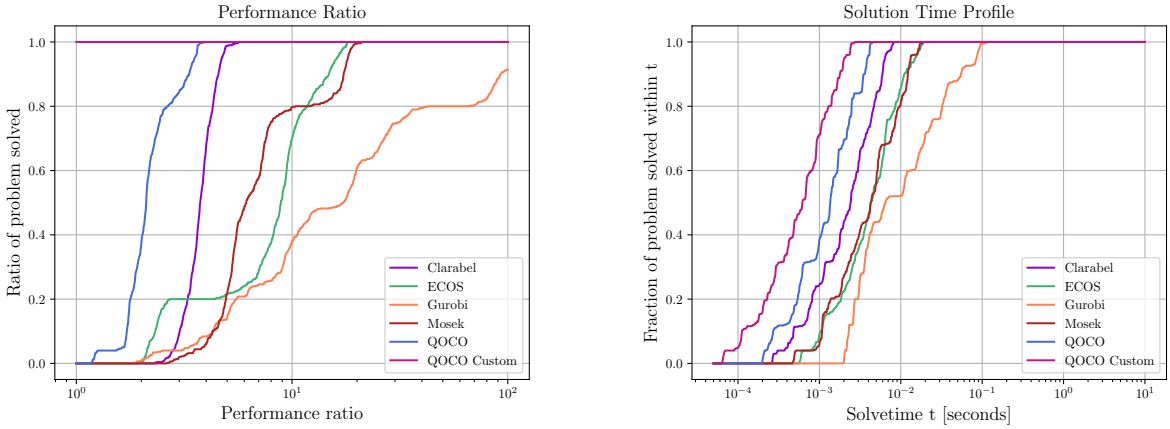
(a) Relative performance profile

(b) Absolute performance profile

	QOCO	**Clarabel**	**ECOS**	**Gurobi**	**Mosek**
Shifted GM	1.0	1.6	2.5	10.2	1.8
Failure Rate (%)	0.0	0.0	0.0	0.0	0.0

(c) Shifted geometric means and failure rates

Figure 3.5: Performance profiles for all benchmark problems



(a) Relative performance profile

(b) Absolute performance profile

	QOCO Custom	**QOCO**	**Clarabel**	**ECOS**	**Gurobi**	**Mosek**
Shifted GM	1.0	2.0	3.5	6.9	23.9	7.0
Failure Rate (%)	0.0	0.0	0.0	0.0	0.0	0.0

(c) Shifted geometric means and failure rates

Figure 3.6: Performance profiles for benchmark problems with custom solver

Problem	Size	Codegen Time (s)	Compile Time (s)	Code Size (KB)	Binary Size (KB)
group_lasso_N_1	761	4.2	4.3	545	234
group_lasso_N_2	2022	16.9	77.0	1292	590
group_lasso_N_3	3783	38.7	37.6	2397	1150
group_lasso_N_4	6044	70.5	69.3	3923	1898
group_lasso_N_5	8805	112.3	96.9	5949	2918
portfolio_N_2	804	2.8	4.1	481	226
portfolio_N_4	2008	11.1	45.0	1082	518
portfolio_N_6	3612	25.3	473.9	1890	918
portfolio_N_8	5616	45.5	46.0	2982	1482
portfolio_N_10	8020	72.9	106.1	6309	3026
lcvx_N_15	644	3.6	19.3	1057	482
lcvx_N_50	2114	39.3	57.0	3795	1590
lcvx_N_75	3164	89.9	85.5	5694	2434
lcvx_N_100	4214	163.8	140.1	7934	3322
lcvx_N_125	5264	263.0	193.3	9675	4026
robust_kalman_filter_N_25	775	5.8	11.3	759	318
robust_kalman_filter_N_50	1550	23.2	128.8	1491	582
robust_kalman_filter_N_75	2325	53.0	34.1	2236	890
robust_kalman_filter_N_125	3875	155.2	76.8	3730	1450
robust_kalman_filter_N_175	5425	311.0	122.5	5235	2022
oscillating_masses_N_8	1344	4.7	22.9	1112	478
oscillating_masses_N_20	3312	28.2	33.3	2785	1150
oscillating_masses_N_32	5280	72.4	60.4	4450	1806
oscillating_masses_N_44	7248	138.5	86.3	6122	2450
oscillating_masses_N_56	9216	226.9	123.6	7839	3102

Table 3.1: Code generation time, compilation time, and resulting code and binary sizes for benchmark problems.

3.5.2 Model-predictive control problems

We consider a set of 64 model-predictive control problems taken from [111] and also used by [92]. Note that we exclude the eight "nonlinear chain" problems as they took too long to generate custom solvers for. We run these problems on our desktop computer with the AMD Ryzen 9 7950X3D, as well as the Raspberry Pi CM4. Similarly to the benchmark problems, we solve each MPC problem 100 times and record the minimum runtime. These problems take the following form

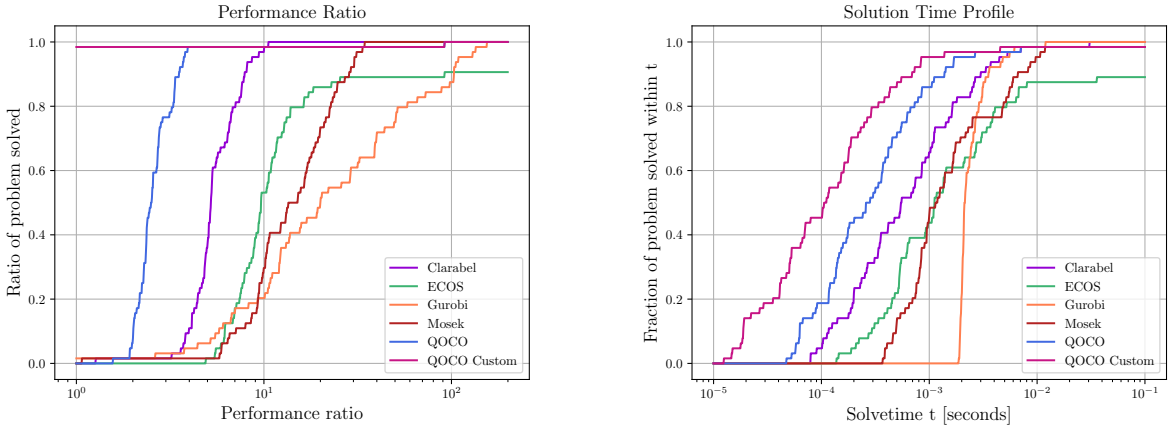
$$
\begin{aligned}
\underset{x,y,u}{\text{minimize}} \quad & \sum_{k=0}^{\tau} \begin{bmatrix} y_k - y_k^r \\ u_k - u_k^r \end{bmatrix}^\top \begin{bmatrix} Q & S \\ S^\top & R \end{bmatrix} \begin{bmatrix} y_k - y_k^r \\ u_k - u_k^r \end{bmatrix} + (x_\tau - x_\tau^r)^\top P (x_\tau - x_\tau^r) \\
\text{subject to} \quad & x_0 = x_{\text{init}} \\
& x_{k+1} = A x_k + B u_k + f_k \\
& y_k = C x_k + D u_k + e_k \\
& d_{\min} \leq M x_k + N u_k \leq d_{\max} \\
& u_{\min} \leq u_k \leq u_{\max} \\
& y_{\min} \leq y_k \leq y_{\max} \\
& d_{\min}^\tau \leq T x_\tau \leq d_{\max}^\tau.
\end{aligned}
$$

Table 3.2 shows that all problems except `SpringMass_1` problem have sizes under 10,000 and combined generation and compilation times under ten minutes. Additionally, among problems with sizes under 10,000, all but `quadcopter_3` require less than five minutes to generate and compile solvers.

Figure 3.7 shows the numerical results on the desktop computer. We see that CLARABEL, GUROBI, and MOSEK solve all problems successfully, QOCO and $\text{QOCO}_{\text{custom}}$ each fail on only one, and ECOS fails on seven. Among the 63 problems solved by both QOCO and $\text{QOCO}_{\text{custom}}$, $\text{QOCO}_{\text{custom}}$ is the fastest solver by a wide margin, followed by QOCO.

Figure 3.8 shows the results on the Raspberry Pi CM4, and we see that all solvers solve the same number of problems as on the desktop. We also see that problems are solved roughly an order magnitude slower on the CM4, as it is a less powerful processor. We again see that among the 63 problems solved by both QOCO and $\text{QOCO}_{\text{custom}}$, $\text{QOCO}_{\text{custom}}$ or QOCO is always the fastest solver, but on the nine largest problems, QOCO is faster than $\text{QOCO}_{\text{custom}}$. This occurs for the largest problems, since the $\text{QOCO}_{\text{custom}}$ binaries are larger than the CM4's L2 cache, likely leading to many instruction cache misses and increased solve time.

We observe QOCO outperforming $\text{QOCO}_{\text{custom}}$ only on the CM4 and not on the desktop computer. This is due to the substantially smaller cache on the CM4 (1 MB of L2 cache) compared to the AMD Ryzen 9 7950X3D (16 MB of L2 cache and 128 MB of L3 cache). On the desktop processor, the $\text{QOCO}_{\text{custom}}$ binaries for all tested problems fit comfortably within the L2 cache. If we were to benchmark problems on the desktop where the $\text{QOCO}_{\text{custom}}$ binary approached the 16 MB L2 cache size, we would expect to see some performance degradation. If the binary size approached the 128 MB L3 cache limit, we expect the performance degradation to be significant and could lead to QOCO outperforming $\text{QOCO}_{\text{custom}}$. However, for problems of this scale, code generation and compilation time will be prohibitively long, on the order of several hours to even a day, making the use of QOCOGEN and $\text{QOCO}_{\text{custom}}$ impractical.



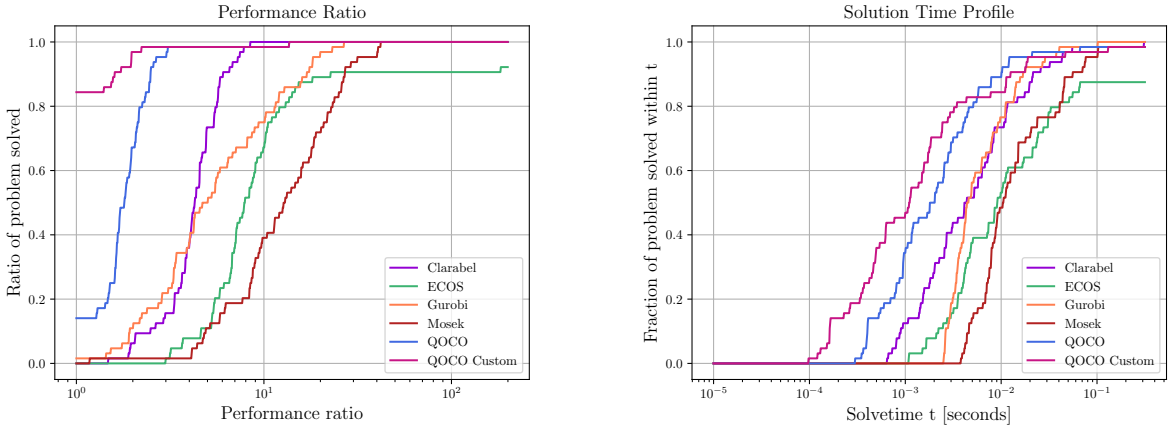
(a) Relative performance profile

(b) Absolute performance profile

	QOCO Custom	**QOCO**	**Clarabel**	**ECOS**	**Gurobi**	**Mosek**
Shifted GM	4.4	4.5	1.0	31.4	1.7	2.5
Failure Rate (%)	1.6	1.6	0.0	10.9	0.0	0.0

(c) Shifted geometric means and failure rates

Figure 3.7: Performance profiles for model-predictive control problems



(a) Relative performance profile

(b) Absolute performance profile

	QOCO Custom	QOCO	Clarabel	ECOS	Gurobi	Mosek
Shifted GM	1.3	1.1	1.4	7.0	1.0	3.6
Failure Rate (%)	1.6	1.6	0.0	10.9	0.0	0.0

(c) Shifted geometric means and failure rates

Figure 3.8: Performance profiles for model-predictive control problems on Raspberry Pi CM4

Table 3.2: Code generation time, compilation time, and resulting code and binary sizes for mpc problem instances.

Problem	Size	Codegen Time (s)	Compile Time (s)	Code Size (KB)	Binary Size (KB)
aircraft_1	504	1.4	2.7	412	182
aircraft_2	524	1.4	2.6	413	186
aircraft_3	464	1.1	2.3	376	170
aircraft_4	504	1.4	2.7	412	182
aircraft_10	584	2.2	3.8	470	198
aircraft_11	636	2.3	3.9	480	202
aircraft_12	636	2.3	3.8	480	202
aircraft_13	3116	56.0	32.7	2230	890
ballOnPlate_1	398	1.5	2.4	328	150
ballOnPlate_2	398	1.5	2.4	328	150
ballOnPlate_3	398	1.5	2.4	328	150
ballOnPlate_4	658	4.1	4.3	509	214
binaryDistillationColumn_1	2936	8.9	179.2	3107	1338
binaryDistillationColumn_2	2936	8.8	179.4	3107	1338
dcMotor_1	564	1.8	3.4	456	194
dcMotor_2	1114	6.7	18.1	858	354
dcMotor_3	5514	175.3	78.4	4184	1602
dcMotor_4	5514	175.3	78.5	4184	1602
dcMotor_5	1114	6.7	18.0	858	354
dcMotor_6	1114	6.8	17.8	857	354
doubleInvertedPendulum_1	502	1.1	2.4	390	178
doubleInvertedPendulum_2	502	1.2	2.4	389	178
doubleInvertedPendulum_3	542	1.4	2.7	425	190
fiordosExample_1	119	0.1	0.6	132	74
fiordosExample_2	139	0.2	0.7	150	78
fiordosExample_3	119	0.1	0.6	132	74

Continued on next page

Problem	Size	Codegen Time (s)	Compile Time (s)	Code Size (KB)	Binary Size (KB)
forcesExample_1	171	0.2	0.8	163	86
forcesExample_2	170	0.2	0.8	161	86
forcesExample_3	189	0.3	0.8	174	90
forcesExample_4	369	1.1	1.8	287	134
helicopter_1	201	0.3	0.9	197	98
helicopter_2	1051	7.7	14.0	863	378
helicopter_3	536	2.1	3.2	449	186
nonlinearCstr_1	1164	7.2	16.3	829	338
nonlinearCstr_2	1164	7.2	16.5	829	338
nonlinearCstr_3	294	0.5	1.3	247	114
pendulum_1	435	1.1	2.0	332	154
pendulum_2	432	1.1	2.0	329	150
pendulum_3	155	0.1	0.7	153	86
quadcopter_1	1592	15.0	118.2	1770	782
quadcopter_2	3172	60.0	57.0	3656	1594
quadcopter_3	7912	409.8	197.9	9408	3974
quadcopter_4	3172	60.0	57.0	3655	1594
quadcopter_5	3072	51.3	55.1	3686	1590
quadcopter_6	1678	15.7	123.5	1797	794
robotArm_1	276	0.6	1.3	243	114
robotArm_2	1056	8.9	12.7	795	330
shell_1	739	4.1	15.7	974	458
shell_2	1469	16.0	146.7	2017	858
shell_3	739	4.0	15.7	974	458
spacecraft_1	1137	8.6	15.6	835	338
spacecraft_2	1137	8.6	15.8	835	338
springMass_1	21239	2521.6	567.3	17247	6326
springMass_2	2159	20.4	178.8	1704	658

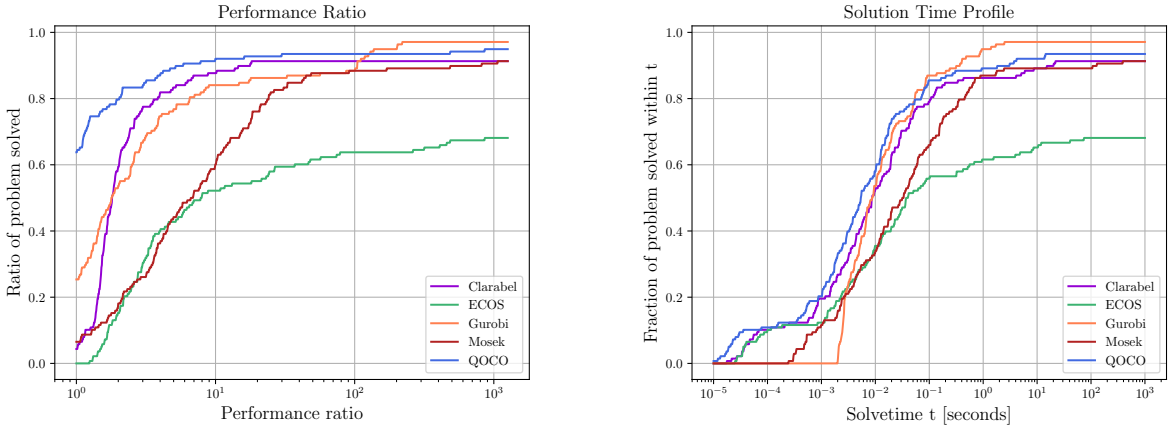
Continued on next page

Problem	Size	Codegen Time (s)	Compile Time (s)	Code Size (KB)	Binary Size (KB)
springMass_3	2144	20.3	167.4	1690	654
springMass_4	4279	82.3	53.5	3382	1314
toyExample_1	199	0.3	0.8	178	90
toyExample_2	389	1.1	1.9	296	138
toyExample_3	199	0.3	0.8	178	90
toyExample_4	959	6.5	8.2	650	278
toyExample_5	1909	26.4	99.8	1259	490
tripleInvertedPendulum_1	2739	22.2	272.7	2109	822
tripleInvertedPendulum_2	2919	28.2	28.3	2291	926
tripleInvertedPendulum_3	2919	28.1	28.3	2289	926

3.5.3 Maros–Mészáros problems

The Maros–Mészáros problems are a set of 138 challenging QPs that include a wide range of problem sizes (from 3 to around 300,000 variables) and contain very difficult problems that have ill-conditioning and poor scaling [130]. These properties make the Maros–Mészáros problems a good test set to assess the robustness of solvers. We test QOCO but not $\text{QOCO}_{\text{custom}}$ on these problems since many problems are far too large to generate code for. Additionally, we run each problem once rather than the 100 runs we do for the benchmark and MPC problems since many of the Maros–Mészáros problems are extremely large, and running them 100 times per solver is extremely cumbersome.

Figure 3.9 shows that the proprietary solver Gurobi successfully solved the largest fraction of the problems followed by QOCO, CLARABEL, MOSEK, and ECOS. We can also see that for the majority of the problems, QOCO was the fastest solver, whereas MOSEK and ECOS tended to be the slowest. The latter result is unsurprising as MOSEK and ECOS can only handle linear objectives and the reformulation of the quadratic objective into a second-order cone likely introduces significant fill-in, slowing these solvers down. We also observe that for many of the largest problems, Gurobi is the fastest solver, due to multithreading in its matrix factorization.



(a) Relative performance profile

(b) Absolute performance profile

	QOCO	Clarabel	ECOS	Gurobi	Mosek
Shifted GM	2.4	3.6	33.9	1.0	4.1
Failure Rate (%)	6.5	8.7	31.9	2.9	8.7

(c) Shifted geometric means and failure rates

Figure 3.9: Performance profiles for Maros–Mészáros problems

3.5.4 SuiteSparse least-squares problems

Finally, we consider a set of 23 least-squares problems $Ax \approx b$, where the matrix $A \in \mathbb{R}^{m \times n}$ is drawn from the SuiteSparse Matrix Collection [65]. Following [177] and [92], we get an approximate solution to each least-squares problem by solving a Huber regression and lasso regression problem for a total of 46 problems, which can both be formulated as constrained quadratic programs [32]. These optimization problems are quite large with some having over a million optimization variables, so we do not test $\text{QOCO}_{\text{custom}}$ due to the prohibitively long code-generation time. Additionally, we run each problem once rather than 100 times.

The Huber regression problem is given by

$$\underset{x}{\text{minimize}} \quad \sum_{i=1}^{m} \phi(a_i^\top x - b_i),$$

where $a_i^\top$ is the i^{th} row of A and $\phi : \mathbb{R} \to \mathbb{R}$ is defined as

$$\phi(z) = \begin{cases} z^2 & \text{if } |z| \leq \delta \\ \delta(2|z| - \delta) & \text{if } |z| > \delta \end{cases},$$

where we use $\delta = 1$.

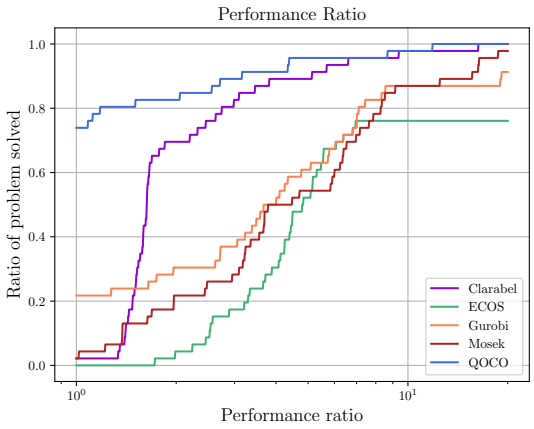
The lasso regression problem is given by

$$\underset{x}{\text{minimize}} \quad \|Ax - b\|_2^2 + \lambda \|x\|_1,$$

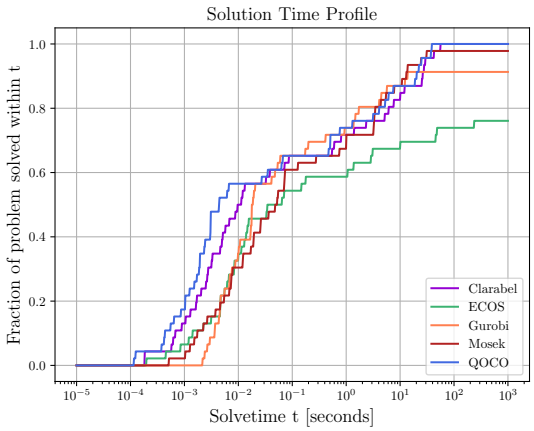
where we set $\lambda = \|A^\top b\|_\infty$.

Figure 3.10 shows that QOCO and CLARABEL successfully solve all problems, with QOCO being the fastest on most instances. After QOCO and CLARABEL, the solvers which solve the largest fraction of problems are MOSEK, GUROBI, and ECOS in that order. As with the Maros–Mészáros problems, we observe that Gurobi is the fastest on some of the largest problems, due to it multithreaded matrix factorization.

In total, there are 184 Maros–Mészáros problems and SuiteSparse least-squares problems, and GUROBI solves 176, QOCO solves 175, CLARABEL solves 172, MOSEK solves 171, and ECOS solves 129.



(a) Relative performance profile



(b) Absolute performance profile

	QOCO	Clarabel	ECOS	Gurobi	Mosek
Shifted GM	1.0	1.2	7.6	1.6	1.2
Failure Rate (%)	0.0	0.0	23.9	8.7	2.2

(c) Shifted geometric means and failure rates

Figure 3.10: Performance profiles for SuiteSparse least-squares problems

3.6 Conclusion

We have presented QOCO, a C-based solver, and QOCOGEN, a custom solver generator for quadratic objective second-order cone programs (SOCPs). Both implement primal-dual interior-point methods, with QOCOGEN generating custom solvers written in C, called $\text{QOCO}_{\text{custom}}$, that exploit the sparsity pattern of problems to gain a computational advantage. We demonstrate that QOCO is a fast and robust solver, and $\text{QOCO}_{\text{custom}}$ is significantly faster than QOCO, making it a useful tool for real-time applications, such as model-predictive control, trajectory optimization, and other domains where computational efficiency is critical.

Since both QOCO and QOCOGEN are open-source, they are accessible to users in both academia and industry. QOCO and QOCOGEN are integrated with CVXPY and CVXPYGEN respectively, allowing users to formulate optimization problems in a natural way following

from math rather than manually converting the problem to the solver-required standard form. This integration enhances usability of our solvers.

Documentation and installation instructions for QOCO and QOCOGEN are available at https://qoco-org.github.io/qoco/index.html.

Chapter 4

GPU ACCELERATION FOR QOCO

We present a GPU-accelerated backend for QOCO, a C-based solver for quadratic objective second-order cone programs (SOCPs) based on a primal-dual interior point method. Our backend uses NVIDIA's cuDSS library to perform a direct sparse $LDL^\top$ factorization of the KKT system at each iteration. We also develop custom CUDA kernels for cone operations and show that parallelizing these operations is essential for achieving peak performance. Additionally, we refactor QOCO to introduce a modular backend abstraction that decouples solver logic from the underlying linear algebra implementations, allowing the existing CPU and new GPU backend to share a unified codebase. This GPU backend is accessible through a direct Python interface and through CVXPY, allowing for easy use. Numerical experiments on a range of large-scale quadratic programs and SOCPs with tens to hundreds of millions of nonzero elements in the KKT matrix, demonstrate speedups of up to 50-70 times over the CPU implementation.

4.1 Introduction

We consider the following quadratic objective second-order cone program (SOCP)

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^\top Px + c^\top x \\ \text{subject to} \quad & Gx \preceq_{\mathcal{K}} h \\ & Ax = b, \end{aligned} \tag{4.1}$$

with optimization variable $x \in \mathbb{R}^n$. The objective is defined by $P \succeq 0$ and $c \in \mathbb{R}^n$. The equality and conic constraints are defined by $A \in \mathbb{R}^{p\times n}$, $G \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^p$, and $h \in \mathbb{R}^m$. The conic inequality $Gx \preceq_{\mathcal{K}} h$ denotes $h - Gx \in \mathcal{K}$, where $\mathcal{K}$ is the Cartesian product

$$\mathcal{K} := \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_K,$$

and $\mathcal{C}_k$ is either the non-negative orthant or a second-order cone. We assume that Problem (4.1) is **feasible** and has a **bounded** optimal objective.

In this work, we focus on solving *large-scale* instances of Problem (4.1) with tens to hundreds of millions of nonzero elements in the KKT matrix using the QOCO solver [46], a C-based solver for quadratic objective SOCPs that implements a primal-dual interior point method [186].

4.1.1 Related work

While GPUs have dramatically accelerated machine learning workloads, leveraging them in optimization algorithms has historically been challenging. The main difficulty is that sparse linear algebra, especially sparse matrix factorizations, is significantly more challenging to accelerate than dense linear algebra [161]. As a result, most GPU accelerated optimization methods have relied on algorithms that avoid direct sparse factorizations [144, 170, 176].

Two main approaches have been considered: first-order optimization methods such as PDLP [121], which require only matrix-vector multiplications that are straightforward to accelerate on a GPU, and using indirect methods such as the preconditioned conjugate gradient method to solve linear systems rather than direct sparse factorizations [144, 170, 176]. However, the first-order methods are highly sensitive to problem conditioning and can struggle to converge to high accuracy solutions.

Interior-point methods (IPMs), however, are more robust and can achieve high accuracy, but have required using indirect methods to solve the linear system for GPU implementation [176]. These indirect methods require only matrix-vector multiplications, which allow them to be accelerated on a GPU, but the iteration count of these methods depends on the condition number of the linear system, which can be extremely large, leading to long per-iteration runtime [143].

In late 2023, NVIDIA released cuDSS, a high-performance direct sparse solver for GPUs, which could be integrated into optimization algorithms. Since its release, cuDSS has been integrated as a linear system solver in the conic solvers CUCLARABEL [53] and MOREAU [14], as well as in the nonlinear programming solver MADNLP [175].

4.1.2 Contribution

We present three main contributions. First, we develop a CUDA backend for the QOCO solver that uses cuDSS to perform a direct sparse $LDL^\top$ factorization of the KKT system [1]. This enables GPU acceleration while avoiding indirect solvers, whose performance is sensitive to conditioning. Second, we implement custom CUDA kernels for parallel cone operations required by the interior-point method and show that these parallelized operations are necessary to achieve good performance on the GPU. Third, we refactor QOCO to use a modular backend abstraction that decouples solver logic from the underlying linear algebra implementation, allowing both CPU and GPU backends to share a unified codebase.

We present benchmarks that demonstrate a $50 - 70\times$ speedup over the CPU version of QOCO on some large problem instances. This GPU-accelerated version of QOCO can be called directly through a Python interface or through CVXPY [67], making it easy to use.

Compared to CUCLARABEL [53], an existing GPU-accelerated IPM for conic optimization, our C-based implementation avoids the just-in-time compilation overhead of Julia, and provides a unified interface for CPU and GPU backends within Python and CVXPY scripts, making prototyping and comparing the backends easier.

4.2 Implementation

Each iteration of QOCO's primal-dual IPM requires solving two linear systems with the same coefficient matrix (the KKT matrix) and performing various cone operations on $\mathcal{K}$. The numerical factorization of the KKT matrix can be accelerated with cuDSS and the cone

[1] Our implementation can be found at https://github.com/qoco-org/qoco

operations are parallelizable because $\mathcal{K}$ is the Cartesian product of many cones, $\mathcal{C}_k$.

4.2.1 Modular backend

To support both CPU and GPU backends without duplicating code, we refactor the main IPM loop of QOCO to be backend-agnostic. All solver code shared between the two backends operates on abstract data types rather than backend-specific data structures.

Specifically, we introduce the types `QOCOMatrix` and `QOCOVector`, which represent matrices and vectors used in QOCO. Each backend provides its own implementation of these types. The CPU implementation stores data in CPU memory, whereas the GPU implementation stores pointers to both CPU and GPU memory.

The linear system solver is abstracted through a `Linsys` interface that defines `initialize`, `factor`, `solve`, and `update` functions. Each backend provides its own implementation of this interface. The CPU backend uses the QDLDL linear system solver [177], while the GPU backend uses cuDSS.

Cone operations such as Jordan products, projecting onto cone $\mathcal{K}$, and computing the Nesterov-Todd scalings are implemented separately for the CPU and GPU backends so that these operations can be parallelized on the GPU.

At compile time, a `CMake` flag selects the desired backend and includes the corresponding implementations of `QOCOMatrix`, `QOCOVector`, `Linsys`, and the cone operations.

The Python interface exposes both backends with `pybind11`, allowing users to select the CPU or GPU implementation at runtime. This is illustrated in Listing 4.1.

4.2.2 Linear system solver

The primary operation accelerated by the GPU backend is the $LDL^\top$ factorization of the KKT system in Equation (4.2), where W_k is the Nesterov-Todd scaling matrix. In the CPU version of QOCO, this factorization dominates the runtime and its cost grows rapidly with problem size. Therefore, we accelerate this factorization on the GPU using cuDSS.

```python
# Solve with CPU backend with Python interface.
solver_cpu = qoco.QOCO(algebra="builtin")
solver_cpu.setup(n, m, p, P, c, A, b, G, h, l, nsoc, q)
result_cpu = solver_cpu.solve()

# Solve with GPU backend with Python interface.
solver_gpu = qoco.QOCO(algebra="cuda")
solver_gpu.setup(n, m, p, P, c, A, b, G, h, l, nsoc, q)
result_gpu = solver_gpu.solve()

# Solve with CPU backend in CVXPY.
problem.solve(solver="QOCO", algebra="builtin")

# Solve with GPU backend in CVXPY.
problem.solve(solver="QOCO", algebra="cuda")
```

Listing 4.1: Calling CPU and GPU backends from Python interface and CVXPY

$$\begin{bmatrix} P & A^\top & G^\top \\ A & 0 & 0 \\ G & 0 & -W_k^\top W_k \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \tag{4.2}$$

cuDSS has three phases: analysis, factor, and solve. The analysis phase is executed once and computes a fill-reducing reordering of the KKT matrix, applies this permutation to the KKT matrix, and computes a symbolic factorization that determines the sparsity pattern of the factors. This phase is the most expensive part of the solve, and for large optimization problems, it can take longer than the IPM iterations themselves. This is because cuDSS computes the reordering on the CPU, since the graph algorithms used for this step are difficult to accelerate on the GPU. The factor phase is executed once per IPM iteration and computes the $LDL^\top$ factor of the KKT matrix. The solve phase is executed twice per IPM iteration and performs triangular solves using the factored system.

4.2.3 GPU performance considerations

Simply using cuDSS as a linear system solver is not sufficient for developing a high-performance GPU backend due to certain implementation and hardware-level challenges that have to be

addressed. Since data transfer between the CPU and GPU is significantly slower than memory access within the GPU, it is important to minimize CPU-GPU data transfers. To address this, during the setup phase of the solver, all internal QOCO data structures are allocated on the CPU and then copied to the GPU. During the solve phase, all computation is performed on the GPU. After convergence, the optimal solution is copied back to the CPU.

Another issue is that all cone operations on $\mathcal{K}$ must also be executed on the GPU, which can be extremely slow if parallelism is not exploited. As discussed earlier, operations on $\mathcal{K}$ decompose into independent operations on each cone $\mathcal{C}_k$. Therefore, these operations are trivially parallelizable.

We have observed in our implementation that this parallelism should be exploited. In earlier versions of our GPU backend, the cone operations were executed serially on a single GPU core. Although this produced a correct algorithm, it was up to an order of magnitude slower, especially on problems with many second-order cones, and the time spent on cone operations exceeded the runtime of the matrix factorization, which should be the main computational bottleneck of the solve phase. The reason is that GPUs are designed for massively parallel workloads, while CPUs outperform GPUs on sequential computations. Therefore, parallelizing cone operations is not merely a performance optimization, but a requirement for an efficient GPU implementation. To do this, we implemented custom CUDA kernels for the cone operations. Each kernel implements the necessary operations for a single cone $\mathcal{C}_k$, and we map cones to GPU threads by launching a grid of thread blocks whose total number of threads matches the number of cones.

4.3 Numerical results

Here, we benchmark QOCO-GPU against the CPU implementation of QOCO, as well as CUCLARABEL, MOSEK, and GUROBI [2]. For all solvers, we use their default settings but set the tolerances $\epsilon_{\text{abs}} = \epsilon_{\text{rel}} = 10^{-7}$. All results were generated on a computer with an Intel

[2]Our benchmarks are publicly available at https://github.com/qoco-org/qoco-gpu-benchmarks

i9-14900k processor, 96 GB of RAM, and an NVIDIA GeForce RTX 5090 with 32 GB of VRAM.

In this section, problem size is defined as the total number of nonzero elements in A, G, and the upper half of P. We evaluate the solvers on a set of benchmark problems which include three quadratic programs (QPs): Huber regression, single-period portfolio optimization [129], and multi-period portfolio optimization [29], and two second-order cone programs (SOCPs): group lasso regression [198] and total variation denoising [42]. The runtime of each solver, which includes setup and solve time, is limited to an hour. To compare the performance of solvers, we use performance profiles [68] and the shifted geometric mean. Details on how these metrics are computed can be found in [46, 53, 92, 177].

4.3.1 Benchmark problems

We consider the following QPs: single period portfolio optimization, multiperiod portfolio optimization, and Huber regression.

The single period portfolio optimization problem is

$$\begin{aligned} \underset{x,y}{\text{minimize}} \quad & x^\top D x + y^\top y - \gamma^{-1}\mu^\top x \\ \text{subject to} \quad & y = F^\top x \\ & x \in \Delta_n \end{aligned}$$

where $F \in \mathbb{R}^{k\times 100k}$ for k: $\{10, 50, 100, 200, 500, 900, 1300, 1800\}$ and Δ_n is the n-dimensional unit simplex.

The multiperiod portfolio optimization problem is

$$
\begin{array}{ll}
\min\limits_{w_t, y_t} & \displaystyle\sum_{t=1}^{T} \left(w_t^\top D w_t + \|y_t\|_2^2 - \frac{1}{\gamma} \mu_t^\top w_t + \|w_t - w_{t-1}\|_2^2 \right) \\
\text{s.t.} & w_0 = \bar{w}_0 \\
& \mathbf{1}^\top w_t = 1, \qquad t = 1, \ldots, T \\
& y_t = F^\top w_t, \qquad t = 1, \ldots, T \\
& 0 \leq y_t \leq 0.01, \qquad t = 1, \ldots, T \\
& \|w_t\|_1 \leq L_{\max}, \qquad t = 1, \ldots, T,
\end{array}
$$

where $F \in \mathbb{R}^{5000 \times 50}$ is the factor loading matrix, and we solve for time horizons T: $\{2, 5, 10, 15, 25, 50, 75, 125\}$.

The Huber regression problem is given by

$$
\begin{array}{ll}
\underset{x}{\text{minimize}} & \displaystyle\sum_{i=1}^{m} \phi(a_i^\top x - b_i),
\end{array}
$$

where $a_i^\top$ denotes the i^{th} row of A and $\phi : \mathbb{R} \to \mathbb{R}$ is the Huber loss. We take $A \in \mathbb{R}^{10N \times N}$ for $N \in \{50, 200, 500, 1000, 2000, 4000, 6000, 10000\}$.

We consider the following SOCPs: group lasso regression and total variation denoising.

The group lasso regression problem is

$$
\begin{array}{ll}
\underset{x}{\text{minimize}} & \|Ax - b\|_2^2 + \lambda \displaystyle\sum_{i=1}^{N} \|x^{(i)}\|_2,
\end{array}
$$

where $x = [x^{(1)}, x^{(2)}, \ldots, x^{(N)}]$ represents a partitioning of the regression variables into groups with each $x^{(i)}$ corresponding to one group. We choose $A \in \mathbb{R}^{250N \times 10N}$ and solve for N: $\{5, 20, 50, 100, 150, 300, 450, 750\}$.

The total variation denoising problem is

$$\min_{U} \quad \mathrm{TV}(U) + \frac{\lambda}{2}\|U - Y\|_F^2,$$

where $\|\cdot\|_F$ is the Frobenius norm, TV is the total-variation operator and Y is the corrupted image. We test with the following images from the `sk-image` collection [185]: `brick`, `camera`, `grass`, `chelsea`, `coffee`, `astronaut`, `immunohistochemistry`, and `logo`.

4.3.2 Benchmark results

Figure 4.1 shows the performance profiles and shifted geometric mean across benchmark problems and Table 4.1 reports the total runtime results, including setup and solve times for each solver, where the cell corresponding to the fastest solver for each problem instance is highlighted. Table 4.1 also includes percentage of the total runtime spent in setup for QOCO-GPU when the solver is called directly through its Python interface, as this information is not available through CVXPY. Missing entries correspond to runs that exceed the one hour time limit or did not converge. For the total variation denoising problems, GUROBI failed to converge to the desired accuracy.

For smaller problems, the CPU version of QOCO outperforms QOCO-GPU, but once the KKT matrix contains around 10^5 nonzeros, the GPU version becomes faster. For the largest problems we observe speedups of up to $70\times$ over QOCO. As problems get larger, around 10^5 to 10^6 nonzeros in the KKT matrix, we also observe that both GPU solvers (QOCO-GPU and CUCLARABEL) outperform all CPU solvers (QOCO, MOSEK, and GUROBI). QOCO-GPU and CUCLARABEL exhibit similar performance on most problems, but for a few problems, such as the largest Huber regression problem and largest group lasso regression problem, QOCO-GPU is approximately $2-3\times$ faster than CUCLARABEL.

Overall, QOCO-GPU has the lowest shifted geometric mean, followed by CUCLARABEL, MOSEK, QOCO, and GUROBI. It is expected that QOCO is amongst the slowest solvers on these large problems, since it uses a single-threaded matrix factorization, whereas MOSEK and GUROBI use multithreaded factorizations.

Finally, for larger problems, up to 75% of QOCO-GPU's runtime is spent in the setup phase, where the dominant cost is the reordering step in cuDSS's analysis phase. This bottleneck has also been observed in [147] and [148]. However, the analysis phase only needs to be performed once. If the problem data changes, the reordering can be reused as long as the sparsity pattern of the problem remains fixed. This makes QOCO-GPU well suited for large parameteric optimization problems, where the initial cost of the analysis phase can be amortized over multiple solves.

Table 4.1: Runtime in seconds for benchmark problems (QOCO-GPU shows setup time percentage in parentheses)

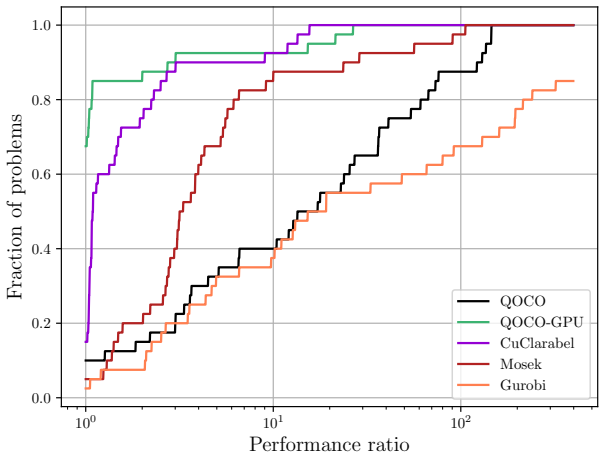
Problem	Size	QOCO-GPU	QOCO	CuClarabel	Mosek	Gurobi
huber_50	5500	0.057 (17%)	0.003	0.036	0.011	0.027
huber_200	52000	0.084 (40%)	0.031	0.069	0.097	0.069
huber_500	280000	0.165 (56%)	0.280	0.152	0.605	0.663
huber_1000	1060000	0.416 (69%)	1.868	0.431	4.153	5.278
huber_2000	4120000	1.534 (78%)	18.510	1.650	36.219	20.051
huber_4000	16240000	5.885 (79%)	159.576	6.435	331.281	89.658
huber_6000	36360000	13.167 (78%)	478.210	25.557	1390.174	252.042
huber_10000	100600000	39.816 (76%)	1459.514	107.326	-	439.726
portfolio_10	8020	0.078 (11%)	0.003	0.046	0.008	0.003
portfolio_50	140100	0.100 (36%)	0.063	0.115	0.064	0.050
portfolio_100	530200	0.159 (51%)	0.350	0.181	0.237	0.191
portfolio_200	2060400	0.393 (64%)	5.282	0.427	1.015	0.813
portfolio_500	12651000	2.206 (76%)	79.760	2.560	6.763	5.564
portfolio_900	40771800	7.501 (71%)	406.031	7.738	20.330	26.248
portfolio_1300	84892600	17.128 (67%)	1149.773	18.409	47.382	61.083
portfolio_1800	162543600	34.458 (65%)	2618.632	37.001	108.825	170.539
multiperiod_portfolio_2	385600	0.247	0.891	0.244	0.302	0.511
multiperiod_portfolio_5	956500	0.568	3.742	0.589	1.148	1.516

Continued on next page

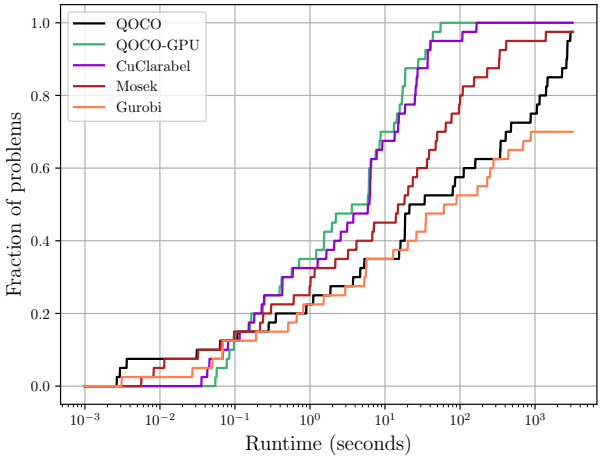
Problem	Size	QOCO-GPU	QOCO	CuClarabel	Mosek	Gurobi
multiperiod_portfolio_10	1908000	1.209	15.418	1.266	3.215	5.662
multiperiod_portfolio_15	2859500	1.955	33.728	2.097	7.102	12.838
multiperiod_portfolio_25	4762500	3.601	85.747	3.776	14.854	35.032
multiperiod_portfolio_50	9520000	8.427	346.344	9.208	76.905	277.693
multiperiod_portfolio_75	14277500	14.317	871.908	15.257	411.356	693.854
multiperiod_portfolio_125	23792500	43.295	2952.094	40.338	229.780	3221.442
group_lasso_5	8805	0.055 (19%)	0.004	0.043	0.006	0.069
group_lasso_20	110220	0.097 (41%)	0.107	0.081	0.032	2.944
group_lasso_50	650550	0.234 (69%)	1.104	0.226	0.216	34.604
group_lasso_100	2551100	0.716 (80%)	4.666	6.445	0.986	229.384
group_lasso_150	5701650	1.543 (83%)	16.067	3.134	2.171	877.229
group_lasso_300	22653300	6.551 (82%)	112.120	6.309	13.908	-
group_lasso_450	50854950	15.627 (79%)	342.902	14.973	49.441	-
group_lasso_750	141008250	54.863 (74%)	1400.521	165.127	336.563	-
tv_denoising_camera	2092037	6.110	18.302	6.229	23.394	-
tv_denoising_grass	2092037	6.064	21.103	5.880	38.575	-
tv_denoising_brick	2092037	6.141	18.471	6.381	18.190	-
tv_denoising_chelsea	2966850	8.706	1058.975	13.401	26.812	-
tv_denoising_coffee	5267013	16.804	2287.905	24.972	64.232	-
tv_denoising_astronaut	5753869	18.558	2702.704	27.056	99.934	-
tv_denoising_immunohistochemistry	5753869	18.457	2681.049	26.344	96.628	-
tv_denoising_logo	7233017	27.720	-	36.955	153.639	-

4.4 Limitations

Although the GPU backend provides substantial speedups on large problems, there are a few limitations. The sparse direct factorization can lead to significant fill-in, which increases memory usage and can lead to out-of-memory errors on the GPU for extremely large problems. This limitation is shared with any solvers, such as CUCLARABEL, that rely on sparse direct methods. Additionally, the GPU backend only offers a speedup for sufficiently large



(a) Relative performance profile



(b) Absolute performance profile

	QOCO-GPU	QOCO	CuClarabel	Mosek	Gurobi
Shifted GM	**1.00**	11.86	1.24	3.79	19.22
Failure Rate (%)	**0.0**	2.5	**0.0**	2.5	27.5

(c) Shifted geometric means and failure rates

Figure 4.1: Performance profiles for benchmark problems

problems. For smaller problems, the overhead of the GPU kernel launches and the CPU-GPU memory transfers can outweigh the speedups offered by the GPU. Finally, the runtime bottleneck of this backend on large problems is the runtime of cuDSS's analysis phase.

Chapter 5

FAST DOUGLAS–RACHFORD SPLITTING

When minimizing the sum of a convex and a strongly convex function, or when finding the zero of the sum of a monotone operator and a strongly monotone operator, Chambolle and Pock (2010) and Davis and Yin (2015) proposed accelerated mechanisms that achieve an $\mathcal{O}(1/N^2)$ convergence rate for the squared distance to the solution, but the optimality of this rate was not established. In this work, we present Fast Douglas–Rachford Splitting (FDR), an accelerated method that improves the constants established in the prior works, and provide a complexity lower bound establishing that both the $\mathcal{O}(1/N^2)$ convergence rate and the leading-order constant of FDR's rate are optimal.

5.1 Introduction

We consider the composite minimization problem

$$\underset{x\in\mathbb{R}^d}{\text{minimize}} \quad f(x) + g(x) \tag{5.1}$$

where $f\colon \mathbb{R}^d \to \mathbb{R}\cup\{\infty\}$ is closed, convex, and proper and $g\colon \mathbb{R}^d \to \mathbb{R}\cup\{\infty\}$ is closed, proper, and μ-strongly convex. When f and g are *proximable*, i.e., their proximal operators can be evaluated efficiently, Douglas–Rachford splitting (DRS) [70, 116]

$$\begin{aligned} x_{k+1/2} &= \operatorname{prox}_{\alpha g}(z_k) \\ x_{k+1} &= \operatorname{prox}_{\alpha f}(2x_{k+1/2} - z_k) \\ z_{k+1} &= z_k + x_{k+1} - x_{k+1/2}, \end{aligned}$$

with stepsize $\alpha > 0$, is a widely used solution method.

While convergence of DRS only requires g to be convex, strong convexity of g produces the rate

$$\|x_N - x_\star\|^2 = \mathcal{O}(1/N).$$

The problem setup can be further generalized to the monotone inclusion problem

$$\operatorname*{find}_{x \in \mathbb{R}^d} \quad 0 \in (\mathbb{A} + \mathbb{B})x \tag{5.2}$$

where $\mathbb{A}$ is maximal monotone and $\mathbb{B}$ is maximal monotone and μ-strongly monotone.

For a maximal monotone operator $\mathbb{A}$, let

$$\mathbb{J}_{\alpha\mathbb{A}} = (\mathbb{I} + \alpha\mathbb{A})^{-1}$$

denote its resolvent. Douglas–Rachford splitting applied to (5.2), with stepsize $\alpha > 0$, is

$$\begin{aligned} x_{k+1/2} &= \mathbb{J}_{\alpha\mathbb{B}}(z_k) \\ x_{k+1} &= \mathbb{J}_{\alpha\mathbb{A}}(2x_{k+1/2} - z_k) \\ z_{k+1} &= z_k + x_{k+1} - x_{k+1/2}, \end{aligned}$$

and attains the same $\mathcal{O}(1/N)$ rate under μ-strong monotonicity of $\mathbb{B}$.

In optimization, an *acceleration* is a modification of a base algorithm that improves the worst-case convergence rate of a given performance measure. While the classical Nesterov acceleration [141] achieves the optimal accelerated rate for function-value suboptimality, the modern theory of accelerations considers a broader range of performance measures, including distance to the solution and the squared fixed-point residual. In particular, the Chambolle–Pock [43] and Davis–Yin splitting [63] methods admit accelerated variants—based on mechanisms distinct from Nesterov's acceleration—that can be applied to Problem (5.1) to obtain an accelerated rate of $\mathcal{O}(1/N^2)$ for the squared distance to the optimal solution. However,

it was unknown whether this $\mathcal{O}(1/N^2)$ rate is optimal for this problem class, or whether the leading constants achieved by these accelerated methods are tight.

Contributions. This work presents two main contributions. First, we present an improved accelerated algorithm for Problems (5.1) and (5.2) whose leading constant improves upon those of the accelerated Chambolle–Pock and Davis–Yin methods by a factor of 4. Specifically, the iterates satisfy

$$\|x_N - x_\star\|^2 \leq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2} \sim \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{4N^2\mu^2},$$

where $x_\star$ denotes an optimal solution, $u_\star$ is a corresponding dual solution, x_0 and u_0 are the primal and dual initializations respectively, μ is the strong convexity parameter, and N is the iteration count. Second, we establish a matching complexity lower bound showing that both the $\mathcal{O}(1/N^2)$ convergence rate and the leading-order term $(\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2)/4N^2\mu^2$ are optimal for this problem class.

5.1.1 Notation and preliminaries

Throughout, $\mathbb{N}$ denotes the set of natural numbers, $\mathbb{R}^d$ the d-dimensional Euclidean space, $\langle \cdot, \cdot \rangle$ the standard inner product, and $\|\cdot\|$ the induced norm.

We recall standard definitions from convex analysis and monotone operator theory [164, 16, 166].

A function $f\colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ is *closed* if its epigraph, defined as

$$\operatorname{epi} f = \{(x, t) \in \mathbb{R}^d \times \mathbb{R} \mid f(x) \leq t\},$$

is a closed set. It is *convex* if

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y) \quad \forall x, y \in \mathbb{R}^d \text{ and } \theta \in [0, 1],$$

and *proper* if it never takes the value $-\infty$ and has finite value for some $x_0 \in \mathbb{R}^d$.

The *subdifferential* of f at x is

$$\partial f(x) = \{v \in \mathbb{R}^d \mid f(y) \geq f(x) + \langle v, y - x\rangle, \forall y \in \mathbb{R}^d\}.$$

We write $f'(x) \in \partial f(x)$ to denote a subgradient.

A function $g\colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ is μ-strongly convex if $g(x) - \frac{\mu}{2}\|x\|^2$ is convex, or equivalently, if for all $x, y \in \mathbb{R}^d$ and all $v \in \partial g(x)$,

$$g(y) \geq g(x) + \langle v, y - x\rangle + \frac{\mu}{2}\|y - x\|^2.$$

If f is closed, convex, and proper, then its subdifferential ∂f is a *monotone operator*, which means that

$$\langle v - w, x - y\rangle \geq 0 \quad \forall x, y \in \mathbb{R}^d,\ \forall v \in \partial f(x),\ \forall w \in \partial f(y).$$

Similarly, if g is closed, proper, and μ-strongly convex, then ∂g is a *μ-strongly monotone operator*, meaning that

$$\langle v - w, x - y\rangle \geq \mu\|x - y\|^2 \quad \forall x, y \in \mathbb{R}^d,\ \forall v \in \partial g(x),\ \forall w \in \partial g(y).$$

For a function $f : \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$, define

$$\operatorname*{argmin}_{x \in \mathbb{R}^d} f(x) = \{z \in \mathbb{R}^d \mid f(z) \leq f(x) \quad \forall x \in \mathbb{R}^d\}.$$

For a closed, convex, and proper function f and $\gamma > 0$, the proximal operator $\operatorname{prox}_{\gamma f} : \mathbb{R}^d \to \mathbb{R}^d$ is

$$\operatorname{prox}_{\gamma f}(x) = \operatorname*{argmin}_{z \in \mathbb{R}^d} \left\{ f(z) + \frac{1}{2\gamma}\|z - x\|^2 \right\}$$

From the optimality conditions of this minimization problem, if $y = \operatorname{prox}_{\gamma f}(x)$, then

$$0 \in \partial f(y) + \frac{1}{\gamma}(y - x).$$

Equivalently,

$$y = x - \gamma f'(y).$$

Consider the primal-dual problem pair

$$\underset{x \in \mathbb{R}^d}{\text{minimize}} \quad f(x) + g(x) \tag{$\mathcal{P}$}$$

$$\underset{u \in \mathbb{R}^d}{\text{maximize}} \quad -f^*(-u) - g^*(u) \tag{$\mathcal{D}$}$$

where $f \colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ is closed, convex, and proper, and $g : \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ is closed, proper, and μ-strongly convex. Assume that the primal problem ($\mathcal{P}$) admits a minimizer $x_\star$, which is unique by strong convexity of g, and total duality holds. Let $u_\star \in \partial g(x_\star)$ denote a corresponding dual solution to ($\mathcal{D}$). Then necessarily $-u_\star \in \partial f(x_\star)$ [166].

Lemma 1. *If $(x_\star, u_\star)$ is a primal-dual solution (i.e. $u_\star \in \partial g(x_\star)$ and $-u_\star \in \partial f(x_\star)$), then $x_\star = \operatorname{prox}_{\gamma g}(x_\star + \gamma u_\star)$ and $x_\star = \operatorname{prox}_{\gamma f}(x_\star - \gamma u_\star)$ for $\gamma > 0$.*

Proof. We can verify the equalities by writing optimality conditions for the proximal operator

$$\begin{aligned} 0 &\in \partial g(x_\star) + \frac{1}{\gamma}\big(x_\star - (x_\star + \gamma u_\star)\big) \\ &= \partial g(x_\star) - u_\star. \end{aligned}$$

Since $u_\star \in \partial g(x_\star)$, the inclusion holds.

$$
\begin{aligned}
0 &\in \partial f(x_\star) + \frac{1}{\gamma}\big(x_\star - (x_\star - \gamma u_\star)\big) \\
&= \partial f(x_\star) + u_\star.
\end{aligned}
$$

Since $-u_\star \in \partial f(x_\star)$, the inclusion holds.

□

5.1.2 Acceleration mechanisms and performance measures

We quickly review other acceleration mechanisms that have been published in prior works. Note that each of these acceleration mechanisms is distinct in the sense that the metric they accelerate differs.

Nesterov-type Acceleration. The classical Nesterov accelerated gradient method (NAG) reduces the function-value suboptimality at an accelerated $\mathcal{O}(1/N^2)$ rate [141]. In the modern literature, the Optimized Gradient Method (OGM) [73, 107, 182] improves the leading constant of NAG by a factor of 2 and attains the exact optimal worst-case rate for this setup with an exact matching complexity lower bound [71].

In the composite minimization setup where one function is smooth and the other is proximable, NAG can be extended to FISTA [18] and OGM to OptISTA [97]. Analogously, OptISTA is exactly optimal for the smooth composite convex minimization setup and improves the leading constant of FISTA by a factor of 2.

In Problem (5.1), however, neither function is assumed to be smooth, so Nesterov-type acceleration does not apply directly. To clarify the distinction, suppose instead that g were L-smooth rather than μ-strongly convex. Then FISTA applied to Problem (5.1) is

$$\begin{aligned} x_{k+1} &= \operatorname{prox}_{f/L}\left(y_k - \frac{1}{L}\nabla g(y_k)\right) \\ y_{k+1} &= x_{k+1} + \left(\frac{t_k - 1}{t_{k+1}}\right)(x_{k+1} - x_k) \end{aligned} \tag{FISTA}$$

where

$$y_1 = x_1 \in \mathbb{R}^d,\ t_1 = 1,\ t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2}.$$

Fact 1. *If $\{x_k\}$ is generated by FISTA, then for any $N \geq 1$, the following bound is satisfied for any minimizer $x_\star$ [18, Theorem 4.4]*

$$f(x_N) + g(x_N) - f(x_\star) - g(x_\star) \leq \frac{2L\|x_1 - x_\star\|^2}{N^2}$$

Nesterov acceleration can be applied to the corresponding dual of (5.1), since the dual involves the sum of a convex and a smooth, convex function. However, the quantity that is reduced at an accelerated rate in this case is the dual-function-value suboptimality, and this is different from the distance to the solution.

Halpern Acceleration. The Halpern acceleration [94], also referred to as the anchor acceleration in the modern literature, reduces the squared fixed-point residual at an accelerated $\mathcal{O}(1/N^2)$ rate [106, 115]. Specifically, suppose $\mathbb{T}\colon \mathbb{R}^d \to \mathbb{R}^d$ is a non-expansive operator. Then, the optimized Halpern method (OHM)

$$x_{k+1} = \frac{1}{k+2}x_0 + \frac{k+1}{k+2}\mathbb{T}x_k$$

exhibits the rate

$$\|\mathbb{T}x_{N-1} - x_{N-1}\|^2 \leq \frac{4\|x_0 - x_\star\|^2}{N^2},$$

where $x_\star \in \operatorname{Fix}\mathbb{T}$ (i.e. $x_\star = \mathbb{T}x_\star$).

OHM can be applied directly to Problem (5.1) by taking $\mathbb{T}$ to be the Peaceman–Rachford

splitting (PRS) operator. This algorithm is

$$\begin{aligned} y_{k+1} &= 2\mathrm{prox}_{\alpha f}(x_k) - x_k \\ x_{k+1} &= \frac{1}{k+2}x_0 + \frac{k+1}{k+2}(2\mathrm{prox}_{\alpha g}(y_{k+1}) - y_{k+1}), \end{aligned} \tag{OHM}$$

where $\alpha > 0$. This yields the rate

$$\|\mathbb{T}_{\mathrm{PRS}} x_{N-1} - x_{N-1}\|^2 = \mathcal{O}(1/N^2),$$

where $\mathbb{T}_{\mathrm{PRS}} = (2\mathrm{prox}_{\alpha g} - \mathbb{I}) \circ (2\mathrm{prox}_{\alpha f} - \mathbb{I})$ is the PRS operator.

However, the quantity that is reduced at an accelerated rate in this case is the fixed-point residual of the PRS operator, and this is different from the distance to the solution.

Chambolle–Pock–Davis–Yin Acceleration. The accelerated Chambolle and Pock method [43, Alg. 2] applied to (5.1) is

$$\begin{aligned} u_{k+1} &= u_k - \sigma_k z_k + \sigma_k \mathrm{prox}_{f/\sigma_k}(z_k - (1/\sigma_k)u_k) \\ x_{k+1} &= \mathrm{prox}_{\tau_k g}(x_k + \tau_k u_{k+1}) \\ z_{k+1} &= x_{k+1} + \theta_k(x_{k+1} - x_k), \end{aligned} \tag{CP}$$

where

$$\tau_{k+1} = \theta_k \tau_k,\ \sigma_{k+1} = \sigma_k/\theta_k\ , \theta_k = \frac{1}{\sqrt{1 + 2\mu\tau_k}},$$

and $\tau_0, \sigma_0 > 0$ with $\tau_0\sigma_0 \leq 1$, and $z_0 = x_0$. Similarly, the accelerated Davis–Yin method [63, Alg. 2] applied to (5.1) is

$$\begin{aligned} x_0 &= \operatorname{prox}_{\gamma_0 g}(y_0) \\ u_0 &= \frac{1}{\gamma_0}(y_0 - x_0) \\ x_k &= \operatorname{prox}_{\gamma_{k-1} g}(y_{k-1} + \gamma_{k-1} u_{k-1}) \\ u_k &= \frac{1}{\gamma_{k-1}}(y_{k-1} + \gamma_{k-1} u_{k-1} - x_k) \\ y_k &= \operatorname{prox}_{\gamma_k f}(x_k - \gamma_k u_k), \end{aligned} \tag{DYS}$$

where $y_0 \in \mathbb{R}^d$ is an initialization, and the proximal stepsizes are given by

$$\gamma_{k+1} = \frac{\gamma_k}{\sqrt{1 + 2\gamma_k \mu}} \text{ for } k = 0, \ldots, N-1,$$

where $\gamma_0 \in (0, \infty)$.

Fact 2. *If $\{x_k\}$ is generated by CP with $\tau_0\sigma_0 = 1$, then for any $\epsilon > 0$, there exists N_0 such that for all $N \geq N_0$ [43, Theorem 2],*

$$\|x_N - x_\star\|^2 \leq \frac{1+\epsilon}{N^2}\left(\frac{\|x_0 - x_\star\|^2}{\mu^2 \tau_0^2} + \frac{\|u_0 - u_\star\|^2}{\mu^2}\right),$$

where $x_\star$ is the minimizer of Problem (5.1) and $u_\star \in \partial g(x_\star)$, equivalently $-u_\star \in \partial f(x_\star)$, is a corresponding dual solution.

Fact 3. *If $\{x_k\}$ is generated by DYS and $x_\star$ is the minimizer of Problem (5.1), then $\|x_N - x_\star\|^2 = \mathcal{O}(1/N^2)$ [63, Theorem 1.2].*

Conceptually, our method in Section 5.2 can be viewed as a Chambolle–Pock–Davis–Yin-type acceleration, but further refined to achieve the optimal constant in the leading order term.

5.1.3 Prior works

Several prior works have sharpened the convergence theory for Douglas–Rachford splitting [70, 116] and its variants under additional regularity assumptions on the functions in (5.1) or operators in (5.2). Giselsson and Boyd [87] prove a tight global linear convergence of Douglas–Rachford splitting when one of the functions is both strongly convex and smooth. Davis and Yin [62] establish improved convergence rates for Douglas–Rachford splitting and ADMM under several regularity assumptions, including strong convexity, smoothness, and linear regularity. Giselsson [86] proves a tight global linear convergence rate for Douglas–Rachford splitting applied to Problem (5.2) assuming one operator is strongly monotone and cocoercive and when one operator is strongly monotone and the other is cocoercive. Ryu et al. [165] develop a performance-estimation framework for operator splitting methods and use it to derive tight contraction factors for Douglas–Rachford splitting in monotone inclusion problems like Problem (5.2) under assumptions such as strong monotonicity, cocoercivity, and Lipschitz continuity. Condat et al. [57] provide a unified treatment of accelerated proximal splitting that recovers the Chambolle–Pock and Davis–Yin methods as special cases, and establish last-iterate $\mathcal{O}(1/N^2)$ rates. Condat et al. [56] propose a stochastic algorithm that recovers the Davis–Yin algorithm as a special case and admits accelerated rates under strong convexity. Patrinos et al. [154] analyze DRS through the Douglas–Rachford envelope and derive accelerated variants in that framework, although their assumptions and performance measures differ from ours. ADMM is an algorithm closely related to DRS, and its linear convergence was shown by França and Bento [83] and Deng and Yin [66]. More recently, Briceño-Arias and Roldán [34] proposed a modified Peaceman–Rachford splitting method, a closely related variant of Douglas–Rachford splitting [78], for problems in which the two functions have combinations of strong convexity and smoothness. The proposed method achieves an improved linear convergence rate relative to previously known bounds in that setting.

If one term in (5.1) is additionally smooth, the problem can be treated by forward–

backward splitting [36, 152] and accelerated variants such as FISTA [18] and APGD [58]. Briceño-Arias [35] studies this smooth composite setting when both smooth and nonsmooth terms may contribute strong convexity and shows that the sharpest theoretical rate for FISTA is obtained by leveraging all available strong convexity through the smooth term.

5.2 Fast Douglas–Rachford Splitting

We now present Fast Douglas–Rachford Splitting (FDR):

$$\begin{aligned} w_0 &= x_0 - \eta_0 u_0 \\ y_{k+1} &= \operatorname{prox}_{\eta_k g}(2x_k - w_k) \\ w_{k+1} &= \left(1 + \frac{\eta_{k+1}}{\eta_k}\right) y_{k+1} - \left(\frac{\eta_{k+1}}{\eta_k}\right)(2x_k - w_k) \\ x_{k+1} &= \operatorname{prox}_{\eta_{k+1} f}(w_{k+1}) \end{aligned} \tag{FDR}$$

for $k = 0, \ldots, N-1$, where $N \in \mathbb{N}$ is a pre-specified total iteration count, $x_0, u_0 \in \mathbb{R}^d$ are respectively primal and dual starting points, and the proximal stepsizes are given by

$$\eta_k = \frac{2N\mu}{1 + 4kN\mu^2} \qquad \text{for } k = 0, \ldots, N.$$

In Section 5.2.1, we establish the following convergence guarantee for FDR.

Theorem 2. *Let $N \geq 1$ and let $f\colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ be closed, convex, and proper, and $g\colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ be closed, proper, and μ-strongly convex with $\mu > 0$. Assume that a primal solution $x_\star \in \operatorname{argmin}(f + g)$ and a dual solution $u_\star$ satisfying $u_\star \in \partial g(x_\star)$, $-u_\star \in \partial f(x_\star)$ exist. Then, the iterates generated by FDR satisfy the rate*

$$\|x_N - x_\star\|^2 \leq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2}.$$

Discussion. For FDR, the total number of iterations N must be specified *a priori*, as the proximal step-size sequence $\{\eta_k\}_{k=0}^N$ explicitly depends on N. Additionally, Theorem 2

provides a guarantee on the final iterate, x_N. Both properties are shared by exact optimal methods such as OGM and OptISTA.

We observe that if the step-sizes are constant ($\eta_0 = \eta_1 = \cdots = \eta_N$), our algorithm reduces to Peaceman–Rachford splitting up to our choice of initialization [155].

Moreover, as $N, k \to \infty$ the ratio $\eta_{k+1}/\eta_k \to 1$, so the later FDR iterations resemble Peaceman–Rachford splitting with varying, decreasing proximal stepsizes.

Theorem 2 shows that FDR improves the leading asymptotic constant of the accelerated Chambolle–Pock and Davis–Yin methods by a factor of four. Details are provided in Appendix B.1.

5.2.1 Proof of Theorem 2

We establish the convergence rate of FDR with a Lyapunov analysis, which is a widely used proof template in the analysis of first-order methods [97, 107, 150, 151]. For notational simplicity, define

$$R^2 = \|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2.$$

Specifically, define

$$\begin{aligned}
\mathcal{V}_{-1} &= \nu R^2 - \nu^2 \| - 2N\mu(x_0 - x_\star) + (u_0 - u_\star)\|^2 \\
\mathcal{V}_0 &= \nu^2 \|x_\star + 2N\mu u_\star - x_0 - 2N\mu u_0\|^2 \\
\mathcal{V}_k &= \nu^2 \|(1 + 4kN\mu^2)x_\star + 2N\mu u_\star - (1 + 4kN\mu^2)(2x_k - w_k)\|^2 \quad \forall k = 1, \dots, N-1 \\
\mathcal{V}_N &= \|x_N - x_\star\|^2 + 4N^2\mu^2\nu^2 \|u_\star + f'(x_N)\|^2,
\end{aligned}$$

where x_k and w_k are generated by (FDR), $f'(x_N) = (w_N - x_N)/\eta_N$, and

$$\nu = \frac{1}{1 + 4N^2\mu^2}.$$

Once we establish that the Lyapunov sequence is dissipative, i.e., that

$$\mathcal{V}_N \leq \mathcal{V}_{N-1} \leq \cdots \leq \mathcal{V}_1 \leq \mathcal{V}_0 \leq \mathcal{V}_{-1},$$

it follows immediately that

$$\|x_N - x_\star\|^2 \leq \mathcal{V}_N \leq \cdots \leq \mathcal{V}_{-1} \leq \nu R^2 = \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2},$$

which implies the stated rate

$$\|x_N - x_\star\|^2 \leq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2}.$$

It remains to show that the Lyapunov sequence is non-increasing.

Case 1: $\mathcal{V}_0 \leq \mathcal{V}_{-1}$

$$\begin{aligned}
\mathcal{V}_0 - \mathcal{V}_{-1} &= \nu^2\|x_\star + 2N\mu u_\star - x_0 - 2N\mu u_0\|^2 - \nu R^2 + \nu^2\| - 2N\mu(x_0 - x_\star) + (u_0 - u_\star)\|^2 \\
&= \nu^2\|x_\star - x_0 + 2N\mu(u_\star - u_0)\|^2 + \nu^2\|2N\mu(x_\star - x_0) + (u_0 - u_\star)\|^2 - \nu R^2 \\
&= \nu^2\left[(1 + 4N^2\mu^2)\|x_0 - x_\star\|^2 + (1 + 4N^2\mu^2)\|u_0 - u_\star\|^2\right] - \nu R^2 \\
&= \nu(\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2 - R^2) \\
&= 0
\end{aligned}$$

Case 2: $\mathcal{V}_1 \leq \mathcal{V}_0$ for $N \geq 2$

First, we will give some formulas and definitions.

$$\|a\|^2 - \|b\|^2 = \langle a+b, a-b\rangle \tag{5.3a}$$

$$\Omega = 1 + 4N\mu^2 \tag{5.3b}$$

$$\Omega\eta_1 = \eta_0 \tag{5.3c}$$

$$g'(y_1) = \frac{x_0 + \eta_0 u_0 - y_1}{\eta_0} \tag{5.3d}$$

$$f'(x_1) = \frac{w_1 - x_1}{\eta_1} \tag{5.3e}$$

$$f'(x_1) + g'(y_1) = \frac{(y_1 - x_1)\Omega}{\eta_0} \tag{5.3f}$$

$$2\eta_0(1+\Omega)\mu = \Omega^2 - 1 \tag{5.3g}$$

$$\begin{aligned}
\mathcal{V}_1 - \mathcal{V}_0 &= \nu^2[\|\Omega x_\star + 2N\mu u_\star - \Omega(2x_1 - w_1)\|^2 - \|x_\star + 2N\mu u_\star - x_0 - 2N\mu u_0\|^2] \\
&= \nu^2[\|\Omega x_\star + \eta_0 u_\star - \Omega(x_1 - \eta_1 f'(x_1))\|^2 - \\
&\|x_\star + \eta_0 u_\star - (y_1 + \eta_0 g'(y_1))\|^2] \qquad \triangleright \text{using (5.3d) and (5.3e)} \\
&= \nu^2[\langle \Omega x_\star + \eta_0 u_\star - \Omega x_1 + \Omega\eta_1 f'(x_1) + x_\star + \eta_0 u_\star - y_1 - \eta_0 g'(y_1), \\
&\Omega x_\star + \eta_0 u_\star - \Omega x_1 + \Omega\eta_1 f'(x_1) - x_\star - \eta_0 u_\star + y_1 + \eta_0 g'(y_1)\rangle] \qquad \triangleright \text{using (5.3a)} \\
&= \nu^2[\langle (1+\Omega)x_\star - y_1 - \Omega x_1 + 2\eta_0 u_\star + \Omega\eta_1 f'(x_1) - \eta_0 g'(y_1), \\
&(-1+\Omega)x_\star + y_1 - \Omega x_1 + \Omega\eta_1 f'(x_1) + \eta_0 g'(y_1)\rangle] \\
&= \nu^2[\langle (1+\Omega)x_\star - y_1 - \Omega x_1 + 2\eta_0 u_\star + \eta_0(f'(x_1) - g'(y_1)), \\
&(-1+\Omega)x_\star + y_1 - \Omega x_1 + (y_1 - x_1)\Omega\rangle] \qquad \triangleright \text{using (5.3c) and (5.3f)} \\
&= \nu^2[\langle (1+\Omega)x_\star - y_1 - \Omega x_1 + 2\eta_0 u_\star + \eta_0(f'(x_1) - g'(y_1)), (1+\Omega)(y_1 - x_\star) + 2\Omega(x_\star - x_1)\rangle] \\
&= \nu^2[\langle (1+\Omega)(y_1 - x_\star), (1+\Omega)x_\star - y_1 - \Omega x_1 + 2\eta_0 u_\star + \eta_0(f'(x_1) - g'(y_1))\rangle + \\
&\langle 2\Omega(x_\star - x_1), (1+\Omega)x_\star - y_1 - \Omega x_1 + 2\eta_0 u_\star + \eta_0(f'(x_1) - g'(y_1))\rangle] \\
&= \nu^2[\langle (1+\Omega)(y_1 - x_\star), (1+\Omega)x_\star - y_1 - \Omega x_1 + 2\eta_0 u_\star + (y_1 - x_1)\Omega - 2\eta_0 g'(y_1)\rangle +
\end{aligned}$$

$$\langle 2\Omega(x_\star - x_1), (1+\Omega)x_\star - y_1 - \Omega x_1 + 2\eta_0 u_\star + 2\eta_0 f'(x_1) - (y_1 - x_1)\Omega\rangle] \qquad \triangleright \text{ using (5.3f)}$$
$$= \nu^2[\langle y_1 - x_\star, (1+\Omega)^2 x_\star + (\Omega^2 - 1)y_1 - 2\Omega(1+\Omega)x_1 + 2\eta_0(1+\Omega)(u_\star - g'(y_1))\rangle +$$
$$\langle x_\star - x_1, 2\Omega(1+\Omega)x_\star - 2\Omega(1+\Omega)y_1 + 4\eta_0\Omega(u_\star + f'(x_1))\rangle]$$
$$= \nu^2[\langle y_1 - x_\star, (1+\Omega)^2 x_\star + (\Omega^2 - 1)y_1 - 2\Omega(1+\Omega)x_\star + 2\eta_0(1+\Omega)(u_\star - g'(y_1))\rangle +$$
$$\langle x_\star - x_1, 4\eta_0\Omega(u_\star + f'(x_1))\rangle]$$
$$= \nu^2[\langle y_1 - x_\star, (\Omega^2 - 1)(y_1 - x_\star) + 2\eta_0(1+\Omega)(u_\star - g'(y_1))\rangle - 4\eta_0\Omega\langle x_\star - x_1, -u_\star - f'(x_1)\rangle]$$
$$= \nu^2\{2\eta_0(1+\Omega)[-\langle y_1 - x_\star, g'(y_1) - u_\star\rangle + \mu\|y_1 - x_\star\|^2] - 4\eta_0\Omega\langle x_\star - x_1, -u_\star - f'(x_1)\rangle\}$$
$$\leq 0,$$

where the final inequality holds by monotonicity of ∂f, μ-strong monotonicity of ∂g, since $-u_\star \in \partial f(x_\star)$, and because $2\eta_0(1+\Omega) \geq 0$ and $4\eta_0\Omega \geq 0$.

Case 3: $\mathcal{V}_{k+1} \leq \mathcal{V}_k \quad \forall k = 1, \ldots, N-2$

First we will give some important formulas and definitions.

$$g'(y_{k+1}) = \frac{2x_k - w_k - y_{k+1}}{\eta_k} \tag{5.4a}$$
$$f'(x_{k+1}) = \frac{w_{k+1} - x_{k+1}}{\eta_{k+1}} \tag{5.4b}$$
$$\Delta_k = 1 + 4kN\mu^2 \tag{5.4c}$$
$$\Delta_{k+1} = 1 + 4(k+1)N\mu^2 \tag{5.4d}$$
$$\eta_k\Delta_k = \eta_{k+1}\Delta_{k+1} = 2N\mu \tag{5.4e}$$
$$f'(x_{k+1}) + g'(y_{k+1}) = \frac{(y_{k+1} - x_{k+1})\Delta_{k+1}}{2N\mu} \tag{5.4f}$$

$$\mathcal{V}_{k+1} - \mathcal{V}_k = \nu^2[\|\Delta_{k+1}x_\star + 2N\mu u_\star - \Delta_{k+1}(2x_{k+1} - w_{k+1})\|^2$$
$$- \|\Delta_k x_\star + 2N\mu u_\star - \Delta_k(2x_k - w_k)\|^2]$$

$$= \nu^2[\|\Delta_{k+1}x_\star + 2N\mu u_\star - \Delta_{k+1}(x_{k+1} - \eta_{k+1}f'(x_{k+1}))\|^2$$

$$- \|\Delta_k x_\star + 2N\mu u_\star - \Delta_k(y_{k+1} + \eta_k g'(y_{k+1}))\|^2] \qquad \triangleright \text{from (5.4a) and (5.4b)}$$

$$= \nu^2[\langle(\Delta_{k+1} + \Delta_k)x_\star + 4N\mu u_\star - \Delta_{k+1}(x_{k+1} - \eta_{k+1}f'(x_{k+1})) - \Delta_k(y_{k+1} + \eta_k g'(y_{k+1})),$$

$$(\Delta_{k+1} - \Delta_k)x_\star - \Delta_{k+1}(x_{k+1} - \eta_{k+1}f'(x_{k+1})) + \Delta_k(y_{k+1} + \eta_k g'(y_{k+1}))\rangle] \qquad \triangleright \text{from (5.3a)}$$

$$= \nu^2[\langle\Delta_{k+1}(x_\star - x_{k+1}) + \Delta_k(x_\star - y_{k+1}) + 4N\mu u_\star + \Delta_{k+1}\eta_{k+1}f'(x_{k+1}) - \Delta_k\eta_k g'(y_{k+1}),$$

$$\Delta_{k+1}(x_\star - x_{k+1}) - \Delta_k(x_\star - y_{k+1}) + \Delta_{k+1}\eta_{k+1}f'(x_{k+1}) + \Delta_k\eta_k g'(y_{k+1})\rangle]$$

$$= \nu^2[\Delta_{k+1}^2\|x_\star - x_{k+1}\|^2 - \Delta_k^2\|x_\star - y_{k+1}\|^2 + 2\Delta_{k+1}^2\langle x_\star - x_{k+1}, \eta_{k+1}f'(x_{k+1})\rangle$$

$$+ 2\Delta_k^2\langle x_\star - y_{k+1}, \eta_k g'(y_{k+1})\rangle + \Delta_{k+1}^2\eta_{k+1}^2\|f'(x_{k+1})\|^2 - \Delta_k^2\eta_k^2\|g'(y_{k+1})\|^2$$

$$+ \langle 4N\mu u_\star, \Delta_{k+1}(x_\star - x_{k+1}) - \Delta_k(x_\star - y_{k+1}) + \Delta_{k+1}\eta_{k+1}f'(x_{k+1}) + \Delta_k\eta_k g'(y_{k+1})\rangle]$$

$$= \nu^2[\langle x_\star - x_{k+1}, 2\Delta_{k+1}^2\eta_{k+1}f'(x_{k+1}) + 4\Delta_{k+1}N\mu u_\star + \Delta_{k+1}^2(x_\star - x_{k+1})\rangle$$

$$+ \langle x_\star - y_{k+1}, 2\Delta_k^2\eta_k g'(y_{k+1}) - 4\Delta_k N\mu u_\star - \Delta_k^2(x_\star - y_{k+1})\rangle$$

$$+ 4N^2\mu^2\langle 2u_\star + f'(x_{k+1}) - g'(y_{k+1}), f'(x_{k+1}) + g'(y_{k+1})\rangle] \qquad \triangleright \text{from (5.4e)}$$

$$= \nu^2[\langle x_\star - x_{k+1}, 2\Delta_{k+1}^2\eta_{k+1}f'(x_{k+1}) + 4\Delta_{k+1}N\mu u_\star + \Delta_{k+1}^2(x_\star - x_{k+1})\rangle$$

$$+ \langle x_\star - y_{k+1}, 2\Delta_k^2\eta_k g'(y_{k+1}) - 4\Delta_k N\mu u_\star - \Delta_k^2(x_\star - y_{k+1})\rangle$$

$$+ 2N\mu\Delta_{k+1}\langle 2u_\star + f'(x_{k+1}) - g'(y_{k+1}), y_{k+1} - x_{k+1}\rangle] \qquad \triangleright \text{from (5.4f)}$$

$$= \nu^2[\langle x_\star - x_{k+1}, 6N\Delta_{k+1}\mu f'(x_{k+1}) + 8N\Delta_{k+1}\mu u_\star - 2N\Delta_{k+1}\mu g'(y_{k+1}) +$$

$$\Delta_{k+1}^2(x_\star - x_{k+1})\rangle + \langle x_\star - y_{k+1}, 2N\mu(2\Delta_k + \Delta_{k+1})g'(y_{k+1}) - 4N\mu(\Delta_k + \Delta_{k+1})u_\star$$

$$- 2N\mu\Delta_{k+1}f'(x_{k+1}) - \Delta_k^2(x_\star - y_{k+1})\rangle] \qquad \triangleright \text{from (5.4e)}$$

$$= \nu^2[\langle x_\star - x_{k+1}, 8N\Delta_{k+1}\mu(f'(x_{k+1}) + u_\star) + \Delta_{k+1}^2(x_\star - y_{k+1})\rangle$$

$$+ \langle x_\star - y_{k+1}, 4N\mu(\Delta_k + \Delta_{k+1})g'(y_{k+1}) - 4N\mu(\Delta_k + \Delta_{k+1})u_\star + (x_{k+1} - y_{k+1})\Delta_{k+1}^2$$

$$- \Delta_k^2(x_\star - y_{k+1})\rangle] \qquad \triangleright \text{from (5.4f)}$$

$$= \nu^2[-8N\Delta_{k+1}\mu\langle x_{k+1} - x_\star, f'(x_{k+1}) + u_\star\rangle$$

$$+ \langle x_\star - y_{k+1}, 4N\mu(\Delta_k + \Delta_{k+1})(g'(y_{k+1}) - u_\star) + (\Delta_{k+1}^2 - \Delta_k^2)(x_\star - y_{k+1})\rangle]$$

$$= \nu^2\{-8N\Delta_{k+1}\mu\langle x_{k+1} - x_\star, f'(x_{k+1}) + u_\star\rangle$$

$$+ 4N\mu(\Delta_k + \Delta_{k+1})[-\langle x_\star - y_{k+1}, u_\star - g'(y_{k+1})\rangle + \mu\|x_\star - y_{k+1}\|^2]\}$$
$$\leq 0$$

where the final inequality holds by monotonicity of ∂f, μ-strong monotonicity of ∂g, since $-u_\star \in \partial f(x_\star)$, and because the coefficients $8N\Delta_{k+1}\mu$ and $4N\mu(\Delta_k + \Delta_{k+1})$ are non-negative.

Case 4: $\mathcal{V}_N \leq \mathcal{V}_{N-1}$ (including the case $\mathcal{V}_1 \leq \mathcal{V}_0$ when $N = 1$)

First, we will give some formulas and definitions.

$$g'(y_N) = \frac{2x_{N-1} - w_{N-1} - y_N}{\eta_{N-1}} \tag{5.5a}$$
$$f'(x_N) = \frac{w_N - x_N}{\eta_N} \tag{5.5b}$$
$$\Delta_{N-1} = 1 + 4(N-1)N\mu^2 \tag{5.5c}$$
$$\Delta_N = 1 + 4N^2\mu^2 \tag{5.5d}$$
$$\Delta_{N-1}\eta_{N-1} = \Delta_N\eta_N = 2N\mu \tag{5.5e}$$
$$f'(x_N) + g'(y_N) = \frac{(y_N - x_N)\Delta_N}{2N\mu} \tag{5.5f}$$

$$\begin{aligned}
\mathcal{V}_N - \mathcal{V}_{N-1} &= \|x_N - x_\star\|^2 + 4N^2\mu^2\nu^2\|u_\star + f'(x_N)\|^2 - \nu^2\|\Delta_{N-1}x_\star + 2N\mu u_\star - \Delta_{N-1}(2x_{N-1} - w_{N-1})\|^2 \\
&= \|x_N - x_\star\|^2 + 4N^2\mu^2\nu^2\|u_\star + f'(x_N)\|^2 - \nu^2\|\Delta_{N-1}x_\star + 2N\mu u_\star - \Delta_{N-1}(y_N + \eta_{N-1}g'(y_N))\|^2 \\
&= \|x_N - x_\star\|^2 + 4N^2\mu^2\nu^2\|u_\star + f'(x_N)\|^2 - \nu^2\|\Delta_{N-1}(x_\star - y_N) + 2N\mu(u_\star - g'(y_N))\|^2 \triangleright \text{(5.5e)} \\
&= \|x_N - x_\star\|^2 + \nu^2\langle 2N\mu(u_\star + f'(x_N)) + \Delta_{N-1}(x_\star - y_N) + 2N\mu(u_\star - g'(y_N)), \\
&\quad 2N\mu(u_\star + f'(x_N)) - \Delta_{N-1}(x_\star - y_N) - 2N\mu(u_\star - g'(y_N))\rangle \qquad \triangleright \text{from (5.3a)} \\
&= \|x_N - x_\star\|^2 + \nu^2\langle 2N\mu(u_\star + f'(x_N)) + \Delta_{N-1}(x_\star - y_N) + 2N\mu(u_\star - g'(y_N)), \\
&\quad (y_N - x_N)\Delta_N - \Delta_{N-1}(x_\star - y_N)\rangle \qquad \triangleright \text{from (5.5f)} \\
&= \|x_N - x_\star\|^2 + \nu^2\langle 2N\mu(u_\star + f'(x_N)) + \Delta_{N-1}(x_\star - y_N) + 2N\mu(u_\star - g'(y_N)),
\end{aligned}$$

$$
\begin{aligned}
&(\Delta_{N-1}+\Delta_N)(y_N-x_\star)-\Delta_N(x_N-x_\star)\rangle \qquad \triangleright \text{add and subtract } \Delta_N x_\star \\
&=\nu^2\langle 2N\mu(u_\star+f'(x_N))-\Delta_{N-1}(y_N-x_\star)+2N\mu(u_\star-g'(y_N)),(\Delta_{N-1}+\Delta_N)(y_N-x_\star)\rangle+ \\
&\nu^2\langle -\Delta_N(x_N-x_\star)+2N\mu(u_\star+f'(x_N))-\Delta_{N-1}(y_N-x_\star)+2N\mu(u_\star-g'(y_N)),-\Delta_N(x_N-x_\star)\rangle \\
&=\nu^2\langle 2N\mu u_\star+(y_N-x_N)\Delta_N-2N\mu g'(y_N)-\Delta_{N-1}(y_N-x_\star)+2N\mu(u_\star-g'(y_N)), \\
&(\Delta_{N-1}+\Delta_N)(y_N-x_\star)\rangle+\nu^2\langle -\Delta_N(x_N-x_\star)+2N\mu(u_\star+f'(x_N))-\Delta_{N-1}(y_N-x_\star)+ \\
&2N\mu u_\star+2N\mu f'(x_N)+(x_N-y_N)\Delta_N,-\Delta_N(x_N-x_\star)\rangle \qquad \triangleright \text{from (5.5f)} \\
&=\nu^2\langle 4N\mu(u_\star-g'(y_N))+(\Delta_N-\Delta_{N-1})(y_N-x_\star)+(x_\star-x_N)\Delta_N,(\Delta_{N-1}+\Delta_N)(y_N-x_\star)\rangle+ \\
&\nu^2\langle 4N\mu(u_\star+f'(x_N))-(\Delta_N+\Delta_{N-1})(y_N-x_\star),-\Delta_N(x_N-x_\star)\rangle \\
&=\nu^2\langle 4N\mu(u_\star-g'(y_N))+(\Delta_N-\Delta_{N-1})(y_N-x_\star),(\Delta_{N-1}+\Delta_N)(y_N-x_\star)\rangle+ \\
&\nu^2\langle 4N\mu(u_\star+f'(x_N)),-\Delta_N(x_N-x_\star)\rangle \\
&=8N\mu(1+2N(2N-1)\mu^2)\nu^2(-\langle g'(y_N)-u_\star,y_N-x_\star\rangle+\mu\|y_N-x_\star\|^2)+ \\
&4N\mu\nu(-\langle f'(x_N)+u_\star,x_N-x_\star\rangle) \\
&\leq 0
\end{aligned}
$$

where the final inequality holds by monotonicity of ∂f, μ-strong monotonicity of ∂g, since $-u_\star \in \partial f(x_\star)$, and because $8N\mu(1+2N(2N-1)\mu^2)\nu^2 \geq 0$ and $4N\mu\nu \geq 0$.

□

5.3 Matching lower bound

In this section, we present a complexity lower bound that matches the $\mathcal{O}(1/N^2)$ rate and its leading constant of Theorem 2.

We follow a standard approach: we first construct a pair of worst-case functions for the class of methods satisfying the *proximal span condition*, and then apply a resisting oracle technique to extend the lower bound to apply to all deterministic algorithms.

5.3.1 Proximal span condition

Intuitively, for first-order methods, the span condition is defined by requiring that each iterate belongs to the span of the previous iterates and the associated first-order information. This notion has been formalized in various forms in prior works [138, 40, 71, 145, 72, 151, 97].

In our particular setting, we consider composite problems accessed through proximal oracles. Before giving the formal definition in Section 5.3.3, we use the term *deterministic N-step prox-prox method* informally to refer to any deterministic algorithm that makes a total of $2N$ proximal oracle calls, each call being to either f or g, and then outputs a point x_N. Since the method has access to two starting points, namely x_0 and u_0, the corresponding *proximal span condition* can be stated as below for fixed iteration count $N > 0$. Specifically, for fixed (x_0, u_0), we say a *trajectory* or a *sequence*

$$\left\{(z_i, \delta_i, \gamma_i)\right\}_{i=0}^{2N-1},\ x_N\Big\}$$

satisfies the *proximal span condition* if

$$\begin{aligned}
&\delta_i \in \{0,1\} && \text{for } i = 0,\dots,2N-1,\\
&z_0 \in x_0 + \operatorname{span}\{u_0\}\\
&d_i = \begin{cases} z_i - \operatorname{prox}_{\gamma_i f}(z_i) & \text{for some } \gamma_i > 0, \text{ if } \delta_i = 0,\\ z_i - \operatorname{prox}_{\gamma_i g}(z_i) & \text{for some } \gamma_i > 0, \text{ if } \delta_i = 1, \end{cases} && \text{for } i = 0,\dots,2N-1,\\
&z_i \in x_0 + \operatorname{span}\{u_0, d_0, \dots, d_{i-1}\} && \text{for } i = 1,\dots,2N-1,\\
&x_N \in x_0 + \operatorname{span}\{u_0, d_0, \dots, d_{2N-1}\}
\end{aligned}$$

and we say a *method* satisfies the *proximal span condition* if the trajectory $\left\{(z_i, \delta_i, \gamma_i)\right\}_{i=0}^{2N-1},\ x_N\Big\}$ produced by the first-order deterministic method satisfies the proximal span condition for any starting point (x_0, u_0).

To clarify, we make no assumption on either the order of the proximal oracle evaluations

or which function is queried at each step. We only assume that the total number of proximal oracle evaluations is $2N$.

Under the proximal span condition, we establish the following lower bound.

Theorem 3. *Fix a positive integer N and let $\mu > 0$. For any initial pair $(x_0, u_0) \in \mathbb{R}^{2N+2} \times \mathbb{R}^{2N+2}$, there exist a closed, proper, and convex function $f\colon \mathbb{R}^{2N+2} \to \mathbb{R} \cup \{\infty\}$, a closed, proper, and μ-strongly convex function $g\colon \mathbb{R}^{2N+2} \to \mathbb{R} \cup \{\infty\}$, $x_\star \in \operatorname{argmin}(f+g)$, and $u_\star \in \partial g(x_\star)$ with $-u_\star \in \partial f(x_\star)$ such that the following holds:*

For every deterministic N-step prox-prox method satisfying the proximal span condition, if x_N denotes its final output when run from (x_0, u_0) on the pair (f, g), then

$$\|x_N - x_\star\|^2 \geq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2 + 4N\mu}.$$

5.3.2 Proof of Theorem 3

We now construct a worst-case instance for first-order deterministic prox-prox methods satisfying the proximal span condition. Throughout this subsection, the starting pair (x_0, u_0) is given and fixed.

Fix $N \in \mathbb{N}$ and $\mu > 0$. Choose an orthonormal set of vectors

$$e_{-1}, e_0, e_1, \ldots, e_{2N}$$

of $\mathbb{R}^{2N+2}$ such that $u_0 \in \operatorname{span}\{e_{-1}\}$. Let $t \in \mathbb{R}^d$ such that

$$t := \sum_{i=0}^{2N} t_i e_i \in \mathbb{R}^d.$$

where $t_i > 0$ for $0 \leq i \leq 2N$. We define two closed convex sets by

$$C_t := \Big\{ y \in \mathbb{R}^{2N+2} : y_{-1} = 0,\ y_0 = t_0,\ (y_{2k-1}, y_{2k}) \in [(0,0), (t_{2k-1}, t_{2k})],\ k = 1, \ldots, N \Big\},$$

and

$$D_t := \Big\{ y \in \mathbb{R}^{2N+2s} : y_{-1} = 0,\ (y_{2k}, y_{2k+1}) \in [(0,0), (t_{2k}, t_{2k+1})],\ k = 0, \ldots, N-1 \Big\},$$

where y_i is shorthand for $\langle y, e_i \rangle$ for $-1 \leq i \leq 2N$ and $[(0,0),(a,b)]$ is the line segment with endpoints $(0,0)$ and (a,b). We then define closed, proper, and μ-strongly convex g

$$g(x) := \frac{\mu}{2}\|x - x_0\|^2 + \langle u_0, x - x_0 \rangle + \delta_{x_0 + C_t}(x),$$

and closed, proper, and convex (possibly non-smooth) f as

$$f(x) := -\langle u_0, x - x_0 \rangle + \delta_{x_0 + D_t}(x).$$

Then minimizing $f + g$ is equivalent to minimizing

$$\frac{\mu}{2}\|x - x_0\|^2$$

over $(x_0 + C_t) \cap (x_0 + D_t)$. Because

$$C_t \cap D_t = \{ y \in \mathbb{R}^{2N+2} : y_{-1} = 0,\ y_i = t_i \text{ for } 0 \leq i \leq 2N \},$$

we have

$$(x_0 + C_t) \cap (x_0 + D_t) = \{ x_0 + y \in \mathbb{R}^{2N+2} : y_{-1} = 0,\ y_i = t_i \text{ for } 0 \leq i \leq 2N \}.$$

Hence $x_\star := x_0 + t$ is the unique minimizer of $f + g$.

Moreover, the proximal mappings admit the explicit representations

$$\operatorname{prox}_{\gamma f}(x_0 + y) = x_0 + \Pi_{D_t}(y + \gamma u_0),$$

and

$$\operatorname{prox}_{\gamma g}(x_0 + y) = x_0 + \Pi_{C_t}\left(\frac{y - \gamma u_0}{1 + \gamma\mu}\right),$$

where Π_{D_t} and Π_{C_t} denote the Euclidean projections onto D_t and C_t, respectively.

For $k = -1, 0, \ldots, 2N$, define

$$S_k := \operatorname{span}\{e_{-1}, e_0, \ldots, e_k\}, \qquad S_{-1} := \operatorname{span}\{e_{-1}\}.$$

The sets C_t and D_t are arranged so that the two proximal operators reveal coordinates one at a time. We will refer to this as the *proximal zero-chain property*, and the next lemma formalizes this construction.

Lemma 4. *For every $\gamma > 0$ and every $k = -1, 0, \ldots, 2N - 1$,*

$$y \in S_k \quad \Longrightarrow \quad \operatorname{prox}_{\gamma f}(x_0 + y) = x_0 + \Pi_{D_t}(y + \gamma u_0) \in x_0 + S_{k+1},$$

and

$$y \in S_k \quad \Longrightarrow \quad \operatorname{prox}_{\gamma g}(x_0 + y) = x_0 + \Pi_{C_t}\left(\frac{y - \gamma u_0}{1 + \gamma\mu}\right) \in x_0 + S_{k+1}.$$

Proof. Both C_t and D_t are Cartesian products of one-dimensional and two-dimensional blocks, and hence their Euclidean projections act blockwise. Since $u_0 \in \operatorname{span}\{e_{-1}\}$ and both C_t and D_t impose the constraint $z_{-1} = 0$, the shift by $\pm\gamma u_0$ only affects the e_{-1}-component and does not reveal any new coordinate. Therefore, if $y \in S_k$, then after projecting onto either C_t or D_t, at most the next coordinate e_{k+1} can become nonzero. This proves the claim. □

This lemma shows that if $y \in S_k$, then

$$(x_0 + y) - \operatorname{prox}_{\gamma g}(x_0 + y) \in S_{k+1}$$

because each proximal evaluation reveals at most one new coordinate. Consequently, for any prox-prox method satisfying the proximal span condition, the final output x_N generated

after at most $2N$ proximal evaluations must belong to

$$x_0 + S_{2N-1}.$$

Since

$$x_\star - x_0 = t = \sum_{i=0}^{2N} t_i e_i$$

has a nonzero e_{2N}-component, we obtain

$$\inf_{y \in x_0 + S_{2N-1}} \|y - x_\star\|^2 = t_{2N}^2 > 0.$$

Thus, after at most $2N$ proximal oracle calls, no method satisfying the proximal span condition can guarantee an error smaller than t_{2N}^2 on this instance. The next lemma rewrites the optimality conditions at the endpoint $x_\star = x_0 + t$ in blockwise form.

Lemma 5. *Let*

$$x_\star = x_0 + t = x_0 + \sum_{i=0}^{2N} t_i e_i,$$

and let

$$g_\star = \sum_{i=0}^{2N} g_i e_i \in \mathbb{R}^d$$

where $g_i \in \mathbb{R}$ for $0 \le i \le 2N$. Assume that

$$g_\star \in \mu t + N_{C_t}(t) \qquad \text{and} \qquad -g_\star \in N_{D_t}(t).$$

Then, with

$$u_\star := u_0 + g_\star,$$

we have

$$u_\star \in \partial g(x_\star), \qquad -u_\star \in \partial f(x_\star).$$

Moreover, the above inclusions on $g_\star$ are equivalent to the blockwise inequalities

$$(g_{2k+1} - \mu t_{2k+1},\, g_{2k+2} - \mu t_{2k+2}) \cdot (t_{2k+1}, t_{2k+2}) \geq 0, \qquad k = 0, \ldots, N-1,$$

$$(g_{2k},\, g_{2k+1}) \cdot (t_{2k}, t_{2k+1}) \leq 0, \qquad k = 0, \ldots, N-1,$$

and $g_{2N} = 0$.

Proof. Recall that

$$g(x) = \frac{\mu}{2}\|x - x_0\|^2 + \langle u_0, x - x_0\rangle + \delta_{x_0+C_t}(x),$$

and

$$f(x) = -\langle u_0, x - x_0\rangle + \delta_{x_0+D_t}(x).$$

Since $x_\star = x_0 + t$, we have

$$\partial g(x_\star) = \mu(x_\star - x_0) + u_0 + N_{x_0+C_t}(x_\star) = \mu t + u_0 + N_{C_t}(t),$$

where we used the translation rule

$$N_{x_0+C_t}(x_0 + t) = N_{C_t}(t).$$

Hence, if

$$g_\star \in \mu t + N_{C_t}(t),$$

then

$$u_\star := u_0 + g_\star \in \partial g(x_\star).$$

Similarly,

$$\partial f(x_\star) = -u_0 + N_{x_0+D_t}(x_\star) = -u_0 + N_{D_t}(t),$$

and therefore

$$-g_\star \in N_{D_t}(t) \quad \Longrightarrow \quad -u_\star = -(u_0 + g_\star) \in \partial f(x_\star).$$

It remains to characterize the normal cone conditions explicitly. Since C_t is the Cartesian product of the singleton constraint $z_0 = t_0$, the constraint $z_{-1} = 0$, and the line segments

$$[(0,0),(t_{2k-1},t_{2k})], \qquad k = 1,\dots,N,$$

its normal cone at t is the product of the corresponding blockwise normal cones. The equality constraints fixing the coordinates -1 and 0 contribute the full normal spaces in those coordinates, and hence impose no restrictions on the corresponding components of a normal vector. Thus, only the two-dimensional segment blocks need to be considered. For a segment

$$[(0,0),(a,b)],$$

the normal cone at the endpoint (a,b) is $\{w \in \mathbb{R}^2 :\ w \cdot (a,b) \geq 0\}$. Applying this with

$$w = (g_{2k+1} - \mu t_{2k+1},\ g_{2k+2} - \mu t_{2k+2})$$

gives

$$(g_{2k+1} - \mu t_{2k+1},\ g_{2k+2} - \mu t_{2k+2}) \cdot (t_{2k+1}, t_{2k+2}) \geq 0, \qquad k = 0,\dots,N-1.$$

Likewise, D_t is the product of the segment blocks

$$[(0,0),(t_{2k},t_{2k+1})], \qquad k = 0,\dots,N-1,$$

together with the constraint $z_{-1} = 0$. Therefore,

$$-g_\star \in N_{D_t}(t)$$

is equivalent to

$$(g_{2k}, g_{2k+1}) \cdot (t_{2k}, t_{2k+1}) \leq 0, \qquad k = 0,\dots,N-1.$$

Finally, D_t leaves the coordinate $2N$ free and hence its normal component there must vanish.

So, $g_{2N} = 0$ and this proves the result. □

We now choose the coefficients of t and $g_\star$ so that every blockwise inequality is satisfied with equality under the normalization

$$\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2 = 1.$$

Lemma 6. *Define*

$$t^\star_{2k} = \sqrt{\frac{\mu}{(1+2N\mu)(1+2k\mu)(1+(2k+1)\mu)}}, \qquad k = 0, \dots, N-1,$$

$$t^\star_{2k+1} = \sqrt{\frac{\mu}{(1+2N\mu)(1+(2k+1)\mu)(1+(2k+2)\mu)}}, \qquad k = 0, \dots, N-1,$$

$$t^\star_{2N} = \frac{1}{1+2N\mu},$$

and

$$g^\star_{2k} = -(1+2k\mu)t^\star_{2k}, \qquad g^\star_{2k+1} = (1+(2k+2)\mu)t^\star_{2k+1}, \qquad k = 0, \dots, N-1,$$

with

$$g^\star_{2N} = 0.$$

Then the pair

$$t = \sum_{i=0}^{2N} t^\star_i e_i, \qquad g_\star = \sum_{i=0}^{2N} g^\star_i e_i$$

with

$$x_\star = x_0 + t, \quad u_\star = u_0 + g_\star$$

satisfies

$$\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2 = \sum_{i=0}^{2N} (t_i^\star)^2 + \sum_{i=0}^{2N} (g_i^\star)^2 = 1,$$
$$(g_{2k+1}^\star - \mu t_{2k+1}^\star,\, g_{2k+2}^\star - \mu t_{2k+2}^\star) \cdot (t_{2k+1}^\star, t_{2k+2}^\star) = 0, \quad k = 0, \ldots, N-1,$$
$$(g_{2k}^\star,\, g_{2k+1}^\star) \cdot (t_{2k}^\star, t_{2k+1}^\star) = 0, \quad k = 0, \ldots, N-1,$$

so that $u_\star \in \partial g(x_\star), -u_\star \in \partial f(x_\star)$, *and therefore* $x_\star \in \operatorname{argmin}(f+g)$.

Proof. We first verify the inequalities.

For each $k = 0, \ldots, N-2$, we have

$$g_{2k+1}^\star - \mu t_{2k+1}^\star = (1 + (2k+1)\mu) t_{2k+1}^\star,$$

and

$$g_{2k+2}^\star - \mu t_{2k+2}^\star = -(1 + (2k+3)\mu) t_{2k+2}^\star.$$

Therefore,

$$\begin{aligned}(g_{2k+1}^\star - \mu t_{2k+1}^\star,\, g_{2k+2}^\star - \mu t_{2k+2}^\star) \cdot (t_{2k+1}^\star, t_{2k+2}^\star) \\ = (1 + (2k+1)\mu)(t_{2k+1}^\star)^2 - (1 + (2k+3)\mu)(t_{2k+2}^\star)^2.\end{aligned}$$

By the definition of $t_{2k+1}^\star$ and $t_{2k+2}^\star$,

$$(1 + (2k+1)\mu)(t_{2k+1}^\star)^2 = \frac{\mu}{(1 + 2N\mu)(1 + (2k+2)\mu)},$$

and

$$(1 + (2k+3)\mu)(t_{2k+2}^\star)^2 = \frac{\mu}{(1 + 2N\mu)(1 + (2k+2)\mu)}.$$

Hence

$$(g_{2k+1}^\star - \mu t_{2k+1}^\star,\, g_{2k+2}^\star - \mu t_{2k+2}^\star) \cdot (t_{2k+1}^\star, t_{2k+2}^\star) = 0.$$

For $k = N - 1$,

$$g^\star_{2N-1} - \mu t^\star_{2N-1} = (1 + (2N-1)\mu)t^\star_{2N-1}, \qquad g^\star_{2N} - \mu t^\star_{2N} = -\mu t^\star_{2N}.$$

Therefore,

$$\begin{aligned}(g^\star_{2N-1} - \mu t^\star_{2N-1},\, g^\star_{2N} - \mu t^\star_{2N}) \cdot (t^\star_{2N-1}, t^\star_{2N}) \\ = (1 + (2N-1)\mu)(t^\star_{2N-1})^2 - \mu(t^\star_{2N})^2.\end{aligned}$$

Now

$$(t^\star_{2N-1})^2 = \frac{\mu}{(1+2N\mu)(1+(2N-1)\mu)(1+2N\mu)} = \frac{\mu}{(1+(2N-1)\mu)(1+2N\mu)^2},$$

so

$$(1 + (2N-1)\mu)(t^\star_{2N-1})^2 = \frac{\mu}{(1+2N\mu)^2}.$$

Also,

$$(t^\star_{2N})^2 = \frac{1}{(1+2N\mu)^2}.$$

Hence

$$(1 + (2N-1)\mu)(t^\star_{2N-1})^2 = \mu(t^\star_{2N})^2,$$

and therefore

$$(g^\star_{2N-1} - \mu t^\star_{2N-1},\, g^\star_{2N} - \mu t^\star_{2N}) \cdot (t^\star_{2N-1}, t^\star_{2N}) = 0.$$

Thus,

$$(g^\star_{2k+1} - \mu t^\star_{2k+1},\, g^\star_{2k+2} - \mu t^\star_{2k+2}) \cdot (t^\star_{2k+1}, t^\star_{2k+2}) = 0, \qquad k = 0, \ldots, N-1.$$

Similarly, for each $k = 0, \ldots, N-1$,

$$(g^\star_{2k}, g^\star_{2k+1}) \cdot (t^\star_{2k}, t^\star_{2k+1}) = -(1+2k\mu)(t^\star_{2k})^2 + (1+(2k+2)\mu)(t^\star_{2k+1})^2.$$

Again, by the definitions,

$$(1+2k\mu)(t^\star_{2k})^2 = \frac{\mu}{(1+2N\mu)(1+(2k+1)\mu)},$$

and

$$(1+(2k+2)\mu)(t^\star_{2k+1})^2 = \frac{\mu}{(1+2N\mu)(1+(2k+1)\mu)}.$$

Therefore,

$$(g^\star_{2k}, g^\star_{2k+1}) \cdot (t^\star_{2k}, t^\star_{2k+1}) = 0, \qquad k = 0, \ldots, N-1.$$

Thus, all block inequalities are satisfied with equality. It remains to verify the normalization constraint. For each $j = 0, \ldots, 2N-1$,

$$(t^\star_j)^2 = \frac{\mu}{(1+2N\mu)(1+j\mu)(1+(j+1)\mu)} = \frac{1}{1+2N\mu}\left(\frac{1}{1+j\mu} - \frac{1}{1+(j+1)\mu}\right).$$

Hence

$$\sum_{j=0}^{2N-1}(t^\star_j)^2 = \frac{1}{1+2N\mu}\sum_{j=0}^{2N-1}\left(\frac{1}{1+j\mu} - \frac{1}{1+(j+1)\mu}\right) = \frac{1}{1+2N\mu}\left(1 - \frac{1}{1+2N\mu}\right).$$

Adding $(t^\star_{2N})^2 = \frac{1}{(1+2N\mu)^2}$ yields

$$\sum_{j=0}^{2N}(t^\star_j)^2 = \frac{1}{1+2N\mu}.$$

Next, for each $k = 0, \ldots, N-1$,

$$(g^\star_{2k})^2 = (1+2k\mu)^2(t^\star_{2k})^2 = \frac{\mu(1+2k\mu)}{(1+2N\mu)(1+(2k+1)\mu)},$$

and

$$(g^\star_{2k+1})^2 = (1+(2k+2)\mu)^2 (t^\star_{2k+1})^2 = \frac{\mu(1+(2k+2)\mu)}{(1+2N\mu)(1+(2k+1)\mu)}.$$

Therefore,

$$\begin{aligned}(g^\star_{2k})^2 + (g^\star_{2k+1})^2 &= \frac{\mu}{1+2N\mu}\left(\frac{1+2k\mu}{1+(2k+1)\mu} + \frac{1+(2k+2)\mu}{1+(2k+1)\mu}\right) \\ &= \frac{\mu}{1+2N\mu}\cdot\frac{2+(4k+2)\mu}{1+(2k+1)\mu} = \frac{2\mu}{1+2N\mu}.\end{aligned}$$

Summing over $k = 0, \ldots, N-1$ and using $g^\star_{2N} = 0$, we obtain

$$\sum_{j=0}^{2N}(g^\star_j)^2 = \sum_{k=0}^{N-1}\left((g^\star_{2k})^2 + (g^\star_{2k+1})^2\right) = \frac{2N\mu}{1+2N\mu}.$$

Combining the two identities gives

$$\sum_{j=0}^{2N}(t^\star_j)^2 + \sum_{j=0}^{2N}(g^\star_j)^2 = \frac{1}{1+2N\mu} + \frac{2N\mu}{1+2N\mu} = 1.$$

□

Together with the fact that

$$\|x_N - x_\star\|^2 \geq \inf_{y \in x_0 + S_{2N-1}} \|y - x_\star\|^2 = (t^\star_{2N})^2$$

and

$$(t^\star_{2N})^2 = \frac{1}{(1+2N\mu)^2} = \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1+4N^2\mu^2+4N\mu},$$

we obtain

$$\|x_N - x_\star\|^2 \geq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1+4N^2\mu^2+4N\mu}.$$

This proves Theorem 3.

5.3.3 Extension using the resisting oracle technique

Next, using the so-called *resisting oracle technique*, we extend the result of Theorem 3 to all deterministic N-step prox-prox methods. First, we make a formal definition.

Let $N > 0$ be a positive integer. A *deterministic N-step prox-prox method* $\mathbf{A}$ is a mapping from an initial pair (x_0, u_0) and a tuple of functions (f, g) to a sequence of outputs

$$\mathbf{A}\big((x_0, u_0), (f, g)\big) = \big\{\{(z_i, \delta_i, \gamma_i)\}_{i=0}^{2N-1}, x_N\big\},$$

where $\delta_i \in \{0, 1\}$ and $\gamma_i > 0$ for $i = 0, \dots, 2N-1$. For clarity, we call $\{z_i\}_{0 \le i \le 2N-1}$ the *query points* and x_N the *final output*. Moreover, $\mathbf{A}$ requires that the output sequence depends on (f, g) only through (x_0, u_0), previous query points, and previous proximal oracle calls. More precisely, letting

$$y_i := \begin{cases} \operatorname{prox}_{\gamma_i f}(z_i), & \delta_i = 0, \\ \operatorname{prox}_{\gamma_i g}(z_i), & \delta_i = 1, \end{cases} \qquad i = 0, \dots, 2N-1,$$

there exist deterministic maps $\mathbf{A}_0, \dots, \mathbf{A}_{2N}$ such that

$$(z_0, \delta_0, \gamma_0) = \mathbf{A}_0(x_0, u_0),$$

$$(z_i, \delta_i, \gamma_i) = \mathbf{A}_i\Big((x_0, u_0), \{(z_j, \delta_j, \gamma_j, y_j)\}_{j=0}^{i-1}\Big), \qquad i = 1, \dots, 2N-1,$$

and

$$x_N = \mathbf{A}_{2N}\Big((x_0, u_0), \{(z_j, \delta_j, \gamma_j, y_j)\}_{j=0}^{2N-1}\Big).$$

With this definition, we are now ready to formally state the more general lower bound.

Theorem 7. *Fix a positive integer N and let $\mu > 0$. Let $d \ge 4N+4$. For every deterministic N-step prox-prox method $\mathbf{A}$ on $\mathbb{R}^d$ and every initial pair $(x_0, u_0) \in \mathbb{R}^d \times \mathbb{R}^d$, there exist a closed, proper, and convex function $f\colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ and a closed, proper, and μ-strongly*

convex function $g\colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$*, together with* $x_\star \in \operatorname{argmin}(f+g)$ *and* $u_\star \in \partial g(x_\star)$ *with* $-u_\star \in \partial f(x_\star)$*, such that, if* x_N *denotes the final output of* $\mathbf{A}$ *when run from* (x_0, u_0) *on* (f, g)*, then*

$$\|x_N - x_\star\|^2 \geq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2 + 4N\mu}.$$

Theorem 7 extends Theorem 3 from methods satisfying the proximal span condition to all deterministic N-step prox-prox methods, using the resisting-oracle technique. To do this, we introduce a *prox-prox zero-respecting condition*. It is closely related to the proximal span condition and may in fact be somewhat redundant here, but we include it for clarity of exposition.

Let $f, g : \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$, and let

$$\left\{\left\{(\tilde{z}_i, \delta_i, \gamma_i)\right\}_{i=0}^{2N-1},\ \tilde{x}_N\right\}$$

be a sequence with $\delta_i \in \{0, 1\}$ and $\gamma_i > 0$ for $i = 0, \ldots, 2N-1$. Define, for each $i = 0, \ldots, 2N-1$,

$$\tilde{d}_i := \begin{cases} \tilde{z}_i - \operatorname{prox}_{\gamma_i f}(\tilde{z}_i), & \delta_i = 0, \\ \tilde{z}_i - \operatorname{prox}_{\gamma_i g}(\tilde{z}_i), & \delta_i = 1. \end{cases}$$

We say that the trajectory

$$\left\{\left\{(\tilde{z}_i, \delta_i, \gamma_i)\right\}_{i=0}^{2N-1},\ \tilde{x}_N\right\}$$

is *prox-prox zero-respecting with respect to* (f, g) if

$$\operatorname{supp}(\tilde{z}_i) \subseteq \bigcup_{k=0}^{i-1} \operatorname{supp}(\tilde{d}_k), \qquad i = 0, \ldots, 2N-1$$

and

$$\operatorname{supp}(\tilde{x}_N) \subseteq \bigcup_{k=0}^{2N-1} \operatorname{supp}(\tilde{d}_k).$$

where the support is taken with respect to the fixed orthonormal basis with the convention

$\mathrm{supp}() = \varnothing$.

The next lemma formalizes that, for functions (f, g) constructed in Theorem 3, the zero-respecting condition is enough to recover the proximal span condition with $(x_0, u_0) = (0, 0)$. The key idea is that proximal residuals can reveal new coordinates only consecutively, one at a time. All supports below are taken with respect to the orthonormal coordinates $e_{-1}, e_0, \ldots, e_{2N}$ used in the construction of (f, g) in Section 5.3.2.

Lemma 8. *Let $f, g : \mathbb{R}^{2N+2} \to \mathbb{R} \cup \{\infty\}$ be the functions defined in Theorem 3. Let*

$$\left\{(\tilde{z}_i, \delta_i, \gamma_i)\right\}_{i=0}^{2N-1},\ \tilde{x}_N\Big\}$$

be prox-prox zero-respecting with respect to (f, g), and define

$$\tilde{d}_i := \begin{cases} \tilde{z}_i - \mathrm{prox}_{\gamma_i f}(\tilde{z}_i), & \delta_i = 0, \\ \tilde{z}_i - \mathrm{prox}_{\gamma_i g}(\tilde{z}_i), & \delta_i = 1, \end{cases} \qquad i = 0, \ldots, 2N-1.$$

Then $\left\{(\tilde{z}_i, \delta_i, \gamma_i)\right\}_{i=0}^{2N-1},\ \tilde{x}_N\Big\}$ satisfies the proximal span condition with initial pair $(x_0, u_0) = (0, 0)$, namely,

$$\begin{aligned} &\tilde{z}_i \in \mathrm{span}\{\tilde{d}_0, \ldots, \tilde{d}_{i-1}\}, \qquad i = 0, \ldots, 2N-1, \\ &\tilde{x}_N \in \mathrm{span}\{\tilde{d}_0, \ldots, \tilde{d}_{2N-1}\}. \end{aligned}$$

Consequently, the proof of Theorem 3 applies to every prox-prox zero-respecting trajectory with $(x_0, u_0) = (0, 0)$.

Proof. For notational simplicity, define

$$E_{-1} := \{0\}, \qquad E_m := \mathrm{span}\{e_0, \ldots, e_m\}, \quad m = 0, \ldots, 2N-1.$$

We prove by induction on $k = 0, \ldots, 2N-1$ that there exists $m_k \in \{-1, 0, \ldots, k\}$ such that

$$\operatorname{span}\{\tilde{d}_0, \ldots, \tilde{d}_k\} = E_{m_k}.$$

For $k = 0$, the zero-respecting property gives

$$\operatorname{supp}(\tilde{z}_0) \subseteq \varnothing,$$

so $\tilde{z}_0 = 0$. Hence, by the prox-zero-chain property,

$$\tilde{d}_0 \in \operatorname{span}\{e_0\}.$$

Therefore $\operatorname{span}\{\tilde{d}_0\}$ is either $\{0\}$ or $\operatorname{span}\{e_0\}$, so the claim holds with $m_0 = -1$ or $m_0 = 0$. Now suppose the claim holds for $k-1$, namely,

$$\operatorname{span}\{\tilde{d}_0, \ldots, \tilde{d}_{k-1}\} = E_{m_{k-1}}.$$

Then

$$\bigcup_{i=0}^{k-1} \operatorname{supp}(\tilde{d}_i) \subseteq \{0, \ldots, m_{k-1}\}.$$

Here, we use the convention $\{0, \ldots, -1\} = \varnothing$ when $m_{k-1} = -1$. Since the trajectory is prox-prox zero-respecting,

$$\operatorname{supp}(\tilde{z}_k) \subseteq \bigcup_{i=0}^{k-1} \operatorname{supp}(\tilde{d}_i),$$

and thus

$$\tilde{z}_k \in \operatorname{span}\{e_0, \ldots, e_{m_{k-1}}\} = E_{m_{k-1}}.$$

Applying the prox-zero-chain property, we obtain

$$\tilde{d}_k \in \operatorname{span}\{e_0, \ldots, e_{m_{k-1}}, e_{m_{k-1}+1}\}.$$

Therefore

$$\operatorname{span}\{\tilde{d}_0, \ldots, \tilde{d}_k\}$$

is either

$$\operatorname{span}\{e_0, \ldots, e_{m_{k-1}}\} = E_{m_{k-1}} \quad \text{or} \quad \operatorname{span}\{e_0, \ldots, e_{m_{k-1}+1}\} = E_{m_{k-1}+1}.$$

Thus by induction, for each $k = 0, \ldots, 2N-1$,

$$\bigcup_{i=0}^{k-1} \operatorname{supp}(\tilde{d}_i) \subseteq \{0, \ldots, m_{k-1}\},$$

so the zero-respecting property implies

$$\tilde{z}_k \in \operatorname{span}\{e_0, \ldots, e_{m_{k-1}}\} = \operatorname{span}\{\tilde{d}_0, \ldots, \tilde{d}_{k-1}\}.$$

Likewise,

$$\operatorname{supp}(\tilde{x}_N) \subseteq \bigcup_{i=0}^{2N-1} \operatorname{supp}(\tilde{d}_i) \subseteq \{0, \ldots, m_{2N-1}\},$$

and therefore

$$\tilde{x}_N \in \operatorname{span}\{e_0, \ldots, e_{m_{2N-1}}\} = \operatorname{span}\{\tilde{d}_0, \ldots, \tilde{d}_{2N-1}\}.$$

This proves that the trajectory satisfies the proximal span condition with initial pair $(x_0, u_0) = (0, 0)$. □

We now turn to the lifting construction used in the resisting-oracle argument.

Lemma 9. *Let $U \in \mathbb{R}^{d' \times d}$ have orthonormal columns with $d' \geq d$. For any vectors $x_0, u_0 \in \mathbb{R}^{d'}$, if $f \colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$ is a closed, proper, and convex function, then $f_U : \mathbb{R}^{d'} \to \mathbb{R} \cup \{\infty\}$ defined by*

$$f_U(x) := f(U^\top (x - x_0)) - \langle u_0, x - x_0 \rangle$$

is a convex, closed, and proper function.

Proof. Since the mapping $x \mapsto U^\top(x - x_0)$ is affine and f is convex, closed, and proper, the composition

$$x \mapsto f\big(U^\top(x - x_0)\big)$$

is convex, closed, and proper. Moreover, the function

$$x \mapsto -\langle u_0, x - x_0\rangle$$

is affine and finite-valued on $\mathbb{R}^{d'}$. Therefore,

$$f_U(x) = f\big(U^\top(x - x_0)\big) - \langle u_0, x - x_0\rangle$$

is the sum of a proper, closed, convex function and an affine function. Hence f_U is proper, closed, and convex. □

Lemma 10. *Let $U \in \mathbb{R}^{d'\times d}$ have orthonormal columns with $d' \geq d$. For any vectors $x_0, u_0 \in \mathbb{R}^{d'}$, if $g\colon \mathbb{R}^d \to \mathbb{R}\cup\{\infty\}$ is closed, proper, and μ-strongly convex, then the function $g_U : \mathbb{R}^{d'} \to \mathbb{R}\cup\{\infty\}$ defined by*

$$g_U(x) := g\big(U^\top(x - x_0)\big) + \langle u_0, x - x_0\rangle + \frac{\mu}{2}\|P(x - x_0)\|^2, \qquad P := I - UU^\top,$$

is μ-strongly convex.

Proof. Fix $x, y \in \mathbb{R}^{d'}$ and let $v \in \partial g_U(x)$ be arbitrary and write it uniquely as

$$v = Us + u_0 + \mu P(x - x_0)$$

where $s \in \partial g\big(U^\top(x - x_0)\big)$. Since g is μ-strongly convex, we have

$$g\big(U^\top(y - x_0)\big) \geq g\big(U^\top(x - x_0)\big) + \big\langle s,\, U^\top(y - x)\big\rangle + \frac{\mu}{2}\|U^\top(y - x)\|^2.$$

Also,

$$\langle u_0, y - x_0\rangle = \langle u_0, x - x_0\rangle + \langle u_0, y - x\rangle,$$

and

$$\frac{\mu}{2}\|P(y - x_0)\|^2 = \frac{\mu}{2}\|P(x - x_0)\|^2 + \mu\langle P(x - x_0), y - x\rangle + \frac{\mu}{2}\|P(y - x)\|^2.$$

Adding these identities and inequalities yields

$$\begin{aligned} g_U(y) \geq g_U(x) &+ \langle Us + u_0 + \mu P(x - x_0),\, y - x\rangle \\ &+ \frac{\mu}{2}\|U^\top(y - x)\|^2 + \frac{\mu}{2}\|P(y - x)\|^2. \end{aligned}$$

Since

$$\|U^\top(y - x)\|^2 + \|P(y - x)\|^2 = \|y - x\|^2,$$

we obtain

$$\begin{aligned} g_U(y) &\geq g_U(x) + \langle Us + u_0 + \mu P(x - x_0),\, y - x\rangle + \frac{\mu}{2}\|y - x\|^2 \\ &= g_U(x) + \langle v,\, y - x\rangle + \frac{\mu}{2}\|y - x\|^2. \end{aligned}$$

Therefore g_U is μ-strongly convex. □

From now on, $U \in \mathbb{R}^{d' \times d}$ has orthonormal columns, x_0, u_0 are fixed, and f_U and g_U are defined as in the previous two lemmas. Next, we derive explicit formulas for the proximal mappings of the lifted functions.

Lemma 11. *For every $z \in \mathbb{R}^{d'}$ and every $\gamma > 0$, assume that $U^\top u_0 = 0$. Write*

$$\tilde{z} := U^\top(z - x_0), \qquad p_z := P(z - x_0),$$

where $P := I - UU^\top$. Then

$$\operatorname{prox}_{\gamma f_U}(z) = x_0 + U \operatorname{prox}_{\gamma f}(\tilde{z}) + p_z + \gamma u_0,$$

and therefore

$$z - \operatorname{prox}_{\gamma f_U}(z) = U\big(\tilde{z} - \operatorname{prox}_{\gamma f}(\tilde{z})\big) - \gamma u_0.$$

In particular,

$$U^\top\big(z - \operatorname{prox}_{\gamma f_U}(z)\big) = \tilde{z} - \operatorname{prox}_{\gamma f}(\tilde{z}).$$

Proof. Since $U^\top U = I_d$ and $P = I - UU^\top$, every $x \in \mathbb{R}^{d'}$ admits a unique decomposition

$$x = x_0 + U\tilde{x} + p, \qquad \tilde{x} \in \mathbb{R}^d, \quad p \in \operatorname{range}(P),$$

and we write

$$z = x_0 + U\tilde{z} + p_z, \qquad \tilde{z} := U^\top(z - x_0), \quad p_z := P(z - x_0).$$

Since $U^\top u_0 = 0$, we have $u_0 \in \operatorname{range}(P)$, and hence

$$\langle u_0, x - x_0\rangle = \langle u_0, U\tilde{x} + p\rangle = \langle u_0, p\rangle.$$

Using $\operatorname{range}(U) \perp \operatorname{range}(P)$, we obtain

$$\|x - z\|^2 = \|\tilde{x} - \tilde{z}\|^2 + \|p - p_z\|^2,$$

and therefore

$$f_U(x) + \frac{1}{2\gamma}\|x - z\|^2 = f(\tilde{x}) - \langle u_0, p\rangle + \frac{1}{2\gamma}\|\tilde{x} - \tilde{z}\|^2 + \frac{1}{2\gamma}\|p - p_z\|^2.$$

Hence the minimization separates. Minimizing with respect to $\tilde{x}$, we get

$$\tilde{x} = \operatorname{prox}_{\gamma f}(\tilde{z}),$$

while the minimizing p satisfies

$$\frac{1}{\gamma}(p - p_z) - u_0 = 0,$$

so

$$p = p_z + \gamma u_0.$$

This proves that

$$\operatorname{prox}_{\gamma f_U}(z) = x_0 + U \operatorname{prox}_{\gamma f}(\tilde{z}) + p_z + \gamma u_0.$$

Subtracting from $z = x_0 + U\tilde{z} + p_z$ gives

$$z - \operatorname{prox}_{\gamma f_U}(z) = U\big(\tilde{z} - \operatorname{prox}_{\gamma f}(\tilde{z})\big) - \gamma u_0.$$

Finally, since $U^\top u_0 = 0$, applying $U^\top$ yields

$$U^\top\big(z - \operatorname{prox}_{\gamma f_U}(z)\big) = \tilde{z} - \operatorname{prox}_{\gamma f}(\tilde{z}).$$

□

Lemma 12. *For every $z \in \mathbb{R}^{d'}$ and every $\gamma > 0$, assume that $U^\top u_0 = 0$. Write*

$$\tilde{z} := U^\top(z - x_0), \qquad p_z := P(z - x_0),$$

where $P := I - UU^\top$. Then

$$\operatorname{prox}_{\gamma g_U}(z) = x_0 + U \operatorname{prox}_{\gamma g}(\tilde{z}) + \frac{1}{1+\gamma\mu}(p_z - \gamma u_0),$$

and

$$z - \operatorname{prox}_{\gamma g_U}(z) = U\big(\tilde{z} - \operatorname{prox}_{\gamma g}(\tilde{z})\big) + \frac{\gamma\mu}{1+\gamma\mu}\, p_z + \frac{\gamma}{1+\gamma\mu}\, u_0.$$

In particular,

$$U^\top\big(z - \operatorname{prox}_{\gamma g_U}(z)\big) = \tilde{z} - \operatorname{prox}_{\gamma g}(\tilde{z}).$$

Proof. Again write

$$x = x_0 + U\tilde{x} + p, \qquad z = x_0 + U\tilde{z} + p_z, \qquad p, p_z \in \mathrm{range}(P).$$

Since $U^\top u_0 = 0$, we have $u_0 \in \mathrm{range}(P)$, and therefore

$$\langle u_0, x - x_0 \rangle = \langle u_0, U\tilde{x} + p \rangle = \langle u_0, p \rangle.$$

Using $\mathrm{range}(U) \perp \mathrm{range}(P)$, we obtain

$$\|x - z\|^2 = \|\tilde{x} - \tilde{z}\|^2 + \|p - p_z\|^2.$$

Hence

$$g_U(x) + \frac{1}{2\gamma}\|x - z\|^2 = g(\tilde{x}) + \langle u_0, p \rangle + \frac{\mu}{2}\|p\|^2 + \frac{1}{2\gamma}\|\tilde{x} - \tilde{z}\|^2 + \frac{1}{2\gamma}\|p - p_z\|^2.$$

The minimization separates. Minimizing with respect to $\tilde{x}$, we get

$$\tilde{x} = \mathrm{prox}_{\gamma g}(\tilde{z}),$$

while the minimizing p satisfies

$$u_0 + \mu p + \frac{1}{\gamma}(p - p_z) = 0.$$

Equivalently,

$$(1 + \gamma\mu)p = p_z - \gamma u_0,$$

so

$$p = \frac{1}{1 + \gamma\mu}(p_z - \gamma u_0).$$

This proves that

$$\operatorname{prox}_{\gamma g_U}(z) = x_0 + U \operatorname{prox}_{\gamma g}(\tilde{z}) + \frac{1}{1+\gamma\mu}(p_z - \gamma u_0).$$

Subtracting from

$$z = x_0 + U\tilde{z} + p_z$$

gives

$$\begin{aligned} z - \operatorname{prox}_{\gamma g_U}(z) &= U\big(\tilde{z} - \operatorname{prox}_{\gamma g}(\tilde{z})\big) + p_z - \frac{1}{1+\gamma\mu}(p_z - \gamma u_0) \\ &= U\big(\tilde{z} - \operatorname{prox}_{\gamma g}(\tilde{z})\big) + \frac{\gamma\mu}{1+\gamma\mu}\, p_z + \frac{\gamma}{1+\gamma\mu}\, u_0. \end{aligned}$$

Finally, since $U^\top u_0 = 0$ and $U^\top p_z = 0$, applying $U^\top$ yields

$$U^\top\big(z - \operatorname{prox}_{\gamma g_U}(z)\big) = \tilde{z} - \operatorname{prox}_{\gamma g}(\tilde{z}).$$

□

These identities allow us to match proximal evaluations in the lifted space with projected proximal evaluations in the original space. We are now ready to carry out the resisting-oracle construction.

Lemma 13. *Let $N > 0$ be a positive integer. Assume $\mu > 0$, $d' \geq d + 2N + 2$, and let $\boldsymbol{A}$ be any deterministic N-step prox-prox method with initial pair $(x_0, u_0) \in \mathbb{R}^{d'} \times \mathbb{R}^{d'}$. Let*

$$f\colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\} \quad \textit{and} \quad g\colon \mathbb{R}^d \to \mathbb{R} \cup \{\infty\}$$

be such that f is closed, proper, and convex and g is closed, proper, and μ-strongly convex. Then there exists a matrix

$$U \in \mathbb{R}^{d' \times d}$$

with orthonormal columns such that $U^\top u_0 = 0$*, and if*

$$\boldsymbol{A}\big((x_0, u_0), (f_U, g_U)\big) = \big\{\big\{(z_i, \delta_i, \gamma_i)\big\}_{i=0}^{2N-1}, x_N\big\},$$

and we define the projected iterates

$$\tilde{z}_i := U^\top(z_i - x_0), \qquad \tilde{x}_N := U^\top(x_N - x_0),$$

then the sequence

$$\big\{\big\{(\tilde{z}_i, \delta_i, \gamma_i)\big\}_{i=0}^{2N-1}, \tilde{x}_N\big\}$$

is prox-prox zero-respecting with respect to (f, g)*.*

Proof. Let $e_0, \ldots, e_{d-1}$ be the standard basis of $\mathbb{R}^d$. First, we recursively construct

$$I_0 \subseteq I_1 \subseteq \cdots \subseteq I_{2N} \subseteq \{0, \ldots, d-1\},$$

an orthonormal family $\{v_j\}_{j \in I_{2N}} \subseteq \mathbb{R}^{d'}$, and a method trajectory

$$(z_0, \delta_0, \gamma_0, y_0), \ldots, (z_{2N-1}, \delta_{2N-1}, \gamma_{2N-1}, y_{2N-1}),$$

where $y_i \in \mathbb{R}^{d'}$ is intended to be the output of the i-th prox call.

We begin with

$$I_0 := \varnothing.$$

Next, fix $i \in \{0, \ldots, 2N-1\}$, and suppose that I_i, the vectors $\{v_j\}_{j \in I_i}$, and the previous trajectory

$$(z_0, \delta_0, \gamma_0, y_0), \ldots, (z_{i-1}, \delta_{i-1}, \gamma_{i-1}, y_{i-1})$$

have already been constructed. Since $\mathbf{A}$ is deterministic, the trajectory determines the next

query triple $(z_i, \delta_i, \gamma_i)$. Define

$$\bar{z}_i := \sum_{j \in I_i} \langle v_j, z_i - x_0 \rangle e_j \in \mathbb{R}^d,$$

so that $\operatorname{supp}(\bar{z}_i) \subseteq I_i$. Next define

$$\bar{y}_i := \begin{cases} \operatorname{prox}_{\gamma_i f}(\bar{z}_i), & \delta_i = 0, \\ \operatorname{prox}_{\gamma_i g}(\bar{z}_i), & \delta_i = 1, \end{cases}$$

and the corresponding residual

$$\bar{d}_i := \bar{z}_i - \bar{y}_i.$$

Set

$$I_{i+1} := I_i \cup \operatorname{supp}(\bar{d}_i).$$

Since $\bar{y}_i = \bar{z}_i - \bar{d}_i$, we also have

$$\operatorname{supp}(\bar{y}_i) \subseteq I_{i+1}.$$

We now choose the new vectors

$$\{v_j\}_{j \in I_{i+1} \setminus I_i}$$

as an orthonormal family in $S_i^{\perp}$, where

$$S_i := \operatorname{span}\Big(\{u_0, z_0 - x_0, \ldots, z_i - x_0\} \cup \{v_j\}_{j \in I_i} \Big).$$

This is possible because

$$\dim S_i \leq (i+2) + |I_i|,$$

hence

$$\dim S_i^{\perp} \geq d' - (i+2) - |I_i| \geq d - |I_i| \geq |I_{i+1} \setminus I_i|.$$

Now define

$$q_i := (z_i - x_0) - \sum_{j \in I_i} \langle v_j, z_i - x_0 \rangle v_j,$$

and $y_i \in \mathbb{R}^{d'}$ by

$$y_i := \begin{cases} x_0 + \sum_{j \in I_{i+1}} (\bar{y}_i)_j \, v_j + q_i + \gamma_i u_0, & \delta_i = 0, \\ x_0 + \sum_{j \in I_{i+1}} (\bar{y}_i)_j \, v_j + \dfrac{1}{1 + \gamma_i \mu}(q_i - \gamma_i u_0), & \delta_i = 1. \end{cases}$$

This completes the recursive construction for $i = 0, \ldots, 2N-1$. Since $\mathbf{A}$ is deterministic, this transcript uniquely determines the final output $x_N \in \mathbb{R}^{d'}$. Then choose the orthonormal vectors indexed by $j \notin I_{2N}$ in the orthonormal complement of

$$S_{2N+1} := \operatorname{span}\Big(\{u_0, z_0 - x_0, \ldots, z_{2N-1} - x_0, \; x_N - x_0\} \cup \{v_j\}_{j \in I_{2N}} \Big).$$

This is possible because

$$\dim S_{2N+1} \leq (2N+2) + |I_{2N}|,$$

so

$$\dim S_{2N+1}^{\perp} \geq d' - (2N+2) - |I_{2N}| \geq d - |I_{2N}|.$$

Now set

$$U := [v_0 | \cdots | v_{d-1}] \in \mathbb{R}^{d' \times d}, \qquad P := I - UU^{\top}.$$

By construction, each column of U is orthogonal to u_0, and therefore $U^{\top} u_0 = 0$.

For the next step, we claim that if $\mathbf{A}$ is run on (f_U, g_U), then after i oracle calls its trajectory is exactly

$$(z_0, \delta_0, \gamma_0, y_0), \ldots, (z_{i-1}, \delta_{i-1}, \gamma_{i-1}, y_{i-1}), \qquad i = 0, \ldots, 2N,$$

and consequently, its final output is x_N. We prove this by induction on i. For $i = 0$, we only

have the initial state (x_0, u_0) and empty transcript, so the claim is immediate. Now assume the claim holds for some $i \in \{0, \dots, 2N-1\}$. Then $\mathbf{A}$ run on (f_U, g_U) has seen exactly the same trajectory

$$(z_0, \delta_0, \gamma_0, y_0), \dots, (z_{i-1}, \delta_{i-1}, \gamma_{i-1}, y_{i-1})$$

as in the recursive construction. Hence, its next query is exactly $(z_i, \delta_i, \gamma_i)$.

Now for every $j \notin I_i$, the vector v_j is orthogonal to $z_i - x_0$, so $\langle v_j, z_i - x_0 \rangle = 0$. Therefore,

$$U^\top (z_i - x_0) = \bar{z}_i, \qquad P(z_i - x_0) = q_i.$$

Now we apply Lemma 11 and Lemma 12. If $\delta_i = 0$, then

$$\begin{aligned} \operatorname{prox}_{\gamma_i f_U}(z_i) &= x_0 + U \operatorname{prox}_{\gamma_i f}\big(U^\top (z_i - x_0)\big) + P(z_i - x_0) + \gamma_i u_0 \\ &= x_0 + U \operatorname{prox}_{\gamma_i f}(\bar{z}_i) + q_i + \gamma_i u_0. \end{aligned}$$

Since $\bar{y}_i = \operatorname{prox}_{\gamma_i f}(\bar{z}_i)$ and $\operatorname{supp}(\bar{y}_i) \subseteq I_{i+1}$, this gives

$$\operatorname{prox}_{\gamma_i f_U}(z_i) = x_0 + \sum_{j \in I_{i+1}} (\bar{y}_i)_j \, v_j + q_i + \gamma_i u_0 = y_i.$$

If $\delta_i = 1$, then

$$\begin{aligned} \operatorname{prox}_{\gamma_i g_U}(z_i) &= x_0 + U \operatorname{prox}_{\gamma_i g}\big(U^\top (z_i - x_0)\big) + \frac{1}{1 + \gamma_i \mu}\big(P(z_i - x_0) - \gamma_i u_0\big) \\ &= x_0 + U \operatorname{prox}_{\gamma_i g}(\bar{z}_i) + \frac{1}{1 + \gamma_i \mu}(q_i - \gamma_i u_0). \end{aligned}$$

Since $\bar{y}_i = \operatorname{prox}_{\gamma_i g}(\bar{z}_i)$ and $\operatorname{supp}(\bar{y}_i) \subseteq I_{i+1}$, this gives

$$\operatorname{prox}_{\gamma_i g_U}(z_i) = x_0 + \sum_{j \in I_{i+1}} (\bar{y}_i)_j \, v_j + \frac{1}{1 + \gamma_i \mu}(q_i - \gamma_i u_0) = y_i.$$

Thus the true i-th oracle output is exactly y_i, which proves the induction step. Hence

the actual run of $\mathbf{A}$ on (f_U, g_U) has exactly the recursively constructed $2N$-step transcript. Since the final output depends only on this trajectory, the final output is exactly x_N.

We now verify the zero-respecting property. For each $i = 0, \ldots, 2N-1$, define

$$\tilde{d}_i := \begin{cases} \tilde{z}_i - \mathrm{prox}_{\gamma_i f}(\tilde{z}_i), & \delta_i = 0, \\ \tilde{z}_i - \mathrm{prox}_{\gamma_i g}(\tilde{z}_i), & \delta_i = 1. \end{cases}$$

Since $y_i = \mathrm{prox}_{\gamma_i f_U}(z_i)$ when $\delta_i = 0$ and $y_i = \mathrm{prox}_{\gamma_i g_U}(z_i)$ when $\delta_i = 1$, Lemma 11 and Lemma 12 give

$$\tilde{d}_i = U^\top (z_i - y_i).$$

Next, we show that

$$U^\top (y_i - x_0) = \bar{y}_i.$$

If $\delta_i = 0$, then by construction,

$$y_i = x_0 + \sum_{j \in I_{i+1}} (\bar{y}_i)_j \, v_j + q_i + \gamma_i u_0.$$

Multiplying by $U^\top$, and using $U^\top q_i = 0$ and $U^\top u_0 = 0$, we obtain

$$U^\top (y_i - x_0) = \bar{y}_i.$$

If $\delta_i = 1$, then

$$y_i = x_0 + \sum_{j \in I_{i+1}} (\bar{y}_i)_j \, v_j + \frac{1}{1 + \gamma_i \mu}(q_i - \gamma_i u_0),$$

and again $U^\top q_i = 0$ and $U^\top u_0 = 0$ imply

$$U^\top (y_i - x_0) = \bar{y}_i.$$

Since also

$$U^\top(z_i - x_0) = \bar{z}_i,$$

we conclude that

$$\tilde{d}_i = U^\top(z_i - y_i) = U^\top(z_i - x_0) - U^\top(y_i - x_0) = \bar{z}_i - \bar{y}_i = \bar{d}_i.$$

Because $I_0 = \varnothing$ and

$$I_{i+1} = I_i \cup \operatorname{supp}(\bar{d}_i) = I_i \cup \operatorname{supp}(\tilde{d}_i),$$

it follows by induction that

$$I_i = \bigcup_{k=0}^{i-1} \operatorname{supp}(\tilde{d}_k), \qquad i = 0, \ldots, 2N.$$

Moreover,

$$\operatorname{supp}(\tilde{z}_i) = \operatorname{supp}(\bar{z}_i) \subseteq I_i = \bigcup_{k=0}^{i-1} \operatorname{supp}(\tilde{d}_k), \qquad i = 0, \ldots, 2N-1.$$

Finally, for every $j \notin I_{2N}$ we have $v_j \in S_{2N+1}^\perp$ and $x_N - x_0 \in S_{2N+1}$, so

$$\langle v_j, x_N - x_0 \rangle = 0.$$

Therefore

$$\operatorname{supp}(\tilde{x}_N) = \operatorname{supp}\big(U^\top(x_N - x_0)\big) \subseteq I_{2N} = \bigcup_{k=0}^{2N-1} \operatorname{supp}(\tilde{d}_k).$$

Hence

$$\operatorname{supp}(\tilde{x}_N) \subseteq \bigcup_{k=0}^{2N-1} \operatorname{supp}(\tilde{d}_k),$$

which is exactly the prox-prox zero-respecting property. □

Combining the resisting-oracle lemma with the zero-respecting property, we obtain the

desired lower bound for arbitrary deterministic prox-prox methods.

Proof. By Theorem 3 with initial pair $(0, 0)$, there exist a closed, proper, and convex function $\tilde{f}\colon \mathbb{R}^{2N+2} \to \mathbb{R} \cup \{\infty\}$ and a closed, proper, and μ-strongly convex function $\tilde{g}\colon \mathbb{R}^{2N+2} \to \mathbb{R} \cup \{\infty\}$ with

$$\tilde{x}_\star \in \operatorname{argmin}(\tilde{f} + \tilde{g}), \qquad \tilde{u}_\star \in \partial\tilde{g}(\tilde{x}_\star),\ -\tilde{u}_\star \in \partial\tilde{f}(\tilde{x}_\star).$$

Since $d \geq 4N + 4 = (2N + 2) + (2N + 2)$, by Lemma 13 there exists a matrix

$$U \in \mathbb{R}^{d\times(2N+2)}$$

with orthonormal columns such that $U^\top u_0 = 0$, and if

$$\mathbf{A}\big((x_0, u_0), (\tilde{f}_U, \tilde{g}_U)\big) = \big\{\{(z_i, \delta_i, \gamma_i)\}_{i=0}^{2N-1}, x_N\big\},$$

and we define

$$\tilde{z}_i := U^\top(z_i - x_0), \qquad \tilde{x}_N := U^\top(x_N - x_0),$$

then the projected sequence

$$\big\{\{(\tilde{z}_i, \delta_i, \gamma_i)\}_{i=0}^{2N-1}, \tilde{x}_N\big\}$$

is prox-prox zero-respecting with respect to $(\tilde{f}, \tilde{g})$. Therefore, by Theorem 3 and Lemma 8,

$$\|\tilde{x}_N - \tilde{x}_\star\|^2 \geq \frac{\|\tilde{x}_\star\|^2 + \|\tilde{u}_\star\|^2}{1 + 4N^2\mu^2 + 4N\mu}. \tag{5.6}$$

Now define

$$f := \tilde{f}_U, \qquad g := \tilde{g}_U,$$

and

$$x_\star := x_0 + U\tilde{x}_\star, \qquad u_\star := u_0 + U\tilde{u}_\star.$$

We first show that $x_\star \in \operatorname{argmin}(f+g)$. Every $z \in \mathbb{R}^d$ can be written uniquely as

$$z = x_0 + Uy + p, \qquad y \in \mathbb{R}^{2N+2}, \quad p \in \operatorname{range}(P),$$

where $P := I - UU^\top$. By definition of f_U and g_U, we have

$$f(z) = \tilde{f}(y) - \langle u_0, Uy + p \rangle,$$

and

$$g(z) = \tilde{g}(y) + \langle u_0, Uy + p \rangle + \frac{\mu}{2}\|p\|^2.$$

Hence

$$(f+g)(z) = \tilde{f}(y) + \tilde{g}(y) + \frac{\mu}{2}\|p\|^2.$$

This is minimized exactly when

$$y \in \operatorname{argmin}(\tilde{f} + \tilde{g}) \quad \text{and} \quad p = 0.$$

Thus

$$x_\star = x_0 + U\tilde{x}_\star \in \operatorname{argmin}(f+g).$$

Next we show that $u_\star \in \partial g(x_\star)$ and $-u_\star \in \partial f(x_\star)$. By definition,

$$g(x) = \tilde{g}\big(U^\top(x - x_0)\big) + \langle u_0, x - x_0 \rangle + \frac{\mu}{2}\|P(x - x_0)\|^2.$$

Hence

$$\partial g(x) = U\, \partial\tilde{g}\big(U^\top(x - x_0)\big) + u_0 + \mu P(x - x_0).$$

Since

$$x_\star = x_0 + U\tilde{x}_\star,$$

we have

$$U^\top(x_\star - x_0) = \tilde{x}_\star, \qquad P(x_\star - x_0) = 0.$$

Therefore

$$\partial g(x_\star) = U\,\partial\tilde{g}(\tilde{x}_\star) + u_0.$$

Since $\tilde{u}_\star \in \partial\tilde{g}(\tilde{x}_\star)$, we conclude that

$$u_\star = u_0 + U\tilde{u}_\star \in \partial g(x_\star).$$

Similarly,

$$f(x) = \tilde{f}\big(U^\top(x - x_0)\big) - \langle u_0, x - x_0\rangle.$$

So,

$$\partial f(x) = U\partial\tilde{f}\big(U^\top(x - x_0)\big) - u_0,$$

and

$$\partial f(x_\star) = U\partial\tilde{f}(\tilde{x}_\star) - u_0.$$

Thus, we conclude that $-u_\star = -u_0 - U\tilde{u}_\star \in \partial f(x_\star)$. Putting altogether, we have

$$\|x_0 - x_\star\|^2 = \|U\tilde{x}_\star\|^2 = \|\tilde{x}_\star\|^2,$$

and

$$\|u_0 - u_\star\|^2 = \|U\tilde{u}_\star\|^2 = \|\tilde{u}_\star\|^2.$$

Thus

$$\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2 = \|\tilde{x}_\star\|^2 + \|\tilde{u}_\star\|^2.$$

Finally,

$$x_N - x_\star = x_N - x_0 - U\tilde{x}_\star = U(\tilde{x}_N - \tilde{x}_\star) + P(x_N - x_0),$$

and the two terms on the right-hand side are orthogonal. Hence

$$\|x_N - x_\star\|^2 = \|\tilde{x}_N - \tilde{x}_\star\|^2 + \|P(x_N - x_0)\|^2 \geq \|\tilde{x}_N - \tilde{x}_\star\|^2.$$

Combining with (5.6) yields

$$\|x_N - x_\star\|^2 \geq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2 + 4N\mu}.$$

□

5.4 Numerical Results

We compare FDR with FISTA, Douglas–Rachford splitting (DRS), accelerated Chambolle–Pock (CP), and accelerated Davis–Yin splitting (DYS) on a family of elastic-net regression problems. Each instance has the form

$$\underset{x}{\text{minimize}} \quad \underbrace{\|Ax - b\|_2^2 + \frac{\mu}{2}\|x\|^2}_{=g(x)} + \underbrace{\lambda\|x\|_1}_{=f(x)}.$$

We randomly generated 100 instances with $A \in \mathbb{R}^{40\times 100}$ and $b \in \mathbb{R}^{40}$ being computed as $Ax_{\text{true}} + \epsilon$, where ϵ is drawn from a Gaussian with standard deviation 0.01, and x_{true} is generated randomly with only 10% of elements being nonzero. We use $\mu = \lambda = 10^{-3}$.

Since the guarantee on FDR's squared distance to the solution is on the final iterate, x_N, the FDR curve in Figure 5.1 reports the terminal error $\|x_k - x_\star\|^2$ obtained by running FDR for $N = k$ iterations. The solid curves are the median of each algorithm's performance, and the shaded region shows the interquartile range for each on the 100 instances.

Figure 5.1 shows that FDR eventually achieves the smallest error among the tested methods. The observed FDR errors also remain below the theoretical bound, which is valid, but we see these errors are much smaller than the theoretical upper bound on these random elastic-net instances.

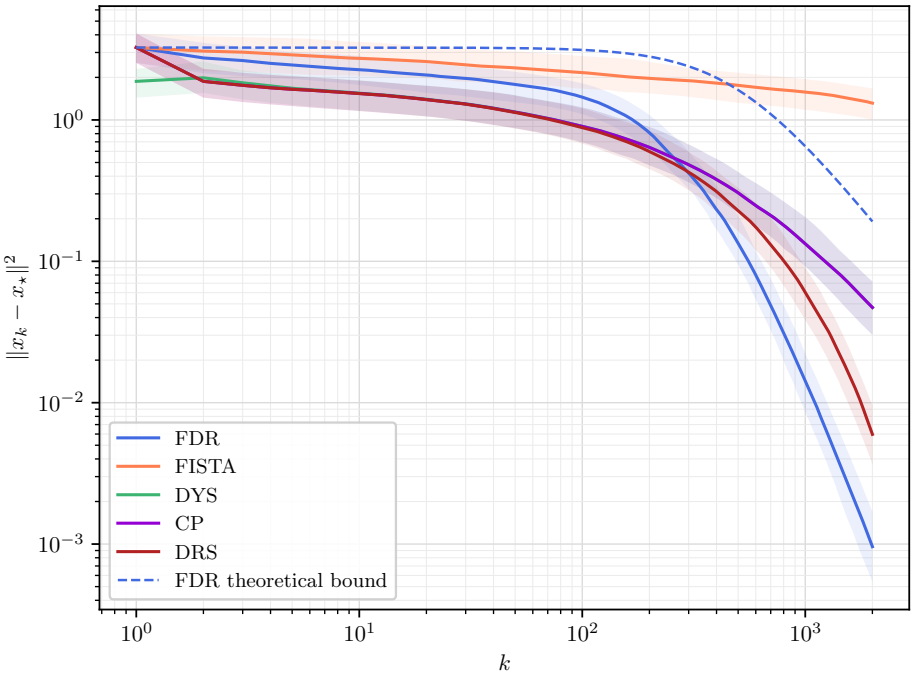


Figure 5.1: Median squared distance to the solution over 100 elastic-net instances. Shaded bands show the interquartile range.

5.5 Conclusion

In this work, we presented Fast Douglas–Rachford Splitting (FDR), an accelerated Douglas–Rachford-type method for minimizing the sum of a convex and a μ-strongly convex function. We established the guarantee

$$\|x_N - x_\star\|^2 \leq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2},$$

which improves upon the leading asymptotic constant of the accelerated methods of Chambolle–Pock and Davis–Yin by a factor of 4. We further established a complexity lower bound showing that both the $\mathcal{O}(1/N^2)$ convergence rate and the leading-order constant achieved by FDR are optimal.

It is worth noting that the acceleration mechanism and the lower-bound construction are

fundamentally different than those of Nesterov acceleration. In this sense, the accelerated methods of Chambolle–Pock, Davis–Yin, and FDR should be viewed as belonging to a class of acceleration mechanism distinct from the Nesterov-type acceleration. Developing a more principled understanding of the different classes of acceleration mechanisms is an interesting direction for future work.

Finally, while our lower bound asymptotically matches the upper bound in the leading-order term, a gap remains in the higher-order terms. Since FDR was discovered through BnB-PEP [61], a framework for discovering optimal first-order algorithms, we conjecture that the upper bound achieved by FDR is in fact exactly optimal, implying that the current lower bound is likely loose. We leave the task of fully tightening the lower bound to future work.

Chapter 6

GPU-ACCELERATED BATCH SEQUENTIAL CONVEX PROGRAMMING

Monte Carlo analysis is critical for certifying the safety and performance of guidance, navigation, and control (GNC) systems [125]. For a 6-DoF powered-descent guidance algorithm, Monte Carlo analysis would involve generating hundreds or even thousands of dispersed initial conditions around the nominal initial condition, computing trajectories with these initial conditions, and assessing statistics such as solve-times, percentage of converged trajectories, and propellant consumption. However, generating thousands of trajectories serially can be time consuming, and can lead to increased development times. Since Monte Carlo simulation is *embarrassingly parallel* [95], we propose performing Monte Carlo analyses on a graphics processing unit (GPU) to exploit its parallel computing capability.

Due to their parallel computing capability and high memory bandwidth, GPUs have become increasingly popular in fields that require solving large-scale problems (in which parallelism can be exploited). Most notably, GPUs have become a staple in machine learning for training deep learning models. For example, OpenAI has a supercomputer with 10,000 GPUs to train its large language models [113], and Tesla, reportedly, also has a 10,000-GPU cluster to perform training for its machine learning-based full self-driving technology [174]. Additionally, many modern machine learning tools such as JAX, TensorFlow, and PyTorch, provide programmers with user-friendly, high-level interfaces to leverage the compute power of GPUs [2, 33, 153].

GPUs have also been used outside of machine learning for a variety of tasks. For example, [96] uses a GPU for running projectile simulations, i.e., propagating the equations of motion for a projectile, in parallel. In [159], the authors develop a framework to perform trajectory

optimization on GPUs; however, they consider a restrictive trajectory optimization problem template that does not model dynamics or generic nonconvex path constraints. The only constraints considered are maximum speed, maximum acceleration, and keepout zone constraints. As a result, the generated trajectories are not guaranteed to obey the equations of motion of the vehicle.

In this paper, we demonstrate a GPU-based parallelized implementation for Monte Carlo analysis of trajectory optimization problems with general nonlinear dynamics and nonconvex path constraints. The solutions obtained are guaranteed to be feasible with respect to the original nonlinear dynamics, and all the imposed constraints are guaranteed to be satisfied at all points in time, i.e., in continuous-time (subject to successful convergence and up to numerical integration and optimization tolerances). This final guarantee on continuous-time constraint satisfaction is a result of the problem formulation proposed in [80]. To demonstrate this framework, we consider the 6-DoF powered-descent guidance algorithm [180].

General trajectory optimization problems, such as the 6-DoF PDG problem, can be posed as nonconvex optimization problems in continuous-time [128]. Given the nonconvexity of these problems, it is typically prohibitively expensive, if not outright impossible, to obtain globally optimal trajectories. As a result, we must turn to solution procedures, such as sequential convex programming (SCP) [127], that yield locally optimal solutions. At a high level, SCP involves (1) convexification (often via linearization) of all nonconvex constraints about some reference trajectory to yield an infinite-dimensional convex optimization problem; (2) discretization, to convert the problem to a numerically tractable finite-dimensional convex optimization problem; and finally (3) finding a solution to this finite-dimensional convex subproblem to update the reference trajectory. This process is applied iteratively until convergence. As the name suggests, this process is inherently sequential in nature. However, we can leverage parallelism to simultaneously solve for many trajectories in Monte Carlo analysis.

There are a number of factors that make developing a GPU-based parallelized implementation of SCP challenging. SCP requires the use of many linear algebra operations and the

solution to a sequence of convex optimization subproblems. For a CPU-based implementation, we can use linear algebra packages such as OpenBLAS [200] and open-source convex optimization solvers such as OSQP and ECOS [69, 177]. However, OpenBLAS binaries are compiled for CPU architectures and NVIDIA's linear algebra libraries, cuBLAS and cuSPARSE, are written such that functions from these libraries must be called by the CPU, even though the computations are performed on the GPU. This is undesirable for our implementation of SCP, since we require all computations for a single trajectory to be performed by a single GPU thread. Similarly, although a GPU-accelerated implementation of OSQP exists, the solver itself must be called by the CPU [170]. Although off-the-shelf convex optimization solvers could be modified to work on the GPU, custom parser frameworks are typically required—to put the convex subproblem in the canonical form prior to being passed to the solver—as well [67, 120]. Thus, for our GPU-based implementation of SCP, we implement Level 1, 2, and 3 BLAS operations for discretization and other such operations, and the first-order proportional-integral projected gradient (PIPG) solver [101, 196], customized to the trajectory optimization problem template [79, 99], allowing us to bypass parsing entirely. Further, as in the sequential conic optimization (SeCO) framework described in [99], this framework completely eliminates the need for sparse linear algebra operations, is entirely matrix-factorization/inversion-free, and is implemented using a library-free codebase.

6.0.1 Notation

$\mathbb{R}n = \mathbb{R}n \times 1$	Vectors are matrices with one column
$1_n, 0_n$	Vector of ones and zeros, respectively, in $\mathbb{R}^n$ (subscript inferred whenever omitted)
$0_{n\times m}$	Matrix of zeros in $\mathbb{R}^{n\times m}$
I_n	Identity matrix in $\mathbb{R}^{n\times n}$
$V_{[1:n]}$	Collection of vectors or matrices V_k, for $k = 1, \ldots, n$
$\lvert v\rvert_+$	$\square \mapsto \max\{0, \square\}$ applied element-wise for vector v
v^2	$\square \mapsto \square^2$ applied element-wise for vector v
(u, v)	Concatenation of vectors $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^m$ to form a vector in $\mathbb{R}^{n+m}$
$[A\, B]$	Concatenation of matrices A and B with same number of rows
$\begin{bmatrix} A \\ B \end{bmatrix}$	Concatenation of matrices A and B with same number of columns

6.1 Problem formulation

We consider a continuous-time dynamical system of the form:

$$\frac{\mathrm{d}\xi(t)}{\mathrm{d}t} = \dot{\xi}(t) = F(\xi(t), \zeta(t)) \tag{6.1}$$

for $t \in [0, t_\mathrm{f}]$, with state $\xi \in \mathbb{R}^{n_\xi}$ and control input $\zeta \in \mathbb{R}^{n_\zeta}$, which are subject to path constraints:

$$g(\xi(t), \zeta(t)) \leq 0_{n_g} \tag{6.2a}$$

$$h(\xi(t), \zeta(t)) = 0_{n_h} \tag{6.2b}$$

where g and h are vector-valued functions. The operators "$\leq$" and "$=$" are interpreted element-wise. Boundary conditions at the initial and final times could also be specified.

The goal of trajectory optimization is to generate a solution $\xi(t), \zeta(t)$ for (6.1) that satisfies the path constraints, boundary conditions, and optimizes a linear function of the

terminal state $\xi(t_\mathrm{f})$, i.e., solve the following Mayer form optimal control problem [114, Sec. 3.3.2]:

$$\begin{aligned}
\underset{t_\mathrm{f},\xi,\zeta}{\text{minimize}} \quad & \xi(t_\mathrm{f})^\top e_\mathrm{cost} && && \text{(6.3a)}\\
\text{subject to} \quad & \dot{\xi}(t) = F(\xi(t),\zeta(t)) && t\in[0,t_\mathrm{f}] && \text{(6.3b)}\\
& g(\xi(t),\zeta(t)) \leq 0 && t\in[0,t_\mathrm{f}] && \text{(6.3c)}\\
& h(\xi(t),\zeta(t)) = 0 && t\in[0,t_\mathrm{f}] && \text{(6.3d)}\\
& E_\mathrm{i}\xi(0) = z_\mathrm{i},\ E_\mathrm{f}\xi(t_\mathrm{f}) = z_\mathrm{f} && && \text{(6.3e)}
\end{aligned}$$

where $E_\mathrm{i} \in \mathbb{R}^{n_\mathrm{i}\times n_\xi}$ and $E_\mathrm{f} \in \mathbb{R}^{n_\mathrm{f}\times n_\xi}$ select components of the initial and final state, respectively.

6.1.1 Time-Dilation and Constraint Reformulation

When the final time, t_f, is an unknown, we adopt a well-known technique called time-dilation [80, 101, 181] to convert a free-final-time problem to an equivalent fixed-final-time problem:

$$\frac{\mathrm{d}\xi(t(\tau))}{\mathrm{d}\tau} = \overset{\circ}{\xi}(\tau) = \frac{\mathrm{d}\xi(\tau)}{\mathrm{d}t}s(\tau) = s(\tau)F(\xi(\tau),\zeta(\tau)) \tag{6.4}$$

for $\tau \in [0,1]$, where $t(\tau)$ is a strictly increasing function with domain $[0,1]$. Note that ξ and ζ are re-defined to be functions of $\tau \in [0,1]$ instead of t. The dilation factor:

$$s(\tau) = \frac{\mathrm{d}t(\tau)}{\mathrm{d}\tau}$$

is treated as an additional control input and is required to be positive. The control input is then re-defined to be $u = (\zeta, s)$, and the final time is given by:

$$t_\mathrm{f} = \int_0^1 s(\tau)\mathrm{d}\tau$$

Next, to address the long-standing issue of inter-sample constraint violation in direct methods [24], we augment the path constraints to the time-dilated system dynamics (6.4) as follows:

$$\overset{\circ}{x}(\tau) = \begin{bmatrix} \overset{\circ}{\xi}(\tau) \\ \overset{\circ}{y}(\tau) \end{bmatrix} = s(\tau) \begin{bmatrix} F(\xi(\tau), \zeta(\tau)) \\ 1^\top |g(\xi(\tau), \zeta(\tau))|_+^2 + 1^\top h(\xi(\tau), \zeta(\tau))^2 \end{bmatrix} = f(x(\tau), u(\tau)) \tag{6.5}$$

for $\tau \in [0, 1]$, where $x = (\xi, y)$ is the re-defined state. Periodic boundary condition on the constraint violation integrator:

$$y(0) = y(1) \tag{6.6}$$

ensure that the path constraints are satisfied for all $\tau \in [0, 1]$. We refer the reader to [80, 167] for further details.

6.1.2 Discretization

Discretization is a crucial step in direct numerical optimal control for computational tractability, i.e., for obtaining an approximate finite-dimensional optimization problem from the original infinite-dimensional optimal control problem. The first step is to generate a grid with N nodes within the interval $[0, 1]$:

$$0 = \tau_1 < \ldots < \tau_N = 1 \tag{6.7}$$

The time duration over sub-interval $[\tau_k, \tau_{k+1}]$ is given by:

$$\Delta t_k = \int_{\tau_k}^{\tau_{k+1}} s(\tau) \mathrm{d}\tau$$

for $k = 1, \ldots, N-1$, and the following constraints are specified on the dilation factor and time duration:

$$
\begin{aligned}
s_{\min} \leq s(\tau) \leq s_{\max} \\
t_{\mathrm{f}} = \sum_{k=1}^{N-1} \Delta t_k \leq t_{\mathrm{f,max}} \\
\Delta t_{\min} \leq \Delta t_k \leq \Delta t_{\max}
\end{aligned}
$$

Next, we parameterize the control input with a first-order-hold (FOH):

$$
u(\tau) = \left(\frac{\tau_{k+1} - \tau}{\tau_{k+1} - \tau_k} \right) u_k + \left(\frac{\tau - \tau_k}{\tau_{k+1} - \tau_k} \right) u_{k+1}
$$

for $\tau \in [\tau_k, \tau_{k+1}]$ and $k = 1, \ldots, N-1$. In practice, the combination of time-dilation and FOH parameterization for control input is sufficient for accurately modeling a large class of optimal control problems. See [125] for a detailed discussion on the relative merits of popular discretization and parameterization techniques in numerical optimal control.

We treat the node-point values of the state and control input, x_k and u_k, respectively, as decision variables, and re-write the system dynamics (6.5) equivalently as:

$$
x_{k+1} = x_k + \int_{\tau_k}^{\tau_{k+1}} f(x(\tau), u(\tau)) \mathrm{d}\tau \tag{6.8}
$$

for $k = 1, \ldots, N-1$, where $x(\tau)$ is the solution to (6.5) over $[\tau_k, \tau_{k+1}]$ with initial condition x_k, and $u(\tau)$ is FOH-parameterized using u_k and u_{k+1}. Then, (6.8) can be compactly expressed through function $f_k : \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$ as follows:

$$
x_{k+1} = f_k(x_k, u_k, u_{k+1}) \tag{6.9}
$$

The boundary condition (6.6) is relaxed to:

$$y_{k+1} - y_k \leq \epsilon$$

for $k = 1, \ldots, N-1$ and a sufficiently small $\epsilon > 0$, so that all feasible solutions do not violate LICQ [80]. Note that y_k is the value of the state y in (6.5) at node τ_k.

The discretized optimal control problem with parameterized control input is given by:

$$\begin{aligned}
\underset{x_k, u_k}{\text{minimize}} \quad & x_N^\top \tilde{e}_{\text{cost}} && && \text{(6.10a)}\\
\text{subject to} \quad & x_{k+1} = f_k(x_k, u_k, u_{k+1}) && k = 1, \ldots, N-1 && \text{(6.10b)}\\
& E_y(x_{k+1} - x_k) \leq \epsilon && k = 1, \ldots, N-1 && \text{(6.10c)}\\
& u_{\min} \leq u_k \leq u_{\max} && k = 1, \ldots, N && \text{(6.10d)}\\
& \tilde{E}_{\text{i}} x_1 = (z_{\text{i}}, 0), \;\; \tilde{E}_{\text{f}} x_N = z_{\text{f}} && && \text{(6.10e)}
\end{aligned}$$

where

$$\tilde{e}_{\text{cost}} = (e_{\text{cost}}, 0) \tag{6.11}$$

$$E_y = [0_{1 \times n_\xi} \;\; 1] \tag{6.12}$$

$$\tilde{E}_{\text{i}} = \begin{bmatrix} E_{\text{i}} & 0_{n_\xi} \\ 0_{1 \times n_\xi} & 1 \end{bmatrix} \tag{6.13}$$

$$\tilde{E}_{\text{f}} = \begin{bmatrix} E_{\text{f}} \; 0_{n_\xi} \end{bmatrix} \tag{6.14}$$

6.1.3 *Sequential Convex Programming*

We adopt the prox-linear method [74], an SCP algorithm, to solve:

$$\underset{x_k,u_k}{\text{minimize}} \quad x_N^\top \tilde{e}_{\text{cost}} + w_{\text{ep}} \sum_{k=1}^{N-1} \|x_{k+1} - f_k(x_k, u_k, u_{k+1})\|_1 \tag{6.15a}$$

$$\text{subject to} \quad E_y(x_{k+1} - x_k) \leq \epsilon \tag{6.15b}$$

$$u_{\min} \leq u_k \leq u_{\max} \tag{6.15c}$$

$$\tilde{E}_{\text{i}} x_1 = (z_{\text{i}}, 0), \;\; \tilde{E}_{\text{f}} x_N = z_{\text{f}} \tag{6.15d}$$

where the (only) nonconvex constraint of (6.10) is exactly penalized with ℓ_1-norm. For a large enough, finite w_{ep}, we can show that a solution of (6.15) that is feasible with respect to (6.10b) is also a solution of (6.10). See [143, Chap. 17] for a detailed discussion.

The penalized problem (6.15) can be compactly stated as:

$$\underset{Z \in \mathcal{Z}}{\text{minimize}} \quad L(Z) + w_{\text{ep}} \|G(Z)\|_1 \tag{6.16}$$

where $Z = (x_1, \ldots, x_N, u_1, \ldots, u_N)$ and $\mathcal{Z}$ is a convex set corresponding to constraints (6.15b), (6.15c), and (6.15d).

The prox-linear method determines a stationary point of (6.16) by solving a sequence of convex subproblems. At iteration $j+1$, it solves the following convex problem:

$$\underset{Z \in \mathcal{Z}}{\text{minimize}} \quad L(Z) + w_{\text{ep}} \|G(Z^j) + \nabla G(Z^j)(Z - Z^j)\|_1 + \frac{w_{\text{prox}}}{2} \|Z - Z^j\|_2^2 \tag{6.17}$$

where Z^j is the solution from iteration j. The sequence Z^j can be shown to converge for an appropriate choice of the proximal term weight w_{prox}. Further, if the converged point is feasible for (6.10), then it is also a KKT point.

There are two important steps in the construction of each convex subproblem–(1) computation of ∇G: the gradient of f_k with respect to the previous iterate, and (2) scaling.

Gradient of Discretized Dynamics

At each iteration of SCP, the discretized dynamics constraint (6.10b) is linearized with respect to the previous iterate, denoted by $\bar{x}_k$, $\bar{u}_k$, for $k = 1, \ldots, N$.

For each $k = 1, \ldots, N-1$, let $\bar{x}^k(\tau)$ denote the solution to (6.5) over $[\tau_k, \tau_{k+1}]$ generated with initial condition $\bar{x}_k$ and control input $\bar{u}(\tau)$, which is parameterized using $\bar{u}_k$, for $k = 1, \ldots, N$. Note that, in general, $\bar{x}^k(\tau_{k+1}) \neq \bar{x}_{k+1}$ before SCP converges [101].

The Jacobians of f in (6.5), evaluated with respect to $\bar{x}^k(\tau)$, $\bar{u}(\tau)$, are denoted by:

$$A^k(\tau) = \frac{\partial f(\bar{x}^k(\tau), \bar{u}(\tau))}{\partial x}$$
$$B^k(\tau) = \frac{\partial f(\bar{x}^k(\tau), \bar{u}(\tau))}{\partial u}$$

for $\tau \in [\tau_k, \tau_{k+1}]$.

The discretized dynamics constraint (6.8) is linearized as follows:

$$x_{k+1} = A_k x_k + B_k^- u_k + B_k^+ u_{k+1} + w_k \tag{6.18}$$

for $k = 1, \ldots, N-1$, where A_k, B_k^-, B_k^+, w_k result from the solution to the following initial value problem over $\tau \in [\tau_k, \tau_{k+1}]$:

$$\begin{aligned}
\mathring{\Phi}_x(\tau, \tau_k) &= A^k(\tau)\Phi_x(\tau, \tau_k) \\
\mathring{\Phi}_u^-(\tau, \tau_k) &= A^k(\tau)\Phi_u^-(\tau, \tau_k) + B^k(\tau)\left(\frac{\tau_{k+1} - \tau}{\tau_{k+1} - \tau_k}\right) \\
\mathring{\Phi}_u^+(\tau, \tau_k) &= A^k(\tau)\Phi_u^+(\tau, \tau_k) + B^k(\tau)\left(\frac{\tau - \tau_k}{\tau_{k+1} - \tau_k}\right) \\
\Phi_x(\tau_k, \tau_k) &= I_{n_x} \\
\Phi_u^-(\tau_k, \tau_k) &= 0_{n_x \times n_u} \\
\Phi_u^+(\tau_k, \tau_k) &= 0_{n_x \times n_u}
\end{aligned}$$

$$
\begin{aligned}
A_k &= \Phi_x(\tau_{k+1}, \tau_k) \\
B_k^- &= \Phi_u^-(\tau_{k+1}, \tau_k) \\
B_k^+ &= \Phi_u^+(\tau_{k+1}, \tau_k) \\
w_k &= \bar{x}^k(\tau_{k+1}) - A_k \bar{x}_k - B_k^- \bar{u}_k - B_k^+ \bar{u}_{k+1}
\end{aligned}
$$

Convex Subproblem with Scaling

The final step in the construction of the convex subproblem involves scaling the decision variables. In particular, we treat $\hat{x}_k$ and $\hat{u}_k$ as the decision variables, which are linearly related to x_k and u_k as follows:

$$x_k = P_x \hat{x}_k + \bar{x}_k \tag{6.19a}$$

$$u_k = P_u \hat{u}_k + \bar{u}_k \tag{6.19b}$$

where P_x and P_u are selected to ensure that $\hat{x}_k$ and $\hat{u}_k$, for $k = 1, \ldots, N$, are roughly on the same order of magnitude, which greatly aids in convergence. Next, we define:

$$\hat{A}_k^- = P_x^{-1} A_k P_x \tag{6.20a}$$

$$\hat{A}_k^+ = -I_{n_x} \tag{6.20b}$$

$$\hat{B}_k^- = P_x^{-1} B_k^- P_u \tag{6.20c}$$

$$\hat{B}_k^+ = P_x^{-1} B_k^+ P_u \tag{6.20d}$$

$$\hat{w}_k = P_x^{-1}(\bar{x}^k(\tau_{k+1}) - \bar{x}_{k+1}) \tag{6.20e}$$

Finally, the convex subproblem (6.17), which is a quadratic program (QP), can be stated

as:

$$\underset{\hat{x}_k,\hat{u}_k}{\text{minimize}} \quad w_{\text{cost}}\hat{x}_N^\top e_{\text{cost}} + \frac{w_{\text{prox}}}{2}\sum_{k=1}^{N}\|\hat{x}_k\|_2^2 + \|\hat{u}_k\|_2^2 + w_{\text{ep}}\sum_{k=1}^{N-1}1_{n_x}^\top(\mu_k^+ + \mu_k^-) \tag{6.21a}$$

$$\text{subject to} \quad \hat{A}_k^-\hat{x}_k + \hat{A}_k^+\hat{x}_{k+1} + \hat{B}_k^-\hat{u}_k + \hat{B}_k^+\hat{u}_{k+1} + \mu_k^+ - \mu_k^- + \hat{w}_k = 0 \tag{6.21b}$$

$$E_y(\hat{x}_{k+1} - \hat{x}_k) \leq \hat{\varepsilon}_k \tag{6.21c}$$

$$\mu_k^+ \geq 0,\ \mu_k^- \geq 0 \tag{6.21d}$$

$$\hat{u}_{k,\min} \leq \hat{u}_k \leq \hat{u}_{k,\max} \tag{6.21e}$$

$$\tilde{E}_{\text{i}}\hat{x}_1 = \hat{z}_{\text{i}},\ \tilde{E}_{\text{f}}\hat{x}_N = \hat{z}_{\text{f}} \tag{6.21f}$$

where

$$\hat{\varepsilon}_k = \epsilon E_y P_x^{-1} E_y^\top - E_y P_x^{-1}(\bar{x}_{k+1} - \bar{x}_k)$$

$$\hat{u}_{k,\min/\max} = P_u^{-1}(u_{\min/\max} - \bar{u}_k)$$

$$\hat{z}_{\text{i}} = \tilde{E}_{\text{i}} P_x^{-1} \tilde{E}_{\text{i}}^\top((z_{\text{i}}, 0) - \tilde{E}_{\text{i}}\bar{x}_1)$$

$$\hat{z}_{\text{f}} = \tilde{E}_{\text{f}} P_x^{-1} \tilde{E}_{\text{f}}^\top(z_{\text{f}} - \tilde{E}_{\text{f}}\bar{x}_N)$$

6.2 Convex optimization solver

The convex subproblem (6.21) is a quadratic program (QP), which can be represented in the canonical form as:

$$\text{minimize} \quad \frac{1}{2}z^\top\hat{P}z + \hat{p}^\top z \tag{6.22a}$$

$$\text{subject to} \quad \hat{G}z = \hat{g} \tag{6.22b}$$

$$\hat{H}z \leq \hat{h} \tag{6.22c}$$

where

$$z = \big(\hat{x}_1, \ldots, \hat{x}_N,\, \hat{u}_1, \ldots, \hat{u}_N,\, \mu_1^-, \ldots, \mu_{N-1}^-,\, \mu_1^+, \ldots, \mu_{N-1}^+\big) \in \mathbb{R}^{(n_x+n_u)N+2n_x(N-1)} \tag{6.23a}$$

$$\hat{P} = \mathrm{blkdiag}(w_{\mathrm{tr}} I_{(n_x+n_u)N},\, 0_{2n_x(N-1)\times 2n_x(N-1)}) \tag{6.23b}$$

$$\hat{p} = w_{\mathrm{cost}}\big(0_{n_x(N-1)},\, e_x,\, 0_{n_u N+2n_x(N-1)}\big) \tag{6.23c}$$

$$+\, w_{\mathrm{vc}}\big(0_{(n_x+n_u)N},\, 1_{2n_x(N-1)}\big) \tag{6.23d}$$

$$\hat{G} = \begin{bmatrix} \hat{G}_x & \hat{G}_u & \hat{G}_\mu \end{bmatrix} \tag{6.23e}$$

$$\hat{G}_x = \begin{bmatrix} \hat{A}^- & 0_{n_x(N-1)\times n_x} \end{bmatrix} + \begin{bmatrix} 0_{n_x(N-1)\times n_x} & \hat{A}^+ \end{bmatrix} \tag{6.23f}$$

$$\hat{G}_u = \begin{bmatrix} \hat{B}^- & 0_{n_x(N-1)\times n_u} \end{bmatrix} + \begin{bmatrix} 0_{n_x(N-1)\times n_u} & \hat{B}^+ \end{bmatrix} \tag{6.23g}$$

$$\hat{G}_\mu = \begin{bmatrix} I_{n_x(N-1)} & -I_{n_x(N-1)} \end{bmatrix} \tag{6.23h}$$

$$\hat{A}^- = \mathrm{blkdiag}(\hat{A}_1^-, \ldots, \hat{A}_{N-1}^-) \tag{6.23i}$$

$$\hat{A}^+ = \mathrm{blkdiag}(\hat{A}_1^+, \ldots, \hat{A}_{N-1}^+) \tag{6.23j}$$

$$\hat{B}^- = \mathrm{blkdiag}(\hat{B}_1^-, \ldots, \hat{B}_{N-1}^-) \tag{6.23k}$$

$$\hat{B}^+ = \mathrm{blkdiag}(\hat{B}_1^+, \ldots, \hat{B}_{N-1}^+) \tag{6.23l}$$

$$\hat{g} = -\big(\hat{w}_1, \ldots, \hat{w}_{N-1}\big) \tag{6.23m}$$

$$\hat{H}_y = -\begin{bmatrix} I_{N-1} \otimes E_y & 0_{n_y(N-1)\times n_x} \end{bmatrix} + \begin{bmatrix} 0_{n_y(N-1)\times n_x} & I_{N-1} \otimes E_y \end{bmatrix} \tag{6.23n}$$

$$\hat{H} = \begin{bmatrix} \hat{H}_y & 0_{n_y(N-1)\times n_u N+2n_x(N-1)} \end{bmatrix} \tag{6.23o}$$

$$\hat{h} = \big(\hat{u}_{1,\max}, \ldots, \hat{u}_{N,\max}, -\hat{u}_{1,\min}, \ldots, -\hat{u}_{N,\min},\, 0_{2n_x(N-1)},\, \hat{\varepsilon}_1, \ldots, \hat{\varepsilon}_{N-1}\big) \tag{6.23p}$$

We adopt the proportional-integral projected gradient (PIPG) algorithm [196] with extrapolation (shown in Algorithm 3) to solve (6.22). The input to the algorithm consists of maximum iterations $j_{\max}$, the extrapolation parameter ρ, the step-size α, which depends on the singular values of $\hat{P}$ and $\hat{H}$, and the step-size ratio ω, which is chosen heuristically to balance the convergence rates of the primal and dual iterates.

Algorithm 3 can be customized to exploits the sparsity structure of the parameters (6.23)

Algorithm 3 PIPG

Inputs: $j_{\max}$, ρ, α, ω

Initialize: ζ^1, η^1, χ^1

$\beta \leftarrow \omega\alpha$
for $j = 1, \ldots, j_{\max}$ **do**
  $z^{j+1} \leftarrow \zeta^j - \alpha(\hat{P}\zeta^j + \hat{p} + \hat{G}^\top \eta^j + \hat{H}^\top \chi^j)$
  $\tilde{E}_{\mathrm{i}} \hat{x}_1^{j+1} \leftarrow \hat{z}_{\mathrm{i}}$
  $\tilde{E}_{\mathrm{f}} \hat{x}_K^{j+1} \leftarrow \hat{z}_{\mathrm{f}}$
  $\hat{u}_k^{j+1} \leftarrow \max\{\hat{u}_{k,\min}, \min\{\hat{u}_{k,\max}, \hat{u}_k^{j+1}\}\}$ $k = 1, \ldots, N$
  $(\mu_k^+)^{j+1} \leftarrow \max\{0, (\mu_k^+)^{j+1}\}$ $k = 1, \ldots, N-1$
  $(\mu_k^-)^{j+1} \leftarrow \max\{0, (\mu_k^-)^{j+1}\}$ $k = 1, \ldots, N-1$
  $w^{j+1} \leftarrow \eta^j + \beta(\hat{G}(2z^{j+1} - \zeta^j) - \hat{g})$
  $v^{j+1} \leftarrow \max\{0, \chi^j + \beta(\hat{H}(2z^{j+1} - \zeta^j) - \hat{h})\}$
  $\zeta^{j+1} \leftarrow (1-\rho)\zeta^j + \rho z^{j+1}$
  $\eta^{j+1} \leftarrow (1-\rho)\eta^j + \rho w^{j+1}$
  $\chi^{j+1} \leftarrow (1-\rho)\chi^j + \rho v^{j+1}$
end for

Return: z^j, w^j, v^j

of (6.22). The recursive calls to PIPG within SCP are warm-started with the primal-dual solution from the previous subproblem to accelerate its convergence. See [79, 99, 196] for further details on custom, real-time implementations of PIPG.

6.3 6-DoF powered descent guidance

The 6-DoF rocket model is based on [181], with the inclusion of an independent torque control input by means of reaction control thrusters (as shown in [99]), and with the aerodynamics forces and moments ignored.

The system dynamics (6.1) and constraint function (6.2) are given by:

$$\xi = (m, \boldsymbol{r}_{\mathcal{I}}, \boldsymbol{v}_{\mathcal{I}}, \boldsymbol{q}_{\mathcal{I}\leftarrow\mathcal{B}}, \boldsymbol{\omega}_{\mathcal{B}}) \tag{6.24a}$$

$$\zeta = (\boldsymbol{T}_{\mathcal{B}}, \boldsymbol{\gamma}_{\mathcal{B}}) \tag{6.24b}$$

$$F(\xi,\zeta) = \begin{bmatrix} -\alpha_{\dot{m}}\|\boldsymbol{T}_{\mathcal{B}}\|_2 \\ \boldsymbol{v}_{\mathcal{I}} \\ \frac{1}{m}C_{\mathcal{I}\leftarrow\mathcal{B}}\boldsymbol{T}_{\mathcal{B}} + \boldsymbol{g}_{\mathcal{I}} \\ \frac{1}{2}\Omega(\boldsymbol{\omega}_{\mathcal{B}})\boldsymbol{q}_{\mathcal{I}\leftarrow\mathcal{B}} \\ J_{\mathcal{B}}^{-1}\left(\boldsymbol{r}_{T,\mathcal{B}}\times\boldsymbol{T}_{\mathcal{B}} - \boldsymbol{\omega}_{\mathcal{B}}\times J_{\mathcal{B}}\boldsymbol{\omega}_{\mathcal{B}} + \boldsymbol{\gamma}_{\mathcal{B}}\right) \end{bmatrix} \tag{6.24c}$$

$$g(\xi,\zeta) = \begin{bmatrix} m_{\text{dry}} - m \\ -\boldsymbol{e}_1^\top\boldsymbol{r}_{\mathcal{I}} \\ \|\boldsymbol{v}_{\mathcal{I}}\|_2^2 - v_{\max}^2 \\ 4\|H_\theta\boldsymbol{q}_{\mathcal{I}\leftarrow\mathcal{B}}\|_2^2 + (1-\cos\theta_{\max})^2 \\ \|\boldsymbol{\omega}_{\mathcal{B}}\|_2^2 - \omega_{\max}^2 \\ \|\boldsymbol{T}_{\mathcal{B}}\|_2 - \boldsymbol{e}_1^\top\boldsymbol{T}_{\mathcal{B}}\sec\delta_{\max} \\ \|\boldsymbol{T}_{\mathcal{B}}\|_2 - T_{\max} \\ -\|\boldsymbol{T}_{\mathcal{B}}\|_2 + T_{\min} \\ \|\boldsymbol{\gamma}_{\mathcal{B}}\|_2^2 - \gamma_{\max}^2 \end{bmatrix} \tag{6.24d}$$

The numerical values for the parameters within the system dynamics ($\alpha_{\dot{m}}, \boldsymbol{g}_{\mathcal{I}}, J_{\mathcal{B}}, \boldsymbol{r}_{T,\mathcal{B}}, H_\theta$),

boundary conditions $(z_\mathrm{i}, z_\mathrm{f})$, and constraint bounds $(m_\mathrm{dry}, \gamma_\mathrm{gs}, v_\mathrm{max}, \theta_\mathrm{max}, T_\mathrm{max}, T_\mathrm{min})$ correspond to the baseline problem in [180]. Note that the non-differentiability of the constraint functions for thrust magnitude bounds at $\boldsymbol{T}_\mathcal{B} = 0_3$ does not affect the linearization operations within SCP in practice with an appropriate initial guess for the thrust vector.

6.4 GPU architecture and code implementation

6.4.1 Overview

Loosely speaking, CPUs are designed to execute code serially. Consider the problem of adding two vectors. A CPU is designed to add these two vectors by looping through the elements of each vector and adding the corresponding elements. In contrast, GPUs have many specialized hardware units that allow them to perform many floating point operations in parallel. A GPU would add two vectors together by adding the corresponding elements in parallel by assigning each scalar addition to a different compute thread.

In GPU programming, a single sequence of instructions is called a *thread*. Back to the example of adding two vectors on a GPU, a single thread would execute a single scalar addition. Threads in turn are organized into a *grid* of *thread-blocks*. In the vector addition example, we would want to have the same number of threads as the number of elements in each vector. To write programs for the GPU, NVIDIA provides the compute unified device architecture (CUDA) platform, which is an extension of the C programming language.

CUDA enables the programmer to write special GPU functions called *kernels*, which, when called, are executed a specified number of times in parallel. When the programmer writes a kernel, they are specifying how a single thread should execute the given code; when calling a kernel, the programmer specifies the number of threads per thread-block and the number of thread-blocks in the grid. The product of these two numbers is the number of threads being called. Each thread of execution has a unique identifier based on its thread-block index within the grid and its thread index within the thread-block. The thread ID in a kernel is an integer in $[0, N)$, where N is the number of threads called. However, before performing

computations on the GPU, we must first ensure that the data the kernel is operating on is in GPU memory rather than in CPU memory. This typically requires the programmer to allocate memory on the GPU to store this data, using the `cudaMalloc()` function, and to then copy the data from the CPU to the GPU using the `cudaMemcpy()` function.

When the kernel is called, the thread-blocks are then distributed to *streaming multi-processors* (SM), which contain a collection of CUDA cores. These CUDA cores execute the threads within a thread-block. When a SM finishes executing all threads on a thread-block, the GPU allocates it another thread-block.

In all of this, the programmer only has to worry about creating a grid of thread-blocks when they call the kernel. The GPU is responsible for allocating thread-blocks to SMs, which are in turn are responsible for allocating threads to CUDA cores. The thread hierarchy is summarized by Figure 6.1.

As an example, consider the code in Listing 6.1. The first function is how one would write a function to add two vectors on a CPU. The second function is how one would write a CUDA kernel to add two vectors in parallel. When launching this kernel, each GPU thread executes the code within the kernel in parallel. The first line in this kernel computes the unique thread ID for the thread; subsequently this thread will perform one scalar addition for the index that matches its thread ID. The `main()` function shows how to call the serial version of this vector addition and how to call the CUDA kernel. We must first allocate the memory for arrays a, b, and c on the GPU and then copy the data in the arrays from the CPU to the GPU. Finally, the kernel can be called with the desired grid size and thread-block size.

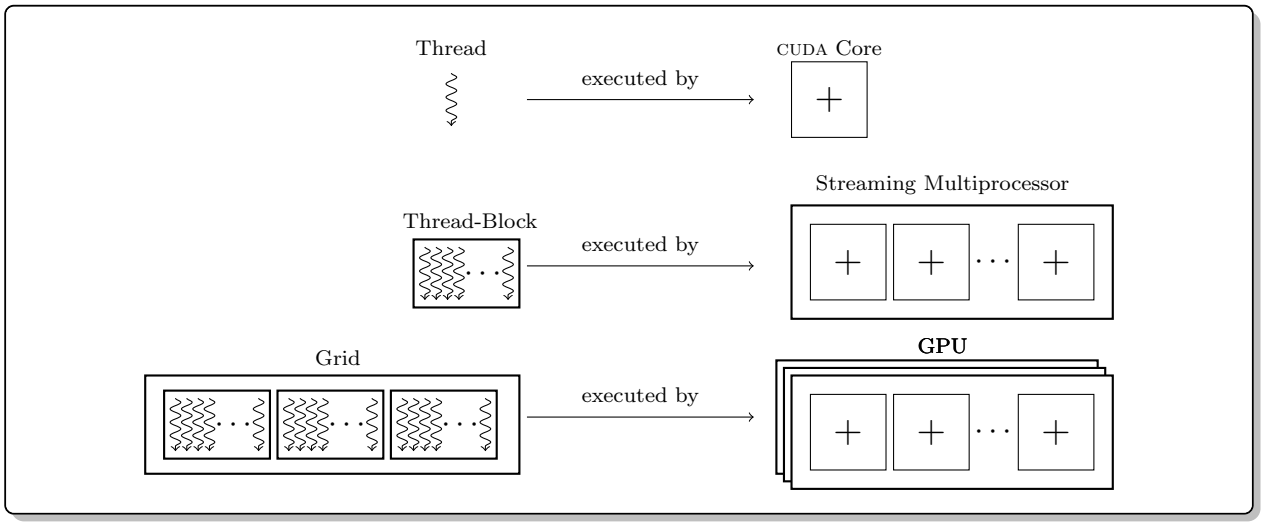


Figure 6.1: Thread hierarchy in the GPU

```
/* Performs the vector addition c = a + b serially, where all vectors have
    N elements. */
void add_vector_serial(int* a, int* b, int* c, int N)
{
    for(int i = 0; i < N; i++)
    {
        c[i] = a[i] + b[i];
    }
}

/* CUDA kernel to perform c = a + b, where all vectors have N elements. */
__global__ void add_vector_parallel(int* a, int* b, int* c, int N)
{
    /* Get the ID of the thread executing this function. */
    int thread_id = blockIdx.x*blockDim.x + threadIdx.x;

    /* Safeguard if the kernel is launched with more threads than the
   number of elements. */
    if (i < N)
    {
        c[i] = a[i] + b[i];
    }
}

int main()
{
    /* Declare pointers for arrays in CPU memory. */
    int* a[N];
    int* b[N];
    int* c[N];

    /* Helper function to fill a and b with data. */
    initialize(a, b);

    /* Declare pointers for arrays in GPU memory. */
    int* d_a, d_b, d_c;

    /* Allocate memory for arrays in GPU memory. */
    cudaMalloc((void**) &d_a, N * sizeof(int));
    cudaMalloc((void**) &d_b, N * sizeof(int)));
    cudaMalloc((void**) &d_c, N * sizeof(int)));

    /* Copy data in array from CPU to GPU memory. */
    cudaMemcpy(d_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_c, c, N * sizeof(int), cudaMemcpyHostToDevice);

    /* Call serial addition function on CPU. */
    add_vector_serial(a, b, c, N);

    /* Call parallel addition kernel on GPU. */
    add_vector_parallel<<<grid_size, block_size>>>(d_a, d_b, d_c, N);
}
```

Listing 6.1: Code for serial (CPU) and parallelized (GPU) vector addition

6.4.2 Code Structure

The data transfer bandwidth between the CPU and the GPU is much slower than that between the CUDA cores and the GPU memory. Thus to achieve peak performance, it is important to minimize data transfer between the CPU and GPU. Consequently, we chose the following execution model: create an array of problem instances with varying initial conditions on the CPU, allocate the necessary memory on the GPU to store these problem instances, copy the problem instance data to the GPU, launch a kernel that solves all of these trajectory optimization problem instances on the GPU, and finally copy back the trajectories from the GPU to the CPU. In this paradigm, we are assigning each thread one trajectory to compute. Each thread is able to access the problem instance it needs to solve by indexing into the array of problem data using its unique thread ID. Figure 6.2 shows the flow of data in our implementation.

By taking this approach we only have to incur the costly overhead of CPU-GPU communication once when sending the problem instances to the GPU and once when copying the computed trajectories back from the GPU.

A key difficulty with implementing this procedure on the GPU is the need to implement all operations from scratch in CUDA, since we are unable to use off-the-shelf linear algebra routines and convex optimization solvers. Therefore, all code necessary to solve for a trajectory was implemented in CUDA, including Level 1, 2, and 3 BLAS operations and the $\text{PIPG}_{\text{custom}}$ algorithm.

6.5 Numerical results

This section presents and analyzes Monte Carlo results for the 6-DoF powered-descent guidance problem outlined in Section 6.3 using the solution procedure in Section 6.1. We then compare the computation time between a serial CPU implementation and parallel GPU implementation for Monte Carlo simulations of various sizes. All results were obtained on a machine with an AMD Ryzen 9 7950X3D CPU and an NVIDIA RTX 3090 GPU.

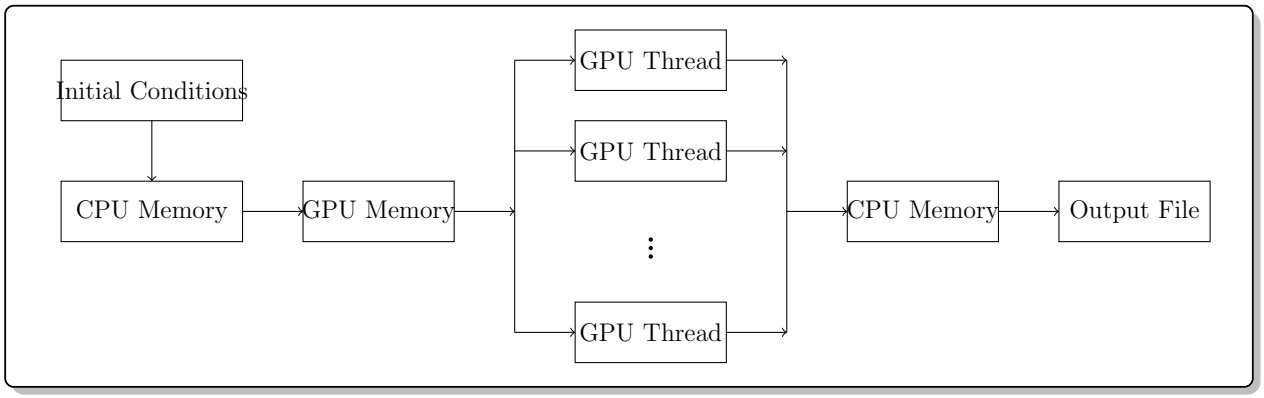


Figure 6.2: Data flow between the CPU and the GPU

When conducting the Monte Carlo, we disperse the initial position according to the following uniform distribution:

$$\boldsymbol{r}_{\mathcal{I}}(0) \sim \begin{bmatrix} \mathcal{U}(6,9) \\ \mathcal{U}(3,6) \\ \mathcal{U}(1,2) \end{bmatrix} \tag{6.25}$$

For all simulations, we use the same weights, w_{cost}, w_{tr}, and w_{vc}, defined in Equation 6.21a, we set the maximum number of SCP iterations to 25, and we use 2500 PIPG iterations with the same hyperparameters ω and ρ for each subproblem solve. We find that SCP converges within 25 iterations for all 256 of these trajectories, which is the first demonstration of robustness to variations in initial conditions for SCP with PIPG as a subproblem solver. Figures 6.3 and 6.4 show the trajectories obtained. An extensive Monte Carlo analysis of the 6-DoF PDG problem is provided in [125].

Figure 6.5 depicts timing results for serial CPU and parallel GPU Monte Carlos for varying Monte Carlo batch sizes and for various thread-block sizes. For the GPU simulations, only the solve-time is reported since execution times for `malloc` and `memcpy` were an order of magnitude smaller than the solve-time.

As mentioned in Section 6.4, calling a kernel requires two parameters: thread-block size

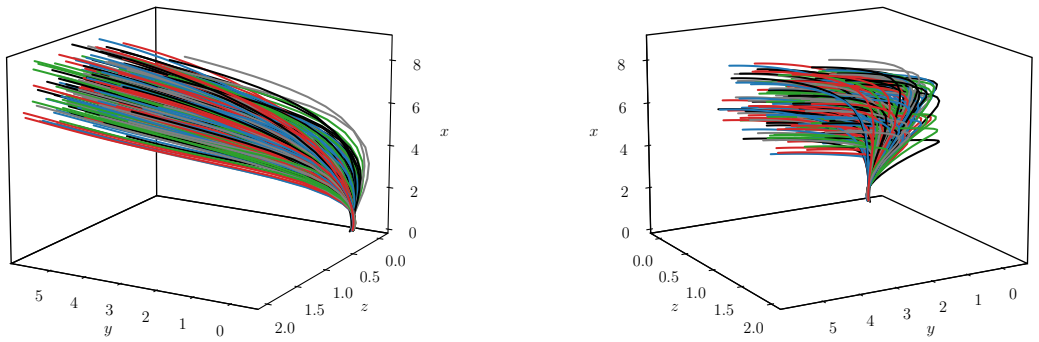
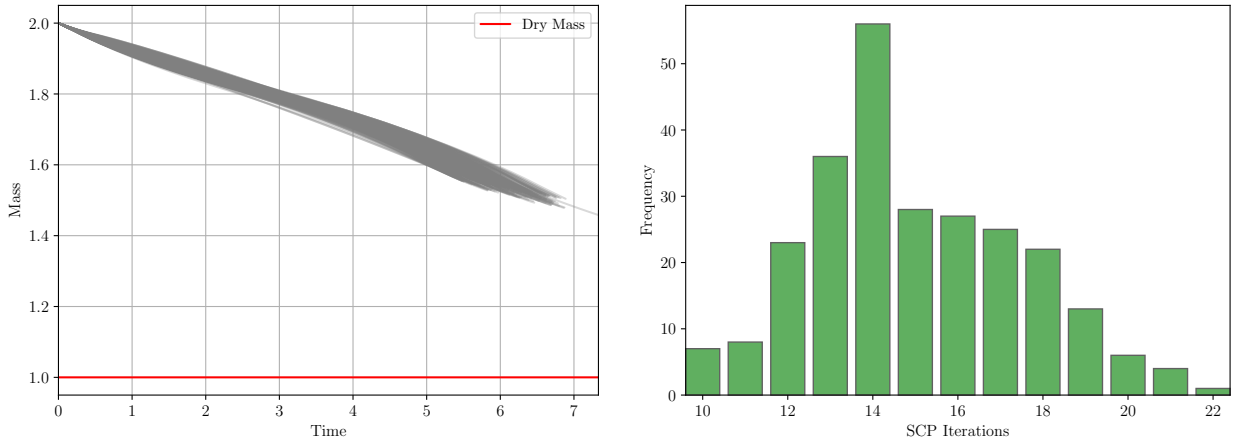


Figure 6.3: Trajectories generated from 256 Monte Carlo runs

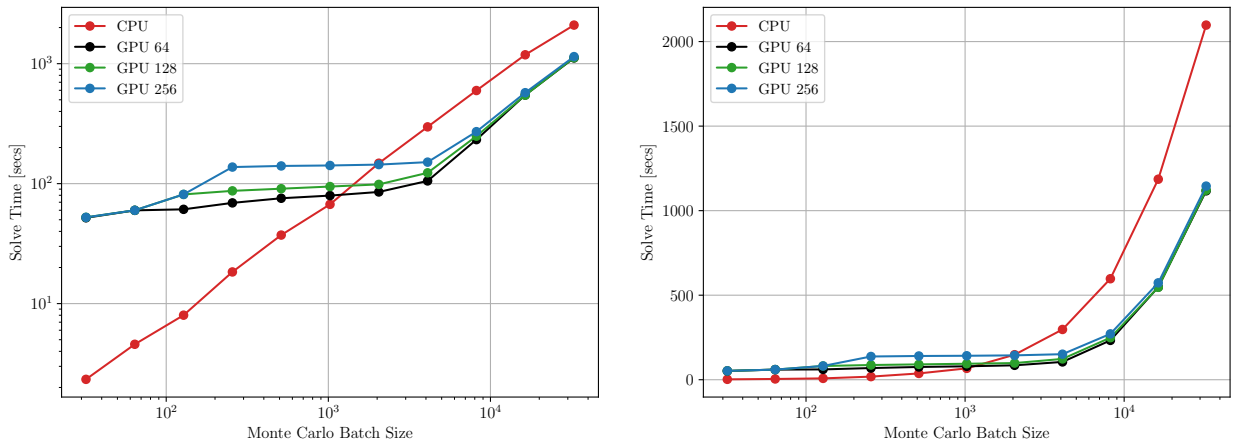


(a) Mass Depletion

(b) Distribution of SCP Iterations

Figure 6.4: Statistics from 256 Monte Carlo runs

and grid size, which is the number of thread-blocks in the grid. The product of these two parameters dictates the number of threads launched. This gives users a choice in choosing the thread-block and grid size when launching a kernel. For example, if we wanted to perform a Monte Carlo with a batch size of 512, we could choose a thread-block size of 128 and grid size of four or a thread-block size of 256 and grid size of two. Here, we perform Monte Carlo



(a) Log-log timing plot

(b) Semi-log timing plot

Figure 6.5: CPU and GPU Monte Carlo solve-times. GPU results are plotted for the indicated thread-block sizes.

runs on the GPU with maximum thread-block sizes of 64, 128, and 256. For batch sizes below the maximum thread-block size, we set the thread-block size to be the batch size, and the grid size to one. For example, if we had a maximum thread-block size of 64 and were conducting a Monte Carlo run with a batch size of four, we would set the thread-block size to 4 and grid size to one. On the other hand, if we were conducting a Monte Carlo run with a batch size of 128, we would set the thread-block size to 64 and the grid size to two.

From Figure 6.5, it is evident that the Monte Carlo run on the GPU is slower than that on the CPU for small batch sizes, and faster for larger batch sizes, with a crossover point roughly between 1024 and 2048 Monte Carlo runs. Since CPUs, unlike GPUs, are optimized for serial tasks such as SCP, it is unsurprising that for smaller runs, the CPU is faster. However, as the batch size increases, the GPU is able to leverage parallelism to provide acceleration. In other words, although the GPU takes longer than the CPU to solve a single SCP problem, it is able to accelerate the Monte Carlo run when given a large number of problems to solve.

The log-log timing plot for the CPU implementation, shown in Figure 6.5a, is linear with

a constant slope, as expected, since doubling the number of runs will double the solve-time for a serial implementation. However, the timing plots for the GPU exhibit three phases as the batch size increases. Initially, the solve-time increases until the maximum block size is reached, then the solve-time stays relatively constant, and finally, it increases again. In the first phase, since all threads are in a single thread-block, all the computations for the Monte Carlo are being performed by a single streaming multiprocessor. After the number of runs exceeds the maximum size of the thread-block, a second thread-block is populated, which allows a second streaming multiprocessor to compute these trajectories in parallel with the first thread-block. This results in the second phase, wherein the solve-time curve is flat. The number of runs after which the curve flattens out is dictated by the maximum thread-block size. The third phase, where increasing the number of runs increases the solve-time again, is likely due to the GPU not being able to allocate any more streaming multiprocessors to the thread-blocks, forcing them to execute thread-blocks serially.

Chapter 7

CONSTRAINT PRECONDITIONING

Solving optimal control problems in real-time is at the heart of model predictive control (MPC). This control scheme is optimization-based, where we solve a fixed time horizon optimal control problem recursively to generate state and control trajectories which satisfy relevant system constraints while minimizing a predefined cost function.

In this chapter, we consider optimal control problems in the form

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & \frac{1}{2} z^\top P z + q^\top z && \text{(7.1a)} \\ \text{subject to} \quad & Hz - g = 0 && \text{(7.1b)} \\ & z \in \mathbb{D} && \text{(7.1c)} \end{aligned}$$

where $H \in \mathbb{R}^{m \times n}$, $m \leq n$, and $P \succ 0$. Constraint 7.1b contains the linear dynamics and set $\mathbb{D}$ contains state and control constraints. We assume that set $\mathbb{D}$ is the Cartesian product of sets that have closed-form projections, such as balls, boxes, second-order cones, half-spaces, and the intersection of a ball and second-order cone [15, 17]. This assumption applies to many typical state and control constraints in MPC [98].

First-order optimization algorithms are popular for solving MPC problems due to their ability to warm-start from any prescribed solution (e.g., one that can be obtained from the previous solution in a recursive MPC scheme), which greatly reduces solve-times and per-iteration computational cost. Unlike second-order methods, first-order methods do not require expensive matrix factorizations, making them attractive for resource constrained embedded systems [13].

However, ill-conditioned optimization problems degrade the performance of methods in this class. Consequently, first-order optimization algorithms use preconditioning and parameter selection to improve their convergence behavior. Preconditioning transforms the given optimization problem into an equivalent problem which is able to be solved in fewer iterations. This is achieved by performing a change of variables or transforming the constraints of the problem to reduce the condition number of a matrix related to the optimization problem, such as the KKT matrix. Whereas preconditioning modifies the given optimization problem, parameter selection modifies the optimization algorithm itself. Parameter selection involves choosing parameters of an optimization algorithm such as step-size or relaxation parameters for fastest convergence [85].

Modified Ruiz equilibriation [177] and the hypersphere preconditioner [99] are two existing preconditioning techniques for first-order primal-dual algorithms. These preconditioners aim to reduce the condition number of the KKT matrix. Both perform a change of variables and additionally, modified Ruiz equilibriation scales the constraints with a diagonal matrix. These two preconditioner transform set $\mathbb{D}$ in Problem 7.1 since they both employ a change of variables. This requires special structure in the matrices defining the change of variables in order to preserve closed-form projections onto $\mathbb{D}$, which is needed to solve Problem 7.1 by using Proportional-Integral Projected Gradient (PIPG), a first-order primal-dual optimization algorithm [195, 197, 196].

Our first contribution is the development of the *QR preconditioner*. This proposed preconditioner does not perform a change of variables, meaning set $\mathbb{D}$ is not transformed and consequently closed-form projections are preserved. Instead, the QR preconditioner transforms the equality constraints by scaling them with a triangular matrix to minimize the condition number of the KKT matrix. Since a triangular matrix is used for transforming the constraints as opposed to a diagonal matrix, the QR preconditioner has more degrees of freedom to transform the constraints than modified Ruiz equilibriation. The QR preconditioner requires a matrix factorization to perform this constraint transformation, unlike the aforementioned preconditioners, making it more computationally expensive. However,

in our numerical examples the factorization time was a small percentage of the overall solve-time. Additionally, in settings requiring the solution of multiple optimization problems with identical equality constraints, such as quasi-convex optimization and MPC, performing the preconditioning offline and solving the preconditioned problem online will result in improved real-time performance.

Our second contribution is a step-size selection heuristic for PIPG. For fastest convergence, PIPG requires carefully selecting step sizes, a process that is currently manually tuned. We eliminate manual tuning of step-sizes by introducing a step-size selection heuristic based on choosing step-sizes to minimize the upper bound on the primal-dual gap.

7.0.1 Related work

In this section we will discuss two existing preconditioning methods.

The hypersphere preconditioner [99] performs a change of variables to make the Hessian of the objective function some positive multiple of the identity matrix, making the objective function perfectly conditioned. This preconditioner avoids transforming the constraints. It is applicable to PIPG when the Hessian of the objective function is diagonal or has an easy to compute Cholesky factorization, and the change of variables induced by the preconditioner preserves the ability to project onto set $\mathbb{D}$ in closed-form.

Modified Ruiz equilibration [177] preconditions the objective function and transforms the constraints by performing a linear change of variables with a diagonal matrix and scaling the constraints with a diagonal matrix to make the preconditioned KKT matrix have equal row and column norms, a heuristic for good conditioning. Similar to the hypersphere preconditioner, to be applicable to PIPG the change of variables induced must preserve the ability to project onto set $\mathbb{D}$ in closed-form.

7.0.2 Notation

We denote the $n \times 1$ vector of zeros as 0_n, the $m \times n$ matrix of zeros as $0_{m\times n}$, the $n \times n$ identity matrix as I_n, the Euclidean projection of point z onto the convex set $\mathbb{D}$ as $\Pi_{\mathbb{D}}[z]$,

the concatenation of vectors u and v as (u, v), the vector formed from the i^{th} through the j^{th} component of z as $z^{i:j}$, and the Euclidean norm of a vector z as $\|z\| := \sqrt{z^\top z}$.

7.1 Proportional-integral projected gradient

Proportional-Integral Projected Gradient (PIPG), shown in Algorithm 4, is a first-order primal-dual conic optimization algorithm capable of infeasibility detection. This algorithm is specialized to handle common constraints sets which arise in optimal control problems, such as boxes, balls, and second-order cones, via closed-form projections [195, 196, 197]. PIPG has been demonstrated to be faster than many state-of-the art optimization solvers such as ECOS, MOSEK, GUROBI, and OSQP [196]. It has also been extensively applied to trajectory optimization problems including three-degree-of-freedom (DoF) powered-descent guidance and has been used as a subproblem solver within sequential convex programming for multi-phase powered-descent guidance, 6-DoF powered-descent-guidance, and spacecraft rendezvous problems [47, 50, 79, 99, 101].

Algorithm 4 PIPG

Require: $k_{\max}, \alpha, \beta, z^1 \in \mathbb{D}, v^1$.
Ensure: z^k.
1: **for** $k = 1, 2, \ldots, k_{\max} - 1$ **do**
2: $\quad w^{k+1} = v^k + \beta(Hz^k - g)$
3: $\quad z^{k+1} = \Pi_{\mathbb{D}}[z^k - \alpha(Pz^k + q + H^\top w^{k+1})]$
4: $\quad v^{k+1} = w^{k+1} + \beta H(z^{k+1} - z^k)$
5: **end for**

In Algorithm 4, z^k and w^k are the iterates of the primal and dual variables respectively. Thus, we refer to α and β as the primal and dual step-sizes. For the convergence results to hold for PIPG, the primal and dual step-sizes must satisfy the algebraic relationship

$$\alpha(\lambda_{\max} + \sigma\beta) < 1.$$

We parameterize this family of step-sizes with $\gamma > 0$ as

$$\alpha < \frac{1}{\lambda_{\max} + \gamma}, \quad \beta = \frac{\gamma}{\sigma} \tag{7.2}$$

where $\lambda_{\max}$ is the maximum eigenvalue of P and σ is the maximum eigenvalue of $H^\top H$. In practice we choose α to be equal to the upper bound and do not observe convergence issues. We will discuss how to choose γ in Section 7.3.

7.2 QR preconditioner

In this section, we derive and provide a geometric interpretation for the QR preconditioner.

7.2.1 Derivation

First, we derive a preconditioner that only modifies Constraint 7.1b of Problem 7.1 and minimizes the condition number of the KKT matrix. First, we need the following assumptions on the constraint matrix H and the Hessian of the objective function P.

Assumption 1. *$H \in \mathbb{R}^{m \times n}$ has full row rank.*

Assumption 2. *P is positive definite and has maximum and minimum eigenvalues $\lambda_{\max}$ and $\lambda_{\min}$ respectively.*

The KKT conditions of Problem 7.1 are as follows

$$\underbrace{\begin{bmatrix} P & H^\top \\ H & 0_{m \times m} \end{bmatrix}}_{\mathcal{K}} \begin{bmatrix} z^\star \\ w^\star \end{bmatrix} + \begin{bmatrix} q \\ -g \end{bmatrix} \in \begin{bmatrix} -\mathbb{N}_{\mathbb{D}}(z^\star) \\ \{0_m\} \end{bmatrix} \tag{7.3}$$

where $z^\star$ and $w^\star$ are the optimal primal and dual solutions respectively and $\mathbb{N}_{\mathbb{D}}(z^\star)$ is the normal cone of set $\mathbb{D}$ at $z^\star$. We denote the KKT matrix as $\mathcal{K}$.

If Assumptions 1 and 2 hold and $\sigma_{\max}$ and $\sigma_{\min}$ are the maximum and minimum singular values of H, the eigenvalues of $\mathcal{K}$ lie in the following set [22]

$$\lambda(\mathcal{K}) \subset \mathcal{I}^- \cup \mathcal{I}^+$$

where

$$\mathcal{I}^- = \left[\frac{1}{2}\left(\lambda_{\min} - \sqrt{\lambda_{\min}^2 + 4\sigma_{\max}^2}\right), \frac{1}{2}\left(\lambda_{\max} - \sqrt{\lambda_{\max}^2 + 4\sigma_{\min}^2}\right)\right]$$

$$\mathcal{I}^+ = \left[\lambda_{\min}, \frac{1}{2}\left(\lambda_{\max} + \sqrt{\lambda_{\max}^2 + 4\sigma_{\max}^2}\right)\right].$$

The spectral condition number of $\mathcal{K}$ is defined as [22]

$$\kappa(\mathcal{K}) = \frac{\max|\lambda(\mathcal{K})|}{\min|\lambda(\mathcal{K})|}.$$

The condition number quantifies the difficulty in terms of the number of iterations to solve Equation 7.3 with an iterative method. Thus, a smaller condition number for $\mathcal{K}$ results in faster convergence for PIPG which solves the KKT conditions in Equation 7.3.

The upper bound for the $\mathcal{I}^+$ interval is larger in magnitude than the lower bound for the $\mathcal{I}^-$ interval since $\lambda_{\max} \geq \lambda_{\min} > 0$ and $\sigma_{\max} > 0$ by Assumptions 1 and 2. We can construct an upper bound on the condition number of $\mathcal{K}$ as follows

$$\kappa(\mathcal{K}) \leq \max\left(\frac{\lambda_{\max} + \sqrt{\lambda_{\max}^2 + 4\sigma_{\max}^2}}{\sqrt{\lambda_{\max}^2 + 4\sigma_{\min}^2} - \lambda_{\max}}, \frac{\lambda_{\max} + \sqrt{\lambda_{\max}^2 + 4\sigma_{\max}^2}}{2\lambda_{\min}}\right). \tag{7.4}$$

Lemma 14. *The singular values for H which minimize the condition number of the KKT matrix are $\sigma_{\min} = \sigma_{\max} = \sqrt{\lambda_{\max}\lambda_{\min} + \lambda_{\min}^2}$*

Proof. We will first prove by contradiction that $\sigma_{\max}$ and $\sigma_{\min}$ which minimize the bound on the condition number given by Equation 7.4 must be equal.

Suppose that we have $\sigma_{\min} < \sigma_{\max}$ which minimize the bound on the condition number. Both arguments of the max operator are monotonically increasing in $\sigma_{\max}$, so we can reduce $\sigma_{\max}$ and further reduce the bound on the condition number which produced a contradiction.

Since $\sigma_{\min} = \sigma_{\max}$ at optimality, the first argument of the max operator and its derivative are

$$h(\sigma_{\max}) = \frac{\lambda_{\max} + \sqrt{\lambda_{\max}^2 + 4\sigma_{\max}^2}}{\sqrt{\lambda_{\max}^2 + 4\sigma_{\max}^2} - \lambda_{\max}}$$

$$h'(\sigma_{\max}) = \frac{-8\lambda_{\max}\sigma_{\max}\sqrt{\lambda_{\max}^2 + 4\sigma_{\max}^2}}{\left(\sqrt{\lambda_{\max}^2 + 4\sigma_{\max}^2} - \lambda_{\max}\right)^2}.$$

Since $\lambda_{\max} > 0$ and $\sigma_{\max} > 0$, $h'(\sigma_{\max})$ is negative and thus $h(\sigma_{\max})$ is monotonically decreasing in $\sigma_{\max}$.

It is also clear that the second argument of the max operator in Equation 7.4 is monotonically increasing in $\sigma_{\max}$. To minimize the maximum of a monotonically decreasing and monotonically increasing function, we must set the two arguments equal. By setting the arguments equal we obtain the minimum and maximum singular values of H which minimize the KKT condition number

$$\sigma_{\max} = \sigma_{\min} = \sqrt{\lambda_{\max}\lambda_{\min} + \lambda_{\min}^2}.$$

□

If $HH^\top = (\lambda_{\min}\lambda_{\max} + \lambda_{\min}^2)I_m$, then the singular values of H are $\sqrt{\lambda_{\max}\lambda_{\min} + \lambda_{\min}^2}$ which minimize the condition number of the KKT matrix by Lemma 14. For $HH^\top$ to be some scaling of the identity matrix, it is sufficient for the rows of H to be orthogonal. To do this, we will apply a QR factorization to $H^\top$, and transform Constraint 7.1b into Constraint 7.5e as follows

$$Hz - g = 0 \tag{7.5a}$$
$$\Rightarrow R^\top Q^\top z - g = 0 \tag{7.5b}$$
$$\Rightarrow Q^\top z - R^{-\top} g = 0 \tag{7.5c}$$
$$\Rightarrow \eta(Q^\top z - R^{-\top} g) = 0 \tag{7.5d}$$
$$\Rightarrow \hat{H} z - \hat{g} = 0 \tag{7.5e}$$

where $H^\top = QR$ is the thin or economy QR factorization of $H^\top$, $\hat{H} = \eta Q^\top$, $\hat{g} = \eta R^{-\top} g$, and Assumption 1 ensures invertibility of $R^\top$ [89]. Note that Assumption 1 is equivalent to requiring that linear independence constraint qualification (LICQ) of Constraint 7.1b holds for Problem 7.1 [143]. We can arbitrarily scale the constraint in Equation 7.5d by some $\eta > 0$ and the rows of $\hat{H}$ will still be orthogonal. If we choose $\eta = \sqrt{\lambda_{\max}\lambda_{\min} + \lambda_{\min}^2}$, then $\hat{H}\hat{H}^\top = (\lambda_{\min}\lambda_{\max} + \lambda_{\min}^2) I_m$, thus minimizing the condition number of the KKT matrix. The QR preconditioner is given by Algorithm 5.

Algorithm 5 QR Preconditioner

function QR_PRECONDITIONER(H, g, $\lambda_{\max}$, $\lambda_{\min}$)
  $\eta = \sqrt{\lambda_{\max}\lambda_{\min} + \lambda_{\min}^2}$
  $Q, R = \mathrm{qr}(H^\top)$ ▷ economy QR factorization
  $\hat{H} = \eta Q^\top$
  $\hat{g} = \eta R^{-\top} g$
  return $\hat{H}$, $\hat{g}$
end function

7.2.2 Geometric Interpretation

To interpret this preconditioner geometrically, we will consider the simplified problem of using PIPG to minimize a bivariate quadratic function subject to two equality constraints with nearly parallel normal vectors. Figures 7.1 and 7.2 depict the level curves of the objective

function and the PIPG iterates before and after the QR preconditioner is applied.

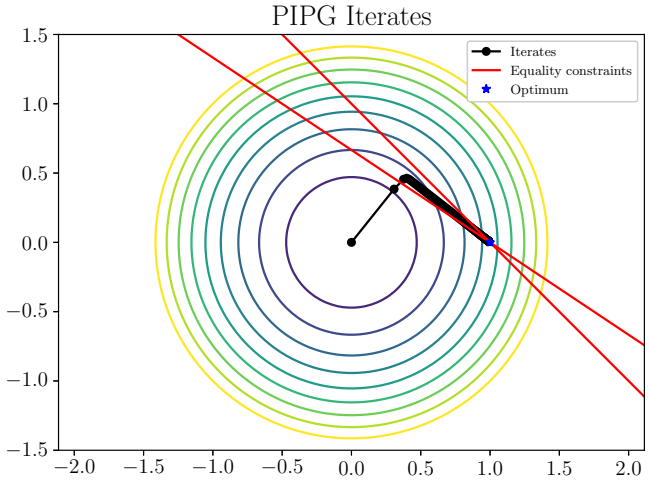


Figure 7.1: PIPG iterates with nearly parallel equality constraints

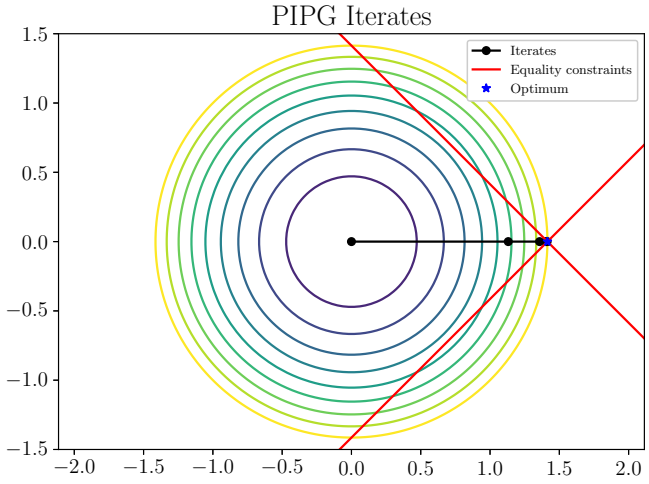


Figure 7.2: PIPG iterates after applying QR preconditioner

We can see that in Figure 7.1, PIPG takes several iterations to converge. Notice that Line 3 of PIPG takes a step in the negative gradient direction of the objective function in order

to minimize the objective, then takes a step in a direction given by a linear combination of the rows of H (the $H^\top w^{k+1}$ term) to satisfy Constraint 7.1b. The rows of H are the normal vectors of the hyperplanes which define the equality constraints in Problem 7.1. If we have two equality constraints with nearly parallel hyperplane normals (as in Figure 7.1), we require smaller coefficients in w^{k+1} for the linear combination of the hyperplane normals, given by $H^\top w^{k+1}$, to move the iterates in a direction nearly parallel to hyperplane normals (up and to the right in Figure 7.1), but require large coefficients for the iterates to move in a direction nearly orthogonal to the hyperplane normals (down and to the right in Figure 7.1). The coefficients of this linear combination are given by our current iterate of the dual variables w^{k+1}. To get large enough w^{k+1} for the iterates to move nearly orthogonal to the hyperplane normals, we must wait for the integrator given by Line 2 to accumulate enough error, resulting in slow convergence.

The QR preconditioner transforms the equality constraints given by Equation 7.1b to a equivalent set of equality constraints with the same feasible set as the original constraints, but one whose hyperplane normal vectors are all orthogonal and of the same length, making it easy for iterates to move in all directions. After applying the QR preconditioner, we can see the resulting iterates in Figure 7.2 where the optimum is achieved in far fewer iterations.

7.3 Parameter selection

In this section we will discuss how to select the step-sizes α and β to minimize the upper bound on the primal-dual gap for the iterates of PIPG.

For convergence of PIPG, the primal and dual step-sizes must satisfy the relationship

$$\alpha(\lambda_{\max} + \sigma\beta) < 1 \tag{7.6}$$

where α and β are the primal and dual step-sizes respectively, $\lambda_{\max}$ is the maximum eigenvalue of P, and σ is the maximum eigenvalue of $H^\top H$.

There are infinite choices for α and β which satisfy the above relationship, and it has

been observed that for fastest convergence a "proper" choice for α and β must be made. From Proposition 1 of [197], we see that zero primal-dual gap is a sufficient condition for optimality of Problem 7.1. Thus, we propose to choose step-sizes which minimize the upper bound on the primal-dual gap.

In [196], the authors introduce a positive, scalar parameter, referred to as ω, to select an α and β which satisfy the relationship given by Equation 7.6. We introduce a different scalar parameter γ, to choose an α and β using Equation 7.2. Using γ instead of the previously introduced ω makes the analysis in this section tractable.

Theorem 1 from [197] provides an upper bound on the primal-dual gap for Algorithm 4. We will rewrite this theorem to explicitly show dependence on γ

$$\mathcal{L}(\bar{z}^k, w^\star) - \mathcal{L}(z^\star, \bar{w}^k) \leq \frac{f(\gamma)}{k} \tag{7.7}$$

$$f(\gamma) = \frac{\lambda_{\max} + \gamma}{2}\|z^1 - z^\star\|^2 + \frac{\sigma}{2\gamma}\|v^1 - w^\star\|^2 \tag{7.8}$$

$$\bar{z}^k = \frac{1}{k}\sum_{j=1}^{k} z^{j+1}, \;\; \bar{w}^k = \frac{1}{k}\sum_{j=1}^{k} w^{j+1}$$

where $\{z^j, w^j\}_{j=1}^k$ are generated by Algorithm 4, $(z^\star, w^\star)$ is the optimal primal-dual solution of Problem 7.1, and $\mathcal{L}(z, w) = \frac{1}{2}z^\top P z + q^\top z + w^\top(Hz - g)$ is the Lagrangian of Problem 7.1. Both γ and σ are defined to be positive real numbers. Under this assumption, we see that $f(\gamma)$ given by Equation 7.8 is convex in γ and has a minimizer given by

$$\gamma^\star = \sqrt{\sigma}\frac{\|v^1 - w^\star\|}{\|z^1 - z^\star\|}.$$

Since $\gamma^\star$ is a function of the optimal primal-dual solution $(z^\star, w^\star)$, we cannot directly choose the optimal step-sizes. Instead, we suggest a heuristic where for every fixed number of iterations of PIPG, we pick the value for γ which minimizes an approximation of Equation 7.8, where the optimal primal-dual solution, $(z^\star, w^\star)$, is replaced with our current iterates of

the primal and dual variables, (z^k, w^k). The value of γ which minimizes this approximation of Equation 7.8 after the k^{th} iteration of PIPG is

$$\hat{\gamma}^\star = \sqrt{\sigma}\frac{\|v^1 - w^k\|}{\|z^1 - z^k\|}.$$

The proposed step-size selection algorithm is given by Algorithm 6. We can combine Algorithms 4, 5, and 6 into Algorithm 7 to show how our proposed algorithms should be used within PIPG.

Although choosing γ to minimize the upper bound on the primal-dual gap given by Equation 7.7 only guarantees improving the worst-case performance of PIPG, we observe improved convergence in our numerical results.

Algorithm 6 Parameter selection for PIPG

function STEP_SELECTION(z^1, v^1, z^k, w^k, $\lambda_{\max}$, σ)
    $\hat{\gamma}^\star = \sqrt{\sigma}(\|z^1 - z^k\|/\|v^1 - w^k\|)$
    $\alpha = 1/(\lambda_{\max} + \hat{\gamma}^\star)$
    $\beta = \hat{\gamma}^\star/\sigma$
    return α, β
end function

Algorithm 7 PIPG w/ QR Preconditioner and Parameter Selection

Require: $k_{\max}, k_{\text{update}}, \alpha_{\text{init}}, \beta_{\text{init}}, z^1 \in \mathbb{D}, v^1, H, g$.
Ensure: z^k.
$\hat{H}, \hat{g} = \text{QR\_PRECONDITIONER}(H, g, \lambda_{\max}, \lambda_{\min})$
$\alpha, \beta = \alpha_{\text{init}}, \beta_{\text{init}}$
for $k = 1, 2, \ldots, k_{\max} - 1$ **do**
    if $\text{mod}(k, k_{\text{update}}) == 0$ **then**
        $\alpha, \beta = \text{STEP\_SELECTION}(z^1, v^1, z^k, w^k, \lambda_{\max}, \sigma)$
    end if
    $w^{k+1} = v^k + \beta(\hat{H}z^k - \hat{g})$
    $z^{k+1} = \Pi_{\mathbb{D}}[z^k - \alpha(Pz^k + q + \hat{H}^\top w^{k+1})]$
    $v^{k+1} = w^{k+1} + \beta\hat{H}(z^{k+1} - z^k)$
end for

7.4 Numerical results

In this section, we assess the performance of Algorithms 5 and 6 on two MPC problems: regulation of an oscillating mass-spring system and quadrotor obstacle avoidance, a quadratic program (QP) and a second-order cone program (SOCP) respectively. We first define the two problems, discuss implementation details of our algorithms, and finally present the numerical results.

7.4.1 Oscillating Mass Control

The first MPC problem considers regulation of a one-dimensional mass-spring system to equilibrium by applying forces to each of the masses [190]. In this problem there are N masses connected to their neighbors by springs with the masses at the end being connected to a wall with springs.

The *state* at time-step t is given by $x_t = (r_t, v_t)$, where $r_t \in \mathbb{R}^N$ is the displacement of the masses from their respective equilibrium positions, and $v_t \in \mathbb{R}^N$ is the velocity of the masses. The *control* at time-step t, $u_t \in \mathbb{R}^N$, is the input forces acting on each of the N masses. We constrain the displacement of the masses (7.9d), the speed of the masses (7.9e), and the force applied to each mass (7.9f). We discretize the dynamics with a zero-order-hold and sample-time of Δt and can write the problem as follows

$$
\begin{aligned}
\underset{x_t, u_t}{\text{minimize}} \quad & \frac{1}{2}\left(\sum_{t=1}^{T} x_t^\top Q_t x_t + \sum_{t=1}^{T-1} u_t^\top R_t u_t\right) && && \text{(7.9a)}\\
\text{subject to} \quad & x_{t+1} = Ax_t + Bu_t && t \in [1, T-1] && \text{(7.9b)}\\
& x_1 = x_{\text{init}} && && \text{(7.9c)}\\
& \|r_t\|_\infty \leq r_{\max} && t \in [1, T] && \text{(7.9d)}\\
& \|v_t\|_\infty \leq v_{\max} && t \in [1, T] && \text{(7.9e)}\\
& \|u_t\|_\infty \leq u_{\max} && t \in [1, T-1]. && \text{(7.9f)}
\end{aligned}
$$

All the masses have unit mass, all the springs have unit spring constant, the following non-dimensional parameters are used,

$$T = 30,\ \Delta t = 0.1,\ N = 8,$$
$$r_{\max} = 0.75,\ v_{\max} = 0.75,\ u_{\max} = 0.5$$
$$Q_t = \text{blkdiag}(I_N, 5I_N),\ R_t = I_N.$$

Each component of the initial state x_{init} is drawn from the uniform distribution on the interval [-0.5, 0.5].

The dynamics given by Equation 7.9b can be written as Constraint 7.1b with appropriate choice of H and g. Constraints 7.9d, 7.9e, and 7.9f can be written as Constraint 7.1c, where $\mathbb{D}$ is the Cartesian product of boxes.

7.4.2 Quadrotor Path Planning

The second MPC problem considered is the three-degree-of-freedom quadrotor path planning problem with obstacle avoidance. The *state* at time-step t is defined as $x_t = (r_t, v_t)$, where $r_t \in \mathbb{R}^3$ is the position of the quadrotor, and $v_t \in \mathbb{R}^3$ is the velocity of the quadrotor. The *control* at time t, $u_t \in \mathbb{R}^3$, is the thrust produced by the quadrotor's propellers.

Unlike the oscillating masses regulation problem, where the goal is to regulate all the masses to their equilibrium positions, the objective in this problem is to track a reference trajectory given by $\hat{x}_t$ which linearly interpolates the initial state x_{init} and the desired target state x_{target}. However, this reference trajectory intersects the obstacle, forcing the optimizer to compute a trajectory that avoids that obstacle. The dynamics of this system are given by a double integrator in three dimensions subject to gravity. These dynamics are discretized with a zero-order-hold to obtain the discrete-time linear system given by 7.10b, where

$$A = \begin{bmatrix} I_3 & (\Delta t) I_3 \\ 0_3 & I_3 \end{bmatrix}$$

$$B = \frac{1}{m} \begin{bmatrix} (1/2)(\Delta t)^2 I_3 \\ (\Delta t) I_3 \end{bmatrix}$$

$$c = \begin{bmatrix} 0_{2\times 1} \\ -(1/2) g (\Delta t)^2 \\ 0_{2\times 1} \\ -g(\Delta t) \end{bmatrix}$$

$g = 9.8$ is the non-dimensional gravitational acceleration and m is the mass of the quadrotor.

We impose two control constraints: an upper bound on thrust (7.10f) and a thrust pointing constraint (7.10g). The thrust pointing constraint is a proxy to constrain the tilt angle of the quadrotor, since we are using a three-degree-of-freedom model and cannot directly constrain the quadrotor's tilt.

A cylindrical keep-out zone can be written as

$$\|r_t^{1:2} - r_c\|_2 \geq \rho$$

where r_c is the position of the center of the obstacle and ρ is the radius of the obstacle. This constraint, however, is nonconvex.

This nonconvex keep-out-zone constraint can be replaced with a convex rotating half-space constraint, as in [195]. This rotating half-space constraint is written as 7.10d with

$$a_t = [\cos(\psi t + \phi) \;\; -\sin(\psi t + \phi)]^\top$$
$$b_t = -a_t^\top r_c - \rho$$

where ψ is the rotation rate of the half-space and ϕ is the initial phase offset of the half-space.

This problem can be written as

$$\underset{x_t,u_t}{\text{minimize}} \ \frac{1}{2}\left(\sum_{t=1}^{T}(x_t-\hat{x}_t)^\top Q_t(x_t-\hat{x}_t) + \sum_{t=1}^{T-1}u_t^\top R_t u_t\right) \tag{7.10a}$$

$$\text{subject to } x_{t+1} = Ax_t + Bu_t + c \qquad t \in [1, T-1] \tag{7.10b}$$

$$x_1 = x_{\text{init}} \tag{7.10c}$$

$$a_t^\top r_t \leq b_t \qquad t \in [1, T] \tag{7.10d}$$

$$\|v_t\|_2 \leq v_{\max} \qquad t \in [1, T] \tag{7.10e}$$

$$\|u_t\|_2 \leq u_{\max} \qquad t \in [1, T-1] \tag{7.10f}$$

$$\cos(\theta_{\max})\|u_t\|_2 \leq u_t^\top e \qquad t \in [1, T-1] \tag{7.10g}$$

where $e = [0\ 0\ 1]^\top$. Additionally, the following non-dimensional parameters are used

$$T = 30,\ \Delta t = 0.2,\ m = 3,\ \psi = -0.5$$

$$\phi = -\pi/4,\ r_c = [2.5\ 2.5]^\top,\ \rho = 0.25$$

$$v_{\max} = 1.5,\ u_{\max} = 35,\ \theta_{\max} = 0.1745$$

$$Q_t = \text{blkdiag}(2I_3, I_3),\ R_t = 0.5I_3$$

$$x_{\text{init}} = [0\ 0\ 5\ 0\ 0\ 0],\ x_{\text{target}} = [5\ 5\ 5\ 0\ 0\ 0].$$

With proper choice for H and g, we can rewrite the dynamics given by Constraint 7.10b in the form of Constraint 7.1b. Constraint 7.10d represents a half-space, Constraint 7.10e represents a ball, and Constraints 7.10f and 7.10g represent the intersection of a ball and a second-order cone. We can write all of these constraints as Constraint 7.1c, where $\mathbb{D}$ is the Cartesian product of the aforementioned sets.

7.4.3 Implementation Details

We generate our numerical results using MATLAB. From Algorithm 5, we determine that the maximum eigenvalue of $H^\top H$ is η^2. If Algorithm 5 is not used, we use power iteration to compute the maximum eigenvalue of $H^\top H$ [89]. To compute maximum and minimum eigenvalues of P, $\lambda_{\max}$ and $\lambda_{\min}$ respectively, we simply take the maximum and minimum diagonal elements of P since P is diagonal in our examples. For general $P \succ 0$, we can compute the maximum eigenvalue with power iteration and minimum eigenvalue with either shifted power iteration or inverse power iteration. In all examples, we use $k_{\text{update}} = 25$ when testing Algorithm 6. When running examples without using Algorithm 6, we set $\gamma = \sigma$ which makes the dual step-size $\beta = 1$ consistent with the implementation in [197]. We run the oscillating-mass problem 50 times with 50 different initial conditions drawn from the distribution described in the previous section.

Figures 7.3 and 7.4 depict the sparsity pattern of H for MPC problems before and after applying the QR preconditioner. Applying the preconditioner results in significant fill-in, i.e., the introduction of nonzero elements in a sparse matrix where they did not exist previously. In the PIPG implementation, H is stored as a sparse matrix when the QR preconditioner is not applied, but H is stored as a dense matrix when the preconditioner is applied. This choice of data structure results in faster matrix-vector multiplication for each case. Without preconditioning, the percentage of nonzero elements in H is small enough that sparse matrix vector multiplication is faster than dense matrix vector multiplication. However, after applying the QR preconditioner, H contains too high of a percentage of nonzero elements for sparse matrix-vector multiplication to be faster than dense matrix-vector multiplication. Consequently, the QR preconditioner is limited to problems and embedded systems where the extra storage requirement for the preconditioned, dense H matrix is not restrictive. Before preconditioning, when we store H for both problems in compressed sparse column (CSC), it takes roughly 187 kilobytes for the oscillating masses problem and 11.6 kilobytes for the quadrotor problem. After applying our preconditioner and storing the preconditioned matrix

as a dense matrix, it takes roughly 2581 kilobytes for the oscillating masses problem and 363 kilobytes of storage for the quadrotor problem. As the problem size increases, the storage requirements for the dense matrix become more demanding.

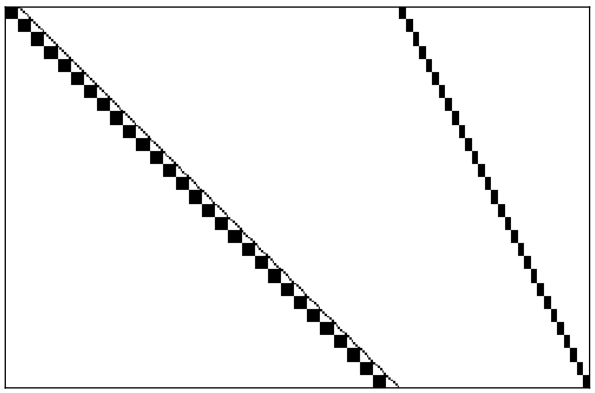


Figure 7.3: Sparsity plot of H for the oscillating mass problem

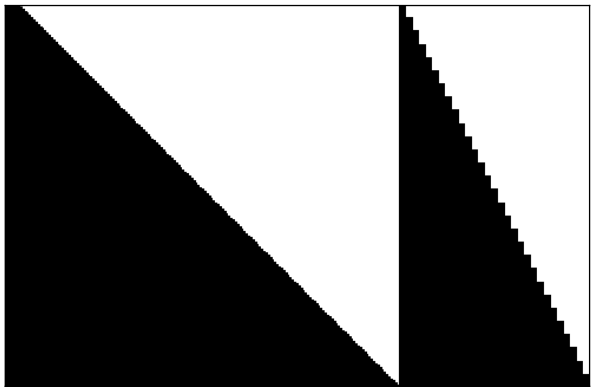


Figure 7.4: Sparsity plot of $\hat{H}$ for the oscillating mass problem

7.4.4 Analysis of results

We assess the effect of the our algorithms using two metrics: relative distance to optimum and feasibility as defined below

$$\text{error}_{\text{opt}} = \frac{\|z^k - z^\star\|_\infty}{\|z^\star\|_\infty}, \ \text{error}_{\text{feas}} = \frac{\|Hz^k - g\|_\infty}{\|z^\star\|_\infty}$$

where $z^\star$ is the solution to 7.1 computed by Gurobi [93] using YALMIP [120]. We terminate PIPG when the relative error, $\text{error}_{\text{opt}}$, decreases below 1.0×10^{-4}, indicating the solution has a relative error of 0.01%.

For the oscillating masses problem, we test the QR preconditioner against modified Ruiz equilibration. For this problem, modified Ruiz equilibration preserves closed-form projection onto set $\mathbb{D}$ since set $\mathbb{D}$ is a box before and after transformation. Additionally, we test the hypersphere preconditioner (HS) on the quadrotor problem, since in this case, the transformed set $\mathbb{D}$ still has a closed-form projection. If our state cost matrix Q_t penalized each component of velocity by different weights or the control cost matrix R_t penalized each component of thrust with different weights, the HS preconditioner will induce a change of variables which will result in the transformed set $\mathbb{D}$ containing ellipsoids and no longer having closed-forms projections. The hypersphere preconditioner involves multiplying the preconditioned cost function by some $\lambda > 0$ similar to how the QR preconditioner multiplies the equality constraints by some $\eta > 0$, however the authors do not provide a way of selecting this λ [99]. In our numerical results, we choose the λ for the hypersphere preconditioner such that the condition number of the KKT matrix given by Equation 7.4 is minimized.

Figure 7.5 depicts the distribution of the absolute value of the eigenvalues for the KKT matrix of the oscillating masses problem and quadrotor problem before and after applying the QR preconditioner. Applying the preconditioner shrinks the interval on which the eigenvalues are distributed, making the maximum and minimum eigenvalues closer in magnitude to each other, reducing the condition number of the KKT matrix.

Figures 7.6 and 7.7 show that the QR preconditioner and the step-size heuristic cause

PIPG to converge in fewer iterations when applied independently. When applied together, we observed a further reduction in iteration count. Applying the QR preconditioner also results in fewer iterations to convergence than applying modified Ruiz equilibration or the hypersphere preconditioner. Note that for the oscillating-mass problem, PIPG without step-selection and the QR preconditioner requires many more than 1500 iterations to meet our stopping criteria, but we only plotted the first 1500 iterations for clarity.

Tables 7.1 and 7.2 contain the solve-times for our numerical examples. The solve-times in the table do not include preconditioning time for any of the preconditioners, since for MPC problems we can perform the preconditioning offline and only have to incur cost of solving the preconditioned problem. We observe that the time to perform the QR preconditioning was very small for the problems we considered: 11.9 and 1.75 milliseconds for the oscillating masses problem and quadrotor problem respectively. Our two proposed algorithms result in solve-time reductions, and we see that the QR preconditioner is more effective at reducing solve-times than modified Ruiz equilibration and the hypersphere preconditioner.

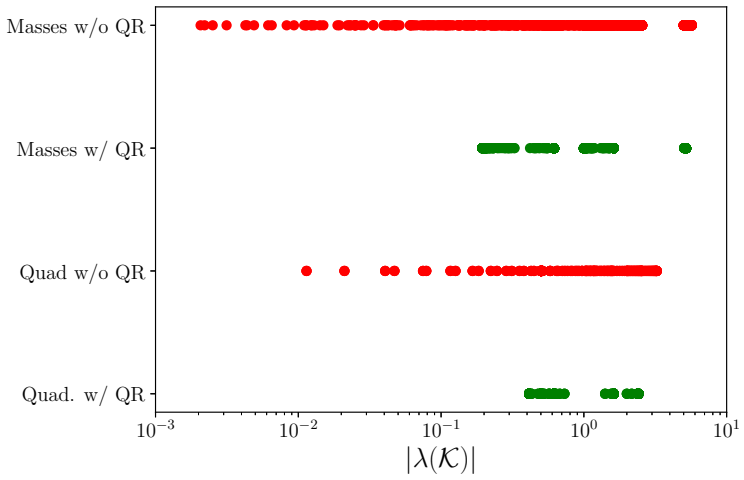


Figure 7.5: Eigenvalue distribution of the KKT matrix for the oscillating masses and quadrotor problems with and without applying the QR preconditioner

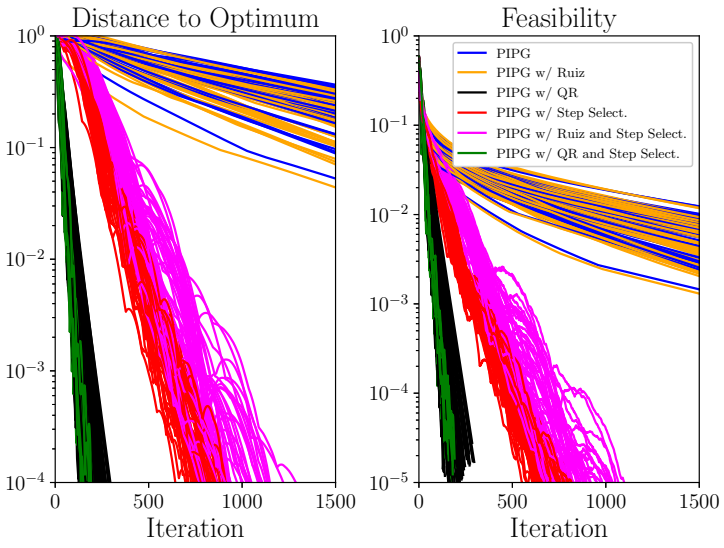


Figure 7.6: $\text{error}_{\text{opt}}$ (left) and $\text{error}_{\text{feas}}$ (right) for the oscillating masses problem

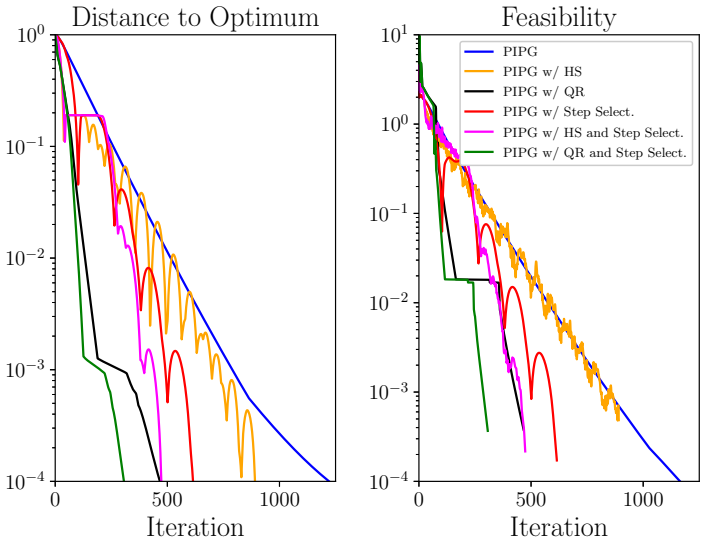


Figure 7.7: $\text{error}_{\text{opt}}$ (left) and $\text{error}_{\text{feas}}$ (right) for the quadrotor problem

	Without Precond.	With Ruiz	With QR
Without Step Select.	651.73	562.80	23.96
With Step Select.	50.85	60.71	**18.69**

Table 7.1: Average solve-time in milliseconds over 50 runs for the oscillating masses problem

	Without Precond.	With HS	With QR
Without Step Select.	226.92	169.50	93.68
With Step Select.	114.91	91.80	**62.04**

Table 7.2: Average solve-time in milliseconds over 100 runs for the quadcopter problem

Chapter 8
REFLECTIONS AND FUTURE DIRECTIONS

This chapter reflects honestly on the technical content, strengths, weaknesses, and future directions of each chapter of this thesis.

8.1 QOCO and QOCOGEN

The main strengths of QOCO are that it is fast on small- to medium-sized problems, robust to ill-conditioned problems, and it is easy to use since it can be called from CVXPY and CVXPYGEN.

The main limitation of QOCO is that it does not have infeasibility detection. When I developed the solver, the intended use case was solving subproblems in sequential convex programming, which are always feasible [126]. However, this lack of infeasibility detection prevents QOCO from being a general-purpose solver. Additionally, QOCO can only handle SOCPs and not SDPs or exponential and power cones.

With infeasibility detection and support for additional cones, QOCO would closely resemble CLARABEL, differing mainly in the implementation language (C rather than Rust) and achieving slightly faster solve times, as shown in the benchmarks of Section 3.5. In this sense, QOCO's value lies less in its novelty and more in providing a clean C-based implementation of the primal-dual IPM for quadratic objective SOCPs.

The novelty of this work is QOCOGEN, the first open-source structure-exploiting custom solver generator for SOCPs. The main strengths of QOCOGEN are the speedups it achieves on small problems, its robustness on ill-conditioned problems, and its applicability to general sparsity patterns. It is also easy to use through its integration with CVXPYGEN, and the generated solvers are library-free with only static memory allocations, making them suitable

for deployment on embedded systems.

The main limitation of QOCOGEN is that the custom linear algebra responsible for its speed also causes code generation and compile times to grow rapidly with problem size, which limits its applicability to small problems.

A promising direction for future work is to develop structure-exploiting techniques that accelerate IPMs without requiring excessive code generation and compile times while being applicable to various problem structures. The methods in [158, 172] target optimal control and multi-stage problems and replace the generic sparse factorization with specialized dense linear algebra routines that exploit the structure of multi-stage KKT systems. The limitation is that these methods only apply to problems with a specific structure. I think the most interesting direction is a generalization of these techniques to problems with arbitrary sparsity patterns. One approach is to compute a permutation of the KKT system that produces dense blocks, then factor using dense operations. Whether such a permutation can always be found efficiently is an open question, but if it can, it would provide solve time speedups without excessive code generation and compile times, while retaining the general applicability of QOCOGEN to arbitrary sparsity patterns.

8.2 GPU acceleration for QOCO

The strength of QOCO's GPU backend is that it enables QOCO to solve much larger problems than a CPU-based IPM implementation can handle, achieving up to a 50× speedup over the CPU backend on large problems. The modular backend abstraction allows the CPU and GPU backends to share a single codebase, making it easy for users to switch between them and making code maintenance easier.

The main limitation is that the scalability is still constrained by the sparse direct factorization, which can introduce significant fill-in, increasing memory usage and causing out-of-memory errors on the GPU for large problems. Instead of using a sparse direct factorization, an indirect method such as preconditioned conjugate gradient (PCG) can be used to solve a condensed KKT system. While this method avoids fill-in and keeps memory usage low,

the condensed KKT system becomes increasingly ill-conditioned as the IPM approaches the solution, causing PCG iteration counts to grow and slowing down the solver [191].

A promising direction for increasing scalability without the drawbacks of indirect methods is to develop a distributed implementation of QOCO that performs the matrix factorization across multiple GPUs or multiple computers, storing the factor across multiple machines. This approach would allow the IPM to scale to problems whose factors exceed the memory of a single machine or single GPU. The cuDSS library [1] supports multi-GPU matrix factorizations and the MUMPS library [8] allows distributed factorizations across multiple computers.

Even a distributed implementation would eventually hit the scaling limits of sparse direct factorization, at which point operator-splitting methods become the only viable option.

8.3 Fast Douglas–Rachford splitting

This chapter presents Fast Douglas–Rachford splitting (FDR), an accelerated operator-splitting algorithm for strongly-convex composite optimization. FDR improves the leading constant in the worst-case $\mathcal{O}(1/N^2)$ convergence rate by a factor of four over the previously best known methods. We also provide a lower bound construction that matches FDR's $\mathcal{O}(1/N^2)$ asymptotic rate and its leading constant. A gap remains in the higher-order terms between FDR's convergence rate and the lower bound. We conjecture that FDR is optimal rather than the lower bound being tight. Closing this gap by finding a tighter lower bound is the most interesting open question raised by this work.

FDR is accelerated in the sense that its worst-case convergence rate is better than that of Douglas–Rachford splitting (DRS), but in practice accelerated algorithms are not universally faster than their unaccelerated variants [166, Chapter 12]. Characterizing the class of problems on which FDR outperforms DRS in practice is an interesting open question. A second limitation is that FDR, like many accelerated methods for strongly-convex minimization, requires knowledge of the strong-convexity constant, which is rarely available in practice. Developing a variant of FDR that does not require this knowledge, potentially by

using adaptive restart strategies [146], is a natural direction for future work.

8.4 GPU-accelerated batch sequential convex programming

We present a GPU-accelerated batch solver for nonconvex trajectory optimization problems that implements the SCP algorithm. This enables fast Monte Carlo analysis and fast multi-start methods where multiple initial guesses are used to generate multiple trajectories and the best is selected. To our knowledge, this was the first GPU implementation of SCP, and the software is open-source and available in the SCVXGEN package [102]. Our results show that these batch solves are about twice as fast as a serial implementation when batch sizes of over 2048 are used.

The main drawback of this design is that it treats the GPU like a CPU. Each GPU thread is given a full nonconvex problem to solve, but the SCP algorithm requires significant control flow, whereas a GPU is optimized for massive parallelism of floating point operations. A better design would run a single SCP instance that contains the entire batch, where the dominant per-iteration operations (discretization and convex subproblem solution) are parallelized on the GPU. This design exposes the GPU to large, regular blocks of floating-point work rather than serial code with frequent branching.

8.5 Constraint preconditioning

This chapter observes that ill-conditioning can occur not only in the objective function but also in the equality constraints, and proposes a preconditioner that reduces the condition number of the equality constraint matrix to exactly 1. Applied to the PIPG algorithm, the preconditioner measurably accelerates convergence on problems with nearly-parallel equality constraints.

The biggest drawback of this preconditioner is that it introduces a significant amount of fill-in to the equality constraint matrix. This limits the preconditioner to small optimization problems, even though it would be most effective for operator-splitting methods, which are most useful for large optimization problems. One useful extension could be a preconditioner

that reduces the condition number of the equality constraints rather than making it unity, while preserving the sparsity pattern.

The preconditioner in this chapter was applied only to PIPG, but it should work for any operator-splitting algorithm for linearly constrained problems, including PDHG [45] (which is equivalent to PIPG in this setting [197]). A formal characterization of which algorithms benefit and under what conditions is an open question. Similarly, the analytical link between constraint preconditioning and the convergence rate of operator-splitting methods was not established in this work. In the equality-constrained case, this connection is clear. The algorithm is a fixed-point iteration of a linear operator, and reducing the condition number of the equality constraint matrix reduces the operator's contraction factor. In the presence of inequality constraints, the connection is more subtle, but the fixed-point operator of an operator-splitting method is piecewise linear, and near the optimal solution, it is linear on the piece corresponding to the optimal active set [124]. Reducing the condition number of the combined equality and active inequality constraints should improve the local linear convergence rate. Building on this observation, a practical extension of the preconditioner to inequality constraints would run the operator-splitting algorithm without preconditioning until the active set is detected, then apply the preconditioner to the equality and active inequality constraints and continue.

BIBLIOGRAPHY

[1] cuDSS: A high-performance cuda library for direct sparse solvers - NVIDIA cuDSS.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Behcet Acikmese, John M. Carson, and Lars Blackmore. Lossless convexification of nonconvex control bound and pointing constraints of the soft landing optimal control problem. *IEEE Transactions on Control Systems Technology*, 21(6):2104–2113, 2013.

[4] Behcet Acikmese and Scott R Ploen. Convex programming approach to powered descent guidance for mars landing. *Journal of Guidance Control and Dynamics*, 30(5):1353–1366, September 2007.

[5] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[6] Farid Alizadeh and Donald Goldfarb. Second-order cone programming. *Mathematical programming*, 95(1):3–51, 2003.

[7] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, October 1996.

[8] P.R. Amestoy, I.S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 184(2):501–520, 2000.

[9] David Applegate, Mateo Díaz, Oliver Hinder, Haihao Lu, Miles Lubin, Brendan O'Donoghue, and Warren Schudy. PDLP: A practical first-order method for large-scale linear programming. *arXiv preprint arXiv:2501.07018*, 2025.

[10] David Applegate, Mateo Díaz, Haihao Lu, and Miles Lubin. Infeasibility detection with primal-dual hybrid gradient for large-scale linear programming. *SIAM Journal on Optimization*, 34(1):459–484, January 2024.

[11] MOSEK ApS. *MOSEK Optimization Suite 10.2.6*, 2024.

[12] Goran Banjac, Paul Goulart, Bartolomeo Stellato, and Stephen Boyd. Infeasibility detection in the alternating direction method of multipliers for convex optimization. *Journal of Optimization Theory and Applications*, 183(2):490–519, 2019.

[13] Goran Banjac, Bartolomeo Stellato, Nicholas Moehle, Paul Goulart, Alberto Bemporad, and Stephen Boyd. Embedded code generation using the osqp solver. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 1906–1911, 2017.

[14] Shane Barratt, Parth Nobel, and Steven Diamond. Moreau: GPU-native differentiable optimization, 2026.

[15] Heinz H. Bauschke, Minh N. Bui, and Xianfu Wang. Projecting onto the intersection of a cone and a sphere. *SIAM Journal on Optimization*, 28(3):2158–2188, 2018.

[16] Heinz H. Bauschke and Patrick L. Combettes. *Convex Analysis and Monotone Operator Theory in Hilbert Spaces*. Springer New York, 2011.

[17] Heinz H. Bauschke and Patrick L. Combettes. Convex analysis and monotone operator theory in hilbert spaces. In *CMS Books in Mathematics*, 2011.

[18] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.

[19] A. Ben-Tal and A. Nemirovski. Robust convex optimization. *Mathematics of Operations Research*, 23(4):769–805, 1998.

[20] A. Ben-Tal and A. Nemirovski. Robust solutions of uncertain linear programs. *Operations Research Letters*, 25(1):1–13, 1999.

[21] Aharon Ben-Tal and Arkadi Nemirovski. *Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications*. Society for Industrial and Applied Mathematics, January 2001.

[22] Michele Benzi, Gene H. Golub, and Jörg Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14:1–137, 2005.

[23] A.J. Berning, Ethan Burnett, and Stefan Bieniawski. Chance-constrained, drift-safe guidance for spacecraft rendezvous. In *AAS Rocky Mountain Guidance, Navigation and Control Conference*. AAS, 2023.

[24] John T Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Advances in Design and Control. SIAM, second edition, January 2010.

[25] Robert E. Bixby and Matthew J. Saltzman. Recovering an optimal LP basis from an interior point solution. *Operations Research Letters*, 15(4):169–178, 1994.

[26] Lars Blackmore. Autonomous precision landing of space rockets. In *in Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2016 Symposium*, volume 46, pages 15–20, 2016.

[27] Paul T Boggs and Jon W Tolle. Sequential quadratic programming. *Acta numerica*, 4:1–51, 1995.

[28] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2017.

[29] Stephen Boyd, Steven Diamond, Kwangmoo Koh, Peter Nystrup, Enzo Busseti, Ronald N Kahn, and Jan Speth. Multi-period trading via convex optimization. *Foundations and Trends in Optimization*, 3(1):1–76, 2017.

[30] Stephen Boyd, Laurent El Ghaoui, Eric Feron, and Venkataramanan Balakrishnan. *Linear matrix inequalities in system and control theory*. SIAM, 1994.

[31] Stephen Boyd, Parikh Neal, Chu Eric, Peleato Borja, and Eckstein Jonathan. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

[32] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004.

[33] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[34] Luis Briceño-Arias and Fernando Roldán. Optimal leveraging of smoothness and strong convexity for peaceman–rachford splitting. *arXiv preprint arXiv:2601.12190*, 2026.

[35] Luis M Briceño-Arias. Lyapunov analysis for fista under strong convexity. *arXiv preprint arXiv:2506.11785*, 2025.

[36] Ronald E Bruck. On the weak convergence of an ergodic iteration for the solution of variational inequalities for monotone operators in hilbert space. *J. Math. Anal. Appl*, 61(1):159–164, 1977.

[37] James R Bunch and Beresford N Parlett. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 8(4):639–655, 1971.

[38] Emmanuel J Candes and Terence Tao. Decoding by linear programming. *IEEE transactions on information theory*, 51(12):4203–4215, 2005.

[39] Emmanuel Candès and Benjamin Recht. Exact matrix completion via convex optimization. *Communications of the ACM*, 55(6):111–119, June 2012.

[40] Yair Carmon, John C Duchi, Oliver Hinder, and Aaron Sidford. Lower bounds for finding stationary points i. *Mathematical Programming*, 184(1):71–120, 2020.

[41] John M. Carson, Michelle M. Munk, Ronald R. Sostaric, Jay N. Estes, Farzin Amzajerdian, James B. Blair, David K. Rutishauser, Carolina I. Restrepo, Alicia M. Dwyer-Cianciolo, George Chen, and Teming Tse. The SPLICE project: Continuing nasa development of gn&c technologies for safe and precise landing. In *AIAA Scitech 2019 Forum*. American Institute of Aeronautics and Astronautics, January 2019.

[42] Antonin Chambolle, Vicent Caselles, Daniel Cremers, Matteo Novaga, Thomas Pock, et al. An introduction to total variation for image analysis. *Theoretical foundations and numerical methods for sparse recovery*, 9(263-340):227, 2010.

[43] Antonin Chambolle and Thomas Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision*, 40(1):120–145, December 2010.

[44] Antonin Chambolle and Thomas Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of mathematical imaging and vision*, 40(1):120–145, 2011.

[45] Antonin Chambolle and Thomas Pock. On the ergodic convergence rates of a first-order primal–dual algorithm. *Mathematical Programming*, 159(1):253–287, 2016.

[46] Govind M. Chari and Behçet Açıkmeşe. QOCO: a quadratic objective conic optimizer with custom solver generation. *Mathematical Programming Computation*, March 2026.

[47] Govind M. Chari and Behçet Açıkmeşe. Spacecraft rendezvous guidance via factorization-free sequential convex programming using a first-order method. *arXiv preprint arXiv:2402.04561*, 2024.

[48] Govind M. Chari and Behçet Açıkmeşe. QOCO-GPU: A quadratic objective conic optimizer with GPU acceleration. *arXiv preprint arXiv:2603.29197*, 2026.

[49] Govind M. Chari, Uijeong Jang, Ernest K. Ryu, and Behçet Açıkmeşe. Optimal acceleration for proximal minimization of the sum of convex and strongly convex functions. *arXiv preprint arXiv:2605.08593*, 2026.

[50] Govind M. Chari, Abhinav G. Kamath, Purnanand Elango, and Behcet Acikmese. Fast monte carlo analysis for 6-dof powered-descent guidance via GPU-accelerated sequential convex programming. In *AIAA SCITECH 2024 Forum*. American Institute of Aeronautics and Astronautics, January 2024.

[51] Govind M. Chari, Yue Yu, and Behçet Açıkmeşe. Constraint preconditioning and parameter selection for a first-order primal-dual method applied to model predictive control. In *2024 IEEE 63rd Conference on Decision and Control (CDC)*, pages 1676–1683. IEEE, 2024.

[52] Yuwen Chen, Paul Goulart, and Colin Jones. A warmstarting technique for general conic optimization in interior point methods. *arXiv preprint arXiv:2512.00693*, 2025.

[53] Yuwen Chen, Danny Tse, Parth Nobel, Paul Goulart, and Stephen Boyd. CuClarabel: GPU acceleration for a conic optimization solver. *arXiv preprint arXiv:2412.19027*, 2024.

[54] Patrick L Combettes and Valérie R Wajs. Signal recovery by proximal forward-backward splitting. *Multiscale modeling & simulation*, 4(4):1168–1200, 2005.

[55] Laurent Condat. A primal–dual splitting method for convex optimization involving lipschitzian, proximable and linear composite terms. *Journal of optimization theory and applications*, 158(2):460–479, 2013.

[56] Laurent Condat, Elnur Gasanov, and Peter Richtárik. The stochastic multi-proximal method for nonsmooth optimization. *arXiv preprint arXiv:2505.12409*, 2025.

[57] Laurent Condat, Grigory Malinovsky, and Peter Richtárik. Distributed proximal splitting algorithms with rates and acceleration. *Frontiers in Signal Processing*, 1:776825, 2022.

[58] Laurent Condat, Abdurakhmon Sadiev, and Peter Richtárik. A nesterov-accelerated primal-dual splitting algorithm for convex nonsmooth optimization. *arXiv preprint arXiv:2604.09245*, 2026.

[59] CVXPY. Robust kalman filtering for vehicle tracking. https://www.cvxpy.org/examples/applications/robust_kalman.html. Accessed: 2024-11-16.

[60] Joachim Dahl and Erling D. Andersen. A primal-dual interior-point algorithm for nonsymmetric exponential-cone optimization. *Mathematical Programming*, 194(1–2):341–370, March 2021.

[61] Shuvomoy Das Gupta, Bart PG Van Parys, and Ernest K Ryu. Branch-and-bound performance estimation programming: A unified methodology for constructing optimal optimization methods. *Mathematical Programming*, 204(1):567–639, 2024.

[62] Damek Davis and Wotao Yin. Faster convergence rates of relaxed peaceman-rachford and ADMM under regularity assumptions. *Mathematics of Operations Research*, 42(3):783–805, 2017.

[63] Damek Davis and Wotao Yin. A three-operator splitting scheme and its optimization applications. *Set-Valued and Variational Analysis*, 25(4):829–858, June 2017.

[64] Timothy A Davis. Algorithm 849: A concise sparse cholesky factorization package. *ACM Transactions on Mathematical Software (TOMS)*, 31(4):587–591, 2005.

[65] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[66] Wei Deng and Wotao Yin. On the global and linear convergence of the generalized alternating direction method of multipliers. *Journal of Scientific Computing*, 66(3):889–916, 2016.

[67] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[68] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, January 2002.

[69] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *European Control Conference (ECC)*, pages 3071–3076, 2013.

[70] Jim Douglas and Henry H Rachford. On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American mathematical Society*, 82(2):421–439, 1956.

[71] Yoel Drori. The exact information-based complexity of smooth convex minimization. *Journal of Complexity*, 39:1–16, 2017.

[72] Yoel Drori and Adrien Taylor. On the oracle complexity of smooth strongly convex minimization. *Journal of Complexity*, 68:101590, 2022.

[73] Yoel Drori and Marc Teboulle. Performance of first-order methods for smooth convex minimization: a novel approach. *Mathematical Programming*, 145(1):451–482, 2014.

[74] Dmitriy Drusvyatskiy and Adrian S Lewis. Error bounds, quadratic growth, and linear convergence of proximal methods. *Mathematics of Operations Research*, 43(3):919–948, 2018.

[75] Daniel Dueri, Behçet Açıkmeşe, Daniel P. Scharf, and Matthew W. Harris. Customized real-time interior-point methods for onboard powered-descent guidance. *Journal of Guidance, Control, and Dynamics*, 40(2):197–212, 2017.

[76] Daniel Dueri, Jing Zhang, and Behcet Açikmese. Automated custom code generation for embedded, real-time second order cone programming. *IFAC Proceedings Volumes*, 47(3):1605–1612, 2014. 19th IFAC World Congress.

[77] Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A Modeling Language for Mathematical Optimization. *SIAM Review*, 59(2):295–320, 2017.

[78] Jonathan Eckstein and Dimitri P Bertsekas. On the douglas—rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical programming*, 55(1):293–318, 1992.

[79] Purnanand Elango, Abhinav G Kamath, Yue Yu, John M Carson III, Mehran Mesbahi, and Behçet Açıkmeşe. A customized first-order solver for real-time powered-descent guidance. In *AIAA SciTech 2022 Forum*, page 0951, 2022.

[80] Purnanand Elango, Dayou Luo, Abhinav G Kamath, Samet Uzun, Taewan Kim, and Behçet Açıkmeşe. Successive convexification for trajectory optimization with continuous-time constraint satisfaction. *arXiv preprint arXiv:2404.16826*, 2024.

[81] Jacques Faraut and Adam Korányi. *Analysis on symmetric cones*. Oxford university press, 1994.

[82] L. R. Ford and D. R. Fulkerson. A suggested computation for maximal multi-commodity network flows. *Management Science*, 5(1):97–101, 1958.

[83] Guilherme França and José Bento. An explicit rate bound for over-relaxed ADMM. *IEEE International Symposium on Information Theory*, 2016.

[84] Michael Garstka, Mark Cannon, and Paul Goulart. Cosmo: A conic operator splitting method for convex conic problems. *Journal of Optimization Theory and Applications*, 190(3):779–810, August 2021.

[85] Euhanna Ghadimi, André Teixeira, Iman Shames, and Mikael Johansson. Optimal parameter selection for the alternating direction method of multipliers (admm): Quadratic problems. *IEEE Transactions on Automatic Control*, 60(3):644–658, 2015.

[86] Pontus Giselsson. Tight global linear convergence rate bounds for douglas–rachford splitting. *Journal of Fixed Point Theory and Applications*, 19(4):2241–2270, 2017.

[87] Pontus Giselsson and Stephen Boyd. Linear convergence and metric selection for douglas-rachford splitting and ADMM. *IEEE Transactions on Automatic Control*, 62(2):532–544, 2016.

[88] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, November 1995.

[89] Gene H Golub and Charles F Van Loan. *Matrix Computations*. Johns Hopkins Series in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 3 edition, October 1996.

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

[91] Jacek Gondzio and Andreas Grothey. A new unblocking technique to warmstart interior point methods based on sensitivity analysis. *SIAM Journal on Optimization*, 19(3):1184–1210, January 2008.

[92] Paul J Goulart and Yuwen Chen. Clarabel: An interior-point solver for conic programs with quadratic objectives. *arXiv preprint arXiv:2405.12762*, 2024.

[93] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[94] Benjamin Halpern. Fixed points of nonexpanding maps. *Bulletin of the American Mathematical Society*, 73(6):957–961, 1967.

[95] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. Newnes, 2020.

[96] Mark Ilg, Jonathan Rogers, and Mark Costello. Projectile Monte-Carlo trajectory analysis using a graphics processing unit. In *AIAA Atmospheric Flight Mechanics Conference*, Guidance, Navigation, and Control and Co-located Conferences, pages 1–18. American Institute of Aeronautics and Astronautics, August 2011.

[97] Uijeong Jang, Shuvomoy Das Gupta, and Ernest K Ryu. Computer-assisted design of accelerated composite optimization methods: Optista. *Mathematical Programming*, pages 1–109, 2025.

[98] Juan L. Jerez, Paul J. Goulart, Stefan Richter, George A. Constantinides, Eric C. Kerrigan, and Manfred Morari. Embedded online optimization for model predictive control at megahertz rates. *IEEE Transactions on Automatic Control*, 59(12):3238–3251, 2014.

[99] Abhinav G Kamath, Purnanand Elango, Taewan Kim, Skye Mceowen, Yue Yu, John M Carson III, Mehran Mesbahi, and Behçet Açıkmeşe. Customized real-time first-order methods for onboard dual quaternion-based 6-dof powered-descent guidance. In *AIAA SciTech 2023 Forum*, page 2003, 2023.

[100] Abhinav G. Kamath, Purnanand Elango, Skye Mceowen, Yue Yu, John M. Carson, Mehran Mesbahi, and Behcet Acikmese. Customized real-time first-order methods for onboard dual quaternion-based 6-dof powered-descent guidance. In *AIAA SCITECH 2023 Forum*. American Institute of Aeronautics and Astronautics, January 2023.

[101] Abhinav G. Kamath, Purnanand Elango, Yue Yu, Skye Mceowen, Govind M. Chari, John M. Carson III, and Behçet Açıkmeşe. Real-time sequential conic optimization for multi-phase rocket landing guidance. *IFAC-PapersOnLine*, 56(2):3118–3125, 2023. 22nd IFAC World Congress.

[102] Abhinav G. Kamath, Samet Uzun, Govind M. Chari, Purnanand Elango, Michael Szmuk, and Behçet Açı kmeşe. SCvxGEN: Successive convexification with automatic

custom code generation for fast and embedded general-purpose trajectory optimization. http://scvxgen.com, 2026.

[103] Petros Karamanakos, Eyke Liegmann, Tobias Geyer, and Ralph Kennel. Model predictive control of power electronic systems: Methods, results, and challenges. *IEEE Open Journal of Industry Applications*, 1:95–114, 2020.

[104] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[105] Leonid G Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[106] Donghwan Kim. Accelerated proximal point method for maximally monotone operators. *Mathematical Programming*, 190(1):57–87, 2021.

[107] Donghwan Kim and Jeffrey A Fessler. Optimized first-order methods for smooth convex minimization. *Mathematical programming*, 159(1):81–107, 2016.

[108] Kitware, Inc. *CMake: Cross-Platform Build System*, 2024.

[109] Victor Klee and George J Minty. How good is the simplex algorithm. *Inequalities*, 3(3):159–175, 1972.

[110] Masakazu Kojima, Shinji Mizuno, and Akiko Yoshise. *A Primal-Dual Interior Point Algorithm for Linear Programming*, page 29–47. Springer New York, 1989.

[111] Dimitris Kouzoupis, Andrea Zanelli, Helfried Peyrl, and Hans Joachim Ferreau. Towards proper assessment of QP algorithms for embedded model predictive control. In *2015 European Control Conference (ECC)*, pages 2609–2616. IEEE, 2015.

[112] Scott Kuindersma, Robin Deits, Maurice Fallon, Andrés Valenzuela, Hongkai Dai, Frank Permenter, Twan Koolen, Pat Marion, and Russ Tedrake. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous robots*, 40(3):429–455, 2016.

[113] Jennifer Langston. Microsoft announces new supercomputer, lays out vision for future AI work. https://news.microsoft.com/source/features/innovation/openai-azure-supercomputer/, Accessed: 2023-12-02.

[114] Daniel Liberzon. *Calculus of Variations and Optimal Control Theory: A Concise Introduction*. Princeton University Press, December 2011.

[115] Felix Lieder. On the convergence rate of the Halpern-iteration. *Optimization letters*, 15(2):405–418, 2021.

[116] Pierre-Louis Lions and Bertrand Mercier. Splitting algorithms for the sum of two nonlinear operators. *SIAM Journal on Numerical Analysis*, 16(6):964–979, 1979.

[117] Thomas Lipp and Stephen Boyd. Variations and extension of the convex–concave procedure. *Optimization and Engineering*, 17:263–287, 2016.

[118] Miguel Sousa Lobo, Maryam Fazel, and Stephen Boyd. Portfolio optimization with linear and fixed transaction costs. *Annals of Operations Research*, 152:341–365, 2007.

[119] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. Applications of second-order cone programming. *Linear algebra and its applications*, 284(1-3):193–228, 1998.

[120] Johan Lofberg. Yalmip: A toolbox for modeling and optimization in matlab. In *2004 IEEE International Conference on Robotics and Automation*, pages 284–289. IEEE, 2004.

[121] Haihao Lu and Jinwen Yang. cuPDLP.jl: A GPU implementation of restarted primal-dual hybrid gradient for linear programming in Julia. *Operations Research*, 73(6):3440–3452, 2025.

[122] Miles Lubin, Oscar Dowson, Joaquim Dias Garcia, Joey Huchette, Benoît Legat, and Juan Pablo Vielma. JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation*, 15:581–589, 2023.

[123] Miles Lubin and Iain Dunning. Computing in Operations Research Using Julia. *INFORMS Journal on Computing*, 27(2):238–248, 2015.

[124] Dayou Luo, Yue Yu, Maryam Fazel, and Behçet Açıkmeşe. Newton-PIPG: A fast hybrid algorithm for quadratic programs in optimal control. *arXiv preprint arXiv:2503.22131*, 2025.

[125] Danylo Malyuta, Taylor Reynolds, Michael Szmuk, Mehran Mesbahi, Behcet Acikmese, and John M Carson III. Discretization performance and accuracy analysis for the rocket powered descent guidance problem. In *AIAA SciTech 2019 Forum*, pages 1–20, Reston, Virginia, January 2019.

[126] Danylo Malyuta, Taylor P. Reynolds, Michael Szmuk, Thomas Lew, Riccardo Bonalli, Marco Pavone, and Behçet Açıkmeşe. Convex optimization for trajectory generation: A tutorial on generating dynamically feasible trajectories reliably and efficiently. *IEEE Control Systems Magazine*, 42(5):40–113, 2022.

[127] Danylo Malyuta, Taylor P. Reynolds, Michael Szmuk, Thomas Lew, Riccardo Bonalli, Marco Pavone, and Behçet Açıkmeşe. Convex optimization for trajectory generation: A tutorial on generating dynamically feasible trajectories reliably and efficiently. *IEEE Control Systems*, 42(5):40–113, October 2022.

[128] Danylo Malyuta, Yue Yu, Purnanand Elango, and Behçet Açıkmeşe. Advances in trajectory optimization for space vehicle control. *Annual Reviews in Control*, 52:282–315, 2021.

[129] Harry Markowitz. Portfolio selection. *J. Finance*, 7(1):77, March 1952.

[130] István Maros and Csaba Mészáros. A repository of convex quadratic programming problems. *Optimization Methods and Software*, 11(1–4):671–681, January 1999.

[131] Jacob Mattingley and Stephen Boyd. Cvxgen: A code generator for embedded convex optimization. *Optimization and Engineering*, 13:1–27, 2012.

[132] Skye Mceowen, Daniel J Calderone, Aman Tiwary, Jason S Zhou, Taewan Kim, Purnanand Elango, and Behcet Acikmese. Auto-tuned primal-dual successive convexification for hypersonic reentry guidance. In *AIAA SCITECH 2025 Forum*, page 1317, 2025.

[133] Skye Mceowen, Abhinav G Kamath, Purnanand Elango, Taewan Kim, Samuel C Buckner, and Behcet Acikmese. High-accuracy 3-dof hypersonic reentry guidance via sequential convex programming. In *AIAA scitech 2023 forum*, page 0300, 2023.

[134] Sanjay Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–601, November 1992.

[135] Sanjay Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–601, 1992.

[136] Gavin Mendeck and Wade May. Space technology mission directorate-game changing development program-fy23 splice annual review presentation. In *NASA Game Changing Development Annual Program Review*, 2023.

[137] Renato D. C. Monteiro and Ilan Adler. Interior path following primal-dual algorithms. part i: Linear programming. *Mathematical Programming*, 44(1–3):27–41, May 1989.

[138] Arkadi S Nemirovsky. On optimality of krylov's information when solving linear operator equations. *Journal of Complexity*, 7(2):121–130, 1991.

[139] Yu E Nesterov and Michael J Todd. Self-scaled barriers and interior-point methods for convex programming. *Mathematics of Operations research*, 22(1):1–42, 1997.

[140] Yu E Nesterov and Michael J Todd. Primal-dual interior-point methods for self-scaled cones. *SIAM Journal on optimization*, 8(2):324–364, 1998.

[141] Yurii Nesterov. A method for solving the convex programming problem with convergence rate o (1/k2). In *Dokl akad nauk Sssr*, volume 269, page 543, 1983.

[142] Yurii Nesterov and Arkadii Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. Society for Industrial and Applied Mathematics, January 1994.

[143] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*, pages 1–664. Springer Series in Operations Research and Financial Engineering. Springer Nature, 2006.

[144] Brendan O'Donoghue. Operator splitting for a homogeneous embedding of the linear complementarity problem. *SIAM Journal on Optimization*, 31:1999–2023, 08 2021.

[145] Yuyuan Ouyang and Yangyang Xu. Lower complexity bounds of first-order methods for convex-concave bilinear saddle-point problems. *Mathematical Programming*, 185(1):1–35, 2021.

[146] Brendan O'donoghue and Emmanuel Candes. Adaptive restart for accelerated gradient schemes. *Foundations of computational mathematics*, 15(3):715–732, 2015.

[147] François Pacaud and Sungho Shin. GPU-accelerated dynamic nonlinear optimization with ExaModels and MadNLP. In *Conference on Decision and Control*, 2024.

[148] François Pacaud, Sungho Shin, Alexis Montoison, Michel Schanen, and Mihai Anitescu. Condensed-space methods for nonlinear programming on GPUs. *arXiv preprint arXiv:2405.14236*, 2024.

[149] Neal Parikh and Stephen Boyd. Proximal algorithms. *Foundations and Trends in optimization*, 1(3):127–239, 2014.

[150] Chanwoo Park, Jisun Park, and Ernest K Ryu. Factor-$\sqrt{2}$ acceleration of accelerated gradient methods. *Applied Mathematics & Optimization*, 88(3):77, 2023.

[151] Jisun Park and Ernest K Ryu. Exact optimal accelerated complexity for fixed-point iterations. In *International Conference on Machine Learning*, pages 17420–17457. PMLR, 2022.

[152] Gregory B Passty. Ergodic convergence to a zero of the sum of monotone operators in hilbert space. *Journal of Mathematical Analysis and Applications*, 72(2):383–390, 1979.

[153] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[154] Panagiotis Patrinos, Lorenzo Stella, and Alberto Bemporad. Douglas-rachford splitting: Complexity estimates and accelerated variants. *IEEE Conference on Decision and Control*, 2014.

[155] D. W. Peaceman and H. H. Rachford, Jr. The numerical solution of parabolic and elliptic differential equations. *Journal of the Society for Industrial and Applied Mathematics*, 3(1):28–41, 1955.

[156] Frank Permenter, Henrik A. Friberg, and Erling D. Andersen. Solving conic optimization problems via self-dual embedding and facial reduction: A unified approach. *SIAM Journal on Optimization*, 27(3):1257–1282, January 2017.

[157] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, USA, 3 edition, 2007.

[158] Christopher V. Rao, Stephen J. Wright, and James B. Rawlings. Application of interior-point methods to model predictive control. *Journal of Optimization Theory and Applications*, 99(3):723–757, December 1998.

[159] Fatemeh Rastgar, Houman Masnavi, Karl Kruusamäe, Alvo Aabloo, and Arun Kumar Singh. GPU accelerated batch trajectory optimization for autonomous navigation. In *2023 American Control Conference (ACC)*, pages 718–725. IEEE, May 2023.

[160] J.B. Rawlings, E.S. Meadows, and K.R. Muske. Nonlinear model predictive control: A tutorial and survey. *IFAC Proceedings Volumes*, 27(2):185–197, 1994. IFAC Symposium on Advanced Control of Chemical Processes, Kyoto, Japan, 25-27 May 1994.

[161] Steven C. Rennich, Darko Stosic, and Timothy A. Davis. Accelerating sparse cholesky factorization on GPUs. *Parallel Computing*, 59:140–150, 2016. Theory and Practice of Irregular Applications.

[162] Taylor P. Reynolds, Michael Szmuk, Danylo Malyuta, Mehran Mesbahi, Behçet Açıkmeşe, and John M. Carson. Dual quaternion-based powered descent guidance with state-triggered constraints. *Journal of Guidance, Control, and Dynamics*, 43(9):1584–1599, September 2020.

[163] R Tyrrell Rockafellar. Monotone operators and the proximal point algorithm. *SIAM journal on control and optimization*, 14(5):877–898, 1976.

[164] Ralph Tyrell Rockafellar. *Convex Analysis*. Princeton University Press, December 1970.

[165] Ernest K Ryu, Adrien B Taylor, Carolina Bergeling, and Pontus Giselsson. Operator splitting performance estimation: Tight contraction factors and optimal parameter selection. *SIAM Journal on Optimization*, 30(3):2251–2271, 2020.

[166] Ernest K. Ryu and Wotao Yin. *Large-Scale Convex Optimization via Monotone Operators*. Cambridge University Press, 2022.

[167] R. W. H. Sargent and G. R. Sullivan. The development of an efficient optimal control package. In *Proceedings of the 8th IFIP Conference on Optimization Techniques, Würzburg, September 5–9, 1977*, volume 7, pages 158–168, Berlin/Heidelberg, 1977. Springer-Verlag.

[168] Maximilian Schaller, Goran Banjac, Steven Diamond, Akshay Agrawal, Bartolomeo Stellato, and Stephen Boyd. Embedded code generation with cvxpy. *IEEE Control Systems Letters*, 6:2653–2658, 2022.

[169] Daniel P. Scharf, Behçet Açıkmeşe, Daniel Dueri, Joel Benito, and Jordi Casoliva. Implementation and experimental demonstration of onboard powered-descent guidance. *Journal of Guidance, Control, and Dynamics*, 40(2):213–229, February 2017.

[170] Michel Schubiger, Goran Banjac, and John Lygeros. GPU acceleration of ADMM for large-scale quadratic programming. *Journal of Parallel and Distributed Computing*, 144:55–67, 2020.

[171] Roland Schwan, Yuning Jiang, Daniel Kuhn, and Colin N. Jones. PIQP: A proximal interior-point quadratic programming solver. In *2023 62nd IEEE Conference on Decision and Control (CDC)*, pages 1088–1093, 2023.

[172] Roland Schwan, Daniel Kuhn, and Colin N. Jones. Exploiting multistage optimization structure in proximal solvers, 2025.

[173] Santiago Akle Serrano. *Algorithms for unsymmetric cone optimization and an implementation for problems with the exponential cone*. Stanford University, 2015.

[174] Anton Shilov. Tesla's $300 million AI cluster is going live today, Aug 2023.

[175] Sungho Shin, Mihai Anitescu, and François Pacaud. Accelerating optimal power flow with GPUs: SIMD abstraction of nonlinear programs and condensed-space interior-point methods. *Electric Power Systems Research*, 236:110651, 2024.

[176] Edmund Smith, Jacek Gondzio, and Julian Hall. GPU acceleration of the matrix-free interior point method. In *International Conference on Parallel Processing and Applied Mathematics*, pages 681–689. Springer, 2011.

[177] Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.

[178] Jos F. Sturm. Using sedumi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11(1-4):625–653, 1999.

[179] Jos F Sturm. Avoiding numerical cancellation in the interior point method for solving semidefinite programs. *Mathematical programming*, 95(2):219–247, 2003.

[180] Michael Szmuk, Taylor P. Reynolds, and Behçet Açıkmeşe. Successive convexification for real-time six-degree-of-freedom powered descent guidance with state-triggered constraints. *Journal of Guidance, Control, and Dynamics*, 43(8):1399–1413, 2020.

[181] Michael Szmuk, Taylor P. Reynolds, and Behçet Açıkmeşe. Successive convexification for real-time six-degree-of-freedom powered descent guidance with state-triggered constraints. *Journal of Guidance, Control, and Dynamics*, 43(8):1399–1413, August 2020.

[182] Adrien B Taylor, Julien M Hendrickx, and François Glineur. Smooth strongly convex interpolation and exact worst-case performance of first-order methods. *Mathematical Programming*, 161(1):307–345, 2017.

[183] Andreas Themelis and Panagiotis Patrinos. Supermann: a superlinearly convergent algorithm for finding fixed points of nonexpansive operators. *IEEE Transactions on Automatic Control*, 64(12):4875–4890, 2019.

[184] Kim-Chuan Toh, Michael J Todd, and Reha H Tütüncü. Sdpt3—a matlab software package for semidefinite programming, version 1.3. *Optimization methods and software*, 11(1-4):545–581, 1999.

[185] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[186] Lieven Vandenberghe. The cvxopt linear and quadratic cone program solvers. *Online: http://cvxopt. org/documentation/coneprog. pdf*, 2010.

[187] Robert J Vanderbei. Symmetric quasidefinite matrices. *SIAM Journal on Optimization*, 5(1):100–113, 1995.

[188] Bang Công Vũ. A splitting algorithm for dual monotone inclusions involving cocoercive operators. *Advances in Computational Mathematics*, 38(3):667–681, 2013.

[189] Maolin Wang, Ian McInerney, Bartolomeo Stellato, Stephen Boyd, and Hayden Kwok-Hay So. Rsqp: Problem-specific architectural customization for accelerated convex quadratic optimization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–12, 2023.

[190] Yang Wang and Stephen Boyd. Fast model predictive control using online optimization. *IEEE Transactions on Control Systems Technology*, 18(2):267–278, 2010.

[191] Margaret Wright. The interior-point revolution in optimization: history, recent developments, and lasting consequences. *Bulletin of the American mathematical society*, 42(1):39–56, 2005.

[192] Stephen J. Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, January 1997.

[193] Mihalis Yannakakis. Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, March 1981.

[194] E. Alper Yildirim and Stephen J. Wright. Warm-start strategies in interior-point methods for linear programming. *SIAM Journal on Optimization*, 12(3):782–810, January 2002.

[195] Yue Yu, Purnanand Elango, and Behçet Açıkmeşe. Proportional-integral projected gradient method for model predictive control. *IEEE Control Systems Letters*, 5(6):2174–2179, 2021.

[196] Yue Yu, Purnanand Elango, Behçet Açıkmeşe, and Ufuk Topcu. Extrapolated proportional-integral projected gradient method for conic optimization. *IEEE Control Systems Letters*, 7:73–78, 2023.

[197] Yue Yu, Purnanand Elango, Ufuk Topcu, and Behçet Açıkmeşe. Proportional–integral projected gradient method for conic optimization. *Automatica*, 142:110359, 2022.

[198] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society Series B*, 68:49–67, 02 2006.

[199] Alan L Yuille and Anand Rangarajan. The concave-convex procedure (cccp). *Advances in neural information processing systems*, 14, 2001.

[200] Xianyi Zhang. Openblas. https://github.com/OpenMathLib/OpenBLAS, 2023.

Appendix A

SUPPLEMENTARY MATERIAL FOR QOCO

A.1 Nonsymmetric Newton system

In this section we will show that Newton steps applied to Equations (3.5a), (3.5b), (3.5c), (3.5d) result in a nonsymmetric linear system which is less efficient to store and factor than the symmetric system presented in Equation (3.19).

We must first introduce the arrow matrix, a matrix that will allow us to rewrite the Jordan product, defined in Equation (3.6), as a matrix-vector multiplication. The arrow matrix can be written for vectors in the non-negative orthant and second-order cone as

$$\text{Arw}(x) = \begin{cases} \text{diag}(x) & \text{if } x \in \mathbb{R}^l_+ \\ \begin{bmatrix} x_0 & x_1^\top \\ x_1 & x_0 I \end{bmatrix} & \text{if } x \in \mathcal{Q}^q. \end{cases}$$

For a vector in cone $\mathcal{K}$ where

$$\mathcal{K} = \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_K,$$

and $\mathcal{C}_i$ is the non-negative orthant or a second-order cone, we can write the arrow matrix as

$$\text{Arw}(x) = \text{blkdiag}(\text{Arw}(x_1), \ldots, \text{Arw}(x_K)) \quad \text{where } x_i \in \mathcal{C}_i,$$

Note that if x is in the interior of $\mathcal{K}$, then $\text{Arw}(x)$ is positive definite and thus is invertible.

We can then write the Jordan product for two vectors $u, v \in \mathcal{K}$ as

$$x \circ y = \text{Arw}(x)y,$$

We now linearize Equations (3.9a) - (3.9e) about the current iterate, (x_k, s_k, y_k, z_k), write the Jordan products using the arrow matrix and get the linear system

$$\begin{bmatrix} P & 0 & A^\top & G^\top \\ 0 & \text{Arw}(z_k) & 0 & \text{Arw}(s_k) \\ A & 0 & 0 & 0 \\ G & I & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} -r_x \\ -r_s \\ -r_y \\ -r_z \end{bmatrix}$$

We can attempt to symmetrize the coefficient matrix by multiplying the second equation by $\text{Arw}(s_k)^{-1}$ to get

$$\begin{bmatrix} P & 0 & A^\top & G^\top \\ 0 & \text{Arw}(s_k)^{-1}\text{Arw}(z_k) & 0 & I \\ A & 0 & 0 & 0 \\ G & I & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} -r_x \\ -\text{Arw}(s_k)^{-1} r_s \\ -r_y \\ -r_z \end{bmatrix}. \tag{A.1}$$

However, the coefficient matrix above will only be symmetric if $\text{Arw}(s_k)^{-1}\text{Arw}(z_k)$ is symmetric. This can only be ensured if $\mathcal{K}$ only consisted of the non-negative orthant and no second-order cones. In other words, the system in (A.1) would only be symmetric if Problem 3.1 was a quadratic program rather than a second-order cone program. Even if we were to eliminate Δs from Equation (A.1), we would still not have a symmetric linear system.

A.2 Custom LDL factorization performance

This section provides empirical evidence supporting our claims in Sections 3.1 and 3.4.1. Specifically, we show that sparse linear algebra incurs extra overhead from identifying and locating nonzero elements, leading to more CPU instructions, memory accesses and cache misses. In contrast, a custom implementation that hardcodes the exact memory accesses

and floating point operations reduces these inefficiencies. We specifically compare QDLDL (used in QOCO) against our custom $LDL^\top$ factorization (used in $\text{QOCO}_{\text{custom}}$), since the most expensive operation in a generic implementation of Algorithm 1 is the $LDL^\top$ factorization. To compare the two matrix factorization implementations, we use Linux `perf`, a profiling tool that provides CPU performance metrics, which will allow us to quantify the computational overhead of sparse versus custom factorizations.

As a representative example of a matrix we would factor, we consider the KKT matrix associated with the Linear Quadratic Regulator (LQR). The LQR problem is

$$\begin{aligned} \underset{x,u}{\text{minimize}} \quad & \frac{1}{2}\left(\sum_{k=1}^{T} x_k^\top Q_k x_k + \sum_{k=1}^{T-1} u_k^\top R_k u_k\right) \\ \text{subject to} \quad & x_{k+1} = A_k x_k + B_k u_k \quad \forall k \in [1, T-1] \\ & x_1 = x_{\text{init}}, \end{aligned}$$

and its associated KKT matrix is

$$K = \begin{bmatrix} P & H^\top \\ H & -\epsilon I \end{bmatrix},$$

where

$$P = \text{blkdiag}(Q_1, Q_2, \ldots, Q_T, R_1, R_2, \ldots, R_{T-1})$$

$$H = \begin{bmatrix} A_1 & -I & & & & & B_1 & & & \\ & A_2 & -I & & & & & B_2 & & \\ & & & \ddots & & & & & \ddots & \\ & & & & A_{N-1} & -I & & & & B_{N-1} \\ I & & & & & & & & & \end{bmatrix}.$$

Notice that we apply static regularization to ensure that the $(2, 2)$ block of K is negative definite, making K quasidefinite. The sparsity pattern of K is depicted by Figure A.1.

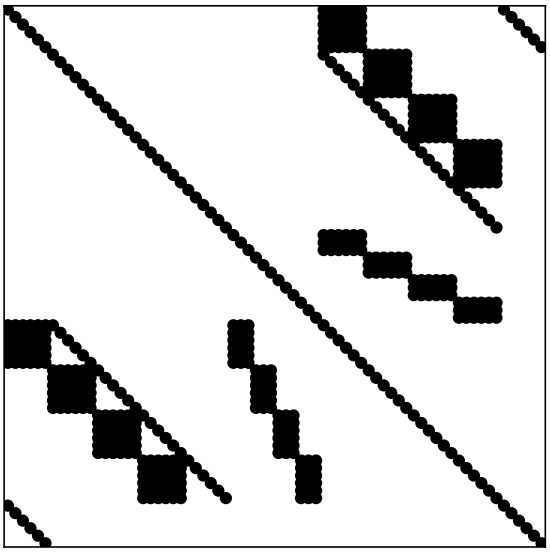


Figure A.1: Sparsity pattern of LQR KKT matrix for T=5

For simplicity, we assume time-invariant system and cost matrices, i.e., $Q_k = R_k = I$ and fixed matrices $A \in \mathbb{R}^{6\times6}$, $B \in \mathbb{R}^{6\times3}$. We choose the elements of A and B randomly and consider time horizons T: $\{5, 15, 50, 75, 100\}$. The actual elements of these matrices do not matter much, as our goal with these experiments is to understand the CPU behavior which (to first order) is unaffected by the value of elements in the matrices. To get accurate profiling results, we run each matrix factorization 1 million times to get the average runtime, number of instructions, number of L1 data cache loads, number of L1 cache misses, number of branches, and number of branch mispredictions.

From Table A.1, we can see that the custom $LDL^\top$ factorization is about four times faster than the sparse factorization because it does much less work: issuing fewer instructions,

performing fewer memory accesses, and incurring fewer L1 cache misses. Although QDLDL has a far more branches than our custom $LDL^\top$ implementation, the branches are quite predictable, and pipeline stalls are avoided.

By hardcoding the exact memory accesses and floating point operations, our custom $LDL^\top$ factorization eliminates much of the overhead associated with sparse matrix operations. This substantial reduction in memory accesses and instructions leads to a significantly faster factorization.

Table A.1: `Perf` results for qdldl and custom $LDL^\top$ factorization

Size	Runtime (us)		Instructions		L1 cache loads		L1 cache misses		Branches		Branch misses	
	qdldl	custom	qdldl	custom	qdldl	custom	qdldl	custom	qdldl	custom	qdldl	custom
5	2.103	0.435	36650	4858	14049	1851	0	0	5419	5	1	0
15	7.507	1.777	132796	18053	50550	6442	572	27	19124	8	4	0
50	27.077	6.117	469289	62046	179709	21468	2539	1204	67081	17	11	1
75	38.848	8.485	709731	93912	279033	32211	3735	1828	101347	24	14	2
100	49.577	12.894	950101	125747	365981	43559	5567	2459	135597	35	18	5

A.3 Problem classes

In this section, we describe the five problem classes considered in our numerical experiments, along with various problem sizes and instances within each problem class. We use the notation $\mathcal{U}(a, b)$ to denote a uniform distribution on the interval $[a, b]$, $\mathcal{N}(\mu, \sigma)$ for a Gaussian distribution with mean μ and standard deviation σ, $0_{m\times n}$ for the matrix of all zeros in $\mathbb{R}^{m\times n}$, and I_n for the identity matrix in $\mathbb{R}^{n\times n}$.

A.3.1 Robust Kalman filter

We consider the robust Kalman filtering problem for vehicle tracking, as outlined in [59]. This problem is a quadratic objective second-order cone program (SOCP) and takes the form

$$\begin{aligned} \underset{x_k, w_k, v_k}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \left(\|w_k\|_2^2 + \tau \phi_\rho(v_k) \right) \\ \text{subject to} \quad & x_{k+1} = Ax_k + Bw_k \quad \forall k \in [0, N-1] \\ & y_k = Cx_k + v_k \quad \forall k \in [0, N-1], \end{aligned} \tag{A.2}$$

where ϕ_ρ is the Huber loss function

$$\phi_\rho(z) = \begin{cases} \|z\|_2^2 & \|z\|_2 \leq \rho \\ 2\rho\|z\|_2 - \rho^2 & \|z\|_2 > \rho. \end{cases}$$

In this problem, $x_k \in \mathbb{R}^4$ represents the state of the vehicle at timestep k, $w_k \in \mathbb{R}^2$ is an unknown force vector acting on the vehicle at timestep k, $v_k \in \mathbb{R}^2$ is the measurement noise vector at timestep k, and $y_k \in \mathbb{R}^2$ represents a noisy measurement of the vehicle's state at timestep k. The matrices A and B are known dynamics matrices for the system, and C is the known observation matrix. To reconstruct the vehicle's state trajectory, we solve Problem A.2.

We use the system matrices

$$A = \begin{bmatrix} 1 & 0 & \left(1 - \frac{\gamma}{2}\Delta t\right)\Delta t & 0 \\ 0 & 1 & 0 & \left(1 - \frac{\gamma}{2}\Delta t\right)\Delta t \\ 0 & 0 & 1 - \gamma\Delta t & 0 \\ 0 & 0 & 0 & 1 - \gamma\Delta t \end{bmatrix}$$

$$B = \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

which correspond to a two-dimensional double integrator model with linear velocity drag and unit mass, where the input is a force and we observe the position of the vehicle. We use $\gamma = 0.05$ for the velocity drag parameter, Δt for the discretization time, and $\rho = 2$ and

$\tau = 2$ for the Huber loss function.

Problem sizes To generate different problem sizes, we vary the number of timesteps N. We consider the following values for N: $\{25, 50, 75, 125, 175, 225, 300, 375, 450, 500\}$. For each of these problem sizes, we set $\Delta t = T/(N-1)$, where $T = 50$ is the time horizon.

Problem instances For each problem size, we generate 20 random problem instances, corresponding to 20 different observation trajectories y_t. To do this, we first randomly generate the force and noise vectors w_t and v_t, respectively, for $t \in [0, N-1]$. Then, we forward-simulate an initial state of $x_0 = [0\ \ 0\ \ 0\ \ 0]$ through the dynamics defined in the constraints of Problem A.2. Each element of w_t and v_t is drawn from a standard Gaussian distribution $\mathcal{N}(0, 1)$. Additionally, for 20 % of the elements of v_t, we introduce outlier noise by drawing elements from $\mathcal{N}(0, 20)$, to simulate noise with a larger magnitude.

A.3.2 Lossless convexification

The losslessly convexified powered-descent guidance problem is an optimal control problem that aims to compute a soft landing trajectory for a rocket while respecting relevant constraints. One important constraint is the lower thrust bound on the engine, which is nonconvex. However, it has been shown that this constraint admits a *lossless* convex relaxation [4, 3]. This means that the solution to the relaxed convex problem guarantees a solution to the original nonconvex problem. The losslessly convexified problem can be written as an SOCP

$$
\begin{aligned}
\underset{x,z,u,\sigma}{\text{minimize}} \quad & -z_T \\
\text{subject to} \quad & x_{k+1} = Ax_k + Bu_k + g \quad \forall k \in [0, T-1] \\
& z_{k+1} = z_k - \alpha\sigma_k \Delta t \quad \forall k \in [0, T-1] \\
& \|u_k\|_2 \leq \sigma_k \quad \forall k \in [0, T-1] \\
& \log(m_{\text{wet}} - \alpha\rho_2 k\Delta t) \leq z_k \leq \log(m_{\text{wet}} - \alpha\rho_1 k\Delta t) \quad \forall k \in [0, T-1] \\
& \mu_{1,k}\left[1 - [z_k - z_{0,k}] + \frac{[z_k - z_{0,k}]^2}{2}\right] \leq \sigma_k \leq \mu_{2,k}[1 - (z_k - z_{0,k})] \quad \forall k \in [0, T-1] \\
& e_3^\top u_k \geq \sigma_k \cos(\theta_{\max}) \quad \forall k \in [0, T-1] \\
& x_0 = x_{\text{init}},\ z_0 = \log(m_{\text{wet}}),\ z_T \geq \log(m_{\text{dry}}),
\end{aligned}
$$

where $x_k \in \mathbb{R}^6$ is the position and velocity of the rocket at timestep k, $u_k \in \mathbb{R}^3$ is the thrust per unit mass at timestep k, $z_k \in \mathbb{R}$ is the natural logarithm of the rocket's mass at timestep k, and $\sigma_k \in \mathbb{R}$ is a slack variable at timestep k. Further we define $z_{0,k} \coloneqq \log(m_{\text{wet}} - \alpha\rho_2 k\Delta t)$, $\mu_{1,k} \coloneqq \rho_1 \mathrm{e}^{-z_0}$, and $\mu_{2,k} \coloneqq \rho_2 \mathrm{e}^{-z_0}$. The parameter $\rho_1 \in \mathbb{R}$ is the minimum thrust of the engine, $\rho_2 \in \mathbb{R}$ is the maximum thrust of the engine, $\alpha \in \mathbb{R}$ is a mass depletion parameter that describes how efficient the engine is, $m_{\text{wet}} \in \mathbb{R}$ is the sum of the dry mass of the rocket and the initial fuel mass, $\theta_{\max} \in \mathbb{R}$ is the maximum angle from vertical that the thrust vector can make, $x_{\text{init}} \in \mathbb{R}^6$ is the initial state of the rocket, $\Delta t \in \mathbb{R}$ is the discretization time, and e_3 is the third canonical basis vector in $\mathbb{R}^3$. The dynamics matrices A and B, and the vector g, are defined as

$$A = \begin{bmatrix} I_3 & \Delta t(I_3) \\ 0_{3\times 3} & I_3 \end{bmatrix}$$
$$B = \begin{bmatrix} \frac{1}{2}\Delta t^2(I_3) \\ \Delta t(I_3) \end{bmatrix}$$
$$g = [0 \;\; 0 \;\; -0.5 g_0 \Delta t^2 \;\; 0 \;\; 0 \;\; -g_0 \Delta t].$$

For this problem class we use the parameters

$$g_0 = 9.807,\ \rho_1 = 100,\ \rho_2 = 500,\ m_{\text{dry}} = 25,\ m_{\text{wet}} = 35,\ \theta_{\max} = \pi/4,\ \alpha = 0.001.$$

Problem sizes To generate different problem sizes, we vary the number of timesteps T. The discretization time is computed as $\Delta t = t_f/(T-1)$, where $t_f = 20$. We consider the following values for T: $\{15, 50, 75, 100, 125, 150, 200, 250, 300, 350\}$.

Problem instances For each problem size (i.e., for each number of timesteps), we generate 20 random problem instances, which involves generating a random initial state x_{init}. We generate x_{init} as

$$x_{\text{init}} \sim \begin{bmatrix} \mathcal{U}(-10, 10) \\ \mathcal{U}(-10, 10) \\ \mathcal{U}(200, 400) \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

A.3.3 Group lasso

The group lasso problem is a quadratic objective SOCP. It is a variation of the least-squares regression problem, where the regression variables are partitioned into disjoint groups, and a regularization term is added to encourage group sparsity. Let $x = [x^{(1)}, x^{(2)}, \ldots, x^{(N)}]$ represent the partitioning of the regression variables, where each $x^{(i)}$ is a group of variables. We can write the problem as

$$\underset{x}{\text{minimize}} \quad \|Ax - b\|_2^2 + \lambda \sum_{i=1}^{N} \|x^{(i)}\|_2.$$

This can be written as an SOCP by introducing a slack variable $t \in \mathbb{R}^N$, and an auxiliary variable $y \in \mathbb{R}^m$ to avoid computing $A^\top A$ and maintain sparsity in the Hessian of the objective function. The problem is now

$$\begin{aligned} \underset{x,t}{\text{minimize}} \quad & \|y\|_2^2 + \lambda \sum_{i=1}^{N} t_i \\ \text{subject to} \quad & \|x^{(i)}\|_2 \leq t_i \quad \forall i \in [1, N] \\ & y = Ax - b, \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$.

Problem sizes To generate different problem sizes, we vary the number of groups N. We consider the following values for N: $\{1, 2, 3, 4, 5, 8, 10, 12, 14, 16\}$.

Problem instances For each problem size (i.e., for each number of groups) we generate 20 random problem instances, corresponding to the generation of the matrix A and the vector b. We set A to have $m = 250N$ rows and $n = 10N$ columns, where x is partitioned into N groups, each containing ten regression variables. The matrix A is sparse, with 10% of its

elements nonzero, and the nonzero elements are drawn from the uniform distribution $\mathcal{U}(0,1)$.

To generate b, we first generate $\hat{x} \in \mathbb{R}^n$ and partition it into N groups of ten variables. Half of the groups are set to zero, and the remaining elements are drawn from the standard Gaussian distribution $\mathcal{N}(0,1)$. Next, we generate an error vector $e \in \mathbb{R}^m$ with elements drawn from $\mathcal{N}(0, 1/n)$. Finally, we compute $b = A\hat{x} + e$. Note that each problem instance will have a different sparsity pattern for A, which requires us to generate a new custom solver for each instance.

A.3.4 Portfolio optimization

We consider the Markowitz portfolio optimization problem with a no-short-selling constraint [129, 32]. This problem is the QP

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \gamma x^\top \Sigma x - \mu^\top x \\ \text{subject to} \quad & \sum_{i=1}^{n} x_i = 1 \\ & x_i \geq 0 \quad \forall i \in [1, n], \end{aligned}$$

where $x \in \mathbb{R}^n$ is the allocation vector, with x_i representing the percentage of resources invested in asset i, $\mu \in \mathbb{R}^n$ is the vector of expected returns, and $\Sigma \in \mathbb{S}^n_+$ is the covariance matrix of asset returns.

Typically, the return covariance matrix Σ is approximated as the sum of a low-rank matrix and a diagonal matrix. Thus, we write $\Sigma = FF^\top + D$, where $F \in \mathbb{R}^{n \times k}$ is a factor matrix of rank k. We then add the auxiliary variable y and the constraint $y = F^\top x$ to avoid computing $FF^\top$ and maintain sparsity in the Hessian of the objective function. The final problem can be rewritten as

$$
\begin{aligned}
\underset{x,y}{\text{minimize}} \quad & x^\top D x + y^\top y - \gamma^{-1} \mu^\top x \\
\text{subject to} \quad & y = F^\top x \\
& \sum_{i=1}^{n} x_i = 1 \\
& x_i \geq 0 \quad \forall i \in [1, n].
\end{aligned}
\tag{A.3}
$$

Problem sizes To generate different problem sizes, we vary the number of factors, k. We consider the following values for k: $\{2, 4, 6, 8, 10, 15, 20, 25, 30, 35\}$.

Problem instances For each problem size (i.e., each number of factors), we generate 20 random problem instances. This corresponds to generating the matrices D, F, and the vector μ. We set the number of assets to $n = 100k$, and for all instances, we take $\gamma = 1$. The diagonal elements of D are drawn from $\mathcal{N}(0, 1)$, and the elements of μ are drawn from $\mathcal{N}(0, \sqrt{k})$. The matrix F is a sparse matrix, with 50% of its elements being nonzero, and its nonzero elements are drawn from $\mathcal{N}(0, 1)$. Note that each problem instance will have a different sparsity pattern for F, which requires us to generate a new custom solver for each instance.

A.3.5 Oscillating masses

The oscillating masses problem is an optimal control problem involving a system of N masses in one dimension, where each mass is connected to its neighboring mass by a spring, and the outermost two masses are attached to a fixed wall. The goal is to apply forces to each mass to bring the system to equilibrium [190]. This problem can be formulated as a QP

$$
\begin{aligned}
\underset{x,u}{\text{minimize}} \quad & \frac{1}{2}\left(\sum_{k=0}^{T} x_k^\top Q x_k + \sum_{k=0}^{T-1} u_k^\top R u_k\right) \\
\text{subject to} \quad & x_{k+1} = A x_k + B u_k \quad \forall k \in [0, T-1] \\
& x_0 = x_{\text{init}} \\
& \|x_k\|_\infty \leq x_{\max} \quad \forall k \in [0, T] \\
& \|u_k\|_\infty \leq u_{\max} \quad \forall k \in [0, T-1].
\end{aligned}
$$

where $x_k \in \mathbb{R}^{2N}$ is the position and velocity of the masses at timestep k, $u_k \in \mathbb{R}^N$ is force applied to each mass at timestep k, and $Q \in \mathbb{S}_+^{2N}$ and $R \in \mathbb{S}_+^N$ penalize deviation from equilibrium and control effort respectively.

The dynamics matrices A and B are derived from an exact discretization of the continuous-time linear dynamics, using a zero-order hold on control input

$$
\begin{aligned}
A &= \mathrm{e}^{A_c \Delta t} \\
B &= A_c^{-1}(A - I_{2N})B_c,
\end{aligned}
$$

where

$$
\begin{aligned}
A_c &= \begin{bmatrix} 0_{N\times N} & I_N \\ L_N & 0_{N\times N} \end{bmatrix} \\
B_c &= \begin{bmatrix} 0_{N\times N} \\ I_N \end{bmatrix}.
\end{aligned}
$$

Here, $L_N \in \mathbb{R}^{N \times N}$ is a symmetric tridiagonal matrix with -2 on the main diagonal and 1 on the subdiagonal and superdiagonal.

For this problem class, we use the parameters

$$N = 4,\ \Delta t = 0.25,\ x_{\max} = 2,\ u_{\max} = 5.$$

Problem sizes To generate different problem sizes, we vary the number of timesteps T. We consider the following values for T: $\{8, 20, 32, 44, 56, 76, 96, 116, 136, 156\}$.

Problem instances For each problem size (i.e., each number of timesteps), we generate 20 random problem instances. This involves generating a random initial state x_{init}, as well as the matrices Q and R. We draw each element of x_{init} from $\mathcal{N}(0, 1)$, then clip the elements such that they lie within the interval $[-0.9x_{\max}, 0.9x_{\max}]$. This ensures that the initial state satisfies the state constraints, making the resulting problem feasible. The matrices Q and R are generated as diagonal matrices, with each diagonal element drawn from the uniform distribution $\mathcal{U}(0, 10)$.

Appendix B

SUPPLEMENTARY MATERIAL FOR FDR

B.1 Leading constants for accelerated splitting methods

In this appendix we provide the asymptotic leading constants of the accelerated Davis–Yin splitting algorithm and the accelerated Chambolle–Pock algorithm when specialized to ($\mathcal{P}$).

B.1.1 Accelerated Davis–Yin Splitting

From Proposition A.1 in [63], the iterates of Algorithm (DYS) satisfy

$$(1+2\gamma_k\mu)\|x_{k+1}-x_\star\|^2+\gamma_k^2\|u_{k+1}-u_\star\|^2 \le \|x_k-x_\star\|^2+\gamma_k^2\|u_k-u_\star\|^2.$$

Furthermore, in the proof of Theorem 1.2 in Appendix A of [63], we see that the proximal step sizes satisfy

$$\frac{1}{\gamma_{k+1}^2}=\frac{1+2\gamma_k\mu}{\gamma_k^2},$$

and

$$\lim_{k\to\infty}(k+1)\gamma_k=\frac{1}{\mu}.$$

After N proximal evaluations of g we obtain

$$\begin{aligned}\|x_{N-1}-x_\star\|^2 &\le \gamma_{N-1}^2\left(\frac{1}{\gamma_0^2}\|x_0-x_\star\|^2+\|u_0-u_\star\|^2\right)\\ &= \frac{\gamma_{N-1}^2}{\gamma_0^2}\left(\|x_0-x_\star\|^2+\gamma_0^2\|u_0-u_\star\|^2\right).\end{aligned}$$

Setting $\gamma_0^2 = 1$ to match our assumed initial condition yields the asymptotic rate

$$\|x_{N-1} - x_\star\|^2 \sim \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{N^2\mu^2}.$$

B.1.2 Accelerated Chambolle–Pock

The convergence rate of the accelerated Chambolle–Pock algorithm is given in [43, Theorem 2], assuming $\sigma_0 = 1/\tau_0$. For any $\epsilon > 0$, there exists N_0 such that for all $N \geq N_0$,

$$\begin{aligned}\|x_N - x_\star\|^2 &\leq \frac{1+\epsilon}{N^2}\left(\frac{\|x_0 - x_\star\|^2}{\mu^2\tau_0^2} + \frac{\|u_0 - u_\star\|^2}{\mu^2}\right) \\ &= \frac{1+\epsilon}{N^2\mu^2\tau_0^2}\left(\|x_0 - x_\star\|^2 + \tau_0^2\|u_0 - u_\star\|^2\right).\end{aligned}$$

Setting $\tau_0^2 = 1$ to match our assumed initial condition and taking $\epsilon \to 0$ yields

$$\|x_N - x_\star\|^2 \sim \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{N^2\mu^2}.$$

B.1.3 Comparison with FDR

Recall that FDR satisfies

$$\|x_N - x_\star\|^2 \leq \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{1 + 4N^2\mu^2} \sim \frac{\|x_0 - x_\star\|^2 + \|u_0 - u_\star\|^2}{4N^2\mu^2}.$$

Thus FDR improves the asymptotic constant of both accelerated Davis–Yin splitting and accelerated Chambolle–Pock by a factor of 4.