Technische Universität München

Department of Mathematics

Master's Thesis

Fast Solvers for Batched Constrained Optimization Problems

Anne Christopher

Supervisor	:	Prof. Felix Brandt
Advisor	:	Stefan Heidekrüger
Submission Date	:	31 July 2020

Technische Universität München

Department of Mathematics

Master's Thesis

Fast Solvers for Batched Constrained Optimization Problems

Effiziente Lösungsmethoden für Bündel Restringierter Optimierungs Probleme

Author	:	Anne Christopher
Supervisor	:	Prof. Felix Brandt
Advisor	:	Stefan Heidekrüger
Submission Date	:	31 July 2020

"I hereby declare that this thesis is entirely my own work and that no other sources have been used except those clearly indicated and referenced."

München, 31.07.2020 Anne Christopher

Acknowledgement

I thank God Almighty for showering his immense blessings on me and making this possible!

A heartfelt thanks to the whole **TUM** community. Thank you, **Prof. Felix Brandt** for your supervision. My sincere gratitude to **Stefan Heidekrüger** for your valuable time and generous guidance and most importantly for believing in my abilities.
I also thank **Paul Sutterer** for your kind help.

Thanks to my Mom for her prayers, love and support, my brother, all my family members and my dear friends for your valuable presence and support. A special thanks to my friends Devika and Aynesh for your helping hands. Finally, thanks to my partner Joval for standing with me even through the toughest days and bringing out the best in me. Watching me in heaven is my Pappa, who would be the happiest soul to see me achieve this!

Thanks to everyone who have been a part of this journey!

Abstract

An under-explored problem in convex optimization is efficiently solving many problems that share the same objective and constraint structure but differ in coefficients in each instance. Such problems appear, for example, in computational game theory and market design. Standard commercial solvers such as Gurobi and CPLEX for convex optimization excel at efficiently computing solutions for a single (even complex) instance but are not able to leverage synergies to efficiently solve many problems of the same kind. The goal of this Master's Thesis is to build a solver for solving a batch of constrained convex optimization problems (Quadratic and Linear problems), where individual problem instances are usually fairly small but applying standard solvers is prohibitive because of lack of parallelism and unnecessary duplicated instantiation overhead. A major part of this thesis focuses on Interior-Point methods for solving constrained quadratic & linear optimization problems. The Primal-Dual Interior-Point Method and parallelisation techniques are exploited in building the fast solver with additional speed-up obtained by using the computational power of a GPU.

Zusammenfassung

Ein wenig erforschtes Problem in der konvexen Optimierung besteht darin, viele Probleme effizient zu lösen, welche dieselbe Struktur der Zielfunktion und Nebenbedingungen haben, sich jedoch in ihren jeweiligen Koeffizienten unterscheiden. Solche Problemstellungen treten beispielsweise in der Spieltheorie und im Market Design auf. Gängige kommerzielle Solver wie Gurobi und CPLEX können zwar Lösungen für eine einzelne Instanzen sehr effizient berechnen, jedoch keine Synergien nutzen, um viele gleichartig Probleme effizient zu lösen. Das Ziel dieser Masterarbeit ist es, einen schnellen Solver zu entwickeln, der ein Bündel von relativ kleinen und strukturell gleichartigen restringierteren konvexen Optimierungsproblemen (Quadratische und Lineare Probleme) parallel lösen kann. Ein Hauptteil dieser Masterarbeit konzentriert sich auf Innere-Punkte-Verfahren zur Lösung von Restringierten Quadratischen & Linearen Optimierungsproblemen. Die Primal-Duale Innere-Punkte-Methode und Parallelisierungstechniken werden genutzt, wobei zusätzlich Hardware-Beschleunigung durch die Verwendung von GPUs erzielt wird.

Contents

Chapter 1

Introduction

Mathematical Optimization is a well-studied topic which plays an important role in domains like operations-research, machine learning, economics and many others. Convex Optimization problems are a sub-class of general mathematical optimization problems that can be solved efficiently. The extensive research on this topic has led to the development of several algorithms for solving the different classes of optimization problems. Furthermore, numerous software packages/solvers have been built based on these algorithms for solving optimization problems on a computer. These solvers are widely used in many applications and are even capable of solving highly complex optimization problems.

However, an under-explored problem in this discipline is solving a huge number (up to millions) of optimization problems with a similar structure in an efficient manner. Such scenarios occur, for example, in the domain of auction theory, where it is required to compute several combinatorial auctions which in turn requires solving several independent optimization problems. Most solvers available till date do not have the capability to solve many similar optimization problems in parallel. Also, not many solvers can utilize the computational power of a GPU (when it is available) to accelerate the processing speed, since they are designed to work only on CPUs. We attempt to resolve these issues and set our goal to *building a fast solver (with GPU support) that can solve a batch of individually relatively simple constrained convex optimization problems (specifically Quadratic/Linear Optimization Problems) which have a similar structure and are independent of each other.*

One among the many optimization algorithms is the class of Interior-Point Methods which were initially developed to solve Linear Optimization Problems and later extended to solve other general convex optimization problems. The fast solver that we develop adopts a variation of the interior-point Methods called the **Mehrotra's Predictor Corrector (MPC)** algorithm. With respect to this algorithm that lays its foundation, we name and further reference the solver we develop as the **'MPC-Solver'**.

The further chapters will walk through the basics of optimization and how we adapt and employ certain methods to build the MPC-Solver. Firstly, Chapter 2 gives an overview of optimization problems which is the required background knowledge to understand the material in the further chapters. Chapter 3 lists some of the conventional software packages used for solving optimization problems, which would later be used to benchmark the results of the MPC-Solver. A reader already familiar with the basics of optimization and the conventional solvers can skip these chapters. In Chapter 4 we discuss in detail about the Primal-Dual Interior-Point Methods and especially the MPC algorithm that the MPC-Solver uses. Chapter 5 sketches how we extend the MPC algorithm to solve thousands of problems in parallel by leveraging the PyTorch functionalities and GPU support. It also describes the mathematical techniques like block-elimination and Cholesky decomposition that helps us to improve the computational performance of the MPC-Solver.

Chapter 6 summarizes the results and estimates the performance and accuracy of the MPC-Solver by comparing it with some of the conventional solvers listed in Chapter 3. Chapter 7 focuses on the application of the MPC-Solver in the context of auction theory for computing many combinatorial auctions efficiently. Chapter 8 lists some of the possible dimensions along which the MPC-Solver could be further extended. Finally, in Chapter 9 we conclude our discussion about the MPC-Solver by highlighting the most important details.

Chapter 2

Background: Optimization Problems

The MPC-Solver is designed to solve Quadratic and Linear Optimization Problems. To study how the MPC-Solver works, one needs a thorough understanding of optimization problems and its associated concepts. This chapter reviews the basic mathematical terms and notions associated with mathematical optimization which will be extensively used in the upcoming chapters.

Section 2.1 introduces the standard form of optimization problems and the relevant terms associated with optimization problems. In Section 2.2 we discuss the class of convex optimization problems especially Linear Optimization Problems (LPs) and Quadratic Optimization Problems (QPs) which is our main point of interest with regards to the MPC-Solver. Since we focus on Primal-Dual methods, in Section 2.3 we describe dual problems and in Section 2.4 we list the Karush-Kuhn-Tucker (KKT) conditions for optimality. Finally, Section 2.5 sketches the most common methods for solving LPs and QPs. A reader familiar with these topics can easily skip this chapter.

2.1 Mathematical Optimization Problems

A general mathematical optimization problem has the form:

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, p \end{array} \tag{2.1}$$

where $f_0 : \mathbf{R}^n \to \mathbf{R}$ is the **objective function**, whose value we are trying to minimize by finding an optimal $x = (x_1, \ldots, x_n)$. The functions $f_i : \mathbf{R}^n \to \mathbf{R}$, $i = 1, \ldots, m$, are called **inequality constraints** and the functions $h_i : \mathbf{R}^n \to \mathbf{R}$, $i = 1, \ldots, p$, are called the **equality constraints**. It is often a common practice to convert the inequality constraints to equality constraints by adding *slack variables* and then defining simpler constraints (like non-negativity) on these slack variables. This technique will be employed in Chapter 4.

The constraint functions define bounds on the optimization variable x. The set of points for which the objective and all constraint functions are defined,

$$\mathcal{D} = \bigcap_{i=0}^{m} \operatorname{dom} f_i \cap \bigcap_{i=1}^{p} \operatorname{dom} h_i,$$

is called the **domain** of the optimization problem (2.1). Every $x \in \mathcal{D}$ that satisfies the constraint functions are called **feasible points**. The collection of all such feasible points, i.e, the solution space defined by the constraint functions, is called the **feasible set**. An optimization problem is said to be *feasible* if there exist at least one feasible point and *infeasible* otherwise.

A vector $x^\star$ is called **optimal**, or a solution of the problem 2.1 if it has the smallest objective value among all the feasible points. The **optimum value $\mathbf{p}^\star$** refers to the value of the objective function ($f_0(x^\star)$) at the optimal point $x^\star$. Therefore,

$$p^\star = \inf \{f_0(x) \mid f_1(x), \dots, f_m(x) \leq 0, h_1(x), \dots, h_p(x) = 0\}$$

2.2 Convex Optimization Problems

The different classes of optimization problems are characterized by the nature of the objective and constraint functions. One such class of problems is the **Convex Optimization Problems** whose objective and constraint functions are **convex functions**. A function is said to be convex if the line segment between any two points on the graph of the function lies above or on the graph. Mathematically, a function $f : \mathbf{R}^n \to \mathbf{R}$ is convex if

$$f(ax + by) \leq af(x) + bf(y) \tag{2.2}$$

for all $x, y \in \mathbf{R}^n$ and all $a, b \in \mathbf{R}$ with $a + b = 1, a \geq 0, b \geq 0$.

A convex optimization problem is one of the form (2.1) where objective function f_0 and inequality constraints $f_1, \dots, f_m$ are convex functions i.e, they obey the inequality (2.2). It also requires the equality constraints $h_i(x) = 0,\ i = 1, \dots, p$ to be affine.

The speciality of convex optimization problems lies in the fact that they can be solved efficiently and reliably. This is primarily because of the fact that the feasible set of a convex optimization problem is always a *convex set* since it is defined by a set of m convex functions and p hyper-planes. A set C is a convex set if the line segment joining any two points in C also lies in C. Some useful properties of convex optimization problems are listed below:

- Any locally optimal point is also (globally) optimal. (In simple terms, a feasible point x is said to be locally optimal if it minimizes the objective function over a neighbourhood of the feasible set and said to be globally optimal if it minimizes the objective over the entire feasible set)

- If the objective function is *strictly* convex, then the problem has at most one optimal point. (The function f is called a *strictly* convex function when the inequality in (2.2) is replaced by a strict inequality, i.e when $f(ax+by) < af(x)+bf(y)$.)

These properties allow most convex optimization problems to be solved using polynomial-time algorithms. Linear Programming Problems, Least-squares problems, Convex Quadratic Problems with linear constraints are some of the convex programming problems with known efficient solving methods. *Convex Optimization* by *Boyd and Vandenberghe* [9] is a recommended read that covers deeper details on convex optimization problems and its sub-classes, efficient solving methods for these problems, the complexity analysis of these methods etc.

2.2.1 Linear Optimization Problems

Linear Optimization Problems (generally called as LPs), as the name suggests, have *linear* objective and *linear* constraint functions. They can be expressed using the general form:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Gx \leq h \\ & Ax = b \end{array} \tag{2.3}$$

where $c^T x$ corresponds to the objective function f_0 in (2.1) and the functions $Gx \leq h$ and $Ax = b$ corresponds to the inequality and equality constraints respectively. Since all the linear functions are convex, all LPs fall under the category of convex optimization problems. There are many effective methods in solving LPs including the simplex method and the interior-point methods which we discuss in Section 2.5.

2.2.2 Convex Quadratic Optimization Problems

Convex Quadratic Programming Problems (generally called as QPs) have a *convex quadratic* objective function and *linear* constraint functions. They can be generally expressed as:

$$\begin{array}{ll} \text{minimize} & (1/2)x^T Q x + q^T x \\ \text{subject to} & Gx \leq h \\ & Ax = b \end{array} \tag{2.4}$$

where $Q \in \mathbf{S}_+^n$[1] & $q \in \mathbf{R}^n$ defines the objective function, $G \in \mathbf{R}^{m \times n}$ & $h \in \mathbf{R}^m$ defines the inequality constraints and $A \in \mathbf{R}^{p \times n}$ & $b \in \mathbf{R}^p$ defines the equality constraints. Q, q, G, h, A and b are collectively called as the **problem data**.

Note that the objective function is convex iff Q is *positive-semi-definite*. This follows from the Second-Order Condition for convexity which states that a twice-differentiable function is convex iff its Hessian is positive semi-definite. As we see from (2.4), the Hessian of the objective function here is Q and hence it needs to be positive semi-definite for convexity to hold.

[1]Here $\mathbf{S}_+^n$ refers to the set of $n \times n$ positive semi-definite matrices

If the objective function and constraint functions are both (convex) quadratic, the problems are then called **Quadratic Constrained Quadratic Programs (QCQP's)**. However, here we only deal with QPs. Its also worth noting that Quadratic Programs generalize Linear Programs i.e, Linear Programs are a special case of Quadratic Programs with Q equal to zero.

2.3 Dual Problems

Consider an optimization problem in its standard form as given by (2.1). As defined in Section 2.1, let $\mathcal{D} = \bigcap_{i=0}^{m} \operatorname{dom} f_i \cap \bigcap_{i=1}^{p} \operatorname{dom} h_i$ represent the domain of the optimization problem, which we assume here to be non-empty and let $p^\star$ be the optimum value.

The **Lagrangian** of the optimization problem is a function $L : \mathbf{R}^n \times \mathbf{R}^m \times \mathbf{R}^p \to \mathbf{R}$ that takes into consideration the objective function and the constraints all at once. It is given by:

$$L(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x) \tag{2.5}$$

where $\lambda_i \in \mathbf{R}$ is the Lagrange multiplier associated with the i th inequality constraint $f_i(x) \leq 0$; and $\nu_i \in \mathbf{R}$ is the Lagrange multiplier associated with the i th equality constraint $h_i(x) = 0$. The vectors $\lambda \in \mathbf{R}^m$ and $\nu \in \mathbf{R}^p$ are called the **dual variables** or the **Lagrange multiplier** vectors associated with the problem (2.1).

The **Lagrange dual function** $g : \mathbf{R}^m \times \mathbf{R}^p \to \mathbf{R}$ is the infimum value of the Lagrangian over x :

$$g(\lambda, \nu) = \inf_{x \in \mathcal{D}} L(x, \lambda, \nu) = \inf_{x \in \mathcal{D}} \left(f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x) \right) \tag{2.6}$$

The importance of dual functions is that it yields a lower bound for the optimal value $p^\star$ of the original optimization problem (2.1), also called the **primal problem**. For any $\lambda \geq 0$ and any ν, we see that all $\lambda_i f_i(x)$ and $\nu_i h_i(x)$ are lesser than or equal to zero, since $f_i(x) \leq 0$ and $h_i(x) = 0$ when $x \in \mathcal{D}$. This implies that the dual function would always have a value lesser than or equal to that of $f_0(x)$, i.e

$$g(\lambda, \nu) \leq p^\star \tag{2.7}$$

The best lower bound for the optimal value of the primal problem can be obtained by solving the **Lagrange dual problem** (commonly called the dual problem):

$$\begin{array}{ll} \text{maximize} & g(\lambda, \nu) \\ \text{subject to} & \lambda \geq 0 \end{array} \tag{2.8}$$

Any (λ, ν) are called dual feasible if $\lambda \geq 0$ and $g(\lambda, \nu) > -\infty$. The multipliers $(\lambda^\star, \nu^\star)$ that gives the optimum value for the dual problem (2.8) are called the **dual optimal** or

optimal Lagrange multipliers. If $d^\star$ denotes the optimal value of the dual problem, then from (2.7), it is clear that

$$d^\star \leq p^\star \tag{2.9}$$

This relation between the optimal dual and primal values is called **weak duality**. Weak duality holds regardless of whether the optimization problem is convex. The difference between the primal and dual optimal values, $p^\star - d^\star$, is called the **duality gap**. A stricter version of duality called **strong duality** holds when

$$d^\star = p^\star \tag{2.10}$$

In this case, the duality gap is zero and hence solving the dual problem yields the solution of the primal problem. There are many results that establish conditions on the optimization problem, under which strong duality holds. These conditions are called **constraint qualifications.**

One simple constraint qualification is the **Slater's condition**. Slater's condition says that if there is some point which is *strictly feasible* (i.e, there exist a point in the interior of the feasible set), and if the problem is convex, then strong duality holds. It can also be refined to form a weaker condition (by disregarding the strict feasibility) when the inequality constraints are affine. When the problem is convex and the constraints are affine, it only requires feasibility for strong duality to hold [9]. This concludes that for both the problems that we focus on: LPs and QPs, when the problem is feasible, strong duality always holds, since by definition the constraint functions of these problems are always affine.

2.4 KKT Conditions

For any optimization problem for which strong duality holds, the **Karush-Kuhn-Tucker (KKT)** conditions provide necessary and sufficient conditions for optimality. Assume that an algorithm produces a sequence of primal feasible $x^{(k)}$ and dual feasible $\left(\lambda^{(k)}, \nu^{(k)}\right)$, for $k = 1, 2, \ldots,$. These generated dual and primal variables can be decided to be the optimal primal and dual variables if they satisfy the KKT conditions. The KKT conditions for optimality include the following:

$$\begin{aligned} f_i(x) &\leq 0, \quad i = 1, \ldots, m \\ h_i(x) &= 0, \quad i = 1, \ldots, p \end{aligned} \tag{2.11}$$

$$\lambda_i \geq 0, \quad i = 1, \ldots, m \tag{2.12}$$

$$\lambda_i f_i(x) = 0, \quad i = 1, \ldots, m \tag{2.13}$$

$$\nabla f_0(x) + \sum_{i=1}^{m} \lambda_i \nabla f_i(x) + \sum_{i=1}^{p} \nu_i \nabla h_i(x) = 0 \tag{2.14}$$

If f_0 convex, f_i and h_i affine, then $x^\star$ is the optimal solution for the primal problem (2.1) and $\lambda^\star, \nu^\star$ form the solutions for the dual problem (2.8) if and only if the above conditions hold for $(x, \lambda, \nu) = (x^\star, \lambda^\star, \nu^\star)$. Also the duality gap which is the difference in the optimal values of the primal and dual problem will be equal to zero.

The KKT conditions are elucidated below:

- (2.11) is called the **Primal Feasibility Condition**. These are the constraints from the primal problem (2.1) and it demands that $x^\star$ can be the primal optimal only if it belongs to the feasible set defined by the constrain functions $f_1 \ldots f_m$ and $h_1 \ldots h_p$.
- Similarly, (2.12) is called the **Dual Feasibility Condition**. It requires the constraints of the dual problem (2.8) to be obeyed by the dual optimal $\lambda^\star$.
- (2.13) is called the **Complementary Slackness Condition**. When strong duality holds, the primal and dual optimal values are equal i.e, $f_0(x^\star) = g(\lambda^\star, \nu^\star)$ (for optimal $(x^\star, \lambda^\star, \nu^\star)$). From (2.6), it is clear that this can be true only when

$$\sum_{i=1}^{m} \lambda_i^\star f_i(x^\star) + \sum_{i=1}^{p} \nu_i^\star h_i(x^\star) = 0 \tag{2.15}$$

  (2.11) implies that each individual term of the summation involving the equality constraints reduces to zero since $h_i(x^\star) = 0, \quad i = 1, \ldots, p$. Therefore for (2.15) to hold, $\sum_{i=1}^{m} \lambda_i^\star f_i(x^\star)$ must also be zero. From (2.11) and (2.12) each individual product $\lambda_i^\star f_i(x^\star)$ is non-positive. A sum of non-positive terms can be zero, only when these terms are zero and hence the complementary slackness condition demands $\lambda_i^\star f_i(x^\star)$ to be 0 for all $i = 1, \ldots, m$
- For the final KKT condition (2.14), we assume that the objective and constraint functions $f_0, f_1 \ldots, f_m, h_1, \ldots, h_p$ are differentiable. From (2.6), L is minimized over x. So if $x^\star, \lambda^\star$ and $\nu^\star$ are the optimal primal and dual variables, it follows that the gradient of the Lagrangian $L(x, \lambda^\star, \nu^\star)$ with respect to x vanishes at $x = x^\star$. This is the requirement put forward by **Stationarity Condition** (2.14) for optimality.

2.5 Solving Optimization Problems

As mentioned before, we are primarily interested in two classes of optimization problems: LPs and QPs. In this section, we discuss the two most commonly used methods for solving these problems.

The **Simplex method**, originally developed by Dantzig in 1948, is one among the oldest and successful methods for solving LPs efficiently. It relies on the fact that, for a feasible LP, the optimal value of the objective functions lies on (at least) one of the vertices of the feasible region (which is a polyhedron for LPs). To find such a point, the simplex method starts from a feasible vertex and moves across vertices of the feasible region as long as there is a progress in the value of the objective function. Even though for some period of time it was believed that this method has polynomial complexity, it was later found out that it has poor exponential complexity in the worst case [20]. Among many other developments made to this method, is the extension of the Simplex Method to solve QPs by Philip Wolfe [33].

Another significant discovery in this field was the **Interior Point Method** by Karmarkar in 1984 [19]. The exciting factor about this paper was its claim on polynomial complexity and also its fast convergence to the optimal solution in a few iterations. A **Primal-Dual** method, which is a subclass of interior-point Methods, is used to build the MPC-Solver. The major distinguishing factor of this algorithm from the Simplex Algorithm is that it starts from a point in the interior of the feasible region and moves along a path through the interior to reach to the optimal point. Chapter 4 describes further details of this method and how it can be used to solve QPs. As mentioned in Section 2.2, solving LPs can be considered as solving a special class of QPs with the coefficient matrix Q set to zero.

Across the years, there has been a lot of research in the field of optimization that led to the development of many more such optimization algorithms. With improvements in computer science, many of these algorithms were implemented to build highly efficient solvers specialized to solve a wide range of highly complex optimization problems within a few seconds which would be beyond human abilities. Nowadays, most mathematicians, practitioners in many industries and other research scientists rely on these solvers for dealing with optimization problems that arise in almost all quantitative disciplines ranging from computer science and engineering to operations research, economics etc. Some of the most commonly used optimization solvers will be discussed in Chapter 3.

Chapter 3

Background: Some Conventional Solvers

There are numerous software packages, both proprietary and open-source, available today for solving different classes of optimization problems. As mentioned before, the MPC-Solver is one such solver designed to solve a huge number of small optimization problems (QPs and LPs) efficiently. Before going into the details of the MPC-Solver, it would be beneficial to look at how some of the most commonly used solvers work, their major advantages and disadvantages.

To understand the functioning of the different solvers, we firstly introduce a simple example of a quadratic optimization problem in Section 3.1, which will later be referenced throughout the chapter. We then discuss some conventional solvers like Gurobi, CVXPY and qpth across Sections 3.2 through 3.4. The results from these solvers will be then used as a benchmark for estimating the performance and accuracy of the MPC-Solver in Chapter 6 and Chapter 7. Section 3.5 also lists some other commonly used solvers for further reference. A reader who is already familiar with the background details and usage of these solvers can skip this chapter.

3.1 Exemplary Optimization Problem

In this section, we introduce a simple quadratic optimization problem and define the problem parameters in matrix form. It also includes a small code-snippet (in Python) that imports the necessary libraries and initializes the problem parameters, which will later be extended in the following sections.

The exemplary QP that we use throughout the chapter is given below. The optimization variable x is in the 2D space, i.e, $x = (x_1, x_2) \in \mathbf{R}^2$.

$$\begin{array}{ll} \text{minimize} & 3x_1^2 + 2x_1x_2 + x_2^2 + x_1 + 6x_2 \\ \text{subject to} & 2x_1 + 3x_2 = 4, \\ & x_1, x_2 \geq 0 \end{array} \tag{3.1}$$

The various solvers differ in the required input format of the problem data. While qpth only supports matrix inputs, Gurobi and CVXPY also supports explicitly defining the objective and constraints as mathematical expressions. However, for ease of comparison and due to better performance, we use the matrix format for all solvers. The problem parameters (as described in Section 2.2.2) for the QP (3.1) in matrix form is given by:

$$Q=\begin{bmatrix} 6 & 2 \\ 2 & 2 \end{bmatrix}, \quad q=\begin{bmatrix} 1 \\ 6 \end{bmatrix}, \quad G=\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad h=\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad A=\begin{bmatrix} 2 & 3 \end{bmatrix}, \quad b=[4]$$

We note that the objective function is defined by the matrices Q and q, where Q is positive semi-definite. Also the inequalities $x_1, x_2 \geq 0$ is transformed to the standard form '$Gx \leq h$' by using negation.

The following code snippet imports all the software packages — Gurobi, CVXPY and qpth — that we use in this Chapter. It also initializes the problem parameters given above as NumPy arrays.

Code Snippet 3.1: Exemplar QP

```python
import numpy as np
import torch
import gurobipy as gp
from gurobipy import GRB
import cvxpy as cp
import qpth.qp as qp

#set problem parameters
n = 2;m = 2;p = 1
Q = np.array([6.0,2.0,2.0,2.0]).reshape(n,n)
q = np.array([1.0,6.0]).reshape(n)
G = np.array([-1.0,0.0,0.0,-1.0]).reshape(m,n)
h = np.array([0.0,0.0]).reshape(m)
A = np.array([2.0,3.0]).reshape(p,n)
b = np.array([4.0]).reshape(p)
```

3.2 Gurobi

The Gurobi Optimizer [18], initially founded in 2008, is a proprietary closed-source software that can solve a wide range of optimization problems like LPs, QPs, Quadratically Constrained Programs (QCPs) etc. One could buy a Gurobi license for a single machine or a network, or pay for hourly computing on the Gurobi cloud. Free trials and free academic licences are also available. A significant feature of Gurobi is that it offers support for a wide range of interfaces for different programming languages like C++, Java, Python, R, Matlab etc.

Let us shortly discuss what approach this solver uses to solve the problems of our interest. Gurobi solves LPs and QPs using either of the following algorithms: Simplex or a parallel barrier method. It can even run both of these algorithms in parallel (using the concurrent setting) and return a solution from the one that reports to finish computation first. The essential difference between the simplex method (outlined in Section 2.5) and the barrier method (a type of interior point method mentioned in Section 2.5) is that the Simplex Method would work with thousands/millions of very cheap iterations on extremely sparse matrices, whereas the barrier method would only do some dozens of expensive iterations, but on denser matrices.

Gurobi has a lot of advantages ranging from the wide range of different classes of optimization problems it can solve to the accuracy of the results it produces. It can deal with highly complex optimization problems and provides a lot of functionalities allowing the user to control different model parameters (for example, easily setting bounds on the optimization variables without using constraints). Gurobi is without a doubt, one of the most commonly used commercial optimization software. However, when it comes to solving a huge number (hundreds of thousands) of relatively small optimization problems, at least the current version of Gurobi spends a long time for this. It is this drawback that we try to overcome with the MPC-Solver, while still managing to produce results as good and accurate as Gurobi would do (as discussed in Chapter 6 and Chapter 7).

In code snippet 3.2 we describe shortly how Gurobi can be used (with its Python Interface) to solve the QP defined by Equation (3.1). The initializations from code snippet 3.1 are assumed.

We start by initializing an object that represents the Gurobi Model and optionally setting the necessary parameters. The `addMVar()` adds a matrix-variable with the given shape and other features like name, lower/upper bound on its value etc. If we are not using matrix variables, this can be replaced by the `addVar()` function (here, to add two variables `x1` and `x2`).

We add the constraints using the `addConstr()` function. Note that instead of using the matrix formulation like we do, one could also specify the constraints explicitly as expressions, for example we could set the equality constraint here using the alternative statement : `gurobi_model.addConstr(2*x1 + 3*x2 == 4.0, "eq")`.

We set the objective using the `setMObjective()` when we have a matrix formulation of the problem. It takes as input the quadratic objective matrix ($0.5 * Q$), the linear objective matrix (q) and the constant term (0). The next three arguments of this function corresponds to the optimization variable which is the left multiplier of Q, the right multiplier of Q and the multiplier of q respectively (in our case all of these corresponds to x). The last argument specifies the optimization sense, i.e `GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization. As already mentioned above, we can also use the statement: `gurobi_model.setObjective( 3*x1*x1 + 2*x1*x2 + x2*x2 + x1 + 6*x2, GRB.MINIMIZE)` as an alternative to the `gurobi_model.setMObjective()` that uses the matrix formulation.

On calling `optimize()` function, Gurobi runs the optimization algorithm and by default displays the optimization status. If it finds a solution, then the `objVal` attribute contains the optimum objective value and `getVars()` helps us to retrieve information of the optimal points. If the problem is infeasible/unbounded, attempting to use these attributes/functions will throw an error.

Code Snippet 3.2: Solving QP with Gurobi 9.0.2 (Release Date: March 2, 2020)

```
# Gurobi
#create model
gurobi_model = gp.Model()
#settings to disable verbose output (optional)
gurobi_model.setParam('OutputFlag', False )
#create optimization variable
x = gurobi_model.addMVar(shape=n,
   name = "x",lb = -GRB.INFINITY,ub = GRB.INFINITY)
#add inequality constraints
gurobi_model.addConstr(G @ x <= h, name="ineq")
#add equality constraints
gurobi_model.addConstr(A @ x == b, name="eq")
#add the objective
gurobi_model.setMObjective(0.5 * Q,q,0.0,x,x,x, GRB.MINIMIZE)
#solve the optimization problem
gurobi_model.optimize()
#retrieve optimal objective value
print('Optimal Objective Value:', gurobi_model.objVal)
#retrieve optimal variable value
print('Optimal Point:')
for v in gurobi_model.getVars():
    print(v.varName, v.x)
```

Output

```
Optimal Objective Value: 9.249999999999998
Optimal point:
x[0] 0.5000000000154424
x[1] 0.9999999999897048
```

3.3 CVXPY

CVXPY is a Python-embedded modeling language for convex optimization problems [14] [1]. CVXPY was designed and implemented by Steven Diamond, with contributions from Stephen Boyd, Eric Chu, Akshay Agrawal, Robin Verschueren, and many others; it was inspired by the MATLAB package CVX. It is an open-source software which relies on further open-source solvers like ECOS, OSQP, SCS etc. It also supports other solvers like CPLEX, XPRESS etc. upon separate installation.

CVXPY handles a wide range of different classes of optimization problems like LPs, QPs, Second-Order Cone Programs (SOCPs), Semi-Definite Programs (SDPs) etc. By default, CVXPY chooses the solver according to the problem type. For example, CVXPY will use the OSQP solver when solving QPs. However, the user can also explicitly set the solver that should be used for optimization depending on the problem. Note that not all the solvers are capable of solving all classes of optimization problems that CVXPY supports. Most of them are specialized only for a subset of the problems that CVXPY can handle. It is interesting that Gurobi (Section 3.2) is also one among the supported solvers for CVXPY i.e, you can use Gurobi to solve an optimization problem using the CVXPY interface.

The optimization algorithms used in solving LPs and QPs are obviously dependent on the solver that CVXPY uses in the background. LPs can potentially be solved by all the solvers that CVXPY supports. However, by default, it uses the ECOS solver for LPs. The Embedded Conic Solver (ECOS) [15], is an interior-point solver for Second-Order Cone Programming (SOCP) designed specifically for embedded applications. ECOS uses a standard interior-point algorithm: the Primal-Dual Mehrotra Predictor-Corrector method. The MPC-Solver that we build is also based on this algorithm (more details in Section 4.4). With CVXPY, QPs are by default solved using the OSQP solver. The Operator Splitting Quadratic Program (OSQP) [30] solver solves convex quadratic programs using the ADMM (Alternating Direction Method of Multipliers) Algorithm [10].

The main advantage of using CVXPY is the wide range of problems it supports along with its flexibility in choosing different solvers. Another advantage is that it automatically transforms the problem into the standard form required by the corresponding solvers, calls the solver and unpacks the results. However, the main drawback is that CVXPY takes a huge amount of time when it comes to solving a lot of optimization problems. Chapter 6 will show that, in practice, the default solvers of CVXPY are even much slower than the Gurobi (Section 3.2) solver itself. However, we also take into consideration the computations from CVXPY for benchmarking the results from the MPC-Solver.

In the code snippet 3.3 we describe shortly how CVXPY can be used to solve the QP defined by (3.1). The initializations from code snippet 3.1 are assumed.

We first create a CVXPY variable `'x'` of the desired size (n=2 in our case). The objective can be set using the `cp.Minimize()` or the `cp.Maximize()` functions accordingly. CVXPY offers many atomic functions that can be applied to CVXPY expressions. `cp.quad_form()` is one such function that returns x^TQx when given the input parameters (x, Q). The constraints are defined using the respective matrices and put together into a list.

The CVXPY optimization problem is initialized using the `cp.Problem()`. The function `prob.solve()` solves this optimization problem and returns the optimal value to `prob.value`. It updates the `prob.status` with the solver status (OPTIMAL, INFEASIBLE, UNBOUNDED etc.) and returns the optimal point using the value field of all the variables in the problem. Even in CVXPY, we can replace the matrix formulations of the objective and constraints by explicit expressions as shown for Gurobi in Section 3.2.

Code Snippet 3.3: Solving QP with CVXPY 1.0.31 (Release Date: April 9, 2020)

```
# CVXPY
#create optimization variable
x = cp.Variable(n)
#create problem with objective, inequality and equality constraints
obj= cp.Minimize( 0.5 * cp.quad_form(x, Q) + q @ x)
cons=[G @ x <= h,
      A @ x == b]
prob = cp.Problem(obj,cons)
#solve the problem
prob.solve()
#retrieve optimal objective value
print('Optimal Objective Value:', prob.value)
#retrieve optimal variable value
print('Optimal Point:',x.value)
```

Output

```
Optimal Objective Value: 9.25
Optimal Point: [0.5 1. ]
```

3.4 OptNet: qpth

OptNet: Differentiable Optimization as a Layer in Neural Networks [3] is a very recent research work published by Brandon Amos in 2019. OptNet is a deep learning library that provides a way of adding a learnable optimization layer to a neural network structure. This learnable optimization layer is facilitated by a QP solver named qpth crafted by Brandon Amos and J. Zico Kolter in conjunction with the OptNet paper. qpth is a fast and differentiable QP solver for PyTorch. The paper talks in detail about the automatically learning quadratic program layers by showcasing examples for signal denoising and learning sudoku games.

qpth by itself is an independent QP solver that can be used to solve LPs and QPs without using other features of the OptNet Neural Network structure. qpth also uses the Primal-Dual Interior-Point Methods (Section 2.5) for solving the optimization problems. It has a batched implementation on PyTorch and also supports optimization on a GPU, which are also goals of the MPC-Solver.

Even though it offers a great advantage of fast solves on a GPU, experiments have proved that results from qpth are inconsistent. It has been experimentally observed that when solving the same set of problems in different batches, qpth returns different solutions, at least for some problems. More details about the accuracy of its results and its inconsistencies are discussed in Chapter 6. The MPC-Solver tries to achieve an implementation with batched-solves and GPU support like qpth while returning accurate and consistent solutions.

The code snippet 3.4 shows how qpth can be used to solve the QP defined by 3.1. The initializations from code snippet 3.1 are assumed.

The first notable difference of qpth from the solvers like Gurobi & CVXPY that we discussed before is that it requires the problem data as torch tensors. This is because the qpth library is an implementation on the PyTorch framework. So unlike in code snippets 3.2 and 3.3 where we used NumPy arrays, in 3.4 we convert the problem data to torch tensors. We then simply pass the problem data to the class QPFunction which does the optimization and returns the optimal point. We explicitly calculate the optimum value from this point, since it is not returned by the qpth library. To perform the optimization on a GPU, one can transfer all the problem data to the cuda device using `torch.cuda()` (for example, to transfer `Q_` to cuda, use `Q_.cuda()`) and then call the `qp.QPFunction()` in the same way.

Code Snippet 3.4: Solving QP with qpth 0.0.15 (Release Date: September 9, 2019)

```
# qpth
#convert numpy arrays to tensors for compatibility to qpth
Q_ = torch.tensor(Q);q_ = torch.tensor(q);G_ = torch.tensor(G)
h_ = torch.tensor(h);A_ = torch.tensor(A);b_ = torch.tensor(b)
#solve the optimization problem
x=qp.QPFunction()(Q_, q_, G_, h_, A_, b_)
#find optimal objective value
print('Optimal Objective Value:',(0.5 * (x @ Q_ @ x.T )+ x @ q_))
#print optimal variable value
print('Optimal Point:',x)
```

Output

```
Optimal Objective Value: tensor([[9.2500]],
                                      dtype=torch.float64)
Optimal Point: tensor([[0.5000, 1.0000]],
                                        dtype=torch.float64)
```

3.5 Other Solvers

There are numerous other software packages/libraries - both closed-source and open-source - available for optimization, where each of them implements one among many of the optimization algorithms developed throughout the years. Few among the optimization software packages specialized in solving linear and quadratic programming problems include CPLEX, MOSEK, Maple etc. Other python packages like scipy offer modules for solving optimization problems.

However, it is rarely possible to use these software-packages/libraries to solve numerous optimization problems in parallel. An exception is the package cvxpylayers, which is a Python library for constructing differentiable convex optimization layers in PyTorch and TensorFlow using CVXPY [2]. This package supports solving numerous problems in parallel, but is considerably slow since it uses the default CVXPY solver in the background which is already observed to be slower than many other solvers like Gurobi.

In the following chapters, we introduce the MPC-Solver, the algorithm it uses, the framework on which it is built, its special features and its performance results. We estimate the efficiency of MPC-Solver by comparing it with the three solvers we discussed in detail above – Gurobi, CVXPY and qpth.

Chapter 4

Primal-Dual Interior Point Methods

The Primal-Dual Interior-Point methods (PDIPMs) are proven to be a class of algorithms that solves a wide range of optimization problems including LPs, QPs, SDPs etc.[34]. They fall under the category of Interior-Point methods and employ Newton's method to solve optimization problems in an efficient manner. The research that gave rise to the field of *Interior-Point Methods* and successively the *Primal-Dual Methods* started with the publication of Karmarkar's paper [19] which marked an important point in the history of optimization algorithms.

This chapter includes a detailed study of PDIPMs. Firstly, we discuss the Newton's Method which is the basic building block of PDIPMs in Section 4.1. We then introduce the idea of interior-point methods and PDIPMs in Section 4.2 and its evolution as a path-following algorithm in Section 4.3. We then talk in detail about a variation of the PDIPMs that the MPC-Solver uses — the Mehrotra's Predictor-Corrector approach — in Section 4.4. We finally look shortly at the complexity and convergence properties of these algorithms in Section 4.5.

4.1 Newton's Method

The main working principle behind PDIPMs is the Newton's Method (also known as the Newton-Raphson method). Newton's method finds the roots (or zeroes) of a function by moving along search directions generated by the linear approximation of the same function starting from a point in the function's domain. It runs several iterations where it produces successively better approximations to the roots of the function.

From the Taylor's theorem [4] the linear approximation ($\bar{f}$) of a function f which is differentiable at a point x_0 is given by:

$$\bar{f}(x) = f(x_0) + (x - x_0)f'(x_0) \tag{4.1}$$

In each iteration, the Newton step Δx_n is obtained by finding the root of this linear approximation of f.

$$f\left(x_n\right) + \Delta x_n f'\left(x_n\right) := 0$$

$$\Rightarrow \quad f'\left(x_n\right) \Delta x_n = f\left(x_n\right) \tag{4.2}$$

In the matrix system the equivalent representation of (4.2) to obtain the Newton step would be

$$J(x_n)\Delta x_n = -F(x_n) \tag{4.3}$$

where $F : \mathbf{R}^n \to \mathbf{R}^n$ is a matrix that comprises the system of equations to be solved and $J(x_n)$ is the Jacobian of $F(x)$ at x_n. These Newton steps are the search directions along which the linear approximation of the function will be iteratively evaluated. The Newton's iterates are expected to converge to the roots of the function $F(x)$[4].

The Primal-Dual Interior-Point methods, like its name says, solves the primal optimization problem and its corresponding dual problem in each iteration using Newton's Method. The function that Newton's Method tries to solve is comprised of the KKT equality conditions (or its modifications as we see later). For constrained quadratic problems, the KKT conditions comprise a set of linear equations which can easily be solved by Newton's method. The underlying idea is that, for any convex optimization problem with differentiable objective and affine constraint functions, any points that satisfy the KKT conditions are primal and dual optimal and have zero duality gap [9], as discussed in Section 2.4. However, it is important to note that one needs to find the inverse of the Jacobian matrix to obtain the Newton step in each iteration until it converges. When the Jacobian is an extremely large matrix, this can become a costly operation.

4.2 Interior-Point Methods and Primal-Dual Interior-Point Methods

The Interior-Point methods for optimization problems derives its name from the underlying principle that the iterates generated by this method are points in the interior of the feasible region - i.e, the iterates generated are strictly feasible (or satisfies the inequality constraints strictly). This property distinguishes interior-point methods from the very common simplex method, which moves along the boundary of the feasible region to get to the optimal vertex. The **Barrier method** and **Primal-Dual Interior-Point methods** are the two most common interior-point methods. We further focus our discussion on Primal-Dual Interior-Point methods (PDIPMs). The derivations in this Section follows Mattingley and Boyd [21].

The PDIPMs generates its iterates by solving the equality conditions of the KKT system (see Section 2.4) using Newton's Method, while restricting the iterates to satisfy the inequality conditions of the KKT system strictly.

Consider the following QP and its associated KKT conditions for optimality (we add a *slack variable* $s \in \mathbf{R}^m$ to the system of inequality constraints $Gx \le h$ so as to convert it into a system of equality constraints and then impose a simpler non-negativity constraint on s).

$$\begin{array}{ll} \text{minimize} & (1/2)x^T Q x + q^T x \\ \text{subject to} & Gx + s = h, \\ & Ax = b, \\ & s \ge 0 \end{array} \tag{4.4}$$

where $Q \in \mathbf{S}_+^n$[1], $q \in \mathbf{R}^n$, $G \in \mathbf{R}^{m\times n}$, $h \in \mathbf{R}^m$, $A \in \mathbf{R}^{p\times n}$ and $b \in \mathbf{R}^p$ comprise the problem data and $x \in \mathbf{R}^n$ and $s \in \mathbf{R}^m$ are the primal variables. Considering the dual variables as $y \in \mathbf{R}^p$ (associated with the equality constraints) and $z \in \mathbf{R}^m$ (associated with the inequality constraints), the KKT conditions for optimality of this problem are:

- $Qx + q + G^T z + A^T y = 0$ Stationarity
- $z_i s_i = 0, \quad i = 1, \ldots, m$ Complementary Slackness
- $Gx + s = h, \quad Ax = b, \quad s \ge 0,$ Primal Feasibility
- $z \ge 0$ Dual Feasibility

The Primal-Dual Interior-Point methods, as mentioned before, finds the solution to the primal and dual problem by solving this KKT system using the Newton's Method. The function F that the Newton's method attempts to solve is comprised of the equality conditions in the KKT system given above.

$$F(x,s,z,y) = \begin{bmatrix} \left(Qx + q + G^T z + A^T y\right) \\ Sz \\ (Gx + s - h) \\ (Ax - b) \end{bmatrix} \tag{4.5}$$

where $S = \operatorname{diag}(s)$ [2].

Now, the update steps can be obtained by the Newton's Method. From (4.3), we have:

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ \Delta z \\ \Delta y \end{bmatrix} = \begin{bmatrix} -\left(A^T y + G^T z + Qx + q\right) \\ -Sz \\ -(Gx + s - h) \\ -(Ax - b) \end{bmatrix} \tag{4.6}$$

where $Z = \operatorname{diag}(z)$[21].

The coefficient matrix on the left-hand side corresponds to the Jacobian of $F(x,s,z,y)$ and is obtained by partially differentiating the function $F(x,s,z,y)$ with respect to x, s, z and y respectively.

All primal-dual methods generate iterates (x,s,z,y) that satisfy the KKT inequality conditions strictly, i.e, after each iteration we have (x,s,z,y) such that $s > 0$ and $z > 0$.

[1]Here $\mathbf{S}_+^n$ refers to the set of $n \times n$ positive semi-definite matrices

[2]*diag*(s) returns a diagonal matrix with the vector s on its diagonal

By respecting these conditions, the method avoids spurious solutions, which are points that satisfy $F(x, s, z, y) = 0$ but not $(s, z) \geq 0$ [34].

Most interior-point methods actually require the iterates to be *strictly feasible*. If we define $\mathcal{F}$ as the primal-dual *feasible set* and $\mathcal{F}^0$ as the *strictly feasible set* by:

$$\mathcal{F} = \{(x, s, z, y) | Qx + q + G^T z + A^T y = 0, Gx + s = h, Ax = b, (s, z) \geq 0\}$$
$$\mathcal{F}^0 = \{(x, s, z, y) | Qx + q + G^T z + A^T y = 0, Gx + s = h, Ax = b, (s, z) > 0\}$$

Then the strict feasibility condition requires:

$$(x, s, z, y) \in \mathcal{F}^0$$

If this is fulfilled, then the terms on the right-hand side of (4.6) vanish to zero, except the term $-Sz$. However, since it is often difficult to find a strictly feasible starting point, we use the **Infeasible Interior-Point** methods. These methods do not require the starting points to belong to the set $\mathcal{F}^0$, rather it only requires the positivity of s and z. Unlike the previously defined case where the RHS of (4.6) reduces to zero, with infeasible interior-point methods these terms result in some non-zero residuals. However, at the optimal point, these residuals also limit to zero, as they obey all the KKT conditions.

Once we obtain the Newton steps by solving (4.6), we update the current set of variables. Usually, according to Newton's method we perform a full-step-update given by $x_n + \Delta x_n$, where Δx_n is the n^{th} Newton step. However, in our constrained setting, it is often not possible to make a full Newton step, since it may violate the inequality constraints $(s, z) > 0$. So we find the best possible *step* that can be made along this step direction without violating the inequality condition using a line search. The updates are then performed as:

$$(x_{n+1}, s_{n+1}, z_{n+1}, y_{n+1}) = (x_n, s_n, z_n, y_n) + \alpha(\Delta x_n, \Delta s_n, \Delta z_n, \Delta y_n) \tag{4.7}$$

for some line search parameter $\alpha \in (0, 1]$.

Unfortunately, the steps that can be taken along this pure Newton-Directions are very small ($\alpha \ll 1$) before violating the inequality condition, and hence makes very little progress towards the solution [34]. This is taken care of by biasing the search direction towards the centre of the search space. This is discussed in the following section.

4.3 Path-Following Algorithms

Often PDIPMs modify the basic Newton's method by biasing the search direction towards the interior of the search space (the non-negative orthant $(s, z) \geq 0$). This is obtained by slightly modifying the KKT conditions by introducing a term τ in the *complementary slackness* condition.

$$z_i s_i = \tau \tag{4.8}$$

The points in $\mathcal{C} = (x_\tau, s_\tau, z_\tau, y_\tau) | \tau > 0)$ form the **Central Path**. This modified KKT system would approximate the original KKT system as τ goes to zero. Also as

$\tau \to 0$, if $\mathcal{C}$ converges, then it's limit is the optimal primal-dual solution. The central path drives us to the solution at a much faster rate than the basic Newton's Method, since the steps taken are longer in the former compared to the later. The biased search direction τ is defined by using a *centering parameter* $\sigma \in [0, 1]$ and a *duality measure* μ defined as given below [34]:

$$\mu = s^T z/m \tag{4.9}$$

Therefore, the modified KKT system is given by:

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ \Delta z \\ \Delta y \end{bmatrix} = \begin{bmatrix} -\left(A^T y + G^T z + Qx + q\right) \\ -Sz + \sigma\mu\mathbf{1} \\ -(Gx + s - h) \\ -(Ax - b) \end{bmatrix} \tag{4.10}$$

where $\mathbf{1} \in \mathbf{R}^m$ represents a vector of ones. When $\sigma = 0$ the above system of equations are identical to (4.6). The steps then generated are the standard Newton steps often called the **affine-scaling direction**.

When $\sigma = 1$, (4.10) provides a **centering direction**. Centering directions are biased strongly towards the interior of the non-negative orthant $(s, z) \geq 0$. The centering steps give points on the central path $\mathcal{C}$, but they help very little in reducing μ significantly (since they add a non-negative quantity to the duality measure). However, they set the scene for major progress in the next iteration (since the next iteration starts near $\mathcal{C}$, it will be able to take a relatively long step — like that of the usual Newton's Method — without leaving the non-negative orthant). To achieve the twin goals of reducing μ and improving centrality, many algorithms use values of σ in the interval (0,1) [34].

These algorithms that restrict the iterates to a neighbourhood of the central path $\mathcal{C}$ and follows the central path to reach the solution are called **Path-Following Algorithms**. Some of the common path-following algorithms are listed below:

- *Short-step* path-following methods: These methods choose the centering parameter σ to be only slightly lesser than 1 and hence make small steps towards the solution. They often take full-Newton steps without violating the inequality constraints.
- *Long-step* path-following methods: They choose smaller values of σ compared to the Short-step methods. The resulting long steps might lead to violation of the inequality constraints, and hence they mostly can't take full-Newton steps.
- *Predictor-Corrector* methods: These methods alternate between predictor steps that use a small value of σ and corrector steps that uses σ close to 1.

The Mehrotra's Predictor Corrector algorithm [24] is a type of Predictor-Corrector method that has shown consistent performance in solving a wide range of problems. This method is described in the following section.

4.4 Mehrotra's Predictor-Corrector Method

Most existing interior-point software packages for general-purpose linear programming are based on Mehrotra's Predictor-Corrector (MPC) algorithm [24]. This method falls under the category of path-following PDIPMs. It generates iterates that satisfy the positivity condition $(s, z) > 0$, but not necessarily belonging to the strictly feasible set $\mathcal{F}^0$ (Section 4.2). This method is characterized by the following three components[34] :

- a pure Newton step or an affine-scaling **predictor** direction
- a **centering** parameter σ which is adaptively chosen based on the affine-Newton step
- a **corrector** direction that compensates for some of the non-linearity in the affine-scaling direction.

Unlike the standard path-following primal-dual methods (Section 4.3), Mehrotra's Predictor Corrector Method computes the affine-scaling direction and the centering term in separate steps. To do so, the centering parameter σ is chosen adaptively based on the improvement offered by the affine-scaling direction (generated by the pure Newton steps).

The affine-scaling directions are obtained from a linear approximation of the equality conditions in the KKT system using Newton's Method as described by (4.3). The affine-scaling direction can be used to assess the error in the linear approximation. This error is used to calculate a *corrector* component — this improves the linear, first-order model of F to a quadratic, second-order model. The affine-scaling direction is also used to determine the centering parameter, that biases the search direction towards the center of the feasible region.

If the affine-scaling direction makes good progress in reducing the duality measure μ without violating the inequality bounds $(s, z) > 0$, then there is no much centering required, and so σ can be set close to zero for this iteration. On the other hand, if very little progress can be made along the affine-scaling direction, the centering parameter σ is set close to 1 so as to drive the iterates closer to the central path so that they can make significant progress in the next iterations (Section 4.3).

One disadvantage of finding the affine scaling direction and centering direction separately is that two linear systems need to be solved in each iteration. However, this can be done efficiently, since the coefficient matrix is the same in both the cases. This will be discussed later in detail in Section 5.3. We discuss below the Mehrotra's Predictor algorithm for solving QPs from Mattingley and Boyd [21].

4.4.1 Outline of MPC Algorithm for Solving QPs

The MPC algorithm first focuses on finding an initial solution which can then be driven along the central path using the further main iterations. These two parts of the algorithm are discussed below.

Initialization

Mehrotra's predictor corrector algorithm works with infeasible iterates and only requires the iterates to obey the positivity condition $(s, z) > 0$. An initial point can be generated in many ways, one among which is given below as described in [32]. This method tries to find the (analytic) solution of the primal and dual problems :

$$\begin{array}{ll} \text{minimize} & (1/2)x^T Qx + q^T x + (1/2)\|s\|_2^2 \\ \text{subject to} & Gx + s = h, \quad Ax = b \end{array}$$

with variables x and s, and

$$\begin{array}{ll} \text{maximize} & -(1/2)w^T Qw - h^T z - b^T y - (1/2)\|z\|_2^2 \\ \text{subject to} & Qw + q + G^T z + A^T y = 0 \end{array}$$

with variables w, y and z.

The solution to the KKT system of these problems would reduce to:

$$\begin{bmatrix} Q & G^T & A^T \\ G & -I & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ z \\ y \end{bmatrix} = \begin{bmatrix} -q \\ h \\ b \end{bmatrix} \tag{4.11}$$

We note that (4.11) represents all the KKT conditions of the QP (Section 4.2) except the complementary slackness. This is because instead of using the inequality constraint $s \geq 0$ we include the term $\|s\|_2^2$ in the objective to obtain a minimum possible value of s. We ensure the positivity of the variables s and z as described below.

The initial primal and dual variables are set to $x^{(0)} = x$ and $y^{(0)} = y$. However, we require the positivity of the variables s and z. For achieving this, we find parameters α_p and α_d for updating s and z.

We get $z = Gx - h$ from solving the system above. We then find α_p such that $\alpha_p = \inf\{\alpha | -z + \alpha \mathbf{1} \geq 0\}$. We use this to set the initial value of s:

$$s^{(0)} = \begin{cases} -z & \alpha_p < 0 \\ -z + (1 + \alpha_p)\,\mathbf{1} & \text{otherwise} \end{cases}$$

Finally, to ensure the positivity of z we find $\alpha_d = \inf\{\alpha | z + \alpha \mathbf{1} \geq 0\}$, and update the initial value of z to:

$$z^{(0)} = \begin{cases} z & \alpha_d < 0 \\ z + (1 + \alpha_d)\,\mathbf{1} & \text{otherwise} \end{cases}$$

This gives the starting point $\left(x^{(0)}, s^{(0)}, z^{(0)}, y^{(0)}\right)$.

Main Iterations

We set a maximum limit '*max_ iter*' on the number of iterations to be performed. The MPC algorithm is believed to converge to the optimal solution (if it exists) in less than 25 iterations [21].

1. Evaluate the stopping criteria and halt if the stopping criteria is satisfied. The solution of the optimization problem can be said to be achieved once the residuals [3] and the duality measure limits to zero. This can be used as an early stopping criterion, for avoiding unnecessary computations until *max_ iter* is reached.

2. The affine-scaling directions are found using pure-Newton step as:

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{aff} \\ \Delta s^{aff} \\ \Delta z^{aff} \\ \Delta y^{aff} \end{bmatrix} = \begin{bmatrix} -\left(A^T y + G^T z + Qx + q\right) \\ -Sz \\ -(Gx + s - h) \\ -(Ax - b) \end{bmatrix}$$

3. Using the affine-scaling directions obtained above, the centering parameter σ is calculated as:

$$\sigma = \left(\frac{\left(s + \alpha \Delta s^{\text{aff}}\right)^T \left(z + \alpha \Delta z^{\text{aff}}\right)}{s^T z} \right)^3 = \left(\frac{\mu^{\text{aff}}}{\mu} \right)^3$$

where,

$$\alpha = \sup \left\{ \alpha \in [0,1] | s + \alpha \Delta s^{\text{aff}} \geq 0, z + \alpha \Delta z^{\text{aff}} \geq 0 \right\}$$

i.e, the centering parameter is adaptively chosen based on the improvement offered by the affine-scaling direction alone. The duality measure μ is calculated as $\mu = s^T z/m$. μ^{aff} refers to the new duality measure if only the affine-scaling direction was used for the update.

4. The centering parameter and the duality measure can be used to determine the centering term ($\sigma\mu\mathbf{1}$) that biases the search direction. We can combine this centering step with the corrector step that accounts for the non-linearity in the affine-step. The combined centering-corrector steps are obtained by solving the following system:

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{cc} \\ \Delta s^{cc} \\ \Delta z^{cc} \\ \Delta y^{cc} \end{bmatrix} = \begin{bmatrix} 0 \\ \sigma\mu\mathbf{1} - \text{diag}\left(\Delta s^{aff}\right) \Delta z^{aff} \\ 0 \\ 0 \end{bmatrix}$$

[3]*residuals* corresponds to the non-zero terms (arising from the infeasible iterates) on the RHS of the KKT system which reduces to zero when the iterates become feasible. They correspond to the differences of the KKT equality conditions from zero. Precisely $-(A^T y + G^T z + Qx + q)$ is the dual residual and the terms $-(Gx + s - h)$ & $-(Ax - b)$ form the primal residual.

5. Once the affine-steps and the centering-corrector-steps are obtained, the update-steps can be calculated as:

$$\begin{aligned}\Delta x &= \Delta x^{\text{aff}} + \Delta x^{cc}\\ \Delta s &= \Delta s^{\text{aff}} + \Delta s^{\text{cc}}\\ \Delta y &= \Delta y^{\text{aff}} + \Delta y^{\text{cc}}\\ \Delta z &= \Delta z^{\text{aff}} + \Delta z^{\text{cc}}\end{aligned}$$

6. Finally we perform the updates of the variables as:

$$\begin{aligned}x &:= x + \alpha\Delta x\\ s &:= s + \alpha\Delta s\\ y &:= y + \alpha\Delta y\\ z &:= z + \alpha\Delta z\end{aligned}$$

where

$$\alpha = \min\{1, 0.99 \sup\{\alpha \geq 0 | s + \alpha\Delta s \geq 0, z + \alpha\Delta z \geq 0\}\}$$

6. Repeat until max_iteration is reached

The MPC algorithm discussed above is the backbone of the MPC-Solver. More details on how this algorithm has been implemented and its special characteristics will be explained in Chapter 5.

4.5 Superlinear Convergence and Polynomial Complexity of PDIPM's

Even though a detailed analysis on the complexity and convergence of the algorithms discussed in the previous sections is beyond the scope, it is roughly outlined in this section with pointers to the corresponding references that provide a thorough numerical analysis of the different classes of interior-point methods.

As explained in Section 4.1, the Primal-Dual Interior-Point Methods have its basis laid on Newton's Method. From the time of its discovery, it is well known that Newton's Method has quadratic convergence [31]. Also, Newton's Method scales well with the problem size: its performance on problems in $\mathbf{R}^{10000}$ is similar to its performance on problems in $\mathbf{R}^{10}$ with only a moderate increase in the number of steps [9]. PDIPMs inherits these advantages of Newton's Method. In addition, the PDIPMs make modifications to Newton's Method including the centering term and the step length calculation that insists the search direction to stay away from the boundary of the non-negative orthant which provides some extra structure that leads to superlinearity [34]. The super-linear convergence of PDIPMs have been proven by quite a number of researchers including Ye [35], Ye et al. [37], Ye and Anstreicher [36].

Complexity theory fundamentally distinguishes the performance of an algorithm by classifying it as *Polynomial Algorithms* (those which requires time that is a polynomial of the problem size) or as *Exponential Algorithms* (those that requires an amount of time that is exponential in the problem size). Polynomial-time algorithms are considered as an efficient class of algorithms. When describing the complexities, we can either consider the worst-case behaviour (which is finding an upper bound on the time required) or the average case performance (which is performance offered on typical problems). Simplex algorithm shows polynomial complexity in the average case, whereas it shows poor exponential complexity in the worst case (as found in 1972 with the Klee-Minty Cube problem [20]). Whereas, for the interior point methods, the worst-case complexity is only a polynomial in n, and the average-case depends only on a fractional power of n or even just $log(n)$ [34]. In fact, the polynomial complexity of the interior point methods can be thought of as a consequence of the algorithm finding a solution in polynomial number of iterations where each iteration takes polynomial time [19]. A lot of research works including those from Ye and Anstreicher [36], McShane [23] and many others have even proved that primal-dual methods exhibits $O(\sqrt{n}L)$ complexity.

Interior Point Algorithm: Theory and Analysis by Ye [35] includes a detailed numerical analysis of the different interior-point methods for linear programming, non-linear convex optimization and non-convex optimization, summarizing the results of research works mentioned above associated with the convergence and complexity analysis of interior-point methods.

Chapter 5

The MPC-Solver – Techniques & Implementation

The MPC-Solver is designed to solve hundreds of thousands of optimization problems in *parallel*, thereby offering exceptionally good performance. The MPC-Solver is based on the MPC algorithm (discussed in Chapter 4) and it currently supports solving LPs and QPs. It additionally offers GPU support, potentially improving its performance by utilizing the extra computational power when it is available.

Now that we know the algorithm that the MPC-Solver uses for optimization, we next talk in detail about how the MPC-Solver has been designed — the techniques it uses & its implementation — so as to achieve the ultimate goal of solving a large number of optimization problems very efficiently. Section 5.1 talks about the Python framework on which the MPC-Solver is built. Section 5.2 describes how the MPC-Solver solves numerous problems stacked into a batch in parallel. Section 5.3 mentions some notable features of the MPC algorithm and describes some additional techniques that the MPC-Solver employs to improve its performance. Section 5.4 outlines the major steps that the MPC-Solver performs for solving QPs. Finally Section 5.5 lists the dependencies of the MPC-Solver and illustrates its usage for some example problems

5.1 PyTorch based Implementation of MPC Algorithm

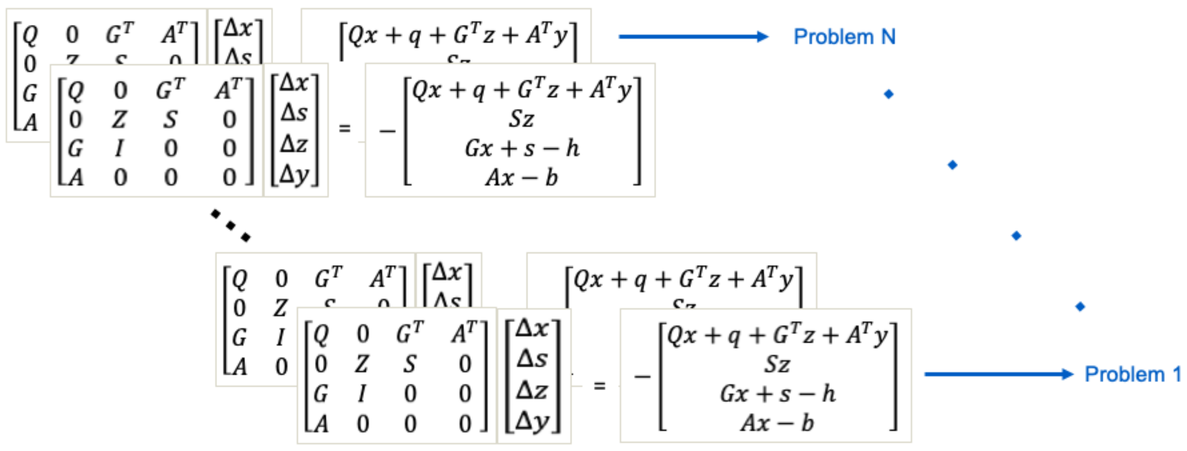
The MPC-Solver is a PyTorch based implementation of the MPC Algorithm (Section 4.4) along with some added techniques to boost up the performance. PyTorch is an open-source machine learning library primarily used with the Python Interface. Similar to the Python NumPy array structure is the PyTorch Tensor (torch.Tensor) which defines a class to store and operate on homogeneous multidimensional rectangular arrays. As usual, a matrix can be represented using a two-dimensional tensor. Further in this chapter, we talk about batched matrices, which is basically stacking matrices together to form a batch. These batched matrices would have three dimensions, where the first one acts as an index for pointing to a matrix that is being referenced and the next two dimensions actually represent the original matrix shape.

The PyTorch library supports parallel operations on a batch. For example, the function `torch.bmm()` takes two input 3D Tensors and performs a batched matrix multiplication i.e, when provided two input sensors with shapes $b \times m \times n$ and $b \times n \times p$, this function would perform b matrix multiplications resulting in a tensor with the shape $b \times m \times p$. It also includes more sophisticated functionalities like LU Factorization, Cholesky Decomposition etc. which helps with solving a system of equations efficiently. The main added advantage of using PyTorch tensors is that it can also be operated on a CUDA-capable Nvidia GPU. All the functions used and mentioned here are also available for tensors on GPU. For large data, GPU operations show better performance compared to those on CPU.

The following sections would describe how these functionalities from the PyTorch library has been exploited in building the MPC-Solver.

5.2 Batched Problems

As stated before, the main goal behind building the MPC-Solver is to solve a huge number of relatively small Quadratic/Linear optimization problems (independent from each other and with a similar structure) in parallel so as to achieve the best performance. Solving the optimization problems in parallel using the Mehrotra's Predictor Corrector Algorithm is accomplished by extending the dimensions of the problem data and hence stacking several problems into a batch on which the required operation can be performed in parallel using functionalities from PyTorch. To illustrate this better, a visual representation of one of the most important steps of the MPC algorithm — the KKT system of equations — for 'N' problems stacked into a batch is shown in Figure 5.1. The ultimate aim is to handle every step of the MPC algorithm on all these problems in parallel just like it is done for a single problem.



Problem N
Problem 1
$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ \Delta z \\ \Delta y \end{bmatrix} = - \begin{bmatrix} Qx + q + G^T z + A^T y \\ Sz \\ Gx + s - h \\ Ax - b \end{bmatrix}$$

Figure 5.1: KKT systems of N optimization problems combined to form a batch

If n_batch corresponds to the batch size (i.e, number of different optimization problems that one would want to solve in parallel using the MPC-Solver), then the MPC-Solver would require the problem data $Q \in \mathbf{S}_{+}^{n\_batch \times n \times n}$, $q \in \mathbf{R}^{n\_batch \times n}$, $G \in \mathbf{R}^{n\_batch \times m \times n}$, $h \in \mathbf{R}^{n\_batch \times m}$, $A \in \mathbf{R}^{n\_batch \times p \times n}$ and $b \in \mathbf{R}^{n\_batch \times p}$ as input tensors. Instead, if it is given an input in two dimensions only, the MPC-Solver assumes that $n\_batch = 1$ i.e, number of problems in the batch is one.

As mentioned in Section 5.1, all basic arithmetic operations that need to performed for each problem can be achieved in parallel using the batch-operations of PyTorch. It is also possible for other mathematical functions like maximum/minimum to be applied along the specified dimensions. This helps it to perform the basic steps of the MPC-algorithm (Section 4.4) like the line search, the update steps of the primal-dual variables, the calculations of duality measure & centering directions etc., in parallel for all the different problems.

However, the step of the MPC algorithm that requires most of the computational effort is in solving the KKT system of equations. There are two steps in each iteration of the MPC algorithm that involves solving the KKT system. This requires computing the inverse (for every problem in the batch) of the KKT matrix and doing batched matrix multiplications to obtain the solution. However, the structure of the KKT matrix allows us to do this in a more efficient way. This will be discussed in detail in Section 5.3.

5.3 Solving the KKT system

The techniques discussed in this section follows Mattingley and Boyd [21] & Amos and Kolter [3].

All the other steps of the MPC algorithm, excluding the solving of KKT systems, are simple arithmetic operations. The only complicated step is the solving of the KKT system of equations and this is done twice in each iteration:

- to find the affine scaling direction,

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{aff} \\ \Delta s^{aff} \\ \Delta z^{aff} \\ \Delta y^{aff} \end{bmatrix} = \begin{bmatrix} -\left(A^T y + G^T z + Qx + q\right) \\ -Sz \\ -(Gx + s - h) \\ -(Ax - b) \end{bmatrix}$$

- and to find the centering-corrector step

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{cc} \\ \Delta s^{cc} \\ \Delta z^{cc} \\ \Delta y^{cc} \end{bmatrix} = \begin{bmatrix} 0 \\ \sigma\mu\mathbf{1} - \operatorname{diag}\left(\Delta s^{aff}\right)\Delta z^{aff} \\ 0 \\ 0 \end{bmatrix}$$

However, it is worth noticing that both these steps of the algorithm have the same coefficient matrix on the LHS. This makes it sufficient to compute the inverse of the KKT matrix only once for the affine step, which can then be reused for the corrector step.

It is also important to note that the coefficient matrix in the KKT system has a special structure. If we have a closer look at the coefficient matrix of the KKT system (4.6), we realize that, all the matrix entries except Z and S (in the second row) remain unchanged across iterations. This can be utilized to improve the efficiency of the solve steps of the KKT system. This matrix can be symmetrized by left-multiplying the whole of the second row with S^{-1} to obtain the following system:

$$\left[\begin{array}{cc|cc} Q & 0 & G^T & A^T \\ 0 & S^{-1}Z & I & 0 \\ \hline G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{array}\right] \begin{bmatrix} \Delta x \\ \Delta s \\ \Delta z \\ \Delta y \end{bmatrix} = \begin{bmatrix} -\left(A^T y + G^T z + Qx + q\right) \\ -z \\ -(Gx + s - h) \\ -(Ax - b) \end{bmatrix} \tag{5.1}$$

Here $S^{-1}Z$ is a diagonal matrix, represented as $D = \text{diag}(d)$ where $d \in \mathbf{R}^m$ is a vector given by $d_i = z_i/s_i, \quad i = 1, \dots m$.

In each iteration of the Mehrotra's Predictor-Corrector algorithm, systems similar to the above with the same coefficient matrix on the left-hand side but different right-hand side would be solved. A solver method that efficiently does this solving procedure by a suitable factorization of the coefficient matrix (once per iteration) would help improve the performance of the algorithm. We have two possible methods for this solve — one using the LU Factorization of the KKT matrix and the second using block elimination techniques with Cholesky decomposition of the sub-matrices of the KKT system.

Before discussing the techniques used to solve the KKT system, let us summarize the important features of the KKT matrix from (5.1):

$$K = \left[\begin{array}{cc|cc} Q & 0 & G^T & A^T \\ 0 & S^{-1}Z & I & 0 \\ \hline G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{array}\right]$$

- K is symmetric, i.e $K^T = K$
- block (1,1) of K is block-diagonal
- block (1,1) of K is positive-semi-definite (since Q is positive semi-definite and $S^{-1}Z$ is a diagonal matrix with positive diagonal entries)
- block (2,2) of K is zero
- block (2,1) is the transpose of block(1,2)
- only the sub-matrix $S^{-1}Z$ in block (1,1) changes across iterations, all others remain the same

5.3.1 LU Factorization

LU Factorization is a commonly used approach in solving a system of linear equations $Ax = b$. The LU factorization factors a matrix A into a product of a lower triangular matrix L and an upper triangular matrix U:

$$A = LU$$

Once the LU factorization is obtained, the system of equations $Ax = b$ can be reformulated as:

$$LUx = b$$

and this new system can be solved in two steps:

1. *Forward Substitution*, to solve $Ly = b$
2. *Backward Substitution*, to solve $Ux = y$

If A is a matrix of size n, then the LU solve of the system $Ax = b$ requires only $(2/3)n^3$ floating-point operations (flops) [1]. This often reduces by a great factor when the matrix A has a specific structure (for example, sparsity).

The `torch.lu()` method from PyTorch does the LU factorization of its input tensor. Also the `torch.lu_solve()` method efficiently solves a system of equation using LU factorization of the coefficient matrix (returned from `torch.lu()`). Both these functions also support the LU factorization and LU solves of tensors on a GPU. This helps to achieve excellent performance when solving many such systems. Thus, the MPC-Solver takes the help of the `torch.lu_solve()` method to solve the KKT system of equations. As mentioned before, the LU factorization of the coefficient matrix needs to be performed only once in each iteration for the two LU solves.

5.3.2 Block Elimination and Cholesky Decomposition

The LU factorization of a matrix attains a special structure when the matrix is symmetric and positive semi-definite. The LU factorization of a symmetric, positive semi-definite matrix A will be of the form:

$$A = LL^T$$

ie, the upper triangular matrix U will be the transpose of the lower triangular matrix L. This factorization is called the **Cholesky Decomposition**. Cholesky Decomposition requires only $(1/3)n^3$ flops compared to LU solves. We note that the KKT matrix, whose factorization we are interested in, is symmetric but not necessarily positive-semidefinite. So we cannot directly solve the KKT system of equations using the Cholesky Decomposition. This is resolved using some additional techniques.

Block elimination is a general technique that is used to solve a system of linear equations $Ax = b$ by eliminating a subset of the variables and then solving a smaller system of linear equations for the remaining variables. This method does not help much when the matrix A does not have any structure. But for example, when the sub-matrices of A associated with the eliminated variables is block diagonal, this method performs substantially better [9].

[1]flops expresses the cost of an algorithm as the total number of floating-point operations it performs in terms of the dimensions of the matrices/vectors involved[9]

Let us look at how block-elimination can be applied to the KKT system of equations (5.1). Let us partition the variables vector into two blocks:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

where $x_1 = [\Delta x, \Delta s]^T$ and $x_2 = [\Delta z, \Delta y]^T$. We conformally partition our KKT system as:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{5.2}$$

Here A_{11}, A_{12}, A_{21} and A_{22} corresponds to the four blocks of the KKT matrix as represented by (5.1). Also $b_1 = [-\left(A^T y + G^T z + Qx + q\right), -z]^T$ and $b_2 = [-(Gx + s - h), -(Ax - b)]^T$. Using (5.2), we can express x_1 in terms of x_2 as:

$$x_1 = A_{11}^{-1}(b_1 - A_{12}x_2) \tag{5.3}$$

Substituting (5.3) into the second equation from (5.2) yields:

$$\left(A_{22} - A_{21}A_{11}^{-1}A_{12}\right)x_2 = b_2 - A_{21}A_{11}^{-1}b_1 \tag{5.4}$$

Equations (5.3) and (5.4) are called *reduced equations*. The coefficient of x_2 is called the **Schur-Complement** S of A_{11}. From Section 5.3, we know that $A_{22} = 0$. This reduces the Schur-Complement to :

$$S = -A_{21}A_{11}^{-1}A_{12}$$

From (5.3) and (5.4), it is evident that we require inverse of A_{11} and S to solve the two systems. This can be done using cholesky decomposition, since:

- A_{11} is positive-semi-definite as explained in Section 5.3
- $-S$ is positive-semi-definite. (From Section 5.3, $A_{21} = A_{12}^T$ and hence $S = -A_{12}^T A_{11}^{-1} A_{12}$ which is negative-semi- definite.)

If $n_1 = n + m$ and $n_2 = m + p$ represents the size of the two blocks of variables x_1 and x_2, then using block elimination to solve (5.2) would require :

$$f_1 + n_2 s + 2n_2^2 n_1 + f_2 \tag{5.5}$$

flops, where f_1 and f_2 corresponds to the the cost of factoring A_{11} and S respectively and s corresponds to the cost solving (5.3) [9]. If the matrices were unstructured and we used the standard LU factorization,(5.5) would reduce to $(2/3)(n_1+n_2)^3$ which is same as solving the larger system of equations using LU factorization without block elimination. However, in our case, f_1 and f_2 reduces by a factor of 2, since we use Cholesky decomposition for factorizing the positive semi-definite matrices A_{11} and $-S$. Thus we combine the techniques of block elimination and Cholesky decomposition to solve the KKT system from MPC algorithm more efficiently.

5.4 Outline of the MPC-Solver

Even though the MPC algorithm is the foundation of the MPC-Solver, we add some techniques to it while implementing it to obtain the best results. Here, we briefly summarize how the MPC-Solver uses the MPC algorithm (Section 4.4) to solve problems efficiently:

- The MPC-Solver receives the input problem data as torch tensors (two/three dimensional). If the tensors are only two-dimensional (i.e, only a single problem is to be solved), the MPC-Solver extends this to three-dimensions for compatibility.
- We add a check on the positive-semi-definiteness of the matrix Q to verify if the problem is a convex QP. If Q is not positive-semi-definite, this throws an error and halts the execution.
- We note that the initialization matrix of the MPC algorithm is slightly different from the KKT matrix for the main iterations. This initial matrix can be easily modified to resemble the structure of the KKT matrix so that we could use the same functions for solving the initial problem and the problems in the subsequent iterations. Therefore, to obtain the initial solution, we solve the following KKT matrix:

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & I & I & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ \Delta z \\ \Delta y \end{bmatrix} = \begin{bmatrix} -q \\ 0 \\ h \\ b \end{bmatrix} \tag{5.6}$$

  This system very closely resembles the KKT system ((5.1)) and can be solved using techniques described in Section 5.3.
- Once we obtain a solution from solving the system of equations given above, we need to update the variables s and z so that they agree to the positivity constraints. This requires a line search for finding parameters α_p and α_d respectively. Since a line search is expensive, we find these parameters by simple arithmetic operations. For example, to find $\alpha_d = \inf\{\alpha \mid z + \alpha\mathbf{1} \geq 0\}$, where $z, \mathbf{1} \in \mathbf{R}^{n\_batch \times m}$ we find the batch-wise maximum of $-z$ to get the step-size $\alpha_p \in \mathbf{R}^{n\_batch}$ for each problem, that can then be used to update z so that all its components turn positive.
- After the initialization of the four variables, we start with the main iterations. We use a default value of 25 for the maximum number of iterations. We also have checks on the duality measure and the residuals in every iteration, to check if the results have already converged to the optimum at an early stage.
- In every iteration, we update the KKT matrix with the current value of s and z. This requires simply updating a block of the KKT matrix with a diagonal matrix $S^{-1}Z$ (as described in Section 5.3). We also perform the factorization of this matrix and solve the first system of equations to obtain the affine scaling directions. The user can demand the solver method to use the LU factorization technique or the

block elimination technique based on the structure of the matrix Q. By default, the MPC-Solver uses LU factorization for the solves.

- The affine-scaling directions are used to compute the new duality measure μ^{aff} and the centering parameter σ. This requires finding a step-size $\alpha \in [0,1]$ that keeps both s and z positive. As we did for initialization, we find the step-size by finding $-z/\Delta z^{aff}$ and $-s/\Delta s^{aff}$ respectively, and then checking for the best possible $\alpha \in [0,1]$ in these values that keeps both s and z positive after the update.
- The next step involves the second solve of the KKT system to obtain the centering-corrector step. This step uses the factorization of the KKT matrix computed for the affine step since both these steps use the same KKT matrix.
- Finally we compute the update steps as the sum of the affine-scaling directions and the centering-corrector steps. The updates are then performed by finding suitable step sizes (uses the same technique described above). These steps are repeated until any of the stopping criteria is met.

5.5 Using the MPC-Solver

The code-base of the MPC-Solver is included in the supplementary material and also available for download at: https://github.com/annechris13/Master-Thesis.

5.5.1 Dependencies

The MPC-Solver package has the following dependencies:

- pytorch >= 1.5.0
- pandas >= 1.0.1
- numpy>= 1.18.1

5.5.2 Example

The MPC-Solver has a user-friendly interface like most of the other solvers. The optimum value and the optimal point of a QP/LP can be obtained on passing the problem data as input. The problem data is expected to be input to the solver as torch tensors (as described in Section 5.2). We include two code snippets in this section. Code Snippet 5.1 solves the QP defined in Section 3.1. In addition to this, Code Snippet 5.2 shows how the MPC-Solver can be used to solve multiple QPs in parallel (and on a GPU).

The code snippet 5.1 solves the QP defined in Section 3.1. Similar to the qpth solver (Section 3.4), we require the problem data to be input to the solver as tensors. Here we simply convert the problem data which are 2D NumPy arrays to 2D tensors.

We initialize the MPC-Solver using the `mpc_class()` function with necessary changes to the default arguments like `max_iter` (maximum number of iterations), `solve_method`

(‘LU’ (default) or ‘BE’(block elimination)) etc. We then invoke the `solve()` method with the problem data. This returns the optimal point and optimal value.

Code Snippet 5.1: Solving QP with MPC

```
#import the mpc module from the source folder where it is located
from solvers import mpc
#convert numpy arrays to tensors
Q_ = torch.tensor(Q);q_ = torch.tensor(q);G_ = torch.tensor(G)
h_ = torch.tensor(h);A_ = torch.tensor(A);b_ = torch.tensor(b)
# intilialize the mpc solver, optionally change max_iter
mpc_solver = mpc.mpc_class(max_iter=20)
#solve the optimization problem and get the optimal point and optimal
   solutions
x, opt_val = mpc_solver.solve(Q_, q_, G_, h_, A_, b_)
#print optimal objective value
print('Optimal Objective Value:',opt_val)
#print optimal variable value
print('Optimal point:',x)
```

Output

```
Optimal Objective Value: tensor( [[[9.2500]]],
                        dtype=torch.float64 )
Optimal Point: tensor( [[0.5000], [1.0000]]],
                        dtype=torch.float64)
```

When solving multiple problems, we require 3D tensors to represent the problem data corresponding to the individual problems. Code snippet 5.2 solves hundred QPs in parallel on a GPU. If a GPU is not available, one can simply ignore the corresponding statements that transfer the tensors to the cuda device. Each problem solved here has 6 inequality and 3 equality constraints. Also the optimization variable has 6 components.

The array ‘seeds’ contains predefined seed values that can be used to randomly generate different QPs that are solvable (generated using other solvers). We first generate the problems one by one and append them to a list, which is finally converted to a 3D tensor.

The output of the Code Snippet 5.2 is omitted since it is lengthy. The `opt_vals` is a tensor of shape `(100,1,1)` that holds the optimum values of the hundred different problems. Similarly, `x` is a tensor of shape `(100,6,1)` that holds the optimal points (each of length 6) of the hundred different problems.

If the residuals & duality measure did not converge for any of the problems, the solver will raise a warning stating the number of problems that did not converge to the optimal solution.

Code Snippet 5.2: Solving multiple QPs with MPC (on GPU)

```
#import the mpc module from the source folder where it is located
from solvers import mpc
#initialize problem sizes
nbatch=100;
nx=6 ; nineq=6 ;neq=3
#initialize lists to store problem parameters
Q=[] ; q=[] ; G=[] ; h=[]; A=[]; b=[]

#iterations to generate the different problems
for i in range(nbatch):
    #generate random problems using seeds
    seed=int(seeds[i])
    random.seed(seed)
    #appends newly generated matrices to the list
    Q.append(make_spd_matrix(nx,random_state=seed).reshape(nx,nx))
    q.append([random.random() for i in range(nx)])
    G.append([random.random() for i in range(nineq*nx)])
    h.append([0 for i in range(nineq)])
    A.append([random.random() for i in range(neq*nx)])
    b.append([random.random() for i in range(neq)])

#convert the list of problem data to 3D tensors
Q=torch.tensor(Q).view(nbatch,nx,nx).type(torch.DoubleTensor)
q=torch.tensor(q).view(nbatch,nx).type(torch.DoubleTensor)
G=torch.tensor(G).view(nbatch,nineq,nx).type(torch.DoubleTensor)
h=torch.tensor(h).view(nbatch,nineq).type(torch.DoubleTensor)
A=torch.tensor(A).view(nbatch,neq,nx).type(torch.DoubleTensor)
b=torch.tensor(b).view(nbatch,neq).type(torch.DoubleTensor)

##########################OPTIONAL!#########################
#transfer the tensors to cuda device
Q, q, G, h, A, b  = [x.cuda() for x in [Q, q, G, h, A, b]]
torch.cuda.synchronize()
###########################################################

#solve using mpc solver
x,opt_vals=mpc_solver.solve(Q,q,G,h,A,b)
#print optimal objective value
print('Optimal Objective Value:',opt_vals)
#print optimal variable value
print('Optimal point:',x)
```

Chapter 6

Results

We talked in great detail about the MPC-Solver in the previous chapters. We also discussed a few other solvers that are widely used for solving optimization problems. We expect the MPC-Solver to be exceptionally faster than these solvers while producing similar (accurate) solutions. In this chapter, we discuss in detail the results from these solvers and judge the performance of the MPC-Solver by comparing it with the other solvers.

In Section 6.1 we talk about the optimization problems that are used to test the performance of the MPC-Solver. In Section 6.2 we discuss the results of the MPC-Solver when it is used for solving QPs. Similarly in Section 6.3 we discuss the results obtained from solving LPs. In Section 6.4 we talk about some relevant differences of the MPC-Solver to that of Gurobi and qpth. Finally, we discuss some details with respect to the memory requirements of the MPC-Solver in Section 6.5.

6.1 Optimization Problems for Testing

To compare the speed and the accuracy of the MPC-Solver with other conventional solvers, we require to test these solvers on a common set of problems. Since the goal of the MPC-Solver is to efficiently solve a huge number of optimization problems, we require a very large set (at least a million) of problems to check if this goal has been achieved.

It is well known that conventional solvers like Gurobi, CVXPY etc. produce very accurate results. Keeping this in mind, we generate random optimization problems and test them on these conventional solvers for its feasibility and boundedness. Those problems that have a unique solution are then used to compare the performance of the MPC-Solver with that of the other solvers. Note that these problems have dense matrices as the problem data since every entry of the problem data matrices are randomly generated.

Let n be the size of the optimization variable, m be the number of inequality constraints and p be the number of equality constraints. We describe below how we generate different LPs and QPs for testing.

QPs

We generate one million QPs each, for the following values for n, m and p:

- $n = 3;\ m = 3;\ p = 1$
- $n = 6;\ m = 6;\ p = 3$
- $n = 10;\ m = 5;\ p = 2$

The problem data is generated randomly as given in code snippet 6.1. In each iteration, we use a pre-defined seed that can generate a solvable QP. The matrix Q is initialized using the `make_spd_matrix()` function from the sklearn python library that generates random positive semi-definite matrices based on a seed.

Code Snippet 6.1: QPs Generation

```
random.seed(s)
Q=make_spd_matrix(n, random_state=s)
q=np.array([random.random()*10 for i in range(n)]).reshape(n)
G=np.array([random.random()*10*((-1)**random.randint(0,1)) for i in
   range(m*n)]).reshape(m,n)
h=np.array([0 for i in range(m)]).reshape(m)
A=np.array([random.random()*10 for i in range(p*n)]).reshape(p,n)
b=np.array([random.random()*10 for i in range(p)]).reshape(p)
```

These problems are solved one after the other with the help of Gurobi and CVXPY. For MPC-Solver and qpth, the problems are stacked into a single batch and solved in parallel. We perform the solves on batches of the following sizes: [100,1000,10000,100000, 500000, 1000000] and stores the results and run-times. We also store all the results and runtimes (for solving the number of problems corresponding to the batch sizes) from Gurobi and CVXPY. The results are discussed in Section 6.2

LPs

We generate 1 million LPs each, with the following values for n, m and p:

- $n = 3;\ m = 3;\ p = 1$
- $n = 6;\ m = 6;\ p = 3$
- $n = 5;\ m = 5;\ p = 2$

The problem data is generated randomly as given in code snippet 6.2. In each iteration, we use the pre-defined seed that can generate a solvable LP. The matrix Q is an identity matrix with very small values on the diagonal (approx 0). This ensures that the problems solved are LPs. Setting Q exactly to zero is not recommended since this might lead to errors when computing the inverse of the KKT matrix with Q as a sub-matrix. Solvers like Gurobi and CVXPY does not even use the matrix Q.

Code Snippet 6.2: LPs Generation

```
random.seed(s)
Q=torch.eye(n)*1e-5
q=np.array(np.random.randint(-5,5,n)).reshape(n)
G=np.array(np.random.randint(-5,5,n*m)).reshape(m,n)
h=np.array([0 for i in range(m)]).reshape(m)
A=np.array(np.random.randint(-5,5,n*p)).reshape(p,n)
b=np.array(np.random.randint(-5,5,p)).reshape(p)
```

Similar to the QPs, these LPs are solved one after the other on Gurobi and CVXPY, and in parallel on MPC-Solver and qpth. Also, the following batch sizes are compared: [100,1000,10000,100000, 500000, 1000000]. The results are discussed in Section 6.3

The computations are performed on a CPU/GPU with the following specifications:

- CPU: Intel(R) Xeon(R) Silver 4116 Processor, 192GB RAM
- GPU: Nvidia GeForce 2080Ti, 11GB VRAM

6.2 Performance on QPs

In this section, we compare the performances of the different solvers when solving 1 million QPs that are generated as defined in Section 6.1.

Table 6.1 summarizes the solve-times (in seconds) of these solvers for solving a given number of problems (denoted by Batch Size), corresponding to each different problem

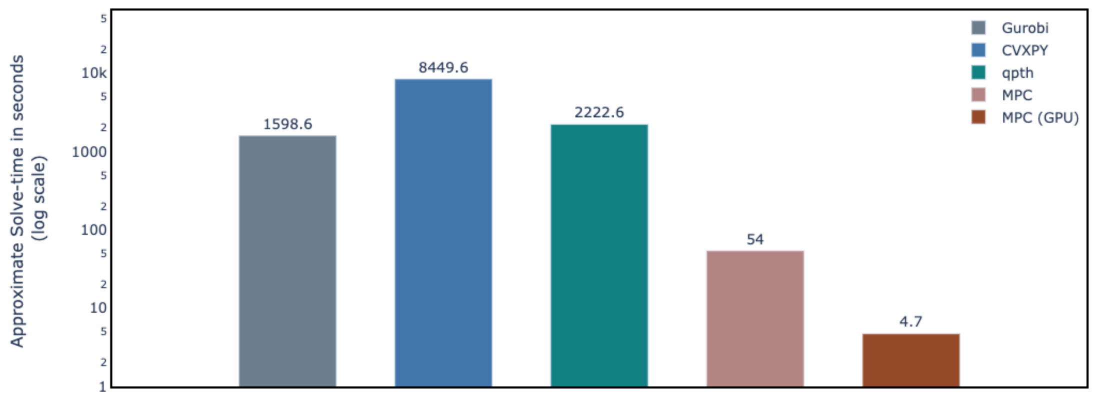


Figure 6.1: Approximate solve-time (in seconds) of 1 million QPs ($n = 3, m = 3, p = 1$) by different solvers

setting. Empty values in the table denoted by '—' correspond to scenarios where the solves could not be performed due to memory overflows. This is discussed in Section 6.5.

Figure 6.1 provides a visual representation of a particular case ($n = 3$, $m = 3$, $p = 1$), Batch Size = 1000000 mentioned in table 6.1 (Note that the y-axis is on log-scale).

Batch Size	**Gurobi** (sec)	**CVXPY** (sec)	**qpth** (sec)	**MPC** (sec)	**MPC** (GPU) (sec)
			n = 3, m = 3, p = 1		
100	0.2	0.9	0.3	0.02	0.1
1000	1.6	8.2	1.7	0.1	0.1
10000	15.8	83.5	15.3	0.8	0.2
100000	160.5	849.9	229.2	6.2	0.5
500000	799.9	4232.2	930.1	28.2	2.3
1000000	1598.6	8449.6	2222.6	54.0	4.7
			n = 6, m = 6, p = 3		
100	0.2	0.9	0.4	0.1	0.1
1000	1.9	8.6	2.3	0.3	0.1
10000	18.8	84.7	19.8	3.5	0.2
100000	184.9	840.7	197.0	32.9	1.1
500000	930.0	4226.4	1124.0	168.5	—
1000000	1857.3	8469.1	2377.4	353.3	—
			n = 10, m = 5, p = 2		
100	0.2	0.9	0.3	0.1	0.1
1000	2.2	9.0	2.2	0.4	0.1
10000	21.7	90.4	20.0	3.7	0.2
100000	215.6	901.2	254.1	36.8	1.2
500000	1047.1	4314.6	1185.3	178.0	—
1000000	2238.5	9364.3	2434.9	380.7	—

Table 6.1: Solve-time Comparison of Different Solvers for QPs

Clearly, we see that the MPC-Solver outperforms all the other solvers. The slowest among all the 4 solvers is the CVXPY default solver which takes around 8450 seconds ($\approx$ 2.5 hours). Gurobi performs better than this, solving 1M QPs in roughly 1599 seconds ($\approx$ 26 minutes), while qpth does this in around 2222 seconds ($\approx$ 37 minutes). The MPC-Solver performs extra-ordinarily well by computing the solutions in just 54 seconds (less than a minute). On GPU, it performs even better, by solving 1M problems in just 5 seconds.

For further settings, we see changes in time required by each solver, but the order of performance remains the same. For Gurobi and CVXPY, the time required does not seem to be largely affected by the problem size. Whereas MPC-Solver seems to take longer to solve larger problems. This can be associated to the large KKT systems corresponding to the huge problem data that needs to be solved as a part of the MPC algorithm. Still, we see that the MPC-Solver performs much better compared to the others.

Next, we compare the accuracy of these solvers. Since we tested these solvers on random problems, it is difficult to judge which among the solvers produced accurate solutions. Since Gurobi is known to be a very accurate solver for years, we assume all the results produced by Gurobi to be 100% accurate. The results from CVXPY,qpth and MPC are compared to the results produced by Gurobi.

We compare the results using two different methods. We define the 'Accuracy %' of the solvers as the percentage of solutions (rounded to 4 decimal places) that are exactly equal to those from Gurobi. We also compute the average 'Root Mean Squared Error (RMSE)'[1] of the solutions from the three solvers with respect to the solutions from Gurobi.

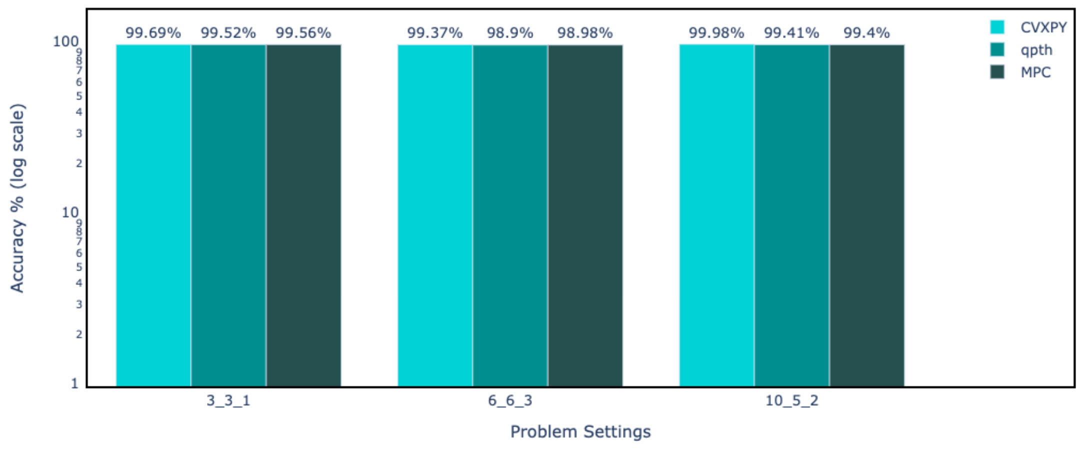


Figure 6.2: Accuracy of solvers with respect to Gurobi for solving QPs with different values of n_m_p given in Table 6.1

Figure 6.2 compares the 'Accuracy %' of CVXPY, qpth and mpc with respect to Gurobi for all the 1M problems generated for the 3 different settings mentioned in Table 6.1. We see that all the solvers return results that are almost always accurate.

Figure 6.3 compares the 'RMSE' of solutions from CVXPY, qpth and mpc with respect to those from Gurobi for all the 1M problems generated for the 3 different settings mentioned in Table 6.1. We see that errors from all the solvers are very small. However, we see in Section 6.4.1 that the results form the qpth solver is not always consistent.

[1] $RMSE = \sqrt{\sum_{i=1}^{n} \frac{(sol_1 - sol_2)^2}{n}}$ where n is the length of the optimization variable, sol_1 & sol_2 are the solutions form the two solvers considered for comparison

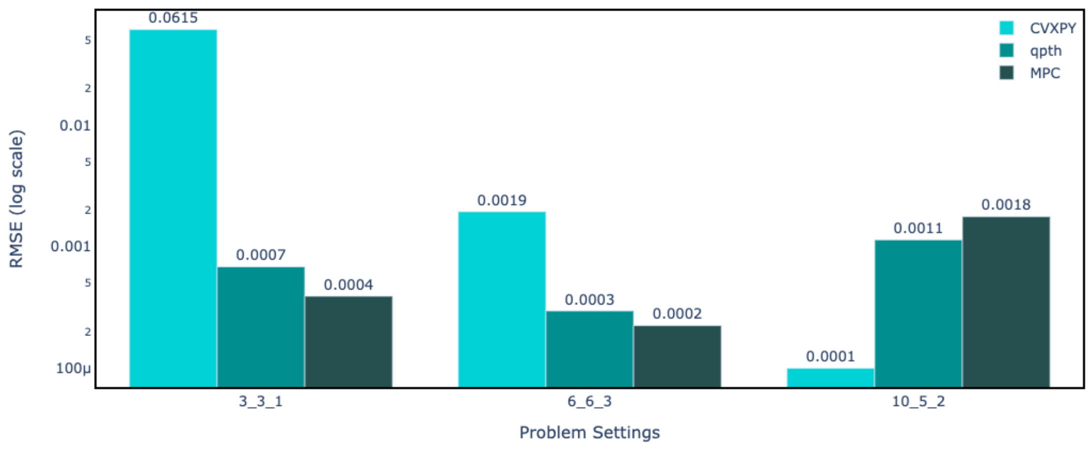


Figure 6.3: RMSE of solutions from different solvers with respect to solutions from Gurobi for QPs with different values of n_m_p given in Table 6.2

6.3 Performance on LPs

In this section, we compare the performance of the different solvers in solving 1 million LPs under the problem settings defined in Section 6.1.

Table 6.2 summarizes the solve-times (in seconds) of these solvers in solving a given number of problems (denoted by Batch Size), corresponding to each different problem setting. Figure 6.4 provides a visual representation of a particular case ($n = 3$, $m = 3$, $p = 1$), Batch Size $= 1000000$ mentioned in the table (Note that the y-axis is on log-scale).

As we saw for the case of QPs, the MPC-Solver outperforms all the other solvers. Even for LPs, the slowest among all the 4 solvers is the CVXPY default solver. We see

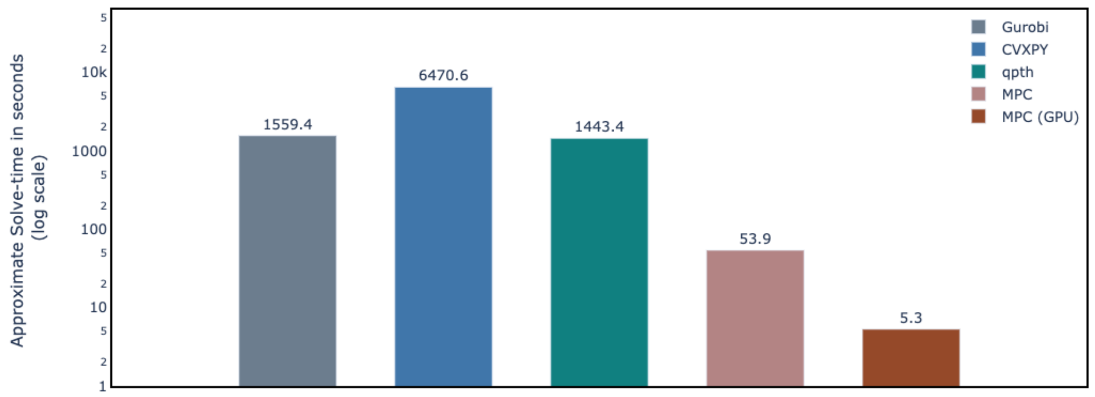


Figure 6.4: Approximate solve-time (in seconds) of 1 Million LPs ($n = 3, m = 3, p = 1$) by different solvers

similar patterns of performance (computation times) to that of the QP solves mentioned in Section 6.2. We see that in all cases MPC-Solver on GPU is the fastest, followed by MPC-Solver on CPU. qpth and Gurobi show competitive solve-times, while CVXPY is always the slowest.

Batch Size	**Gurobi** (sec)	**CVXPY** (sec)	**qpth** (sec)	**MPC** (sec)	**MPC**(GPU) (sec)
			n = 3, m = 3, p = 1		
100	0.2	0.6	0.2	0.03	0.1
1000	1.5	6.3	1.6	0.1	0.1
10000	15.3	63.1	15.1	0.7	0.1
100000	156.7	649.4	144.8	4.7	0.5
500000	785.2	3257.1	728.0	27.5	2.6
1000000	1559.4	6470.6	1443.4	53.9	5.3
			n = 6, m = 6, p = 3		
100	0.2	0.8	0.5	0.1	0.1
1000	1.7	7.8	3.4	0.3	0.1
10000	16.4	75.6	18.8	3.9	0.2
100000	162.6	750.8	206.9	33.7	1.0
500000	809.7	3736.4	1351.6	164.9	—
1000000	1623.4	7500.6	2594.0	355.9	—
			n = 5, m = 5, p = 2		
100	0.2	0.7	0.3	0.04	0.1
1000	1.6	7.2	1.8	0.2	0.1
10000	16.0	70.7	16.9	1.5	0.2
100000	159.7	704.8	241.9	15.6	0.8
500000	797.0	3516.7	1037.6	82.8	—
1000000	1584.7	7001.1	2288.3	181.1	—

Table 6.2: Performance Comparison of Different Solvers for LPs

We see the accuracy results in Figure 6.5 and Figure 6.6. Although all the solvers seem to produce similar results in most cases, the accuracy of qpth and mpc seems to be lesser, when compared to the case of solving QPs. This is probably because of the small error introduced by the matrix Q since it uses the same solver algorithm for both QPs and LPs with $Q \approx 0$. A possible solution to this problem is mentioned in Chapter 8.

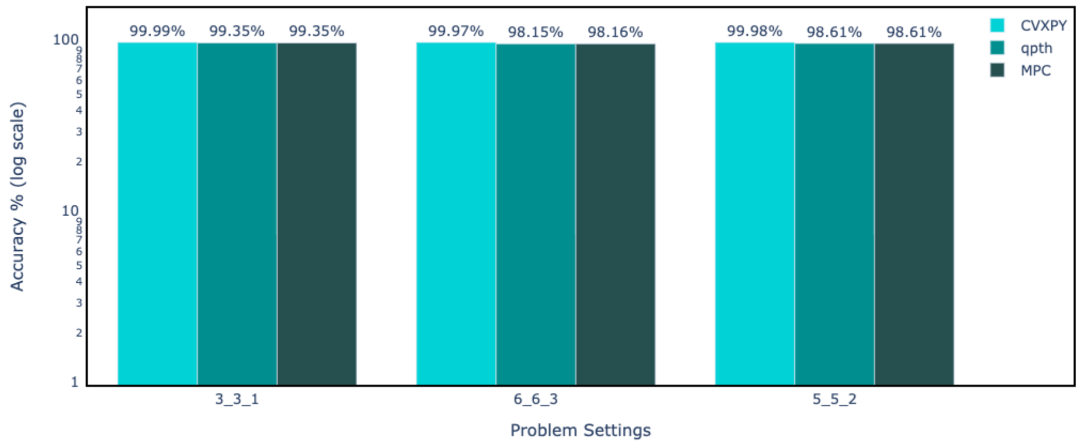


Figure 6.5: Accuracy of solvers with respect to Gurobi for solving LPs with different values of n_m_p given in Table 6.2

6.4 Other Remarkable Differences

We have already looked at the performance and accuracy of the MPC-Solver with respect to few other solvers. Apart from these, we would like to make some additional remarks on the MPC-Solver in comparison to some of these conventional solvers. We discuss this in the following two sub-sections.

6.4.1 Consistency of Solutions

We saw from Chapter 3 that qpth was the only one solver other than the MPC-Solver, that could solve problems in parallel. From Section 6.2 and Section 6.3 we saw that qpth

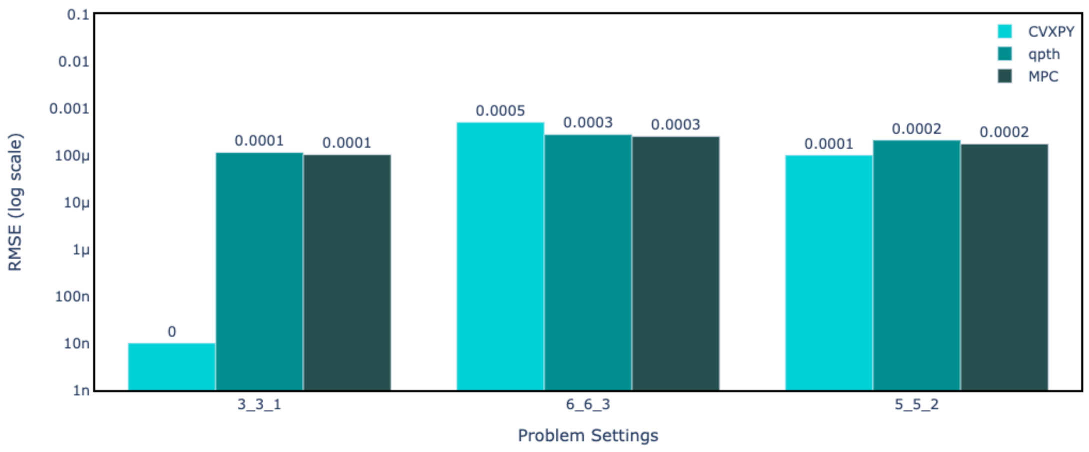


Figure 6.6: RMSE of solutions from different solvers with respect to solutions from Gurobi for LPs with different values of n_m_p given in Table 6.2

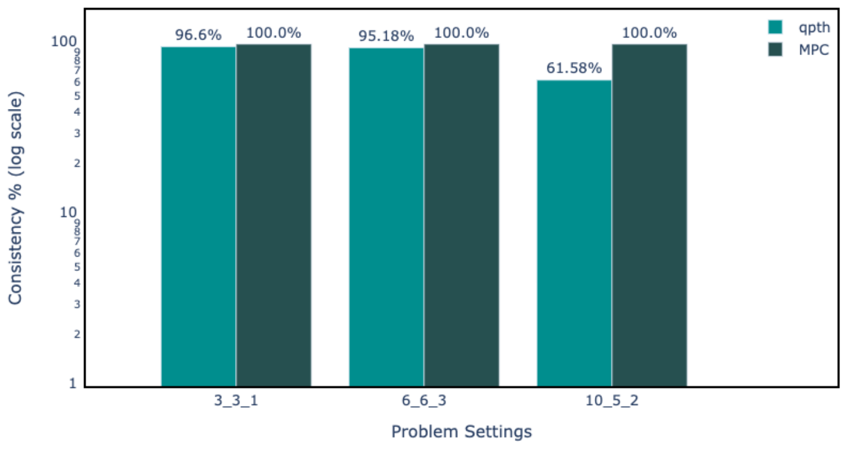


Figure 6.7: Consistency of solutions when solved in parallel vs. serial for different values of n_m_p given in Table 6.1

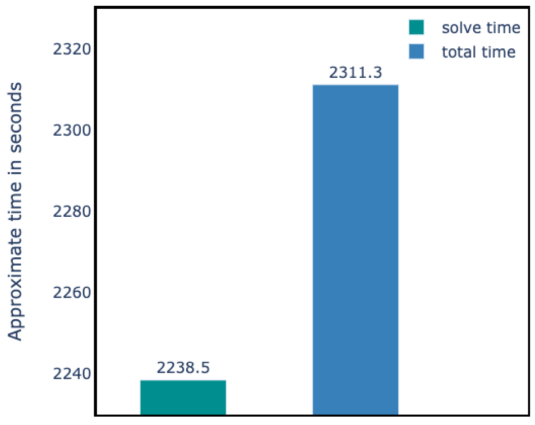
provides solutions that are satisfactory (in terms of accuracy). However, it has been observed that the solutions from the qpth solver are not always consistent. We ran a test, solving ten thousand QPs both serially and in parallel using qpth. The results showed that approximately only 61% of the solutions obtained from solving the problems serially matched the solutions obtained by solving them in parallel for the setting $n = 10$, $m = 5$ and $p = 2$.

Following this inconsistency of the qpth solver, we did the same test on the MPC-Solver. The MPC-Solver showed consistent results where all of the solutions matched. These results for all the different problem settings are summarized in Figure 6.7.

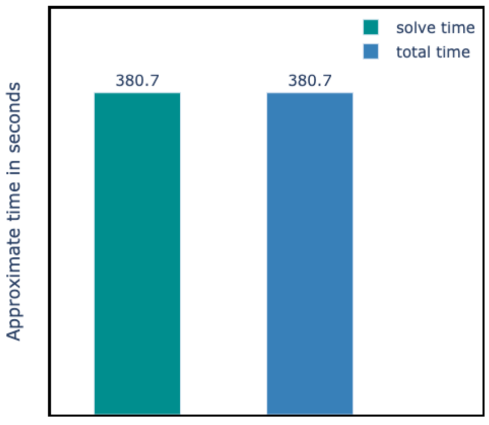
6.4.2 Solve Time vs. Total Time

We have primarily considered Gurobi, because of its accurate results. We saw the approximate times that Gurobi would take to solve a problem in Section 6.2 and Section 6.3. However, in practice, it takes more time to obtain the results from the Gurobi Solver. This is because Gurobi returns an object that is packed with many details including the optimal points, its variable names, optimal solution etc. Retrieving these values from the resulting Gurobi object takes more time compared to those from any other solver. This is problematic when the number of problems (and hence the number of solutions to be retrieved from the object) is large.

We performed a comparison of the solve time and total time (time to solve problems & retrieve solutions to a NumPy array) of Gurobi and the MPC solver. The results are shown in Figure 6.8 . We see that the total time taken by Gurobi is higher than the time it takes to simply solve the problem. Whereas, the MPC-Solver has comparable solve time and total time. This is because the MPC-Solver returns the optimal point and optimal solution as separate tensors which can easily be converted to NumPy arrays.



(a) Gurobi



(b) MPC

Figure 6.8: Comparison of solve-time and total-time of Gurobi (a) and MPC (b) for obtaining solutions of 1 Million QPs

6.5 Memory Requirements

Potentially, we would want to solve the maximum number of problems in parallel. However, it is important that one needs to take care of the memory requirements while forming a batch of problems. The total memory requirement depends obviously on the number of problems and the size of the individual problem data. Most of the memory we use is to store the problem data and for the factorization of the KKT matrix in each iteration. Solving the KKT system also requires a part of the memory. This becomes more important when we are working with a limited memory environment. For example, when working with a GPU, it is often likely that there are restrictions on memory usage. This is because GPUs work on dedicated VRAMs and can't access the computer's RAM directly. As we see in Table 6.1, we could not solve 1 million problems on a GPU because of insufficient memory. We recommend users to be aware of this fact and decide the batch sizes accordingly. A possible solution to this problem is discussed in Section 8.3.

Chapter 7

Application to Auction Theory

Auction theory is a branch of economics that studies auction markets, with a focus on developing auction designs (or set of rules), assessing the efficiency of these designs, finding optimal and equilibrium bidding strategies etc. Combinatorial Auctions is an important branch of Auction Theory. In this chapter, we mainly discuss a quadratic programming technique used in computing combinatorial auctions put forward by Robert W. Day and Peter Cramton [12] and the relevance of the MPC-Solver in this discipline.

In Section 7.1 we discuss auctions and introduce the topic of Combinatorial Auctions. We also state the winner determination problem and discuss the different pricing rules including the 'nearest-VCG' rule put forward by Day and Cramton [12]. We also describe the optimization problem formulations that they introduced to solve the payment determination problem. We then talk about a specific combinatorial setting that we are interested in, the LLLLGG setting, in Section 7.2. Finally, in Section 7.3, we discuss the implementation and compare results from the two solvers — Gurobi and MPC-Solver — for solving the optimization problems for payment determination, hence drawing conclusions on improved performance achieved with the help of the MPC-Solver.

7.1 Combinatorial Auctions

In simple terms, an auction is a process of buying and selling goods or services by offering them up for bid, taking bids, and then selling the item to the highest bidder or buying the item from the lowest bidder. However, auctions can take many forms, each characterized by different rules. In most auctions, the bidders usually know their personal benefit/valuation of the item being bid upon, which is unknown to the other bidders and the seller. Therefore auctions are often considered as Bayesian games with incomplete information [22]. Auctions have a long history and they are still used today in many real-world scenarios, including the sale of radio spectrum licenses [11], the procurement of industrial goods [28], the allocation of TV ad slots [17] and many more.

Combinatorial Auctions (CA) are auctions for multiple items in which bidders submit bids on combinations/packages of items. CAs are also referred to as “package auctions” or auctions with “package bidding”[12]. CAs allow allocation of multiple, indivisible goods to multiple bidders. This allows bidders to express complex preferences on the bundles of goods, taking into account that goods can be complements or substitutes[11].

However, package bidding often puts forward great challenges to both the bidders and the bid-taker. For example, if there are n items available, then the bidders can bid on 2^n different packages. In practice, the bidders cannot enumerate all these different packages. Also, the auctioneer cannot accept this full set of package bids from each bidder, and so limits the number of package bids they will accept. The bidders thus face the difficulty of deciding which are the “best” packages to bid on, in addition to the problem of deciding their value for any single package [12].

7.1.1 Winner Determination Problem

One of the main goals of CAs is in identifying the maximizing assignment of the packages given the bids while ensuring maximum social welfare (efficiency). This is called the **winner-determination problem**. Following is the mathematical formulation of this problem as stated in [12].

Let $M = \{1, 2, \ldots, m\}$ represent the set of m items being auctioned and $N = \{1, 2, \ldots, n\}$ represent the set of n bidders. Assume that each bidder has submitted a collection of bundle bids, with $b_j(S)$ representing bidder j ’s monetary bid on any bundle $S \subseteq M$. The efficient winner determination problem over the set of bidders N is defined by the following integer program, which maximizes the value of the accepted bids :

$$\begin{aligned} wd(N) = \max & \sum_{j \in N} \sum_{S \subseteq M} b_j(S) \cdot x_j(S) \\ \text{subject to } & \sum_{S \supseteq (i)} \sum_{j \in N} x_j(S) \leq 1, \quad \forall i \in M, \\ & \sum_{S \subseteq M} x_j(S) \leq 1, \quad \forall j \in N \\ & x_j(S) \in \{0, 1\}, \quad \forall (S, j), \text{ such that a bid } b_j(S) \text{ was submitted} \end{aligned} \tag{7.1}$$

(7.1) is a LP with exponential problem size given by $n2^m$ (n bidders with 2^m bids). $x_j(S)$ is a binary variable that states whether a bundle S is assigned to a bidder j or not. The first constraint ensures that the same item is not sold to more than one bidder. The second constraint implies that no two bids made by the same bidder may be accepted by the auctioneer.

7.1.2 Payment Determination

Another critical element of any CA is the pricing rule, which determines the payment that each winner makes for the package won. Most common pricing rules are the first-price and second-price rules.

With the **first-price rule**, bidders submit a sealed bid for the single item being auctioned, and the highest bid wins the item and the winner pays the amount of their bid. The first-price rule is simple, but not incentive-compatible (see below for definition).

Similar to the first-price rule is the **second-price rule**, with the difference being that the winner pays the second-highest bid for the item. Essentially, the first-price rule follows a pay-as-bid format, whereas the second-price rule corrects the winner's bid to what they should have bid to just tie the bid of the next highest bidder if they had known how much that was. The following are a few important properties of the second-price rule [12]:

1. Individual rationality: each bidder expects a non-negative payoff for participating.
2. Efficiency: the highest valued bid wins.
3. Dominant strategy incentive-compatibility: misreporting one's value for the item(s) never gives an advantage.
4. The "core" property: no coalition (a subset of all players) can form a mutually beneficial renegotiation among themselves.

Unfortunately, second-price rules cannot be applied to CAs. The **VCG mechanism** is its generalization for CAs that satisfies all the properties given above, except the core-property. The **nearest-VCG** rule, implemented as QPs, finds payments that are closest to the VCG-payments but obeys the core-property. Rather than having incentive-compatibility as a constraint, nearest-VCG only minimizes the deviation from this property. It is important to note that only the VCG mechanism fulfils properties 1, 2 and 3 given above. In the following sub-sections, we explain in detail the VCG mechanism and then the nearest-VCG mechanism for payment determination in combinatorial settings.

VCG Mechanism

The Vickrey auction, also known as the Vickrey-Clarke-Groves or VCG mechanism, implements the efficient solution described by (7.1) and each winning bidder j receives a discount from her winning bid amount, equal to $wd(N) - wd(N\backslash\{j\})$. This induces a bidder to bid honestly. Unfortunately, VCG mechanism has a few drawbacks, one among which is that it doesn't always follow the core-property that says no coalition can form a mutually beneficial renegotiation among themselves. In the case of an auction, this simply means that the seller would not prefer to ignore the outcome dictated by the auction and renegotiate with a subset of the bidders[12]. This property is however satisfied by the second-price sealed-bid auctions for single item.

A simple example (from [6]) of the VCG mechanism and its inability to satisfy the core-property is illustrated here. Consider a two-item auction for items A, and B, with bids by three bidders $b_1(A) = 2$, $b_2(B) = 2$ and $b_3(A, B) = 2$,. The winning bidders will be 1 and 2. From the VCG rule, here bidder 1 and bidder 2 needs to pay only $b_1(A) + b_2(B) - b_3(A, B)$. Thus the VCG payments are both zero for winning bidders 1 and 2, despite a competing bid by bidder 3 on these items. This shows that core property is not upheld by the VCG auction. Here, both the seller and bidder 3 would prefer to renegotiate for both items at any price in the open interval (0,2).

Nearest-VCG Rule

Day and Cramton [12] contributes "core-selecting mechanisms" or "auctions with core pricing" that provide a usable generalization of the second-price sealed-bid auction paradigm to the combinatorial setting. Let us shortly discuss the additional notations and also an example problem (from [12]) that could help us understand this mechanism better.

Let payment vector $p \in R_+^n$ represent the non-negative vector of payments for each bidder, and let $\pi_j = b_j(S_j) - p_j$ represent the observable surplus or profit experienced by bidder j when the auction awards bidder j set S_j. Also, let $\pi_0 = \sum_{j \in N} p_j$ be the profit of the seller.

An outcome is represented by a feasible solution to problem (7.1), which is specified by the set of awarded (possibly empty) bundles $\{S_j\}$ for each bidder and a payment vector p, thus inducing a profit vector π. An outcome is said to be blocked by coalition $C \subseteq N$ if there is some alternative outcome with awarded bundles $\{\bar{S}_j\}$ and payments $\bar{p}$, such that $\bar{\pi}_j = b_j(\bar{S}_j) - \bar{p}_j \geq \pi_j$ for all $j \in C$ and for which $\bar{\pi}_0 = \sum_{j \in C} \bar{p}_j > \pi_0$. An outcome that is not blocked in this context is said to be in the core with respect to the submitted bids.

Let us try to understand the mechanism better through a simple example [12]. Consider $m = 2$ items, A and $B, n = 5$ bidders, and let bids be as follows.

$$b_1(A) = 28, \quad b_2(B) = 20, \quad b_3(AB) = 32, \quad b_4(A) = 14, \quad b_5(B) = 12$$

Note that here each bidder submits only one bid each, which is not always the case. Here, it is easy to see that the unique winners determined by (7.1) are bidders 1 and 2 and the maximum possible total value is 48. Also we see that the VCG payment of bidder 1 is $p_1^{\mathrm{VCG}} = (48 - (20 + 14)) = 14$ and $p_2^{\mathrm{VCG}} = (48 - 32) = 12$. The core formed here is shown graphically in Figure 7.1.

We see that here, the constraints defining the core are simply the bids of the losing bidders (this is not always the case). In particular, the bidder 4 would always object (block) if the bidder 1 paid less than 14 for item A, and hence we have the constraint $p_1 \geq 14$. Similarly, bidder 5 blocks if bidder 2 payed less than 12 and hence the constraint $p_2 \geq 12$. Bidder 3 would object if bidders 1 and 2 together pays less than his bid on the two items together and thus $p_1 + p_2 \geq 32$. Upper bounds on payments are given by the bids themselves.

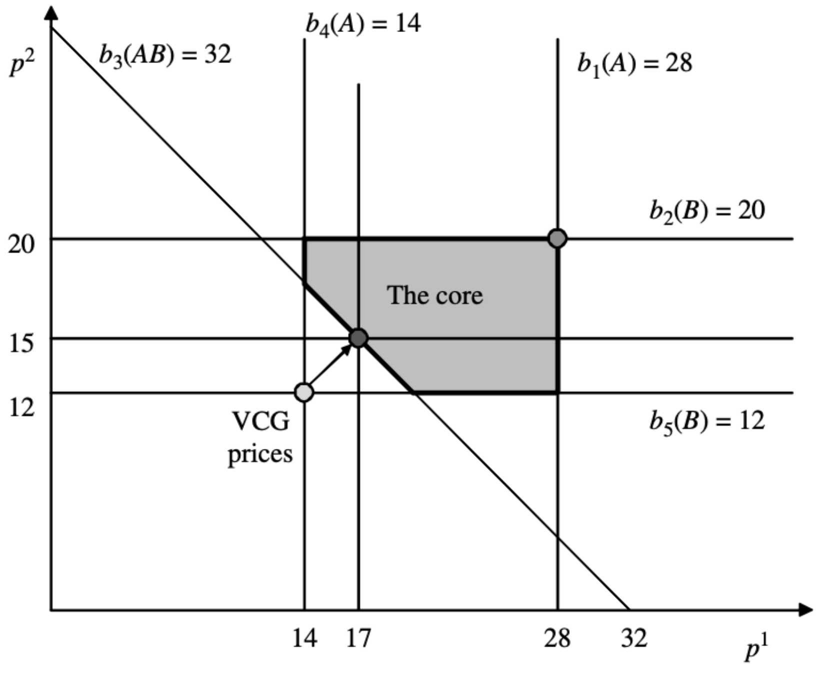


Figure 7.1: The core VCG payments(Source: Day & Cramton [12])

In this case, we see that the VCG payment $(14, 12)$ is not in the core. From the constraints defined above, bidder 3 alone forms a blocking coalition. Using the technique of Day and Raghavan [13], one can guarantee bidder-optimality by minimizing total payments over the core. For our Example problem, we could determine any payment vector on the line segment connecting the point $(14, 18)$ to $(20, 12)$, any of which will not be blocked and hence in the core. Day and Cramton [12] finds a lack of precision in the Day and Raghavan [13] algorithm as it does not specify which of these bidder-optimal points should be chosen. Motivated by the observation of Parkes et al. [27] that says that the difference between a final payment and the VCG payment represents a measure of "residual incentive to misreport," and so should be minimized, Day and Cramton [12] proposes the following refinement to the procedure: "over all the total-payment minimizing core points, select the one that minimizes the sum of square deviations from the VCG payment point". This is equivalent to minimizing the positive square root of this amount, and hence the core point selected as payment vector is that which has minimum Euclidean distance from VCG (formulated as a QP). This rule is referred to as the **VCG-nearest** or **Vickrey-nearest rule**.

Core Formulations and Quadratic Rules for Payment Determination

Usually a divide-and-conquer approach is used, first solving the winner-determination problem and then computing core payments once a particular set of winning bundles $\{S_j\}$ has been determined. If W denotes the set of winning bidders who win non-empty bundles in the winner determination problem, then we require the payment vectors to obey [12]:

$$\sum_{j\in W} p_j \geq wd(C) - \sum_{j\in C} \left(b_j\left(S_j\right) - p_j\right) \quad \forall C \subseteq N \tag{7.2}$$

Here the right-hand side reflects what coalition C is willing to offer to the seller at payment vector p; they will offer as much as can be obtained from them as a group, $wd(C)$, minus the profit they are already making at payment vector p, which is $\sum_{j\in C}\left(b_j\left(S_j\right) - p_j\right)$ [12]. We can segregate decision variables and constants on their respective sides of the inequality by simply canceling payment terms appearing on both sides. This yields :

$$\sum_{j\in W\setminus C} p_j \geq wd(C) - \sum_{j\in C} b_j\left(S_j\right) \quad \forall C \subseteq N \tag{7.3}$$

Day and Cramton [12] proposes the use of this formulation for the actual computations of core prices, which is found by quadratic optimization over the core. They define the quadratic optimization problem as given below. Consider:

$$\beta_C = wd(C) - \sum_{j\in C} b_j\left(S_j\right),$$

If β, denotes the vector of all such β_C, then 7.3 can be written more compactly as:

$$Ap \geq \beta$$

where each a_C is the characteristic vector of the complementary set of winners. (That is, the j th entry in a_C equals 0 if bidder j is in set C and equals 1 if bidder j is not in C. Since non-winners never pay, the dimension of each a_C is $|W| \times 1$, rather than $n \times 1$.) The core-selection region is defined by these constraints as well as the individual rationality constraints that says each winner pays an amount less than or equal to her bid:

$$p \leq b$$

where each component b_j in the vector b is given by $b_j = b_j\left(S_j\right)$

Motivated by the works of Erdil and Klemperer [16] and Day and Raghavan [13], Day and Cramton [12] presents a class of algorithm for core-selection based on quadratic programming by employing rules that are referred to as reference rules. These rules determine the payment vector by minimizing the Euclidean distance to a reference vector of prices. A p^0 -reference rule finds final payments p^* that minimize the sum of squared deviations from payment reference point p^0. Also, the MRC -reference rules limits the feasible set of payments to those core points that minimize total revenue, referred to as the minimum revenue core or MRC. We first find minimal core payments by solving the LP:

$$\mu = \begin{array}{ll} \text{minimize} & \mathbf{1}^T p \\ \text{subject to} & A\,p \geq \beta \\ & p \leq b \end{array} \tag{7.4}$$

Then determine final payments p^* as the optimal solution to the following QP :

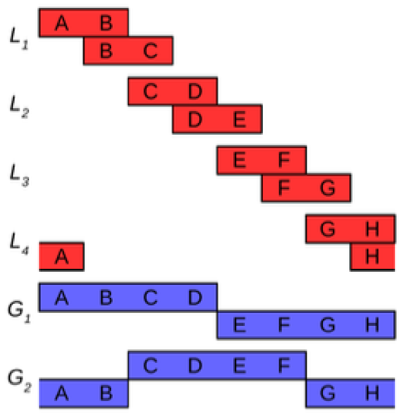
$$\begin{array}{ll} \text{minimize} & \left(p - p^0\right)^T \left(p - p^0\right) \\ \text{subject to} & A\,p \geq \beta \\ & p \leq b \\ & \mathbf{1}^T p = \mu \end{array} \tag{7.5}$$

These are the set of optimization problems that we would want to solve with the MPC-Solver. As previously mentioned, in a combinatorial auction, there are many possible combinations of bids that the bidders can bid upon and the auctioneers can choose as winning bids. In a practical scenario, this leads to a requirement of solving thousands of such optimization problems. The MPC-Solver has been tested to solve these problems for the LLLLGG setting described in the following section.

7.2 The LLLLGG setting

Bidder	Bundle 1	Bundle 2
L_1	AB	BC
L_2	CD	DE
L_3	EF	FG
L_4	GH	HA
G_1	ABCD	EFGH
G_2	CDEF	GHAB

(a) Bundles of interest of each bidder.



(b) Graphical representation of bundle overlap.

Figure 7.2: The multi-minded LLLLGG domain (Source: Bosshard, Vitor and Bünz, Benedikt and Lubin, Benjamin and Seuken, Sven[8])

The Local-Local-Global (LLG) domain (Ausubel et al. [7]) is one of the smallest and most widely-studied examples of an auction where combinatorial interactions between bidders arise. There are two local bidders, each of whom is interested in a different single good, and a global bidder who is interested in the package of both goods. A closed-form solution to obtaining core-price rules in this setting already exists [5]. However, a bigger and more complex version of this setting is introduced in [8] called the LLLLGG setting. This domain has six bidders (L1-L4, G1-G2) and eight goods (labelled A$-$H), with each bidder being interested in two bundles as shown in Figure 7.2. There are four "local" bidders L1 - L4 who draw each of their two bundle values from $\mathcal{U}[0,1]$, and two "global" bidders G1 and G2 who draw their two bundle values from $\mathcal{U}[0,2]$; all draws are independent. For the global bidders, the two bundles are perfect substitutes (even though they are not overlapping), such that a bidder will never want to win both bundles at the same time. For the local bidders, their bundles of interest are partially overlapping such that they can obviously only ever win one of them[8].

7.3 Payment Determination for the LLLLGG setting

Ongoing research by Stefan Heidekrüger, Paul Sutterer, Nils Kohring and Martin Bichler [29] proposes the Neural Pseudogradient Ascent (NPGA) as an equilibrium learning method that uses gradient dynamics to learn equilibrium strategies using iterative machine learning methods. Equilibrium learning in games differs from most learning tasks in that it suffers from the non-stationarity problem: Each player's objective depends on other agents' actions [29]. For the learning purpose, NPGA requires computing the expected utility (via Monte Carlo Sampling) in a given strategy-profile (decision vector comprising decisions of all bidders) over all possible inputs. NPGA is evaluated on the local-global auctions for benchmarking the results.

In the LLLLGG setting, the different possible valid package bids on the 6 items by the 8 bidders are enumerated. Combinatorial auctions are then performed on each of these input combinations to learn the utility of each strategy profile. The payment vectors in the LLLLGG setting is computed according to the nearest-VCG rule. In this Neural-Network setting, each experiment is repeated 10 times over 5,000 iterations each, where each iteration requires the computation of $P(n+1)$ auctions, where n is the number of bidders ($n = 6$ in the LLLLGG setting) and P is a perturbation parameter (with typical size 64 or 128). This gives rise to the requirement of solving a huge number of optimization problems — LPs (7.4) followed by QPs(7.5) — as discussed in Section 7.1.2. The optimization problems are individually relatively small with the problem variable having a length of 6 corresponding to the number of players and the number of constraints below 100. Ultimately, we would want to perform these large number of optimization problems in each iteration in parallel for better performance. This is where we find the possibility of taking advantage of the MPC-Solver for performing fast solves of these huge number of optimization problems.

Since an implementation that uses the Gurobi solver for solving these optimization problems was available, we could easily test for the accuracy and performance of the MPC-Solver using the results from Gurobi. We first compare the time required by both the solvers for solving problems of different batch sizes (denoting the total number of optimization problems solved). Since Gurobi does not have an inbuilt batch-wise solving capability, it does the solves in serial. However, the MPC-Solver does the solves in parallel on a GPU (Nvidia GeForce 2080Ti, 11GB VRAM).

Figure 7.3 compares the run-time of both the solvers in solving different batches of optimization problems for determining the payment vector as described in Section 7.1.2. Note that the y-axis that shows the approximate run-times in seconds is on log-scale. The difference in run-times is enormous, especially with the increase in the batch-size. When Gurobi takes around 836 seconds to solve 2^{12} problems, MPC-Solver does this on a GPU in just 13 seconds. The ultimate aim would be to solve as many optimization problems as possible in parallel. However, when working on a GPU, it is more likely that there are more memory constraints than that on a CPU. For the problem settings that we used and the memory available, the MPC-Solver could only manage a batch-size of 2^{12}. A possible solution to this problem is discussed in Section 8.3.

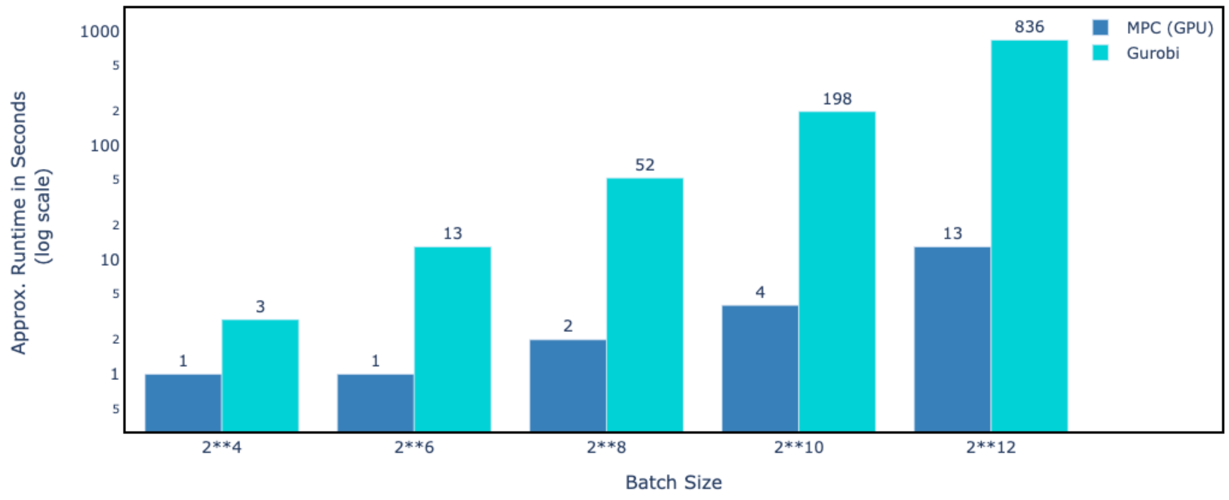


Figure 7.3: Run-time comparison of MPC-Solver(on GPU) and Gurobi for different batch-sizes

Now, since we know that the MPC-Solver produces much faster results than Gurobi, we need to make sure that these results are reliable. For this purpose, we compare the payment vectors returned by both the solvers. We assume that the Gurobi solver is 100% accurate and hence score the results from the MPC-Solver based on this. As already shown in Chapter 6, we compare the results using two methods — percentage of exactly agreeing solutions (rounded to some decimals) and Root Mean Squared Error (RMSE) of the solutions.

Figure 7.4 shows the accuracy as a percentage of exactly agreeing solutions (rounded to the specified decimals) of the MPC-Solver and Gurobi when solving different number

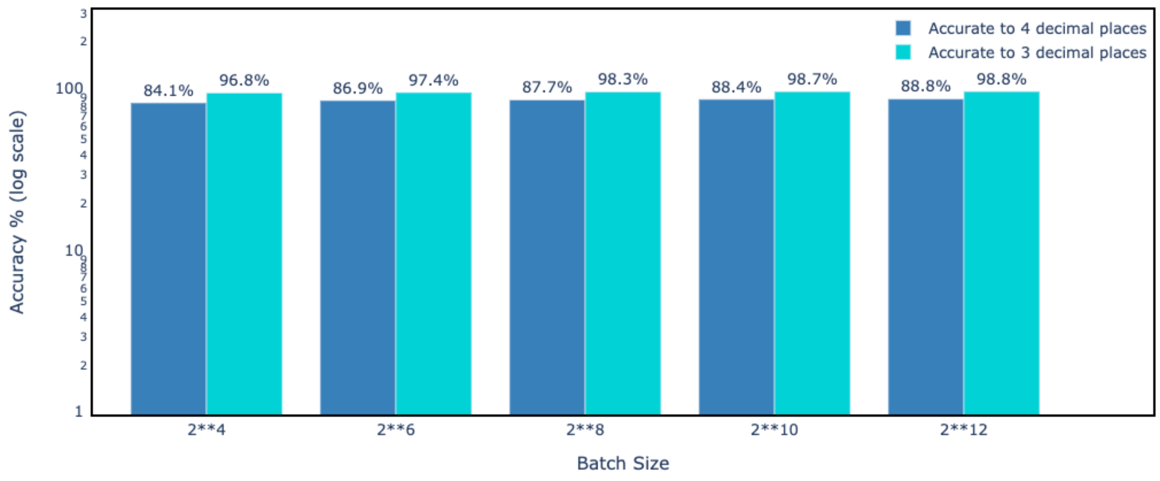


Figure 7.4: Accuracy of MPC-Solver(on GPU) with respect to ‘Gurobi’ on different batch-sizes

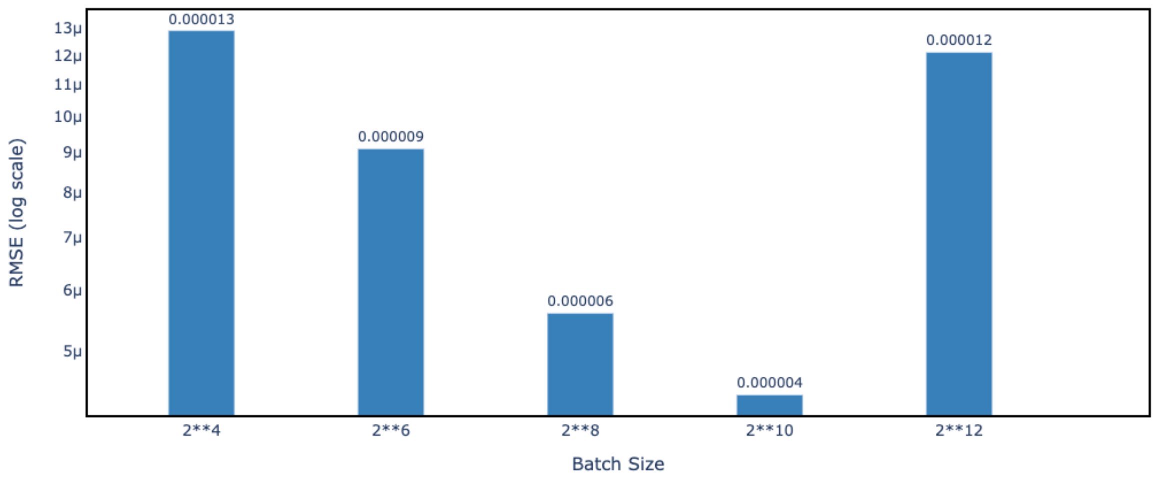


Figure 7.5: RMSE of solutions from MPC-Solver with respect to those from Gurobi

of optimization problems (as specified by the x-axis). Note that here also we have the y-axis on log-scale. We see that when solving 2^{12} problems, approximately 98.8% of the results from both the solvers agree.

We also compare the solutions from both the solvers using the RMSE. Figure 7.5 shows the RMSE of solutions obtained from MPC-Solver with respect to those from Gurobi for the different batch sizes. Note that the y-axis is on log-scale. We see that the errors are very small, of the order or 10^{-5}.

Thus, we conclude that the MPC-Solver produces accurate solutions compared to those from Gurobi in a very small amount of time using its techniques of parallelisation and exploitation of GPU computational power. This helps when computing results of a large number of combinatorial auctions, as in the scenario that we described above.

Chapter 8

Future Work

We have had a thorough discussion on optimization problems, the software-packages currently available to solve them, and especially the MPC-Solver that we introduced for the purpose of solving a large number of problems very efficiently. We saw interesting results in Chapter 6, where we compared the performance of the different solvers and ensured that the MPC-Solver performs the best. However, the MPC-Solver can be further extended by adding multiple features to it. In this chapter, we discuss the possible future extensions that can be made to the MPC-Solver.

In Section 8.1, we discuss the possibility of adding a specific LP solver to the MPC-Solver, rather than using the QP solver to solve the LPs. In Section 8.2 we discuss the possibility of extending the MPC-Solver to solve the classes of convex optimization problems other than QPs and LPs. In Section 8.3 we propose the idea of splitting the batch-size based on the available memory, so as to avoid issues with memory-overflows. We discuss the idea of using a warm-start option for the MPC-Solver in Section 8.4.

8.1 Specific LP Solver

Currently, the MPC-Solver uses the MPC Algorithm for solving QPs to also solve the LPs by setting the problem matrix Q to a very small number (≈ 0). However, it is possible to add separate solvers to the MPC-Solver that can deal with LPs and QPs separately. The main difference would be in the structure of the KKT Matrix since the objective function would no longer have the matrix Q in it. This can potentially increase the accuracy of the LP solves by the MPC-Solver, while maintaining the computational performance that it currently offers.

8.2 Other Convex Programming Problems

As we have discussed so far, the MPC-Solver is currently able to solve only two classes of convex optimization problems: LPs and QPs. However, there are other classes of convex optimization problems that can be solved in the same way using the primal-dual interior-point methods.

Monotone Linear Complementary Problems (LCP) is a general class of problems that encompasses LPs and QPs. Monteiro and Wright [25] showed the convergence of interior-point methods for monotone LCPs.

Semi-Definite Programming (SDP) is also an extension of Linear Programming in which symmetric matrices, as well as real vectors, are included among the variables and positive semi-definiteness conditions on the matrix variables are included in the constraints [34]. Solving SDPs using interior-point methods have also been an important area of research in recent years. Nemirovski [26] showed that it is possible to devise polynomial interior-point algorithms for SDPs.

Second Order Conic Programs (SOCPs) are also capable of being solved by interior-point methods. They are very similar to the case of solving QPs, with just a few differences in the form of the KKT matrix used for the search direction computations, the only substantial difference being that the matrices Z and S are not diagonal. [21]

Thus, it is possible to extend the scope of the MPC-Solver to solve these additional classes of problems that are proven to be solvable with the interior-point methods. Since the topic of interior-point is extensively being researched on, there are possibilities of other classes of algorithms to be included on the list of solvable problems by interior-point methods with time.

8.3 Memory Based Batch Splits

We saw in Chapter 6 that one needs to be careful while using the batched operations from the MPC-Solver. When stacking a batch with a huge number of problems — more than what the memory could hold at the same time — there are chances of memory overflows. This happens mainly when we work with restricted memory environments. For example when we work with a GPU, it uses a VRAM and cannot acces the computer's RAM directly. Often these VRAMs are very much smaller compared to the computer's RAM.

A possible solution to this problem is to add a functionality to the MPC-Solver that splits a huge batch of problems into smaller batches, even when the user does not specify this. The solves can be then done on the smaller batches and the results can be combined and returned to the user. In such a case, the user need not be concerned about the maximum batch-size that can be handled by the memory when using the MPC-Solver, instead, the solver takes care of this. This can potentially solve the memory-overflow problems that we saw in Section 6.2 and Section 6.3.

8.4 Warm Start

Some optimization problem solvers support ‘warm-start’ for achieving better performance when solving similar problems. A warm-start refers to using the historical information of a related/simplified optimization problem to provide the initial values for the problem of interest. The solver can use the previous solution as an initial point for the current problem or reuse cached matrix factorizations. The hope is that the warm-start helps the convergence to the optimal solution faster.

The MPC-Solver currently does not support the warm-start functionality. However, this is a possible further step that might help with faster convergence if solutions, especially when solving similar problems stacked into different batches.

Chapter 9

Conclusion

In this chapter, we conclude our discussion on optimization problems and the newly developed MPC-Solver by highlighting some important details.

We started our discussion by introducing the very basics of optimization problems in Chapter 2. We introduced the notation and definitions associated with optimization problems that have been used throughout the document. The main problems of our interest - LPs and QPs were stated and their associated Dual Problems and KKT systems were also defined.

In Chapter 3, we saw some of the conventionally used solvers like Gurobi, CVXPY and qpth that are used for solving optimization problems. We familiarized ourselves with the usage of these solvers, their advantages and disadvantages.

Chapter 4 included all the theoretical details regarding interior-point methods that lay the background of the MPC-Solver. Most importantly we discussed the MPC Algorithm which we used to build the MPC-Solver. We also briefly discussed the convergence properties and complexity of the Primal-Dual Interior Point Methods in this Chapter.

After providing sufficient background knowledge, we introduced the concepts of the MPC-Solver in Chapter 5. We introduced the idea of stacking several problems into a batch and solving them in parallel using the MPC algorithm. We also stated the mathematical techniques — LU factorization, block-elimination and Cholesky decomposition — that we exploited in order to achieve the best performance results. We roughly outlined the functioning of the MPC-Solver and also stated examples to help understand the usage of this solver better.

We summarized our results in Chapter 6. We described how we generated 1 million test problems (both QPs and LPs) randomly. We compared the performance of these solvers in solving these different problems and also tested their accuracy. The MPC-Solver showed extraordinarily better solve times compared to other solvers and produced results similar to those from Gurobi - hence achieving our goal! We also made some remarks on the consistency of solutions and the memory requirements.

In Chapter 7, we introduced a real-life problem, where we need to solve thousands of optimization problems and hence finds the MPC-Solver very useful. We talked about auctions and combinatorial auctions, and the necessity of computing solutions to optimization problems to determine payment vectors with the nearest-VCG rule. We also compared the results of Gurobi with the MPC-Solver for a LLLLGG setting and saw that the MPC-Solver performs solves much faster than Gurobi, while giving similar solutions.

Finally, in Chapter 8 we discussed the possible dimensions along which the MPC-Solver could be further extended in future, like extending the solver to solve other different classes of convex optimization problems, managing batch-splits based on available memory and adding the warm-start functionality for faster convergence.

References

[1] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1), 2018.

[2] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter. Differentiable convex optimization layers. In *Advances in neural information processing systems*, 2019.

[3] B. Amos and J. Z. Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*. PMLR, 2017.

[4] T. M. Apostol. Mathematical analysis. 1964.

[5] L. M. Ausubel and O. V. Baranov. Core-selecting auctions with incomplete information. *University of Maryland*, 121, 2010.

[6] L. M. Ausubel and P. R. Milgrom. Ascending auctions with package bidding. *The BE Journal of Theoretical Economics*, 1(1), 2002.

[7] L. M. Ausubel, P. Milgrom, et al. The lovely but lonely vickrey auction. *Combinatorial auctions*, 2006.

[8] V. Bosshard, B. Bünz, B. Lubin, and S. Seuken. Computing bayes-nash equilibria in combinatorial auctions with verification. *arXiv preprint arXiv:1812.01955*, 2018.

[9] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[10] S. Boyd, N. Parikh, and E. Chu. *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc, 2011.

[11] P. Cramton, Y. Shoham, R. Steinberg, et al. Combinatorial auctions. Technical report, University of Maryland, Department of Economics-Peter Cramton, 2004.

[12] R. W. Day and P. Cramton. Quadratic core-selecting payment rules for combinatorial auctions. *Operations Research*, 60(3), 2012.

[13] R. W. Day and S. Raghavan. Fair payments for efficient allocations in public sector combinatorial auctions. *Management science*, 2007.

[14] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83), 2016.

[15] A. Domahidi, E. Chu, and S. Boyd. Ecos: An socp solver for embedded systems. In *2013 European Control Conference (ECC)*. IEEE, 2013.

[16] A. Erdil and P. Klemperer. A new payment rule for core-selecting package auctions. *Journal of the European Economic Association*, 2010.

[17] A. Goetzendorff, M. Bichler, P. Shabalin, and R. W. Day. Compact bid languages and core pricing in large multi-item auctions. *Management Science*, 2015.

[18] L. Gurobi Optimization. Gurobi optimizer reference manual, 2020. URL `http://www.gurobi.com`.

[19] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, 1984.

[20] V. Klee and G. J. Minty. How good is the simplex algorithm. *Inequalities*, 3(3), 1972.

[21] J. Mattingley and S. Boyd. Cvxgen: A code generator for embedded convex optimization. *Optimization and Engineering*, 13(1), 2012.

[22] R. P. McAfee and J. McMillan. Auctions and bidding. *Journal of economic literature*, 1987.

[23] K. McShane. Superlinearly convergent $o(\sqrt{n}l)$-iteration interior-point algorithms for linear programming and the monotone linear complementarity problem. *SIAM Journal on Optimization*, 1994.

[24] S. Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on optimization*, 2(4), 1992.

[25] R. D. Monteiro and S. J. Wright. Local convergence of interior-point algorithms for degenerate monotone lcp. *Computational Optimization and Applications*, 3(2), 1994.

[26] A. Nemirovski. Interior point polynomial time methods in convex programming. *Lecture notes*, 2004.

[27] D. C. Parkes, J. R. Kalagnanam, and M. Eso. Achieving budget-balance with vickrey-based payment schemes in exchanges. 2001.

[28] T. Sandholm. Very-large-scale generalized combinatorial multi-attribute auctions: Lessons from conducting $60 billion of sourcing. 2013.

[29] Stefan Heidekrüger, Paul Sutterer, Nils Kohring and Martin Bichler. Equilibrium learning in combinatorial auctions: Computing approximate bayesian nash equilibria via pseudogradient dynamics. Manuscript submitted for publication, 2020.

[30] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: An operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 2020. doi: 10.1007/s12532-020-00179-2. URL `https://doi.org/10.1007/s12532-020-00179-2`.

[31] E. Süli and D. F. Mayers. *An introduction to numerical analysis.* Cambridge university press, 2003.

[32] L. Vandenberghe. The cvxopt linear and quadratic cone program solvers. *Online: http://cvxopt. org/documentation/coneprog. pdf*, 2010.

[33] P. Wolfe. The simplex method for quadratic programming. *Econometrica: Journal of the Econometric Society*, 1959.

[34] S. J. Wright. *Primal-dual interior-point methods*, volume 54. Siam, 1997.

[35] Y. Ye. *Interior point algorithms: theory and analysis*, volume 44. John Wiley & Sons, 2011.

[36] Y. Ye and K. Anstreicher. On quadratic and $o(\sqrt{n}l)$convergence of a predictor—corrector algorithm for lcp. *Mathematical Programming*, 1993.

[37] Y. Ye, O. Güler, R. A. Tapia, and Y. Zhang. A quadratically convergent $o(\sqrt{n}l)$-iteration algorithm for linear programming. *Mathematical programming*, 1993.