Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Tor Kristian Ertsvik Andreassen

Automatic Penalty Parameter Selection for the Distributed Alternating Direction Method of Multipliers

Master's thesis in Cybernetics and Robotics
Supervisor: Damiano Varagnolo
Co-supervisor: Behdad Aminian
April 2023

NTNU
Norwegian University of
Science and Technology

Tor Kristian Ertsvik Andreassen

Automatic Penalty Parameter Selection for the Distributed Alternating Direction Method of Multipliers

Master's thesis in Cybernetics and Robotics
Supervisor: Damiano Varagnolo
Co-supervisor: Behdad Aminian
April 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Abstract

Distributed optimization algorithms have become popular, in part due to the increasing prevalence of embedded systems with communication capabilities. The Alternating Direction Method of Multipliers is one such algorithm. One concern is that its performance is sensitive to the selection of the so-called penalty parameter. The contribution of this thesis is to explore novel methods for automatically selecting the penalty parameter. Two methods were explored, but only one method saw promising results. A limitation of the method is that its hyperparameters must be selected by the user, but it is easier to select these hyperparameters than to select the penalty parameter, due to their straightforward interpretation. Another limitation is that the optimization algorithm tended to diverge when not using complete graphs, but the specific issue underlying this problem was identified.

Sammendrag

Distribuerte optimaliseringsalgoritmer har blitt populære, delvis på grunn av den økende utbredelsen av innvevde systemer som kan kommunisere. En slik algoritme er Alternating Direction Method of Multipliers. En utfordring er at ytelsen er følsom for valget av den såkalte straffeparameteren. Denne oppgavens hovedformål er å utforske nye metoder for automatisk valg av straffeparameteren. To metoder ble evaluert, men bare én metode ga lovende resultater. En begrensning ved metoden er at dens hyperparametre må velges av brukeren, men siden metoden baserer seg på enkle prinsipper er det lettere å velge disse hyperparametrene enn å velge straffeparameteren. En annen begrensning er at optimaliseringsalgoritmen hadde en tendens til å divergere når komplette grafer ikke ble brukt, men det spesifikke problemet som lå til grunn for dette ble identifisert.

Acknowledgements

I would like to express my gratitude to my supervisor, Damiano Varagnolo, for his feedback and support during the course of my thesis. I am also grateful to Behdad Aminian, who served as co-supervisor. Working with them has significantly enhanced my skills as an engineer. Finally, I would like to thank my family for their unconditional support during my studies.

Contents

List of Figures

List of Algorithms

Acronyms

ADMM Alternating Direction Method of Multipliers.

FFT Fast Fourier Transform.

IoT Internet of Things.

Chapter 1

Introduction

1.1 Background

It is becoming increasingly common for embedded systems to have communication capabilities. This gives rise to ad-hoc networks of sensors and actuators that can generate massive amounts of data. This phenomenon is often referred to as the Internet of Things (IoT). The data produced can be analyzed to extract valuable insights, and to make better decisions. Examples of applications are regression, classification, control systems, and resource allocation.

Mathematical optimization algorithms are needed for many classes of data analysis and decision making. However, classical optimization algorithms are typically posed in a *centralized* context. That is, the algorithms assume the optimization is performed by a single entity with access to all of the data.

In the aforementioned *distributed* context of an ad-hoc network of entities, there is no pre-defined central entity responsible for data analysis. Furthermore, introducing a central entity might be infeasible. For instance, the bandwidth of the network might be too low to transfer all data to the central entity for processing. In addition, a central entity might not have enough computational power to process all of the data in a reasonable amount of time. Instead, it might be desirable to exploit the computational power of all the entities in the network. For these reasons, distributed optimization algorithms are sought.

The Alternating Direction Method of Multipliers (ADMM) is an algorithm for solving convex optimization problems [1]. It is widely used in a variety of domains, including machine learning and control systems. ADMM is commonly presented in a centralized form, but it

turns out that the algorithm is amenable to distributed computation as well [2], which makes it interesting for IoT applications.

1.2 Problem statement

In a distributed setting, each step of an optimization algorithm requires communication. Because bandwidth may be at a premium, it is desirable to solve the optimization problem with as few steps as possible. The number of steps k needed to arrive at a solution is thus the performance metric of interest in this thesis.

Similar to other optimization algorithms, ADMM can be tuned in order to increase its performance. For ADMM, the so-called *penalty parameter* is critical. While ADMM can be competitive with state-of-the-art methods, selecting a bad penalty parameter can ruin the performance.

The goal of this thesis is to find new methods for selecting the ADMM penalty parameter in a distributed setting. In particular, I want to explore methods that use information from neighboring entities to select the parameter. That is, the penalty parameter should be selected based on how other entities are performing.

1.3 Research delimitation

ADMM is applicable to a wide variety of problems, but this thesis cannot close all knowledge gaps. Therefore, the following delimitations were made.

- Only convex optimization problems are considered.
- There are many ways to formulate ADMM as a distributed algorithm. In this thesis, the only formulation used is the one developed by Notarstefano *et al.* [2]. An implementation of this version of distributed ADMM is provided in the software library DISROPT [3], and this implementation was used for performing all experiments.
- Experiments were not performed on real-world cost functions. Since the goal of this thesis is to develop new methods, using arbitrary cost functions was deemed sufficient for testing. Therefore, only quadratic and exponential cost functions were evaluated.

- The goal is not to find methods for selecting the penalty parameter with zero input from the user. Instead, it is adequate to formulate methods that have hyperparameters that may be tuned. The idea is that it should be easier to tune the hyperparameters than to select the penalty parameter.

1.4 Related work

While automatic penalty parameter selection is still an open field of research, some methods have been proposed. For the centralized version of ADMM, the most common method is *residual balancing* [1, 4]. This method works by ensuring that the so-called residuals – measures of primal and dual feasibility – are of similar magnitudes. The residual balancing scheme has also been adapted to distributed contexts [5]. In this setting each entity has it's own local residuals that are balanced.

Unfortunately, residual balancing suffers from two major issues. First, the scaling of the decision variable impacts the size of the residuals. For instance, if the decision variable x is scaled such that $x \leftarrow \alpha x$ (with $\alpha \gg 1$), the solution of the optimization problem remains unchanged. However, the residuals will not be the same as in the unscaled problem. This sensitivity to scaling results in subpar performance for unfavorably scaled problems [4].

Secondly, residual balancing does not guarantee convergence. The solution to this is to set the penalty parameter to a fixed value after a given number of iterations [1]. If that happens, and if the residual balancing scheme fails to find a good penalty parameter before it is turned off, the performance will suffer.

Another method for selecting the penalty parameter is known as Adaptive ADMM [6]. This method uses estimates of the dual function's curvature to select the penalty parameter, and is similar to rules used for gradient descent. A distributed version of this method has been developed, the so-called Adaptive Consensus ADMM [7]. These methods are promising, but they are still relatively new compared to residual balancing, so only a limited set of data is available regarding their performance in different contexts.

1.5 Report outline

This thesis is organized as follows. Chapter 2 presents the theoretical background needed to understand ADMM. ADMM is presented both in the centralized form and in the distributed form. Chapter 3 details two new algorithms for automatically selecting the penalty parameter. Chapter 4 presents the experiments performed to assess the performance of the two aforementioned algorithms for automatic penalty parameter selection. Chapter 5 discusses the results of the experiments. Finally, a conclusion is provided in Chapter 6.

Chapter 2

Theoretical background

2.1 Alternating Direction Method of Multipliers (ADMM)

The Alternating Direction Method of Multipliers (ADMM) is a popular optimization method that is easily adapted to distributed problems. While this thesis focuses on ADMM in a distributed context, I begin by providing a review of ADMM in a centralized setting, following Boyd *et al.* [1] and Notarstefano *et al.* [2, Appendix A.4].

Suppose that we have an optimization problem of the form

$$\begin{aligned} \underset{x \in \mathbb{R}^n, z \in \mathbb{R}^m}{\text{minimize}} \quad & f(x) + g(z) \\ \text{subject to} \quad & Ax + Bz = c, \\ & x \in \mathcal{C}_x, \\ & z \in \mathcal{C}_z \end{aligned} \tag{2.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^m \to \mathbb{R}$ are convex functions. The constraint sets $\mathcal{C}_x$ and $\mathcal{C}_z$ are convex, closed, and nonempty. Furthermore, $A \in \mathbb{R}^{p \times n}$, $B \in \mathbb{R}^{p \times m}$, and $c \in \mathbb{R}^p$.

The augmented Lagrangian of Equation (2.1) is

$$L_\rho(x, z, \lambda) = f(x) + g(z) + \lambda^T(Ax - Bz - c) + \frac{\rho}{2}\|Ax + Bx - c\|_2^2 \tag{2.2}$$

where $\lambda \in \mathbb{R}^p$ is the Lagrange multiplier (or dual variable), and $\rho > 0$ is the penalty parameter.

ADMM is conducted as follows. At each iteration $k \geq 0$ the following steps are carried out sequentially

$$x^{k+1} = \underset{x \in \mathcal{C}_x}{\operatorname{argmin}} L_\rho(x, z^k, \lambda^k) \tag{2.3}$$

$$z^{k+1} = \underset{z \in \mathcal{C}_z}{\operatorname{argmin}} L_\rho(x^{k+1}, z, \lambda^k) \tag{2.4}$$

$$\lambda^{k+1} = \lambda^k + \rho(Ax^{k+1} + Bz^{k+1} - c) \tag{2.5}$$

where z^0 and λ^0 can be selected arbitrarily. Algorithm 2.1 describes the entire centralized ADMM algorithm.

It can be shown that ADMM converges to a globally optimum solution under rather mild conditions [1, pp. 15–17].

Algorithm 2.1: Centralized ADMM

Initialization: z^0 and λ^0
for $k = 0, 1, \ldots$ **do**
  Compute x^{k+1} using Equation (2.3)
  Compute z^{k+1} using Equation (2.4)
  Compute λ^{k+1} using Equation (2.5)
end for

2.2 Distributed optimization

In Section 2.1 I provided a brief review of ADMM in a centralized context. That is, the algorithm makes two key assumptions that render it unsuitable for distributed computation: the optimization routine is performed by a single entity, and that single entity has access to all data.

In a distributed environment, these requirements can often not be met. As mentioned in Chapter 1, there are several limitations inherent to the distributed setting. Specifically, the communication bandwidth between entities may not be big enough to transmit all data to a central entity in a reasonable amount of time. In addition, it might be desirable to exploit the computational power of all the entities in a network, as opposed to only utilizing a single entity's power. Therefore, methods that take these limitations into account are needed.

Distributed optimization algorithms takes said limitations into

account.[1] These algorithms do not rely upon having a single entity perform the entire optimization algorithm, and they do not require a single entity to have access to all data. Instead, the entities, called *agents*, cooperate to solve the optimization problem.

When considering a distributed algorithm, the form of the optimization problem (such as the one presented in Section 2.1 in the case of centralized ADMM) must be modified to incorporate the notion of several agents. In addition, the communication between agents must be modeled, using ideas from graph theory.

There is an important caveat to the communication model and optimization set-up I present in this section. There are several choices that can be made when describing the communication model and the form of the optimization problem. For instance, one could allow the communication graph to be time-varying. Regarding the optimization problem, one must for instance decide if all agents share the same constraints, or if each agent has its own individual constraints. However, the focus of this paper is not on evaluating complex models, but rather finding novel approaches for penalty parameter selection. Therefore, I will only consider simple models. If more complex models are needed, Notarstefano *et al.* [2] provides an excellent overview of distributed optimization, with different types of optimization set-ups and algorithms.

2.2.1 Communication model

We model the communication between agents as an undirected graph $\mathcal{G}$, with N nodes representing the agents. $\mathcal{E}$ is the set of edges. Agent i and agent j are only able to communicate directly if there is an edge from i to j in $\mathcal{E}$. In this case they are said to be neighbors. The set of neighbors reachable from agent i is denoted $\mathcal{N}_i$. Finally, the graph is not time-varying, and it is connected. A more formal description of communication models for distributed optimization can be found in [2].

[1] As an aside, so-called *parallel* optimization algorithms exist. Like distributed algorithms, they exploit the computational power of all entities in a network. However, they are not suitable for distributed environments because, unlike distributed algorithms, they rely upon transmitting intermediate results to and from a central entity. On the other hand, distributed optimization algorithms only require an entity to exchange data with it's neighbors. Parallel optimization algorithms are not a topic of this thesis, and as such are not elaborated on further.

2.2.2 Optimization set-up

The optimization set-up used in this thesis is known as *cost-coupled optimization* [2]. This set-up is suitable when the cost function can be written as a sum of local cost functions $f_i(x)$ that are only known by agent i. Furthermore, there are N agents in total, and they all share the same constraint set $\mathcal{C}$. More formally, the problem can be written as

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & \sum_{i=1}^{N} f_i(x) \\ \text{subject to} \quad & x \in \mathcal{C}. \end{aligned} \tag{2.6}$$

2.3 Distributed ADMM

In this section I describe a version of ADMM that can solve a problem of the form in Equation (2.6), following Notarstefano *et al.* [2]. First, let each agent i have its own copy $x_i \in \mathbb{R}^n$ of the optimization variable x. Now, we need to ensure consistency between all copies $x_1, \ldots, x_N$. Therefore, we add a so-called *coherence constraint* by introducing an auxiliary variable $z \in \mathbb{R}^n$.

$$\begin{aligned} \underset{\substack{x_1,\ldots,x_N \\ z}}{\text{minimize}} \quad & \sum_{i=1}^{N} f_i(x_i) \\ \text{subject to} \quad & x_i \in \mathcal{C} \quad i \in \{1, \ldots, N\}, \\ & x_i = z, \quad i \in \{1, \ldots, N\}. \end{aligned} \tag{2.7}$$

The optimization set-up in Equation (2.7) is more amenable to distributed optimization than the previous one, because it no longer relies upon each agent having global knowledge of the optimization variable x. Instead, each agent keeps a local copy x_i, which will be enforced to be similar among all agents during optimization due to the coherence constraint $x_i = z$.

Unfortunately, the coherence constraint in its current form is posed as a global variable z among all agents, in a similar manner to how x used to be global. We want to get rid of this global variable, such that each agent only needs to communicate with its neighbors. Therefore, we amend the coherence constraint such that it only requires coherence among an agent's neighbors.

$$\begin{aligned} \underset{\substack{x_1,\dots,x_N \\ z_1,\dots,z_N}}{\text{minimize}} \quad & \sum_{i=1}^{N} f_i(x_i) \\ \text{subject to} \quad & x_i \in \mathcal{C}, \quad i \in \{1,\dots,N\}, \\ & x_i = z_i, \quad i \in \{1,\dots,N\}, \\ & x_i = z_j, \quad (i,j) \in \mathcal{E}. \end{aligned} \tag{2.8}$$

We can derive the augmented Lagrangian for Equation (2.8). We'll give each coherence constraint its own dual variable λ_{ij}. Define X, Z, and Λ as three vectors stacking all the copies of the primal, auxiliary primal, and dual variables, respectively. Then the augmented Lagrangian is

$$L_\rho(X,Z,\Lambda) = \sum_{i=1}^{N} \left[f_i(x_i) + \sum_{j \in \mathcal{N}_i \cup \{i\}} \lambda_{ij}^T (x_i - z_j) + \frac{\rho}{2} \sum_{j \in \mathcal{N}_i \cup \{i\}} \|x_i - z_j^t\|^2 \right]. \tag{2.9}$$

The optimization problem we arrived at in Equation (2.8) can be solved using the ADMM algorithm described in Section 2.1. The optimization variables x and z are replaced by the vectors X and Z stacking the copies of the primal and auxiliary variables. Furthermore, $f(X) = \sum_{i=1}^{N} f_i(x_i)$ and $g(Z) = 0$. As for the constraint sets, $\mathcal{C}_x = \mathcal{C}$ and $\mathcal{C}_z = \mathbb{R}^{N \cdot n}$. The coefficients in the equality constraint $AX + BZ = c$ are

$$A = \begin{bmatrix} I_{N \cdot n} \\ I_{N \cdot n} \end{bmatrix} \tag{2.10}$$

$$B = \begin{bmatrix} \text{Adj} \otimes I_n \\ I_{N \cdot n} \end{bmatrix} \tag{2.11}$$

$$c = 0 \tag{2.12}$$

where $\otimes$ is the Kronecker product, Adj is the adjacency matrix of $\mathcal{G}$, I_n is the $n \times n$ identity matrix, and $I_{N \cdot n}$ is the $Nn \times Nn$ identity matrix.

As mentioned, by using the definitions above, we can apply ADMM to Equation (2.8) [2]. The optimization steps are

$$x_i^{k+1} = \underset{x_i \in \mathcal{C}}{\operatorname{argmin}}\ f_i(x_i) + \left(\sum_{j \in \mathcal{N}_i \cup \{i\}} \left(\lambda_{ij}^k\right)^T \right) x_i + \frac{\rho}{2} \sum_{j \in \mathcal{N}_i \cup \{i\}} \|x_i - z_j^k\|^2 \tag{2.13}$$

$$z_i^{k+1} = \underset{z_i}{\operatorname{argmin}} -\left(\sum_{j \in \mathcal{N}_i \cup \{i\}} \left(\lambda_{ji}^k\right)^T \right) z_i + \frac{\rho}{2} \sum_{j \in \mathcal{N}_i \cup \{i\}} \|x_j^{k+1} - z_i\|^2 \tag{2.14}$$

$$\lambda_{ii}^{k+1} = \lambda_{ii}^k + \rho \left(x_i^{k+1} - z_i^{k+1} \right) \tag{2.15}$$

$$\lambda_{ij}^{k+1} = \lambda_{ij}^k + \rho \left(x_i^{k+1} - z_j^{k+1} \right) \tag{2.16}$$

Because the optimization step for z_i is an unconstrained quadratic program, we can find an explicit solution:

$$z_i^{k+1} = \frac{\sum_{j \in \mathcal{N}_i \cup \{i\}} x_j^{k+1}}{|\mathcal{N}_i| + 1} + \frac{\sum_{j \in \mathcal{N}_i \cup \{i\}} \lambda_{ji}^k}{\rho \left(|\mathcal{N}_i| + 1 \right)}. \tag{2.17}$$

The full algorithm for distributed ADMM, from the perspective of agent i, is given in Algorithm 2.2.

Algorithm 2.2: Distributed ADMM

Initialization: ρ, z_i^0, λ_{ii}^0, and λ_{ij}^0 for all $j \in \mathcal{N}_i$
for $k = 0, 1, \ldots$ **do**
  Gather λ_{ji}^k from neighbors $j \in \mathcal{N}_i$
  Compute x_i^{k+1} using Equation (2.13)
  Gather x_j^{k+1} from neighbors $j \in \mathcal{N}_i$
  Compute z_i^{k+1} using Equation (2.17)
  Gather z_j^{k+1} from neighbors $j \in \mathcal{N}_i$
  Compute λ_{ii}^{k+1} using Equation (2.15)
  foreach $j \in \mathcal{N}_i$ **do**
    Compute λ_{ij}^{k+1} using Equation (2.16)
  end foreach
end for

Chapter 3

Automatic penalty parameter selection

In this chapter, I describe two algorithms developed for automatic penalty parameter selection. To allow for varying the penalty parameter, Algorithm 2.2 is slightly modified. Instead of a fixed penalty parameter ρ, there is now a ρ^k for each iteration k. The modified algorithm, henceforth referred to as Distributed Automatic ADMM, is described in Algorithm 3.1.

Algorithm 3.1: Distributed Automatic ADMM

Initialization: ρ^0, z_i^0, λ_{ii}^0, and λ_{ij}^0 for all $j \in \mathcal{N}_i$
for $k = 0, 1, \ldots$ **do**
  Gather λ_{ji}^k from neighbors $j \in \mathcal{N}_i$
  Compute x_i^{k+1} using Equation (2.13)
  Gather x_j^{k+1} from neighbors $j \in \mathcal{N}_i$
  Compute z_i^{k+1} using Equation (2.17)
  Gather z_j^{k+1} from neighbors $j \in \mathcal{N}_i$
  Compute λ_{ii}^{k+1} using Equation (2.15)
  foreach $j \in \mathcal{N}_i$ **do**
    Compute λ_{ij}^{k+1} using Equation (2.16)
  end foreach
  Compute ρ^{k+1} using one of the algorithms presented in this thesis
end for

3.1 Using the Fast Fourier Transform

The idea behind this algorithm is to consider the reliability of neighboring estimates. That is, if the neighboring x^k estimates are unreliable, then we should focus on consensus. If instead they are reliable, we should focus on optimization.

A measure of reliability is needed. The idea is that if the sequence of x_j^k received from agent j is changing a lot, it means that the estimates are unreliable. We should therefore focus on consensus, as mentioned. If on the other hand the sequence of x_j^k is stable, we may place more emphasis on optimization of the objective function. The idea is presented visually in Figure 3.1a.

A simple measure of reliability is as follows. Consider a vector w containing the l last x^k received from neighbor j. The vector is essentially a sliding window over the sequence of x_j^k received from neighbor j. We use the Fast Fourier Transform (FFT) to assess to which degree the estimates in w are changing. A set of frequency-domain magnitudes M is computed from the FFT of w. Each magnitude M_i is normalized by dividing it by the first element M_0 of M. Then, the maximum (normalized) magnitude $M^j_{\text{max, normalized}}$ of the set $\{ M_i : i \neq 0 \}$ is found.

This routine is performed for each neighbor in $\mathcal{N}_i$, and a maximum normalized magnitude M^* among an agent's neighbors is found:

$$M^* = \max_{j \in \mathcal{N}_i} M^j_{\text{max, normalized}}. \tag{3.1}$$

This magnitude M^* will be the measure of reliability used. The full algorithm is described in Algorithm 3.2.

To justify why the magnitudes from the FFT are used, consider the following intuition. Assume that the sequences of x^k can be modeled as a linear trend plus a sinusoidal component. If the sinusoidal component has a lot of power compared to the linear part, the x estimates will oscillate a great deal, and thus they are unreliable. Therefore, we want to avoid a high M_i for $i > 0$, and thus the maximum M^* defined in Equation (3.1) is of interest. This concept is illustrated in Figure 3.1b.

Now that a measure of reliability is established, an algorithm for updating the penalty parameter based on reliability can be derived. Algorithm 3.3 is a simple implementation of this idea. If the normalized maximum amplitude M^* is greater than some threshold ϵ, the estimates are considered to be reliable, and the penalty parameter is decreased by 5%. That is, $\rho^{k+1} = \rho^k \times 0.95$. On the other hand,

Algorithm 3.2: A measure of reliability based on the Fast Fourier Transform.

function `FindMaxMagnitude(`$\mathcal{N}_i$, l`)`
  input: set of neighbors $\mathcal{N}_i$; length l of sliding window w to use for computing FFT
  output: normalized maximum magnitude M^* among neighbors' x sequences
  initialization: $M^* = 0$

  foreach $j \in \mathcal{N}_i$ **do**
    $w = \left[x_j^{k-(l-1)}, \ldots, x_j^k\right]$
    $w^{\text{FFT}} =$ `FFT(`w`)`

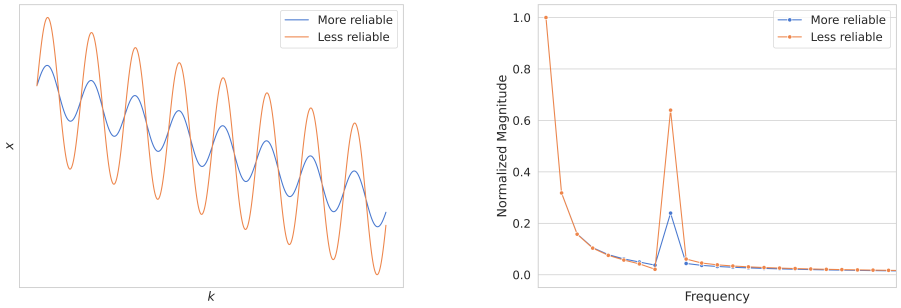
    $M = \emptyset$
    for $i \leftarrow 0$ **to** l **do**
      $M_i = \left|w_i^{\text{FFT}}\right|$
    end for

    `// find maximum value of` M `except for first element`
    $M_{\max} = \max_{i \neq 0} M_i$

    `// normalize by` M_0
    $M_{\text{max, normalized}} = \frac{M_{\max}}{M_0}$

    if $M_{max,\ normalized} > M^*$ **then**
      $M^* = M_{\text{max, normalized}}$
    end if
  end foreach

  return M^*
end function



(a) Sequence of x estimates **(b)** Fast Fourier Transform

Figure 3.1: A measure of reliability of an agent's x estimates. The blue sequence of x estimates in (a) is considered to be more reliable than the orange sequence. In (b) the (normalized) magnitudes obtained from the Fast Fourier Transform of (a) is shown. The maximum magnitude (excluding the first one, which is used for normalization) gives an indication of the reliability of the x estimates.

if M^* is less than ϵ, the estimates are considered to be less reliable, and the penalty parameter is increased by 5%.

Note that, since M_i is normalized by M_0 for all $i > 0$, a threshold of, for instance, $\epsilon = 0.1$ would mean that if the largest sinusoidal component has 10% or more of the power of the linear component, we are no longer considering the estimates to be reliable, and vice versa.

Algorithm 3.3 can be used to select ρ^k in each iteration of Algorithm 3.1.

3.2 Comparing finite differences of estimates

Another measure of reliability is based on comparing the set of

$$\|x_j^k - x_j^{k-1}\| \tag{3.2}$$

received from all neighbors. The idea is that the change in x from one iteration to the other should be similar in magnitude for all agents. If they are disparate, it indicates that we should focus more on consensus. If they are similar, we may instead place more emphasis on optimization.

This gives rise to an issue. By simply letting $\rho \to \infty$ we can ensure that $\|x_j^k - x_j^{k-1}\|$ is always similar for all agents. The question

Algorithm 3.3: Automatic penalty parameter selection using the Fast Fourier Transform.

input: set of neighbors $\mathcal{N}_i$; length of sliding window l; iteration number k; previous penalty parameter ρ^k; threshold ϵ for the reliability of M^*
output: new penalty parameter ρ^{k+1}

if $k < l$ **then**
  /* wait until we have received enough x estimates to create a sliding window of length l */
  return ρ^k
end if

$M^* =$ `FindMaxMagnitude(`$\mathcal{N}_i$, l`)`
if $M^* > \epsilon$ **then**
  $\rho^{k+1} = \rho^k \times 1.05$
else
  $\rho^{k+1} = \rho^k \times 0.95$
end if

is then, how can we ensure that the magnitudes are similar, while still ensuring that optimization of the cost function takes place? In other words, how do we know if we increased ρ by too much?

To this end, consider z^k. Since this auxiliary variable is the coherence constraint, it gives an indication of what the x^k values are across all neighbors. Then $z^k - z^{k-1}$ must be a measure of the speed of convergence among all neighbors, and the finite difference

$$\left(z^k - z^{k-1}\right) - \left(z^{k-1} - z^{k-2}\right) \tag{3.3}$$

gives an indication of the acceleration of x^k steps across all neighbors.

The algorithm for updating ρ is based on balancing the two quantities

$$\left\| x_j^k - x_j^{k-1} \right\| \tag{3.4}$$

and

$$\left\| \left(z^k - z^{k-1}\right) - \left(z^{k-1} - z^{k-2}\right) \right\|. \tag{3.5}$$

It follows a similar intuition to residual balancing, but uses different quantities for comparison.[1]

The algorithm works as follows. If the set of neighboring x step magnitudes are different, increase ρ. If not, and if the magnitude of z acceleration is low, decrease ρ. The full algorithm is described in Algorithm 3.4.

[1] Residual balancing is based on comparing $\left\| x^k - z^k \right\|$ and $\left\| z^k - z^{k-1} \right\|$.

Algorithm 3.4: Automatic penalty parameter selection based on comparing finite differences of x and z.

input: set of neighbors $\mathcal{N}_i$; iteration number k; previous penalty parameter ρ^k
output: new penalty parameter ρ^{k+1}

if $k < 3$ **then**
  `/* wait until we have received enough` x
    `estimates to compute second-order finite differences */`
  return ρ^k
end if

compute $\|\Delta x_i\|$ using Equation (3.4)
compute $\|\Delta x_j\|$ for each neighbor j using Equation (3.4)
compute $\|\Delta^2 z\|$ using Equation (3.5)

compute median M of $\{\Delta x_j : j \in \mathcal{N}_i \cup i\}$
compute maximum M^+ of $\{\Delta x_j : j \in \mathcal{N}_i \cup i\}$
compute minimum M^- of $\{\Delta x_j : j \in \mathcal{N}_i \cup i\}$

if $M^+ > 10M$ **or** $M^- < 0.1M$ **then**
  $\rho^{k+1} = \rho^k \times 1.2$
else if $\|\Delta^2 z\| < 10\|\Delta x_i\|$ **then**
  $\rho^{k+1} = \rho^k \times 0.8$
else
  $\rho^{k+1} = \rho^k$
end if

Chapter 4

Experiments

4.1 Common experimental setup

4.1.1 Performance metric

The goal of this thesis is to evaluate different methods for automatic penalty parameter selection. The performance metric selected was therefore how many iterations would be needed to reach a certain level of accuracy.

To this end, the centralized solution $x^\star$ to the global problem

$$\operatorname*{argmin}_{x \in \mathcal{C}} \sum_i f_i(x)$$

was computed ahead of time. This makes it possible to evaluate the performance metric as the number of iterations k needed for all solutions x_i to be within some small distance ϵ of $x^\star$.

4.1.2 ADMM algorithm with performance metric

To measure the performance, the versions of ADMM proposed in Algorithm 2.2 and Algorithm 3.1 were used, with a small modification. At the end of each iteration, the maximum solution error

$$E = \max_i \|x_i^k - x^\star\| \tag{4.1}$$

was computed, and if at some k the error E was less than the solution error tolerance ϵ the algorithm would terminate. If not, the algorithm would continue until it reached the maximum allowed number of iterations K.

Algorithm 4.1: A modified version of Distributed Automatic ADMM used for assessing performance of different penalty parameter selection schemes.

Initialization: $x^\star$, ϵ, K, ρ^0, z_i^0, λ_{ii}^0, and λ_{ij}^0 for all $j \in \mathcal{N}_i$
for $k = 0, 1, \ldots, K$ **do**
  Gather λ_{ji}^k from neighbors $j \in \mathcal{N}_i$
  Compute x_i^{k+1} using Equation (2.13)
  Gather x_j^{k+1} from neighbors $j \in \mathcal{N}_i$
  Compute z_i^{k+1} using Equation (2.17)
  Gather z_j^{k+1} from neighbors $j \in \mathcal{N}_i$
  Compute λ_{ii}^{k+1} using Equation (2.15)
  foreach $j \in \mathcal{N}_i$ **do**
    Compute λ_{ij}^{k+1} using Equation (2.16)
  end foreach

  $E = \max_i \|x_i^k - x^\star\|$
  if $E < \epsilon$ **then**
    Terminate algorithm early
  else
    Compute ρ^{k+1}
  end if
end for

The full algorithm used for Distributed Automatic ADMM is described in Algorithm 4.1. The extension to Distributed ADMM with a fixed penalty parameter is immediate (in that case, $\rho^{k+1} = \rho^k$ always).

To get a reasonable perspective of the performance of the various algorithms for automatically selecting ρ, it is not enough to look at only one run of Algorithm 4.1. Instead, N runs of the algorithm were performed, with different cost functions for each $n \in \{1, \ldots, N\}$. Furthermore, each agent was assigned its own, random cost function. This general framework for assessing performance is described in Algorithm 4.2. Clearly, after running the algorithm, the performance of N different runs of ADMM had been collected, and these N runs were used to compute the average performance.

Algorithm 4.2: A general framework for assessing the performance of automatic penalty parameter selection methods.

initialization: number of experiments to perform N; set P of (initial) penalty parameters; maximum solution error tolerance ϵ

for $n = 1, \ldots, N$ **do**
  create a new graph $\mathcal{G}$
  create a new cost function $f_i(x_i)$ at random
  compute solution $x^\star = \operatorname{argmin} \sum_i f_i(x)$

  foreach $\rho \in P$ **do**
    run `DistributedADMM` (Algorithm 4.1 with fixed penalty parameter $\rho^k = \rho$)
    run `DistributedAutomaticADMM` (Algorithm 4.1 with initial penalty parameter $\rho^0 = \rho$, and automatic selection of subsequent ρ^k)
  end foreach
end for

4.1.3 Creating random cost functions

To get a convex quadratic cost function at random, the following procedure was followed. Each agent i has it's own local cost function $f_i(x_i) = A_i x^2 + B_i x$. The coefficients must be selected at random such that $f_i(x_i)$ is convex. The coefficient for the quadratic term was selected by drawing a random number from the standard normal distribution, and then squaring it. The coefficient for the linear term was selected by drawing another random number from the standard normal distribution, and multiplying it by 3. In other words, the normal distribution was sampled

$$a_i \sim \mathcal{N}(0, 1),$$
$$b_i \sim \mathcal{N}(0, 1),$$

and the coefficients

$$A_i = a_i^2,$$
$$B_i = 3b_i$$

associated with $f_i(x_i)$ were computed.

A similar procedure was followed to get random convex exponential functions. For each agent i, two random numbers α_i and β_i were sampled from the standard normal distribution. The random exponential function associated with agent i is then given by $f_i(x_i) = \alpha_i^2 \exp(\beta_i x)$.

4.2 Using the Fast Fourier Transform

4.2.1 Experimental setup

The goal of this experiment was to determine whether using the Fast Fourier Transform to select the penalty parameter (Algorithm 3.3) is a practical method. Because of this, the experiments were performed under ideal conditions. By ideal, I mean that the network topology was fully connected, and only quadratic cost functions were used.

The experiment was conducted according to Algorithm 4.2. The decision variable x was mono-dimensional, and a complete graph was used. $N = 100$ iterations of the loop were performed. 10 agents were used. 10 penalty parameters were evaluated, spaced evenly on a logarithmic scale from 10^{-4} to 10^2. A random quadratic cost function was assigned to each agent for each run of the experiment, and the common constraint set was $1 > x_i > -1$ for all iterations. The solution error tolerance ϵ was 10^{-6}.

4.2.2 Results

Figure 4.1 shows the mean performance of the algorithm, that is, the mean number of iterations needed to reach the given solution error tolerance. This mean is plotted for each penalty parameter $\rho \in P$ evaluated. Figure 4.2 shows the mean maximum solution error among all agents (mean of Equation (4.1)) achieved for each penalty parameter. Both figures compare the results of using the FFT algorithm with the usage of a fixed penalty parameter.

Figure 4.1 shows that the mean number of iterations needed was slightly less compared with using a fixed ρ. However, Figure 4.2 shows that the algorithm tended to diverge for small initial ρ.

The fact that the mean number of iterations required was less for the FFT algorithm (despite the algorithm in general being worse than using a fixed ρ) can be explained by Figure 4.3, which shows box plots of the number of iterations needed. The FFT algorithm had more outliers for small ρ. The higher number of outliers gives a justification for why the *average* number of iterations required was smaller than when using a fixed penalty parameter.

4.3 Comparing finite differences on problems with quadratic cost functions

4.3.1 Experimental setup

The experiment was conducted according to Algorithm 4.2 (and it was similar to the FFT experiment in Section 4.2). The decision variable x was mono-dimensional, and a complete graph was used. $N = 100$ iterations of the loop were performed. 10 agents were used. 10 penalty parameters were evaluated, spaced evenly on a logarithmic scale from 10^{-4} to 10^{2}. A random quadratic cost function was assigned to each agent for each run of the experiment, and the common constraint set was $1 > x_i > -1$ for all iterations. The solution error tolerance ϵ was 10^{-6}.

4.3.2 Results

Similar plots as in the FFT experiment (Section 4.2) were generated. Figure 4.4 shows the mean number of iterations required to reach the solution error tolerance. Figure 4.5 shows the average maximum solution error. Figure 4.6 shows the box plots of the number of iterations performed.

In this experiment, the automatic algorithm for selecting ρ performed better than using a fixed ρ, on average. The mean number of iterations required (Figure 4.5) was significantly lower than for the fixed penalty parameter, except close to the best fixed penalty parameter (among the ρ tested).

For the mean maximum solution error (Figure 4.5), the fixed penalty parameter outperformed the automatic tuning algorithm close to the best fixed ρ. However, the automatic tuning algorithm was less sensitive to an initially chosen bad ρ.

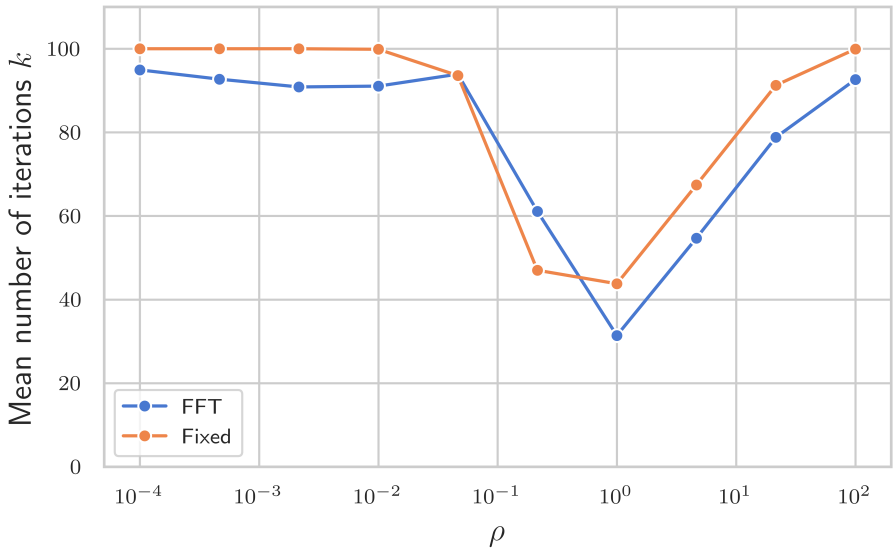


Figure 4.1: The performance of the FFT method for selecting ρ. Mean number of iterations k required to achieve the solution error tolerance. The results were compared with using a fixed ρ to solve the same optimization problems. The maximum allowed number of iterations was 100.

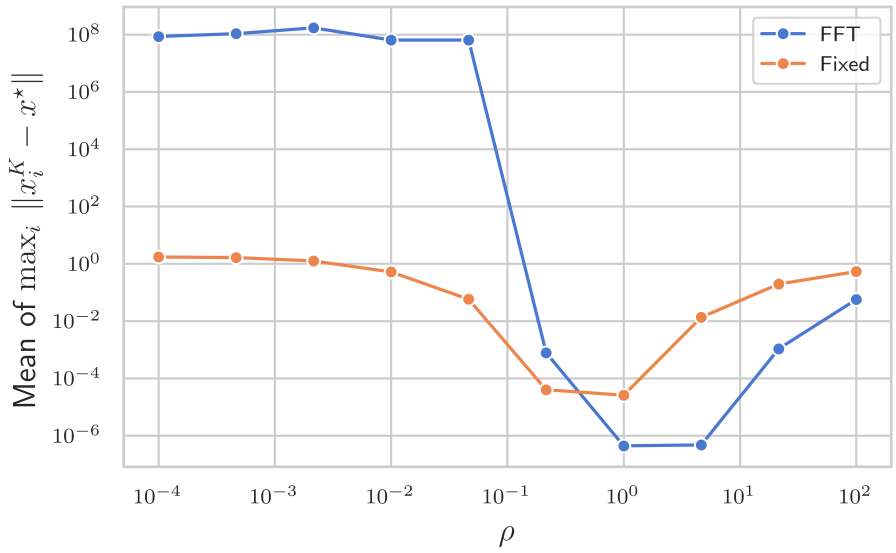
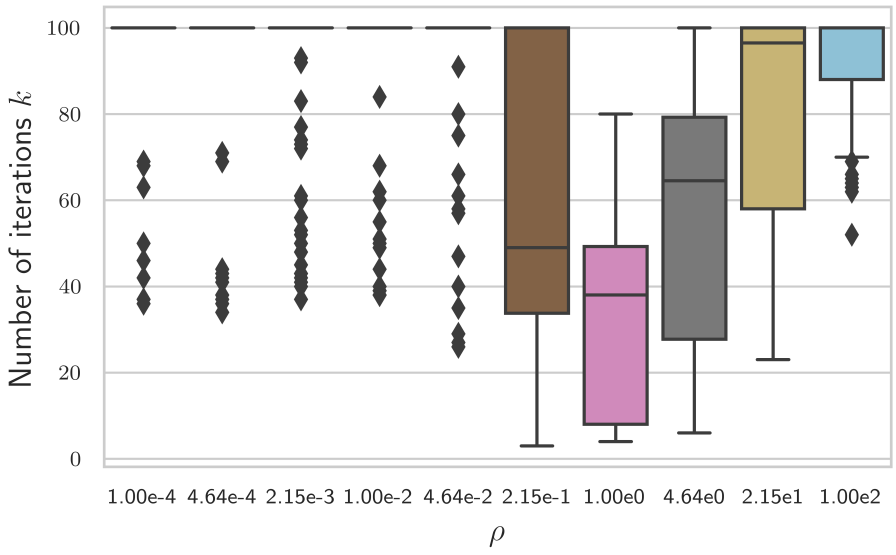
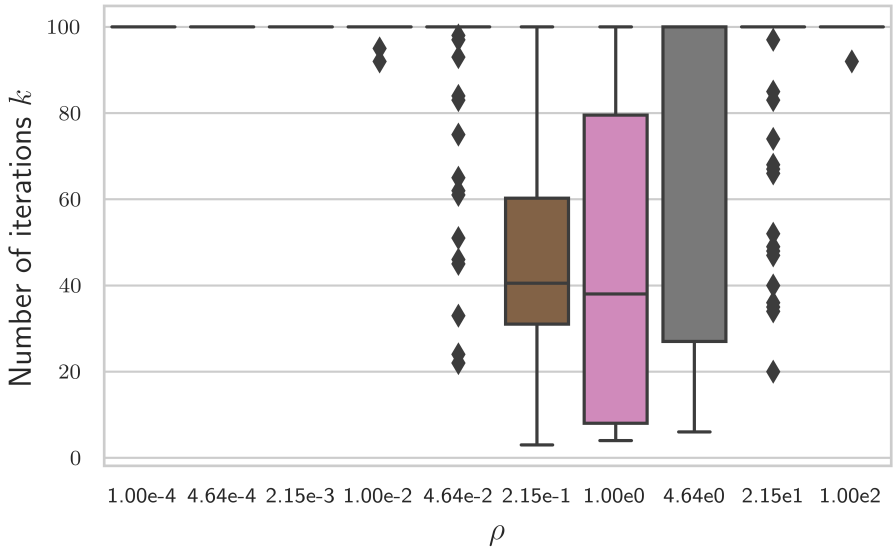


Figure 4.2: The mean maximum solution error when using the FFT method to select ρ. The result was compared with using a fixed ρ.



(a) Automatically selected ρ



(b) Fixed ρ

Figure 4.3: Box plots of the number of number of iterations required to achieve the solution error tolerance. The FFT method was compared with using a fixed ρ. The maximum number of allowed iterations was 100.

On average, both algorithms failed to reach the solution error tolerance $\epsilon = 10^{-6}$. The box plots (Figure 4.6) show that there was always a large number of experiments that terminated at $k = 100$ iterations without achieving the desired solution error tolerance.

4.4 Comparing finite differences on problems with exponential cost functions

4.4.1 Experimental setup

The results in Section 4.3 were promising, but the usage of quadratic cost functions could give a false impression of how well the algorithm performed, since they typically are quite easy to solve. Therefore, a similar experiment as before – except with exponential cost functions – was performed.

The experiment was conducted according to Algorithm 4.2. The decision variable x was mono-dimensional, and a complete graph was used. $N = 100$ iterations of the loop were performed. 10 agents were used. 10 penalty parameters were evaluated, spaced evenly on a logarithmic scale from 10^{-4} to 10^{2}. A random exponential cost function was assigned to each agent for each run of the experiment, and the common constraint set was $1 > x_i > -1$ for all iterations. The solution error tolerance ϵ was 10^{-6}.

4.4.2 Results

The same plots as before were generated. Figure 4.7 shows the mean number of iterations required to reach the solution error tolerance. Figure 4.8 shows the average maximum solution error. Figure 4.9 shows the box plots of the number of iterations performed.

The mean number of iterations required (Figure 4.7) was quite close between the two methods, but the automatic penalty parameter selection algorithm still came out ahead. The average maximum solution error (Figure 4.8) was lower when using the automatic tuning. Once again, the exception to both these statements was when the initial penalty was selected close to the best $\rho \in P$.

The box plots (Figure 4.9) once more show how there still was a significant amount of iterations that terminated at $k = 100$ without achieving the desired solution error tolerance. However, when automatically selecting ρ, if the initial ρ was bad, a larger proportion

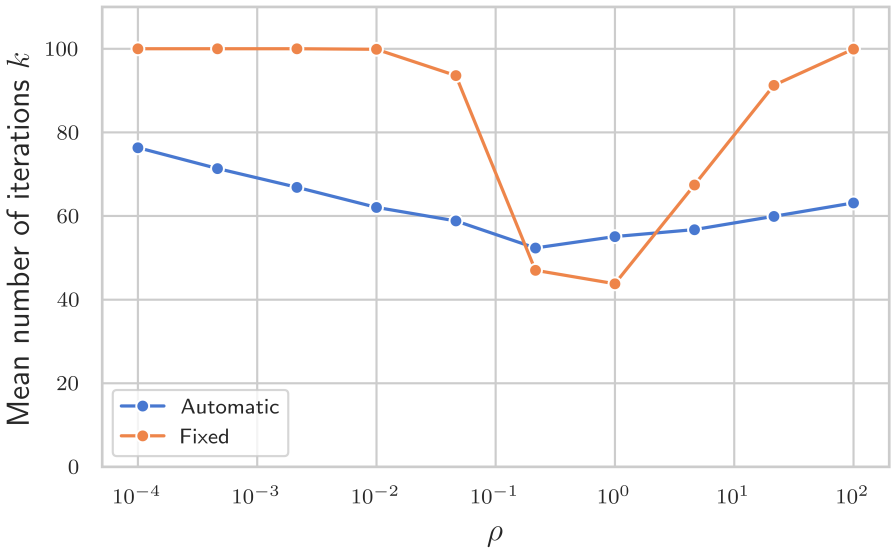


Figure 4.4: The performance of the finite differences method for selecting ρ, when using a complete graph and quadratic cost functions. The results were compared with using a fixed ρ to solve the same optimization problems. The maximum allowed number of iterations was 100.

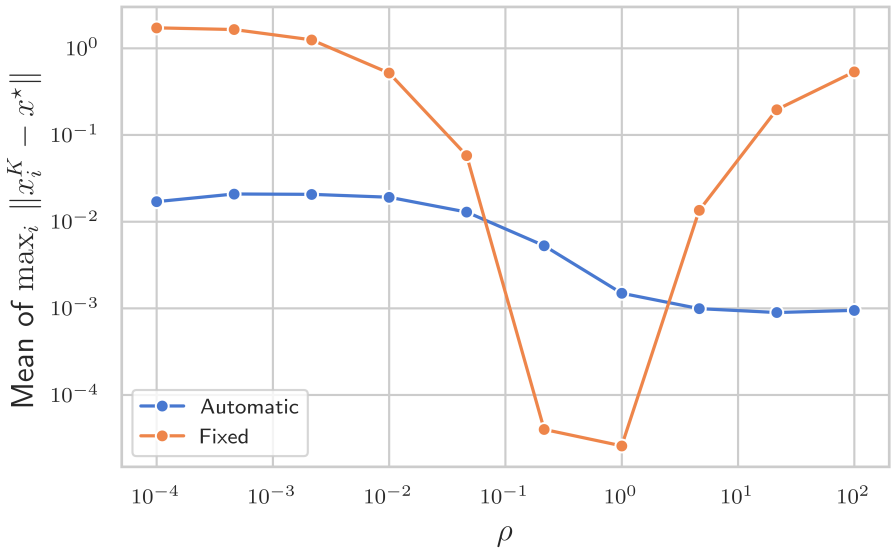
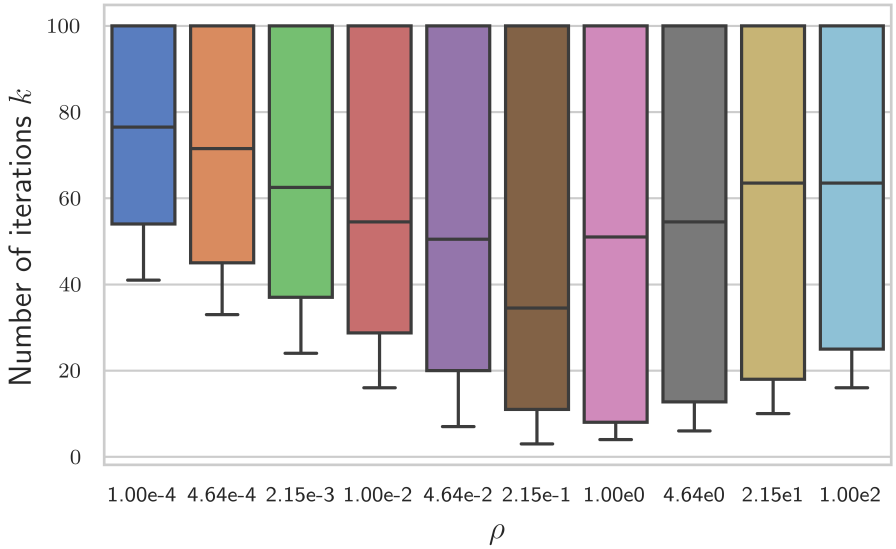
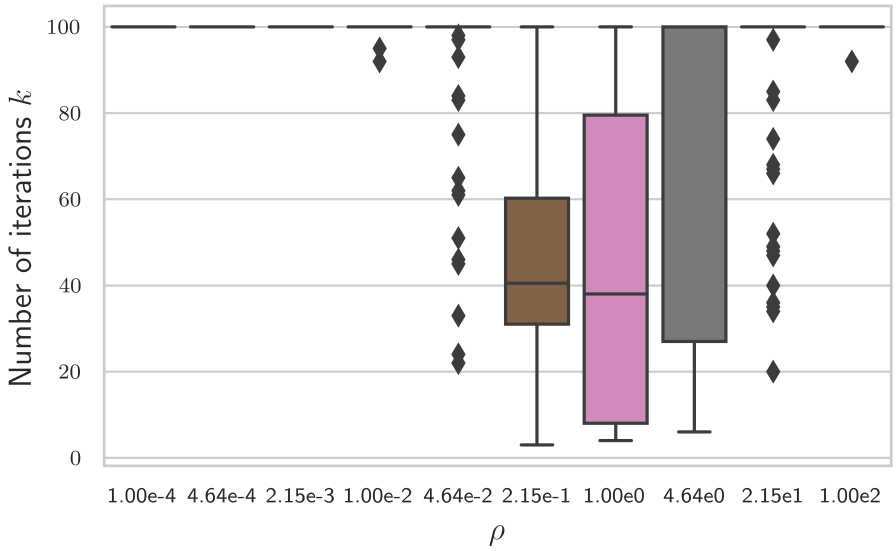


Figure 4.5: The mean maximum solution error when using the finite differences method, using a complete graph and quadratic cost functions. The result was compared with using a fixed ρ.



(a) Automatically selected ρ



(b) Fixed ρ

Figure 4.6: Box plots of the number of number of iterations required to achieve the solution error tolerance, when using a complete graph and quadratic cost functions. The finite differences method was compared with using a fixed ρ. The maximum number of allowed iterations was 100.

of experiments achieved the desired tolerance before $k = 100$.

Finally, Figure 4.10 shows nine examples of how the maximum solution error (among all agents) progresses at each step k. The examples were selected at random from the set of experiments that converged before $k = 100$, with $\rho^0 = 100$. The figure shows that the convergence speed tended to be superlinear if the algorithm converged.

4.5 Comparing finite differences on problems with quadratic cost functions, with a random binomial graph

4.5.1 Experimental setup

Using a complete graph might not be realistic. In real world systems, it might not be possible to establish a connection between all agents in a distributed network. Therefore, the algorithm for selecting ρ must also perform well on graphs that are not fully connected. Thus, an experiment with random graphs was performed.

The experiment was conducted according to Algorithm 4.2. The decision variable x was mono-dimensional, and a random binomial graph was used with edge probability $p = 0.3$. $N = 100$ iterations of the loop were performed. 10 agents were used. 10 penalty parameters were evaluated, spaced evenly on a logarithmic scale from 10^{-4} to 10^2. A random quadratic cost function was assigned to each agent for each run of the experiment, and the common constraint set was $1 > x_i > -1$ for all iterations. The solution error tolerance ϵ was 10^{-4}.

Due to this thesis being time constrained, combined with the fact that running these experiments took a significant amount of time, the difficulty of the experiment had to be lowered somewhat. Therefore, the solution error tolerance was chosen higher than before. Furthermore, quadratic cost functions were used.

4.5.2 Results

The same plots as before were generated. Figure 4.11 shows the mean number of iterations required to reach the solution error tolerance. The automatic tuning performs slightly better than when using a fixed ρ.

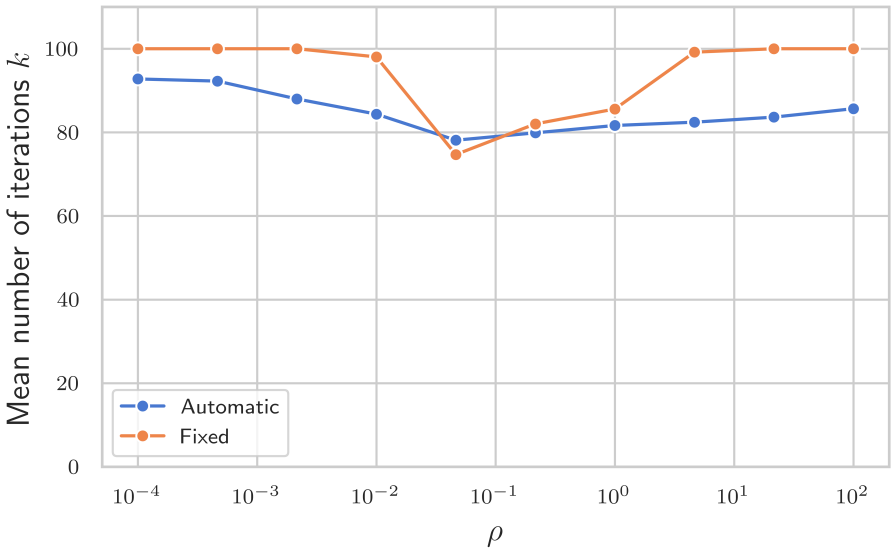


Figure 4.7: The performance of the finite differences method for selecting ρ, when using a complete graph and exponential cost functions. The results were compared with using a fixed ρ to solve the same optimization problems. The maximum allowed number of iterations was 100.

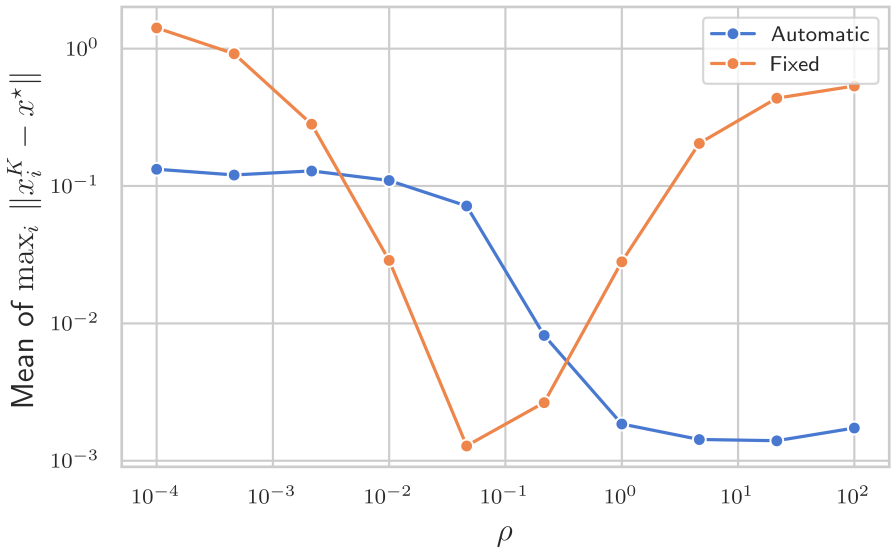
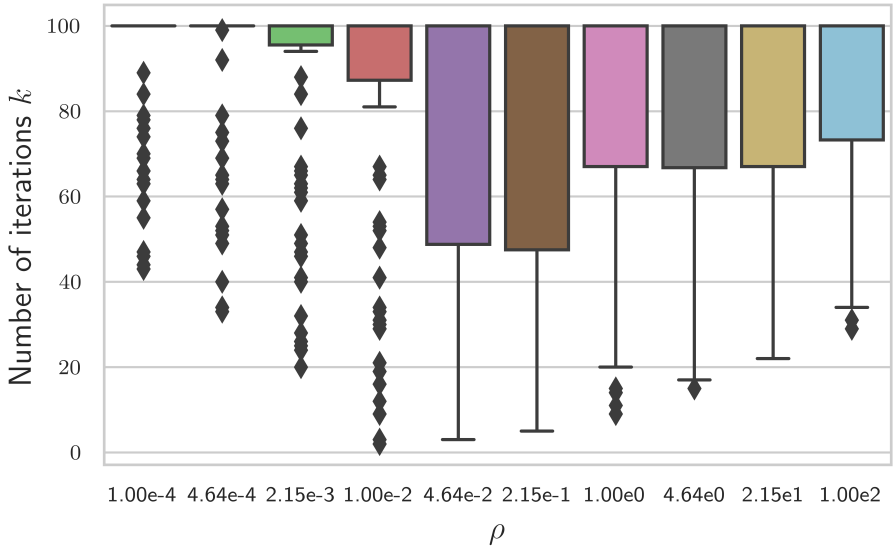
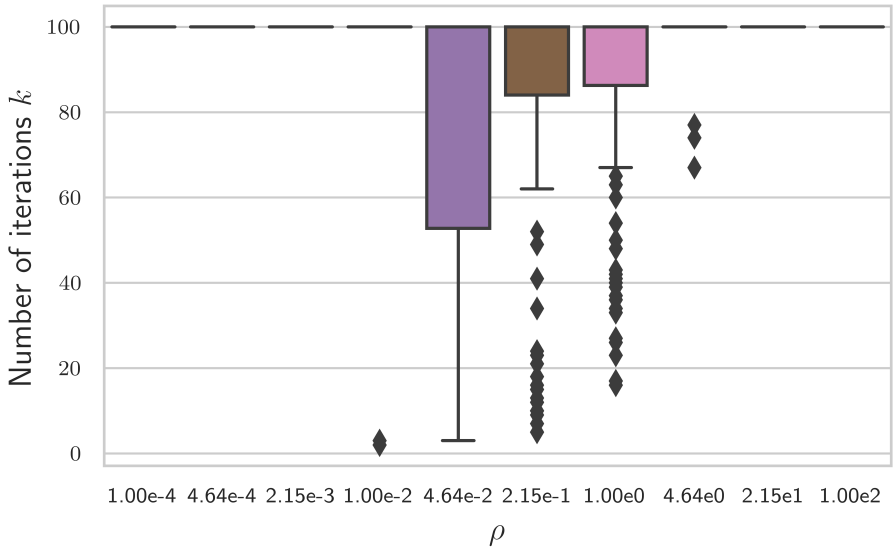


Figure 4.8: The mean maximum solution error when using the finite differences method, using a complete graph and exponential cost functions. The result was compared with using a fixed ρ.



(a) Automatically selected ρ



(b) Fixed ρ

Figure 4.9: Box plots of the number of number of iterations required to achieve the solution error tolerance, when using a complete graph and exponential cost functions. The finite differences method was compared with using a fixed ρ. The maximum number of allowed iterations was 100.

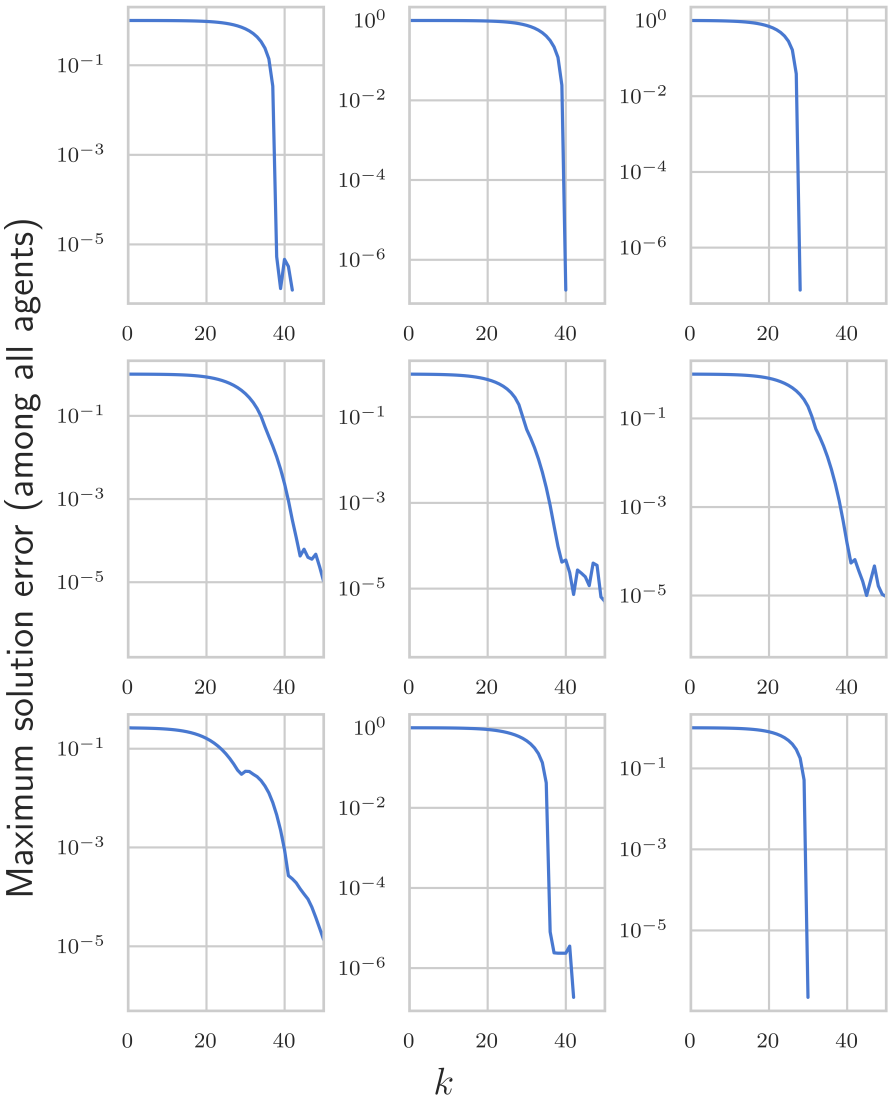


Figure 4.10: Nine examples of the maximum solution error at each iteration k, when the finite differences method was used. A complete graph and exponential cost functions were used, and $\rho^0 = 100$. The experiments plotted were selected at random from the set experiments that converged to a solution before $k = 100$.

Figure 4.12 shows the average maximum solution error. The extremely high average shows that the optimization algorithm diverged. However, the box plots (Figure 4.13) show that there was a lot of outliers that did achieve the desired solution tolerance before $k = 100$ iterations. Therefore, the algorithm sometimes converged, but this fact did not show up in Figure 4.12 due to the mean error of the diverging estimates dominating the ones that converged.

Finally, Figure 4.14 shows nine examples of how the maximum solution error (among all agents) progresses at each step k. The examples were selected at random from the set of experiments that converged before $k = 100$, with $\rho^0 = 100$. The figure shows that the convergence speed tended to be superlinear if the algorithm converged.

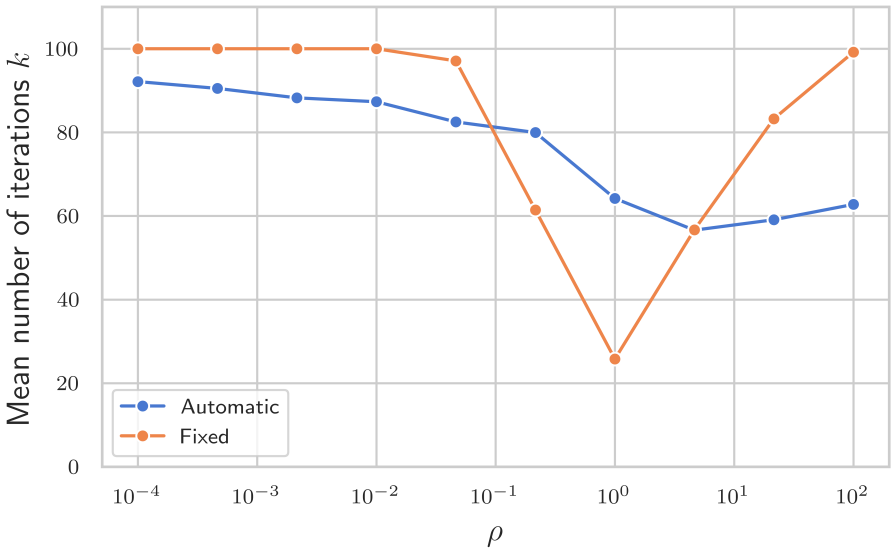


Figure 4.11: The performance of the finite differences method for selecting ρ, when using a random binomial graph and quadratic cost functions. The results were compared with using a fixed ρ to solve the same optimization problems. The maximum allowed number of iterations was 100.

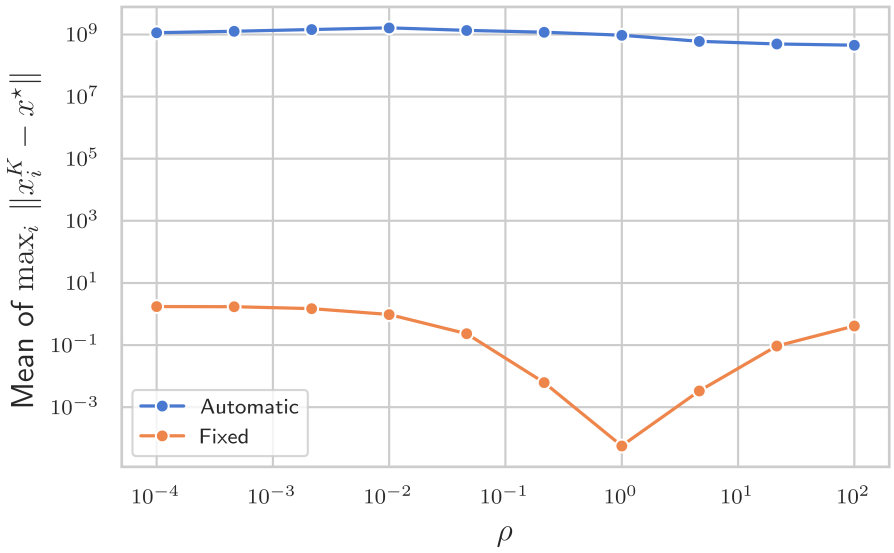
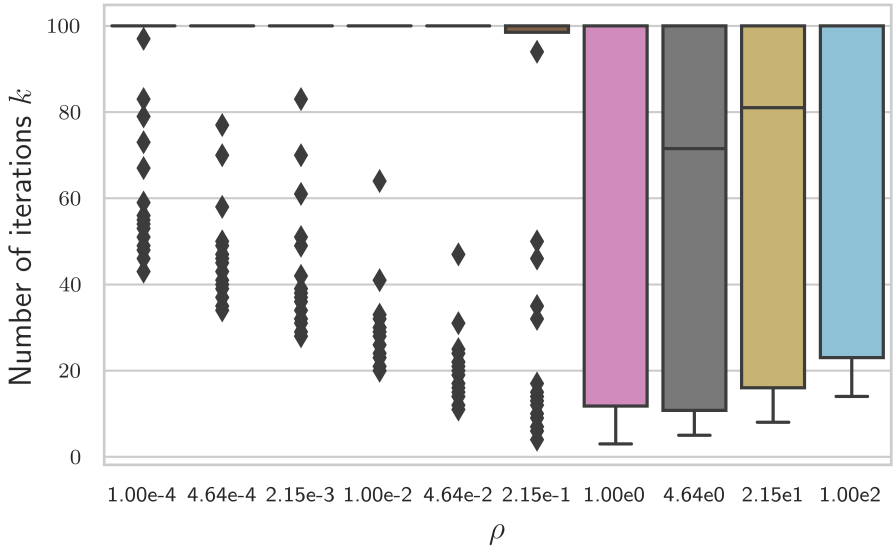
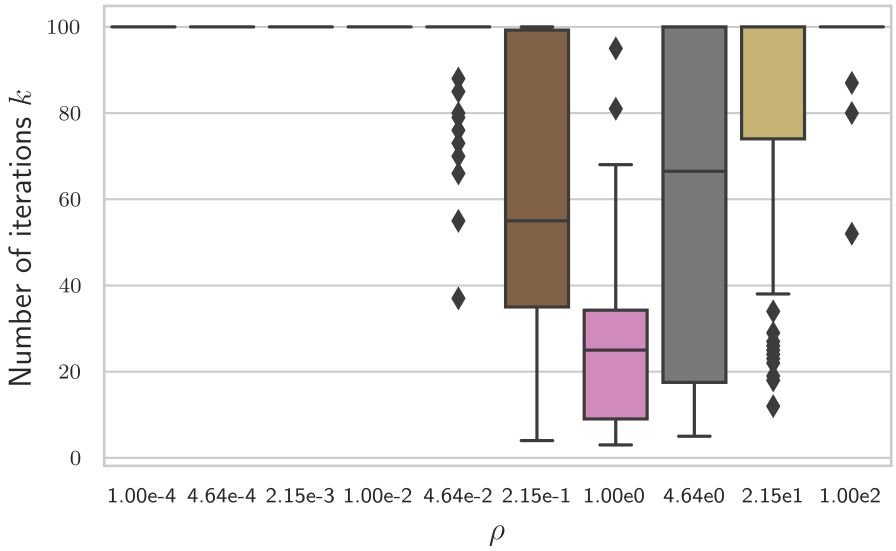


Figure 4.12: The mean maximum solution error when using the finite differences method, using a random binomial graph and quadratic cost functions. The result was compared with using a fixed ρ.



(a) Automatically selected ρ



(b) Fixed ρ

Figure 4.13: Box plots of the number of number of iterations required to achieve the solution error tolerance, when using a random binomial graph and quadratic cost functions. The finite differences method was compared with using a fixed ρ. The maximum number of allowed iterations was 100.

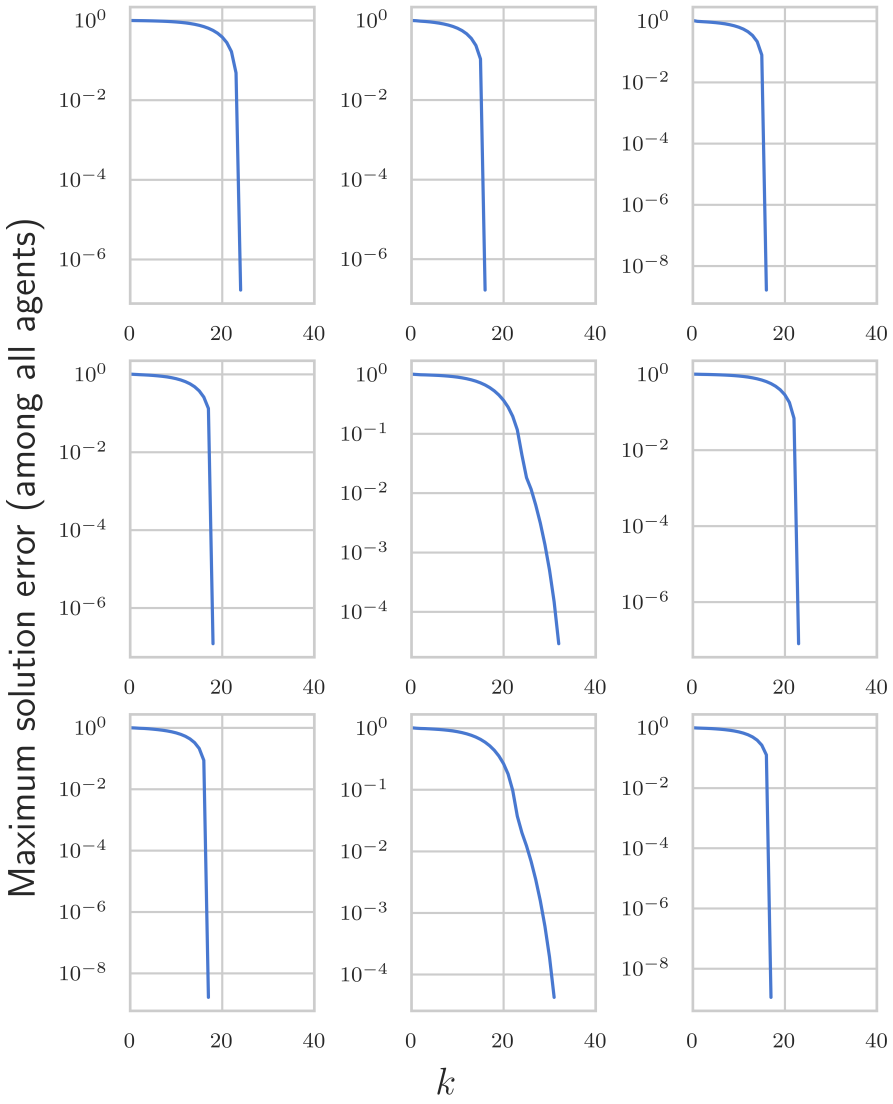


Figure 4.14: Nine examples of the maximum solution error at each iteration k, when the finite differences method was used. A random binomial graph and quadratic cost functions were used, and $\rho^0 = 100$. The experiments plotted were selected at random from the set experiments that converged to a solution before $k = 100$.

Chapter 5

Discussion

5.1 Using the Fast Fourier Transform

Selecting the penalty parameter using the Fast Fourier Transform did not work as expected. The performance for low initial penalty parameters was bad, with large mean solution errors. However, for large initial ρ, the performance of the FFT method was better.

The reason the algorithm did not work is that the sequence of x^k estimates received from neighbors were quite smooth. This smoothness is present whether the initial penalty parameter is selected too low or too high. For instance, if ρ^0 is too high, too much focus is placed on consensus over optimization. Therefore, the estimates will quickly converge to a common value for all agents. The estimates will then remain at that value, since almost no emphasis is placed on optimizing $\sum_i f_i(x)$.

Similarly, if the penalty parameter is selected too low, almost no emphasis will be put on consensus. Therefore, each agent's x_i will quickly converge to the local solution of $f_i(x)$. The estimates will remain there, since almost no emphasis is put on consensus.

As mentioned, in both of these cases the sequence of x^k received from neighbors will be smooth. Due to this, the Fast Fourier Transform will always return small magnitudes (see Algorithm 3.2).

The small magnitudes will lead to a decrease in ρ (see Algorithm 3.3). This is a good choice when ρ^0 is selected too high. This explains the good performance for large initial penalty parameters. However, as mentioned, the x^k sequences are always smooth, and therefore ρ will also decrease when ρ^0 is selected too low. This is clearly the wrong choice. This explains the bad performance for small initial penalty parameters.

For these reasons, the algorithm proposed in Algorithm 3.2 is not a good measure of the reliability of neighboring estimates. Therefore, the proposed FFT method for selecting ρ does not work.

During the course of this thesis, a large amount of time was spent trying to improve this method. Despite this, no improvement was found, and near the end of the thesis I ceased experimenting with the FFT method.

5.2 Comparing finite differences on complete graphs

When evaluating the finite differences method with complete graphs and quadratic cost functions, the results were encouraging. The average maximum solution error varied between 10^{-2} to 10^{-3}, while the average for the fixed ρ varied between 10^{0} and 10^{-4}. While using a fixed ρ resulted in better average performance in the best case, the performance was worse than the finite differences method when ρ was selected badly.

The results for quadratic cost functions were promising. However, optimization problems with quadratic cost functions tend to be quite easy to solve, so performing experiments with other types of cost functions was deemed necessary.

When performing experiments with the same complete graphs, but with exponential cost functions, the results remained favorable for the finite differences method. If a good initial penalty parameter was selected, the performance was still better when using a fixed ρ. However, for all other ρ^0, the performance was better for the finite differences method.

The fact that the convergence speed tended to be superlinear (if the algorithm converged) is very interesting. ADMM can achieve superlinear performance, but it is only proven under very specific conditions [1].

The results of the experiments with exponential cost functions solidified the confidence in the finite differences method. A summary of the results is that the sensitivity to the initially chosen penalty parameter was much lower when using the finite differences method than when using a fixed penalty parameter. It is difficult to select the best possible ρ^0 ahead of time, so the fact that the finite differences method was less sensitive indicates that the method has potential for application in the real world.

5.3 Comparing finite differences on random binomial graphs

The finite differences method was evaluated on random binomial graphs. As mentioned, due to time constraints and the large computational burden required to run the experiments, the solution error tolerance had to be increased. Quadratic cost functions were also used, since the primary goal was to observe the performance on a graph that was not fully connected.

The mean number of iterations required to achieve the solution tolerance remained less than when using a fixed ρ (Figure 4.11) . However, the mean solution distance (Figure 4.12) was extremely large, at 10^8 for all $\rho \in P$. The interpretation of this is that the algorithm tended to diverge.

It is important to note that the algorithm did not always diverge. The mean solution distance was large because, when computing the mean, the results from diverging experiments dominated the results where the solution error was small. A more complete picture of the performance is shown in the box plots (Figure 4.13), where there are many outliers that reach the desired solution tolerance.

Having an algorithm that tends to diverge is clearly bad. However, the performance on complete graphs makes me hopeful that the algorithm can be amended to also work on random graphs. Sadly, the thesis was time-constrained, and therefore there was no more time for developing the algorithm further.

Nevertheless, I can provide some thoughts on why the algorithm would diverge on a graph that is not fully connected. Consider for instance a random graph where there is one agent that is much less connected than all other agents. It follows that the set of agents that are very connected have more information available to select ρ. Because of this, the less connected agent may choose a ρ which is very different compared to the other agents.

This can become a problem. Recall that each agent i chooses its dual variable associated with neighbor j according to

$$\lambda_{ij}^{k+1} = \lambda_{ij}^{k} + \rho\left(x_i^{k+1} - z_j^{k+1}\right),$$

and updates its auxiliary primal variable z_i according to

$$z_i^{k+1} = \frac{\sum_{j\in\mathcal{N}_i\cup\{i\}} x_j^{k+1}}{|\mathcal{N}_i|+1} + \frac{\sum_{j\in\mathcal{N}_i\cup\{i\}} \lambda_{ji}^{k}}{\rho\left(|\mathcal{N}_i|+1\right)}.$$

In the z_i update, if we assume that ρ is fixed, the weighted average of the dual variables λ_{ij} essentially becomes the average of the integrals of the errors $x_i - z_j$ from $k = 0$ up to the current iteration. That is, each $\frac{\lambda_{ij}}{\rho}$ becomes an integral of the errors $x_i - z_j$. This is because ρ cancels out in the expression for λ_{ij}.

Now, if we assume that ρ is no longer fixed, the following might happen. Let ρ_N denote the penalty parameter selected by the less connected agent, and assume that all other agents select penalty parameters close to ρ_C. Then, if ρ_N is very different from ρ_C the algorithm may diverge.

The reasoning is as follows. In the expression for z_i^{k+1}, dividing by ρ_N will cause each λ_{ij}^{k+1} received from neighbors to be divided by ρ_N. To become the proper integral of errors, the expression for λ_{ij}^{k+1} should have been divided by ρ_C instead. If for instance $\rho_C \gg \rho_N$, the error will be interpreted as larger than it actually is.

This will obviously shift the value of z_i, and since z_i is used for the coherence constraints, this will also shift the solution of the primal problem. The erroneously computed z_i will also be transmitted to neighbors, causing further issues.

The result is that the expression for the z_i update is no longer valid when varying the penalty parameter. However, if the graph is complete, the agents may select similar enough values of ρ, such that it is still a good approximation. This explains the results in the previous sections. However, when introducing a random graph, agents will no longer have the same information available, and the z_i expression no longer holds.

Finally, the fact that the convergence speed tended to be superlinear also for random binomial graphs (once again, if the algorithm converged) is extremely interesting. Unfortunately there was not enough time left for me to explore this result further in this thesis.

5.4 Limitations and further work

The most important limitation is that, in a certain sense, we have moved the problem from selecting ρ to selecting hyperparameters for the finite differences method. However, my assertion is that it is more difficult to select a good ρ than good hyperparameters for the finite differences method. There is no clear interpretation of the penalty parameter, whereas the finite differences are more interpretable in the sense that they are approximations of derivatives.

Despite this, it can be worth illuminating some issues with the hyperparameters used in the experiments. First, ρ is increased or decreased by 20% in each iteration. This will clearly be too aggressive in some situations. In other situations, for instance when the optimal $\rho = 1$ but $\rho^0 = 100$, we will spend many iterations decreasing ρ by 20%. A similar argument follows for the threshold for deciding whether to increase or decrease ρ.

In this thesis, the hyperparameters were selected somewhat arbitrarily. It is not certain that these were the optimal parameters. For instance, the divergence might have been exacerbated by the selection of hyperparameters. More research is needed to determine whether it is possible to select the hyperparameters in a better way.

Chapter 6

Conclusion

Selecting the penalty parameter for the Alternating Direction Method of Multipliers is a challenging problem. In this thesis, I proposed two novel methods. The FFT-based method did not work due to the relative stability of the x sequences received from neighbors. The method based on comparing finite differences is promising, and saw good performance when performing experiments with complete graphs.

There are two limitations to the finite differences method. First, the method tends to diverge under challenging conditions, illustrated by the experiments on random binomial graphs. Secondly, the method is not completely automatic, because it has hyperparameters that must be tuned.

Above all, this project has demonstrated that using neighboring x estimates to compute is a promising idea. However, the particular method I arrived at in this thesis has, as mentioned, several issues. There was not enough time to fix these issues during the course of this thesis, but I will explore solutions to these limitations in further research.

Bibliography

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011, ISSN: 1935-8237. DOI: 10.1561/2200000016.

[2] G. Notarstefano, I. Notarnicola, and A. Camisa, "Distributed Optimization for Smart Cyber-Physical Networks," *Foundations and Trends® in Systems and Control*, vol. 7, no. 3, pp. 253–383, 2019, ISSN: 2325-6818. DOI: 10.1561/2600000020.

[3] F. Farina, A. Camisa, A. Testa, I. Notarnicola, and G. Notarstefano, "DISROPT: a Python Framework for Distributed Optimization," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 2666–2671, 2020, 21st IFAC World Congress, ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2020.12.382.

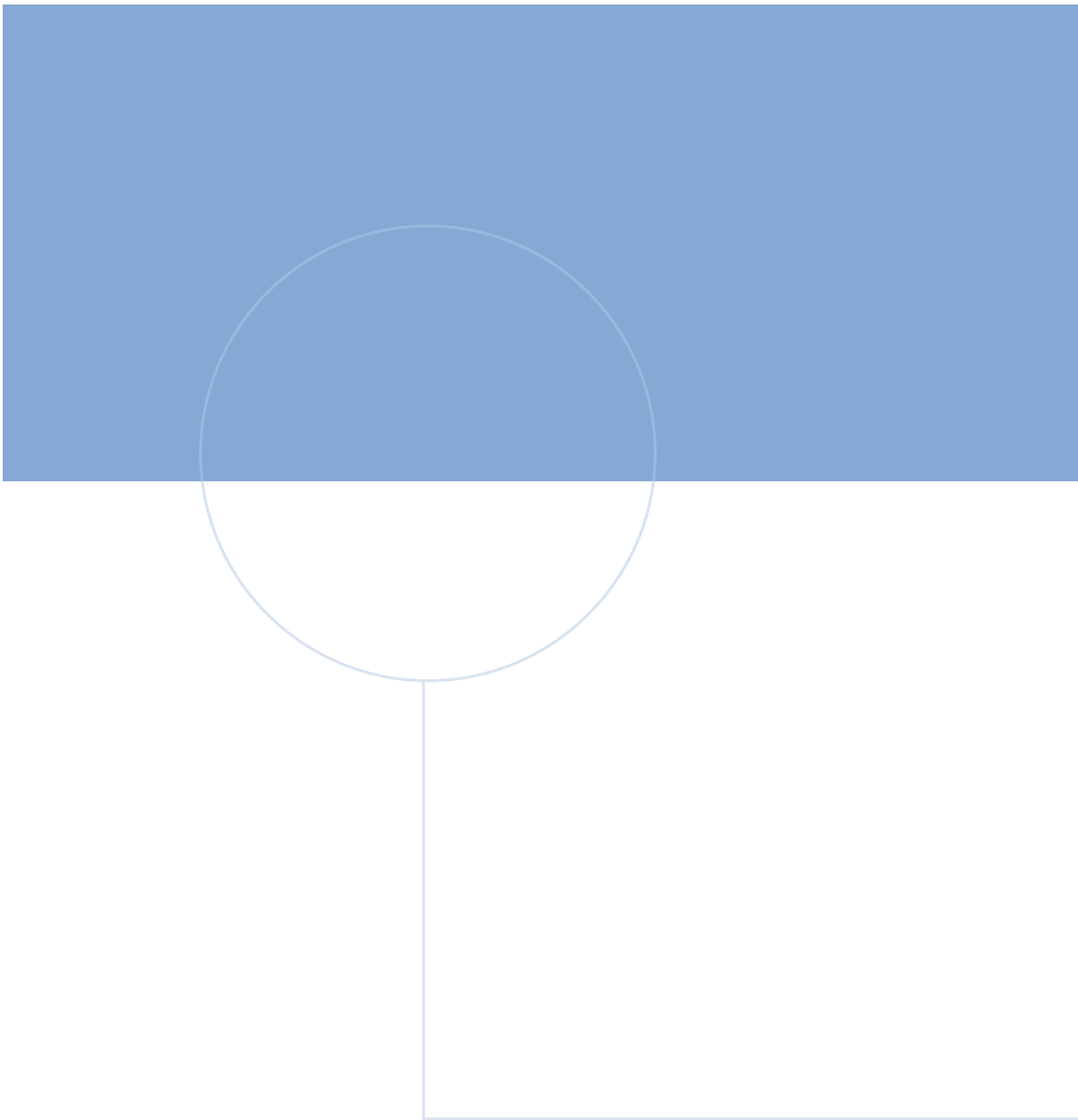
[4] B. Wohlberg, "ADMM penalty parameter selection by residual balancing," 2017. arXiv: 1704.06209.

[5] C. Song, S. Yoon, and V. Pavlovic, "Fast ADMM Algorithm for Distributed Optimization with Adaptive Penalty," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, Feb. 2016. DOI: 10.1609/aaai.v30i1.10069.

[6] Z. Xu, M. Figueiredo, and T. Goldstein, "Adaptive ADMM with Spectral Penalty Parameter Selection," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, A. Singh and J. Zhu, Eds., ser. Proceedings of Machine Learning Research, vol. 54, PMLR, 20–22 Apr 2017, pp. 718–727. [Online]. Available: https://proceedings.mlr.press/v54/xu17a.html.

[7] Z. Xu, G. Taylor, H. Li, M. A. T. Figueiredo, X. Yuan, and T. Goldstein, "Adaptive Consensus ADMM for Distributed Optimization," in *Proceedings of the 34th International Conference on*

Machine Learning, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, Jun. 2017, pp. 3841–3850. [Online]. Available: https://proceedings.mlr.press/v70/xu17c.html.



NTNU

Norwegian University of
Science and Technology