Inexact Distributed Optimization Schemes

A convergence analysis using monotone operator theory

N. (Niels) van Wijngaarden

TU Delft

Inexact Distributed Optimization Schemes

A convergence analysis using monotone operator theory

by

N. (Niels) van Wijngaarden

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday July 17, 2018 at 13:30.

Student number:	4063899	
Thesis committee:	Dr. ir. R. Heusdens,	TU Delft, supervisor
	Dr. ir. R.C. Hendriks,	TU Delft
	Dr. ir. J.C.A. van der Lubbe,	TU Delft
	T.W. Sherson	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Distributed optimization has been an extensively studied field for years. Recent developments in the area of sensors makes it possible to create networks consisting of a large number of nodes. The focus of this thesis will be optimizing distributed problems over a decentralized network. These distributed optimization schemes operate in an iterative matter as follows. First each node performs some local computations, after which the data is transmitted to its neighbours. The purpose of this study is to investigate the effects of approximating these local computations inexactly on the convergence of distributed optimization schemes. Although we consider many optimization schemes in general, the primal-dual method of multipliers (PDMM) is used during the simulations. Therefore we start off by deriving the inexact iteration for PDMM which shows how the inexactness propagates through the iterates. This derivation also suggests that the inexactness depends on the optimization constant, which was verified during the simulations. After that, the convergence of distributed optimization schemes is analyzed by making use of monotone operator theory to investigate under which conditions convergence will be reached. This convergence analysis has two main results. It firstly shows that distributed optimization schemes converge to a fixed point if the error is summable and secondly that an error has less influence as iterations pass. Thereafter simulations are presented that suggest that the inexactness affects how far the algorithm converges, thus what the remaining error is when convergence is reached. Decreasing the error when convergence is reached causes the inexact PDMM iteration to resume converging at the rate of the standard PDMM algorithm. These observations holds in synchronous as well as asynchronous operation. Introducing packet loss only influences the convergence rate of the inexact PDMM iteration.

Preface

Before you lies my master thesis submitted to obtain the degree Master of Science (MSc) in Electrical Engineering at the Delft University of Technology (TU Delft), Netherlands. Writing this thesis has been a challenging but enriching experience and would not have been possible without people supporting me.

Hereby, I would like to take the opportunity to express my gratitude to a few people. The research in this thesis has been conducted under the supervision of Dr. ir. R. Heusdens and Ph.D student T. Sherson to whom I am both very grateful for their valuable support and guidance throughout this research. Secondly I wish to thank my colleagues, with a special thanks to Jake Jonkman for the valuable discussions, who has been kind enough to proofread large parts of my thesis and has contributed to the nice ambiance. Lastly, I would like to thank my family for their support throughout the process. I would not have been able to complete my thesis without all of you.

I sincerely hope you will enjoy reading my thesis.

N. van Wijngaarden
Delft,
July 5, 2018

Contents

1 Introduction **1**

2 Background **5**
2.1 Introduction . . . 5
2.2 Iterative methods . . . 7

3 Monotone operator theory **9**
3.1 Fixed point iteration . . . 12
3.2 Inexact Krasnosel'skii-Mann iteration . . . 12
3.3 Operator splitting methods . . . 13
3.3.1 Peaceman-Rachford splitting method . . . 13
3.3.2 Douglas-Rachford splitting method . . . 14
3.4 Primal-dual method of multipliers . . . 15
3.5 Inexact primal-dual method of multipliers . . . 16
3.6 Determining the optimal z . . . 17
3.7 Asynchronous updates and packet loss . . . 19

4 Convergence analysis **21**

5 Results **25**
5.1 Distributed average consensus . . . 25
5.1.1 Comparison between domains. . . . 26
5.1.2 Inexact PDMM iteration . . . 27
5.1.3 Synchronous . . . 28
5.1.4 Asynchronous . . . 30
5.2 P-norm optimization . . . 31
5.2.1 Synchronous . . . 31
5.2.2 Asynchronous . . . 33
5.3 Channel capacity problem . . . 35
5.3.1 Synchronous . . . 36
5.3.2 Asynchronous . . . 38

6 Conclusion and Future Work **41**
6.1 Conclusion . . . 41
6.2 Future work . . . 42

A Primal-dual method of multipliers **45**
A.1 General PDMM . . . 45

B Distributed optimization problems **51**
B.1 Distributed average consensus . . . 51
B.2 P-norm optimization . . . 53
B.3 Channel capacity optimization . . . 55
B.3.1 Inexact Channel capacity problem . . . 59

Bibliography **61**

1 Introduction

Distributed optimization has been an extensively studied field for many years [8, 17]. Solving a problem over a decentralized network has certain advantages such as the ease to alter the topology, and the added robustness since the network is not completely dependent on one central node meaning that the network does not contain a single point of failure. Recent developments in the area of sensors make it possible to have cheap sensors that have some processing and communication capabilities to both perform some local computations and to share data with their neighbors. This has increased the interest in distributed optimization schemes even more.

A sensor network consists of a large number of nodes. These can for example be distributed in a random way, meaning that each node does not need to have a specified location. Each node consists of a sensor, a processing unit and has the ability to communicate with his neighbors. Interested readers are referred to [1] for a more comprehensive overview of sensor networks in general.

The main interest will be distributed problems that are optimized over a decentralized network. This means that there will not be a central processing point present, in contrast to a centralized network. All nodes will have the capability to do some computations and communicate. Because of these capabilities, the networks can be modelled as undirected connected graphs. With undirected meaning that if there is a connection between two nodes, the connection is bidirectional. Also a graph is called connected if there exists a path between every pair of nodes in the network, thus none of the nodes is unreachable. The concept of a decentralized network in comparison to a centralized network is graphically displayed in Figure 1.1.

There are many practical examples for which these decentralized sensor networks can be used. For example, consider disaster management or more specifically forest fire detection. The United States Forest Service has spent more than $2 billion on fire suppression over the last year alone [19], which indicates the impact forest fires have. With the changing climate only contributing to the increase of forest fires, the need to find ways to fight forest fires also increases. Besides prevention, early detection is of crucial importance. This is one example of a problem where distributed optimization can be used. Imagine placing a large number of cheap temperature sensors within a forest, which creates a wireless sensor network where each of the sensors represents a node of the network. These sensors could collaboratively collect and process data to detect a forest fire in an early stage, which ensures that the damage can be controlled as much as possible.

This thesis focuses on algorithms that solve the minimization of the sum of convex functions in a distributed way. These distributed problems will be optimized over a graphical model $G = (V, E)$, where V is the vertex set (nodes) and E the set of undirected edges or connections between nodes. We also define the number of nodes as $n = |V|$ and the number of vertices as $m = |E|$. The considered problems take on the following form

$$\begin{aligned} \min_{x} \quad & \sum_{i \in V} f_i(x_i), \\ s.t. \quad & A_{ij}x_i + A_{ji}x_j = b_{ij}, \qquad \forall (i,j) \in E, \end{aligned} \tag{1.1}$$

with variable $x \in \mathbb{R}^n$, $f_i : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ closed convex and proper (CCP) functions, f_i the objective function of node i, $A \in \mathbb{R}^{m \times n}$ defining the connections and $b \in \mathbb{R}^m$ defining the constraints between the nodes. The

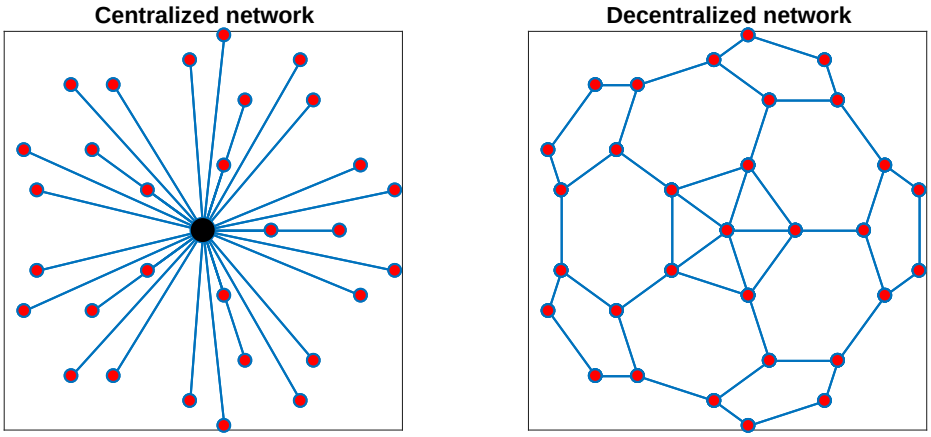

Figure 1.1: The difference between a centralized network and a decentralized network both consisting of 31 nodes. The central processing point is displayed in black, the nodes in red and the edges are given by the blue lines

A matrix is not necessarily full rank. However, the linearly dependent constraints defined by A and b can be removed if the matrix is not full rank. The newly constructed full rank matrix combined with the new vector can be substituted for the old A matrix and b vector, meaning that it can be assumed that A is full rank.

There are two approaches that are mainly used to solve a problem of this form (1.1), one based on probabilistic inference and the other on convex optimization. Probabilistic inference determines the probability of a certain instance happening, as the name suggests. Examples of probabilistic inference algorithms are the sum-product and the max-sum algorithm [21]. These algorithms can solve quadratic objective functions by message passing, which means that information is sent forwardly and backwardly through a network. Another probabilistic inference algorithm is the linear coordinate descent algorithm, which does not have the requirement that the objective function has to be quadratic [29].

The second option to solve problems that attain the form of (1.1) is by means of convex optimization, which is where this research is focused on. One of the earliest methods based on convex optimization is the dual-ascent method [4, 13]. This method uses gradient ascent to solve the dual problem, which means that the assumption is made that the dual function is differentiable. There are objective function without differentiable dual function for which the dual-ascent method can still be used, however, this can only be done under certain strong assumptions that do not hold in many applications [8, p. 8]. In this case the method becomes a subgradient method and has a much slower convergence rate. The method of multipliers was proposed in order to have an algorithm that is more useful in practical applications. This algorithm uses the augmented Lagrangian, causing it to need fewer restrictions in order to guarantee convergence. The drawback this method has is that if the objective function is separable, the augmented Lagrangian will not be separable meaning that the updates cannot be done in parallel. An algorithm that combines the advantages both of the previously discussed algorithms have is the alternating direction method of multipliers (ADMM) [8, 11]. This algorithm does not require strong assumptions while it still has the benefit of seperability, meaning that it is possible to update in parallel. Another promising algorithm, called the primal-dual method of mulitpliers (PDMM), has recently been proposed [30–33]. The advantage of ADMM is that the algorithm is guaranteed to converge for every problem, whereas the PDMM algorithm has the advantage of an higher convergence rate but does not always converge.

The previously discussed distributed algorithms operate in an iterative manner. First, a minimization is solved locally at each node. The solution of this minimization is then manipulated with linear operations, after which the result is transmitted to its neighbors. These steps are repeatedly performed in order to get a good approximation of the optimal point. The local computation consists of a minimization for which it is not always possible to derive an algebraic expression. If an algebraic expression for the local computation does not exist, the solution is found by using an iterative scheme such as a proximal point method or the

Table 1.1: The different optimization problems that are considered for the simulations. The objective function of the channel capacity optimization problem is optimized over both μ_i and λ_i

Problem	Corresponding objective function
Distributed average consensus	$\sum_{i\in V} \frac{1}{2}(x_i - x_i^{(0)})^2$
P-norm optimization	$\|x-a\|_p^p$
Channel capacity optimization	$\sum \left[\log_2\left(\frac{-1}{\mu_i+\lambda_i}\right) - \mu_i\sigma_i^2 - \lambda_i\sigma_i^2 - \frac{\mu_i}{N}\right]$

Newton-Raphson method. Since a finite number of iterations is used, this will cause the approximation to be inexact, where the error of the solution depends on the accuracy of the minimization done locally at a node. Therefore it is interesting to investigate the convergence properties with this inexactness incorporated. The first thing to look at are the mathematical properties of distributed optimization schemes when inexactness is introduced. After that it is important to validate these results with simulations. During the simulations, the PDMM algorithm is used to solve three selected problems that can be found in Table 1.1. These simulations minimize an objective function, which is of the form of (1.1), in a distributed way. To confirm that the results are not problem dependent, three different objective functions will be considered. The first is an averaging problem, where each node gets a value assigned and the goal is to find the average of these values. The objective of the second problem is to minimize a p-norm to the power p. The final problem is the channel capacity problem, where each node represents a channel and is characterized by its noise variance. The aim of this problem is to optimize the capacity of all channels combined.

Each of these objective functions (Table 1.1) is locally minimized by each node. The nodes often use an iterative method to approximate the solution to this problem, which is where the inexactness arises. This means that the chosen objective function as well as the iterative method influences the magnitude of the inexactness. The iterative method that will be used in this thesis for this approximation is the Newton-Raphson method, which appears to have first been used in 1669 [12, p. 64]. This method makes a second order approximation of a desired function at a given point within the domain of the function [6, p. 484].

To solve the averaging problem, a quadratic objective function is minimized. The second order approximation of the Newton-Raphson method will give an accurate solution in a single step. Secondly a p-norm optimization problem is considered to investigate the behavior for higher powers than two (where p defines the order of the problem). A second order approximation of an higher power than two will not be accurate anymore, which results in inexactness. Finally a Gaussian channel capacity problem is formulated. This problem differentiates itself from the others in the sense that it includes a logarithm in the objective function and must be optimized over two variables. This means that the objective function that belongs to this problem can also not be accurately approximated by the Newton-Raphson method and therefore results in inexactness. The Gaussian channel capacity problem is in essence not a distributed problem and is therefore rewritten, this causes the objective function to be dependent on two optimization variables, instead of one. The actual objective functions for each of these problems are summarized in Table 1.1 and are discussed in depth in Appendix B.

One more differentiation will be made during the simulations, the synchronous and asynchronous updating of the variables. Synchronously updating entails all the nodes sending their local update at the exact same moment. From a practical standpoint the asynchronous updating schemes are of particular interest, the reason being that the clock of all the nodes must be perfectly synchronized in order to have the network operate synchronously. This makes the distributed optimization scheme still dependent on a global clock, which goes against the motivation to use a distributed optimization scheme. A second argument as to why asynchronous operation is important, is that this makes the optimization scheme more robust against changes in topology since nodes can easily be added or removed without having to synchronize it to the rest of the network. Finally, the convergence of the algorithm in the case of packet loss is investigated. This is implemented in the asynchronous operation by only updating a part of the neighbors of the chosen node, where the probability of updating a certain neighbor is predefined.

To summarize the content of this chapter and define the direction of the research, the following research question is formulated:

Research question:

What is the influence of inexact local updates on the convergence of distributed optimization schemes? Will the effects be different in asynchronous operations in comparison to synchronous operation and does packet loss have an influence on the convergence?

The remainder of this document is organized as follows: Chapter 2 will start with a background on some mathematical principles and a short introduction into monotone operator theory. The PDMM algorithm, with the influence of the inexact update included, is derived in Chapter 3. Chapter 4 gives the mathematical proof that distributed optimization schemes still converge in the case of inexactness. The three different optimization problems are defined and cast into the PDMM iterates to study how PDMM can be used to solve the problems in Chapter 5. Chapter 6 contains the results from the simulations. A conclusion, discussion and future work are discussed in Chapter 7.

2

Background

This chapter gives an overview of the important mathematical definitions that will be used throughout this document. Some relevant convex set and function theory will be presented first. Then a basic introduction into monotone operator theory is given. Finally, the fixed point iteration, the Krasnosel'skii-Mann iteration and two different splitting methods are discussed. This chapter will only scratch the surface of most subjects. A more in depth explanation of convex functions can be found in [3, 6], while readers are referred to [24] for a more comprehensive introduction into monotone operator theory.

Unless otherwise specified, $\|\cdot\|$ is the 2-norm and $\log(\cdot)$ is the base-2 logarithm throughout the remainder of this document.

2.1. Introduction

To start off, a few important properties of convex functions are explained that will serve as a foundation for the remainder of this work.

Convex set. A set C is a convex set if for any $x, y \in C$ and $0 \leq \theta \leq 1$, we have

$$\theta x + (1-\theta) y \in C.$$

Intuitively this means that the line segment between any 2 points of the set lies fully in the set. The concept of a convex set is graphically demonstrated in Figure 2.1.

Convex function. A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if the domain of the function is a convex set and if for all $x, y \in \mathbb{R}^n$, and with θ restricted by $0 \leq \theta \leq 1$, the following holds

$$f(\theta x + (1-\theta) y) \leq \theta f(x) + (1-\theta) f(y). \tag{2.1}$$

To develop some intuition, this means that the chord from x to y completely lies above or on the graph of f. A function is called strictly convex if (2.1) holds with strict inequality. An example of a convex function is given in Figure 2.2.

Subgradient. A function does not necessarily have to be differentiable. We define the subgradient, which holds for the points where a function is differentiable as well as the points where a function is non differentiable. The subgradient $g \in \mathbb{R}^n$ of the function $f : \mathbb{R}^n \to \mathbb{R}$ at the point $x \in \mathbb{R}^n$ is defined if for all $y \in \mathbb{R}^n$ the following holds

$$f(y) \geq f(x) + g^T (y - x).$$

Intuitively this means that the function should never lie below the tangent line to the function at the point x with gradient g. Note that for differentiable points we have $g = \{\nabla f(x)\}$, meaning that the subgradient becomes the derivative. A graphical illustration with three subgradients can be found in Figure 2.3.

Subdifferential. The set containing all subgradients of a function f is called the subdifferential and is defined as

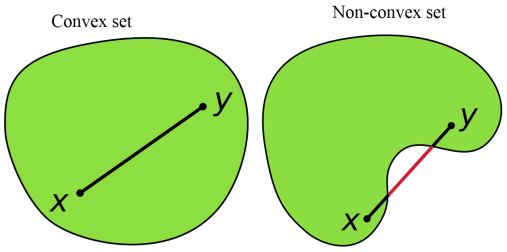

Figure 2.1: Convex set

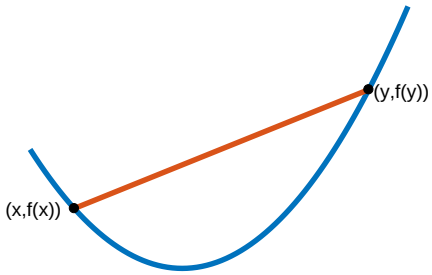

Figure 2.2: (strictly) Convex function

$$\partial f = \{(x, g) | x \in \mathbb{R}^n, \forall y \in \mathbb{R}^n : f(y) \geq f(x) + g^T(y - x)\}.$$

Convex function. Functions that contain non differentiable points can still be convex. To include these functions, the convexity requirement (2.1) is rewritten. A function $f : \mathbb{R}^n \to \mathbb{R}$ is a convex function of for all $x, y \in \mathbb{R}^n$ the following holds

$$f(y) \geq f(x) + g^T(y - x).$$

This requirement is the same as the definition of the subgradient, with the added constraint that it has to hold for all x in the domain of f. This is a crucial difference since the subgradient is not necessarily defined for all x in the domain of f. Recall that in the case that f is differentiable at x, we have $\partial f(x) = \{\nabla f(x)\}$, which is a singleton. This means that this requirement is applicable to both differentiable functions as well as non differentiable functions.

Indicator function. For an example of a subdifferential we first consider the indicator function, $I_C(x)$. The indicator function to a closed convex set C is defined as

$$I_C = \begin{cases} 0, & x \in C, \\ \infty, & x \notin C. \end{cases}$$

Normal cone operator. The subdifferential of the indicator function, $\partial I_C(x)$, is given by the normal cone operator as

$$N_C(x) = \begin{cases} \{g | g^T(y - x) \leq 0, \forall y \in C\}, & x \in C \\ \emptyset, & x \notin C. \end{cases}$$

At the boundary of the domain C, $N_C(x)$ is pointing outwards. This means that the number of subgradients at a given x depends on the smoothness of the boundary of C at that point. A boundary that is not smooth will result in multiple subgradients at the point where the non smoothness occurs. Another reason that both these functions are introduces here is the fact that both are needed for the derivation of the PDMM algorithm.

Closed function. A function $f : \mathbb{R}^n \to \mathbb{R}$ is closed if it satisfies the following three conditions that are equivalent: [3, Proposition 2.5]

1. The function f is a lower-semicontinuous function.
2. For all $\alpha \in \mathbb{R}$ the level set $\{x \in \mathbb{R}^n | f(x) \leq \alpha\}$ is closed.
3. The epigraph of the function defined as $\text{epi} f = \{(x, \lambda) | x \in \mathbb{R}^n, \lambda \in \mathbb{R}, f(x) \leq \lambda\}$ is closed.

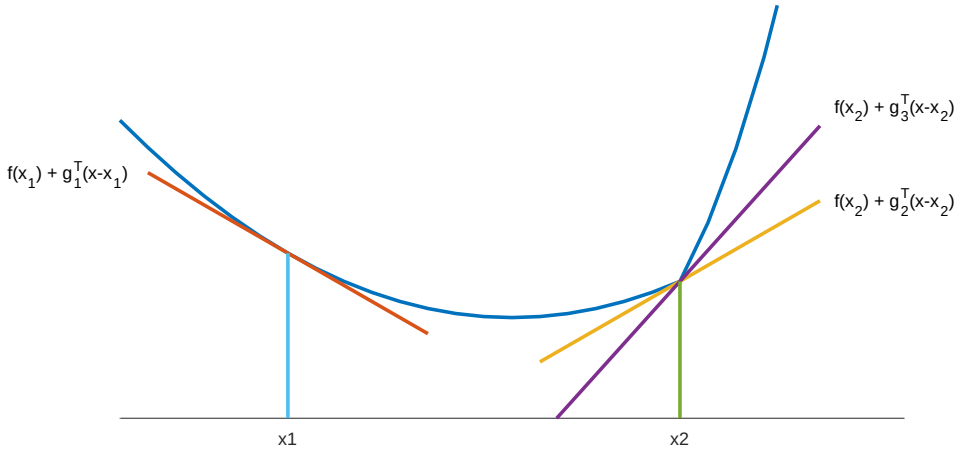

Figure 2.3: Three subgradients of the function in blue, subgradient g_1 at the point x_1 and subgradients g_2 and g_3 at the point x_2

Proper convex function. A convex function f that is not the constant function $+\infty$ and for which $f(x) > -\infty$ holds for every x in its domain is called a proper convex function.

The functions in the remainder of this document are assumed to be closed, convex and proper (CCP).

Lipschitz continuity. An important tool that will be used in the convergence analysis later on is Lipschitz continuity. A function f is Lipschitz continuous with parameter $L \geq 0$ if for all $x, y \in \mathbb{R}^n$ the following holds

$$||f(y) - f(x)|| \leq L||y - x||. \tag{2.2}$$

The Lipschitz parameter L is a bound on the derivative of the function f. This means that a function that is Lipschitz continuous is limited in the magnitude of its change by its Lipschitz constant L.

Conjugate function. Another operation that plays a vital role later on in the derivation of the PDMM algorithm in Appendix A is the conjugate of a function. The conjugate $f^* : \mathbb{R}^n \to \mathbb{R}$ of the function $f : \mathbb{R}^n \to \mathbb{R}$ is given by

$$f^*(y) = \sup_x (y^T x - f(x)), \tag{2.3}$$

where $x, y \in \mathbb{R}^n$. Intuitively in the case of a function $f : \mathbb{R}^n \to \mathbb{R}$, the conjugate function can be interpreted as the maximum gap between the function $f(x)$ and the linear line xy. If the function $f(x)$ is closed convex and proper, which is assumed throughout this document, we have $f^{**} = f$ [23, Theorem 12.2]. Another useful identity of the conjugate function is $\partial f^* = (\partial f)^{-1}$ [24, p. 6].

2.2. Iterative methods

This thesis focuses on methods that solve distributed convex optimization problems, more specifically minimizing a convex function, in an iterative way. The aim of an iterative method is to generates a sequence of successive approximations, where each approximation makes use of the previous one, to reach a desired solution. Such a sequence of approximations is also called a minimizing sequence and is of the form $x^{(0)}, x^{(1)}, \ldots$ with $x^{(0)}$ some starting point and each of the points in the domain of the function.

Assuming that an optimal point, x^*, for the minimization exists we have $x^{(k)} \to x^{(*)}$ for $k \to \infty$. This means that the iterative algorithm goes from a starting point $x^{(0)}$ and moves towards the optimal point as iterations pass. A stopping tolerance $\epsilon > 0$ is specified to determine when the algorithm should stop. The algorithm is terminated at iteration k for which $\left\| x^{(k)} - x^* \right\| \leq \epsilon$ holds.

3

Monotone operator theory

Monotone operator theory is an important tool that can be used to derive many minimization algorithms. Besides the derivation, the properties of the monotone operators can also be used in the mathematical proof for the convergence of the algorithm.

In this work both the derivation of the PDMM algorithm in Appendix A as well as the convergence analysis in Chapter 4 make use of monotone operator theory. For the derivation of the PDMM algorithm a general minimization problem will be rewritten where the objective becomes to find the zero of a monotone operator. The PDMM algorithm is derived from this monotonic inclusion problem by applying a splitting method that is discussed later on in Section 3.3.1. The convergence analysis makes use of properties of monotone operators, such as Lipschitz continuity, in order to ensure convergence when the iterates are updates inexactly.

Operator. We begin by defining a relation or operator T that maps a point $x \in \mathbb{R}^n$ to a set $y \in \mathbb{R}^n$ and is formally defined as

$$T(x) = \{y \in \mathbb{R}^n | \exists x \in \mathbb{R}^n : (x, y) \in T\}.$$

In the case that an operator $T(x)$ is a point-to-point mapping instead of a point-to-set mapping for all x in the domain of T, the operator is called a function and will from now on be written as $T(x) = y$ in contrast to $T(x) = \{y\}$, which is the correct notations. An example of an operator is the identity operator given by $I = \{(x, x) | x \in \mathbb{R}^n\}$.

Domain. The domain of the operator T is defined as

$$\mathbf{dom}\, T = \{x \in \mathbb{R}^n | T(x) \neq \emptyset\}.$$

It is possible to have operators interact with each other in ways of summation and composition. These two operations are now, for the sake of clarity, defined.

Addition. First the sum of the two operators R and S is given by

$$R + S = \{(x, y + z) | (x, y) \in R, (x, z) \in S\}.$$

Composition. Secondly, the composition of the operators R and S is defined as

$$R \circ S = \{(x, y) | \exists z (x, z) \in S, (z, y) \in R\}.$$

To develop some intuition, consider the case where we have $R \circ S(x)$. First the operator S is applied on x, resulting in the set z as $S(x) = z$. Then the second operator R is applied to z, yielding the set y as $R(z) = y$ which is the solution of the composition applied on x.

Inverse. An important property of an operator is the inverse relation. The inverse of an operator always exists, even when the operator is a point-to-set mapping, and is defined as

$$T^{-1} = \{(x, y) | (y, x) \in T\}.$$

The name inverse could give the false intuition that the composition of an operator with its inverse results in identity, which is not the case. Written in a formal mathematical notation this means $T^{-1} \circ T \neq I$. As a counter example consider the zero relation defined as $0 = \{(x,0)|x \in \mathbb{R}^n\}$. Even though the composition of an operator with its inverse is not necessarily identity, an equality that does hold is $T^{-1} \circ T(x) = x$ if T^{-1} is a function.

Zero set. Many problems can be posed as finding the zero of an operator. This is written as $0 \in T(x)$, which means that the point x is a zero of the operator T. The zero set of an operator is the set containing all the zeros and is formally defined as

$$T^{-1}(\{0\}) = \{x|(x,0) \in T\}.$$

Monotone operator. The operator T is monotone if for all $u \in T(x)$, $v \in T(y)$ and $x, y \in \mathbb{R}^n$ it satisfies

$$(u-v)^T(x-y) \geq 0.$$

For some intuition, consider the two dimensional case, with $x, y \in \mathbb{R}$, where the operator T maps point-to-point and is therefore a function. A monotone function preserves the order, which means that the function is either entirely non-decreasing or entirely non-increasing. The monotonicity condition can in functional notation also be written as

$$(T(x) - T(y))^T(x-y) \geq 0. \tag{3.1}$$

Strongly monotone. An operator T is called strongly monotone or coercive with parameter m if for $x, y \in \mathbb{R}^n$, the following holds

$$(T(y) - T(x))^T(y-x) \geq m||y-x||^2.$$

Again consider the two dimensional case with $x, y \in \mathbb{R}$. Roughly speaking the inequality to determine whether an operator (or function in this case) is monotone (3.1) will become a strict inequality, causing the function to be either completely increasing or completely decreasing, where the minimal slope is determined by the parameter m. The inverse of a strongly monotone operator is single valued and Lipschitz continuous with parameter $L = 1/m$.

Maximal monotone. A stronger condition on an operator is maximal monotonicity. The operator T is called maximal monotone if T has no proper monotone extension. This means that there is no other monotone operator that properly contains it. The operator T not being a maximal monotone operator means that there exist a pair $(x,u) \notin T$ such that $T \cup \{(x,u)\}$ is still monotone.

Lipschitz continuity. Recall the definition of Lipschitz continuity for a function (2.2). This also applies to operators. Rewriting Lipschitz continuity for an operator T gives

$$||T(y) - T(x)|| \leq L||y-x||, \tag{3.2}$$

for all $(x,y) \in \mathbf{dom}\, T$. If the operator is Lipschitz continuous with parameter $L = 1$, the operator is nonexpansive and when $L < 1$ the operator is a contraction. A contractions implies that when the points x and y are mapped by the operator, the distance between the points becomes smaller. Or in other words, for iterative methods the distance between the points x and y decreases at each successive iteration and after sufficient iterations of applying the operator, a fixed point will be reached. For a nonexpansive operator it is guaranteed that the distance does not become larger when the points are mapped, however the distance can stay equal meaning that a fixed point will not be reached.

Now consider the two operators R and S with respectively Lipschitz constant L and $\tilde{L}$. The Lipschitz constant of the contraction $R \circ S$ has the Lipschitz constant $L\tilde{L}$ [24]. This implies that when either one of the operators is a contraction and the other is nonexpansive, the composition will be nonexpansive. This notion is important since the composition of two operators will be something that we will come across often in the remainder of this thesis.

Two other things that will be used very frequently throughout this document are the resolvent of an operator and the Cayley operation on an operator. More specifically, these are used for the operator splitting

methods in Section 3.3, the derivation of the PDMM algorithm in Appendix A as well as the convergence analysis in Chapter 4.

Resolvent. We start off by defining the resolvent of an operator. The resolvent with constant $c \in \mathbb{R}$ and $cT = \{(x, cy)|(x, y) \in T\}$ of an operator T is defined as

$$J_{cT} = (I + cT)^{-1},$$

Cayley operator. The Cayley operator or reflection operation on an operator T is defined as

$$C_{cT} = 2J_{cT} - I. \tag{3.3}$$

In the case that T is a monotone operator, both the resolvent and the Cayley operator are nonexpansive functions. Another property is that the composition of two nonexpansive operators is also nonexpansive, as discussed before. These notions are essential since the composition of two Cayley operations is assumed to be nonexpansive during the convergence analysis later on.

Cayley operator identity. At this point an identity of the Cayley operator needs to be derived. This identity is needed for the derivation of the operator splitting methods in section 3.3.

$$C_T \circ (I + \alpha T) = I - \alpha T \tag{3.4}$$

Proof.

$$\begin{aligned} C_T \circ (I + \alpha T)(x) &= 2(I + \alpha T)^{-1} \circ (I + \alpha T)(x) - (I + \alpha T)(x) \\ &= 2I(x) - (I + \alpha T)(x) \\ &= (I - \alpha T)(x) \end{aligned}$$

□

Proximal operator. Another operation needed during the derivation of the PDMM algorithm is the proximal operator or proximity operator. This operator is the resolvent of the subdifferential of a function. The proximal operator associated with the function f and parameter $c > 0$ is given by [24, p. 25]

$$J_{c\partial f}(x) = \text{prox}_{cf}(x) = \arg\min_u \left[f(u) + \frac{1}{2c} \|u - x\|^2 \right]. \tag{3.5}$$

The resolvent $J_{c\partial f} = (I + c\partial f)^{-1}$ is called the proximal operator. In the case that the function f is CCP, we have $\mathbf{dom} J_{c\partial f}(x) = \mathbb{R}^n$, even when $\mathbf{dom} f \neq \mathbb{R}^n$ [24, p. 25].

Fixed point set. A fixed point of the operator T is a point that is mapped to itself by the operator. The point x is a fixed point of an operator $T : \mathbb{R}^n \mapsto \mathbb{R}^n$ if $T(x) = x$. The set containing all fixed points is called the fixed point set of the operator T and is formally defined as

$$\mathbf{Fix} T = \{x \in \mathbb{R}^n | T(x) = x\},$$

and is a closed and convex set. The fixed point set can be empty, contain one point, but can also contain many points. In the case that the operator T is a contraction and its domain is $\mathbb{R}^n$, the fixed point set will contain exactly one point. [24, p. 6]

Representation lemma. The final thing to introduce about monotone operators is the representation lemma, which is needed for the derivation of the splitting methods later on. Every $z \in \mathbb{R}^n$ can be represented in at most one way by $z = x + cy$, where $y \in T(x)$, $c > 0$ and T a monotone operator on $\mathbb{R}^n$. In the case that the operator T is maximal monotone, any $z \in \mathbb{R}^n$ can uniquely be written as $z = x + cy$.

Proof.

$$\begin{aligned} &z = x + cy, \\ &z \in x + cT(x), \\ &z \in (I + cT)(x), \\ &x = (I + cT)^{-1}(z), \\ &x = J_{cT}(z). \end{aligned}$$

□

For a maximal monotone operator T, the resolvent J_{cT} is single valued and has full domain. This results in the fact that every $z \in \mathbb{R}^n$ can be uniquely expressed by $z = x + cy$.

In a similar fashion it can be derived that every $z \in \mathbb{R}^n$ can be expressed as $z = x - cy$, where $-y \in T(x)$ and $x = J_{cT}(z)$. These two notions are essential in the derivation of the Peaceman-Rachford splitting method in Section 3.3.1

3.1. Fixed point iteration

A fixed point iteration is a method for iteratively approximating the fixed points of monotone operators. The PDMM algorithm is an example of a fixed point method. Recall that the point x of an operator $T : \mathbb{R}^n \mapsto \mathbb{R}^n$ is a fixed point of T if $T(x) = x$. The fixed point iteration is therefore given by

$$x^{(k+1)} = T(x^{(k)}),$$

which is initialized at some starting point $x^{(0)} \in \mathbb{R}^n$. This iteration is applied repeatedly in order to find the fixed point of the operator T. There are two ways to guarantee that the fixed point iteration converges; the operator T being contractive or by averaging the operator, where only the latter will be discussed in depth here. Interested readers are referred to [24] for the convergence proof in the case of a contractive operator.

3.2. Inexact Krasnosel'skii-Mann iteration

A theory that links the fixed point iteration to the splitting methods later on in Section 3.3 is the Krasnosel'skii-Mann iteration. This iteration averages the fixed point iteration and provides some interesting insight into the convergence properties. The relevance of the Krasnosel'skii-Mann iteration will become clear when splitting methods are discussed. These splitting methods can be cast as specific cases of the Krasnosel'skii-Mann iteration. The source of this section, that can also be used for a more detailed explanation is [16].

Krasnosel'skii-Mann theorem. The fixed point iteration is guaranteed to converge for every starting point $x^{(0)} \in \mathbb{R}^n$ if the operator $T : \mathbb{R}^n \mapsto \mathbb{R}^n$ is averaged. In this case the fixed point iteration is also called the Krasnosel'skii-Mann iteration [15, 18, 24].

Krasnosel'skii-Mann iteration. The Krasnosel'skii-Mann iteration with $T : \mathbb{R}^n \mapsto \mathbb{R}^n$ a nonexpansive operator, averaging parameter $0 \leq \lambda^{(k)} \leq 1$ and $\mathbf{Fix}T \neq \emptyset$ is given by

$$z^{(k+1)} = (1 - \lambda^{(k)})z^k + \lambda^{(k)} T(z^{(k)}). \tag{3.6}$$

Intuitively this means that the newly computed z is a weighted combination of the previous value of z and the newly computed z by mapping the previous z by the operator T.

This thesis focuses on the case where the operator is applied inexactly. In order to take this into account, the standard Krasnosel'skii-Mann iteration can be slightly adjusted in order to incorporate an error as

$$z^{(k+1)} = (1 - \lambda^{(k)})z^k + \lambda^{(k)}(T(z^{(k)}) + \epsilon^{(k)}), \tag{3.7}$$

where $\epsilon^{(k)}$ the error caused by the inexact approximation of the mapping of the operator T. The error of an iteration is now defined as

$$e^{(k)} = (I - T)(z^{(k)}) = \frac{z^{(k)} - z^{(k+1)}}{\lambda^{(k)}} + \epsilon^{(k)}.$$

It is known that the iteration error $e^{(k)}$ of the inexact Krasnosel'skii-Mann iteration converges to zero, meaning that a fixed point will be reached, if the inexact Krasnosel'skii-Mann iteration (3.7) satisfies the following two conditions

- $(\lambda^{(k)}(1 - \lambda^{(k)}))_{k\in\mathbb{N}} \notin l^1_+$,
- $(\lambda^{(k)} \left\| \epsilon^{(k)} \right\|)_{k\in\mathbb{N}} \in l^1_+$,

with l_+^1 the set of summable sequences in $[0,\infty[$ [16, p. 4]. Meaning that if something is summed over all iterations and it results in either zero or a finite positive value it is within the set l_+^1. Intuitively the first requirement means that for all k, $0 < \lambda^{(k)} < 1$ must hold, so the averaging parameter cannot be either 0 or 1 for any k. In order to fulfill the second requirement, either $\lambda^{(k)}$ or $\epsilon^{(k)}$ must become zero at some point. This means that the error of approximating the operator must decrease and eventually become zero since the first requirement caused $\lambda^{(k)} \neq 0$.

3.3. Operator splitting methods

Solving optimization problems often entails finding the zero of a monotone operator. It is required to evaluate the resolvents of an operator in order to find the zero. Recall that the resolvent of an operator T is given by $J_{cT} = (I + cT)^{-1}$. This inversion could cause difficulties while evaluating the resolvent. Therefore we will look at two different splitting methods that can be applied to split one operator into two maximal monotone operators, such that $T = T_1 + T_2$, where T the original operator. The reason behind this is that it can be easier to separately evaluate the resolvents of T_1 and T_2 than evaluating the resolvent of T. The two discussed splitting methods are the Peaceman-Rachford splitting method and the Douglas-Rachford splitting method.

3.3.1. Peaceman-Rachford splitting method

One operator splitting method is the so called Peaceman-Rachford splitting method. As previously discussed, the aim is often to find a zero of a monotone operator. This leads, for the monotone operator T, to the following problem

$$0 \in T(x).$$

To decrease the difficulty of the problem the original operator is now split up into two operators. The original operator T can be split up as $T = T_1 + T_2$, with T_1 and T_2 maximal monotone operators as

$$\begin{aligned} 0 &\in (T_1 + T_2)(x), \\ 0 &\in T_1(x) + T_2(x). \end{aligned} \tag{3.8}$$

Now the operators T_1 and T_2 can be evaluated separately instead of evaluating the original operator $T(x)$. At this point it is important to note that the only way that x minimizes the original operator T is if the following holds

$$\begin{aligned} y &\in T_1(x), \\ -y &\in T_2(x), \end{aligned}$$

such that the summation of both equals zero. Now recall the representation lemma. This lemma stated that for the monotone operator T and with $y \in T(x)$, every $z \in \mathbb{R}^n$ can be represented in at most one way as $z = x + cy$ with $x = J_{cT}(z)$. With a comparable result for $-y \in \mathbb{R}^n$ resulting in the following two things:

1. $z = x + cy$, where $y \in T(x)$ and $x = J_{cT}(x + cy)$.
2. $z = x - cy$, where $-y \in T(x)$ and $x = J_{cT}(x - cy)$.

Recognize that the first one can be applied to $y \in T_1(x)$ and the second statement can be applied to $-y \in T_2(x)$. Analyzing each of these notions separately gives

$\mathbf{z = x + cy}, \quad \mathbf{x = J_{cT_1}(x + cy)}, \quad \mathbf{y \in T_1(x)}$	$\mathbf{z = x - cy}, \quad \mathbf{x = J_{cT_2}(x - cy)}, \quad \mathbf{-y \in T_2(x)}$
$x - cy = J_{cT_1}(x + cy) + x - (x + cy)$	$x + cy = J_{cT_2}(x - cy) + x - (x - cy)$
$= 2J_{cT_1}(x + cy) - (x + cy)$	$= 2J_{cT_2}(x - cy) - (x - cy)$
$= (2J_{cT_1} - I)(x + cy)$	$= (2J_{cT_2} - I)(x - cy)$
$= C_{cT_1}(x + cy)$	$= C_{cT_2}(x - cy)$

These results can be combined to rewrite the problem with two separate operators (3.8). At this point we start off with the result at the right side, which is $x + cy = C_{cT_2}(x - cy)$. Then the $x - cy$ within the brackets can be replaced by the result on the left side; $x - cy = C_{cT_1}(x + cy)$. Following these steps results in

$$\begin{aligned} x + cy &= C_{cT_2}(x - cy), \\ &= C_{cT_2} \circ C_{cT_1}(x + cy). \end{aligned}$$

Which, by using the representation lemma again, is equal to

$$z = C_{cT_2} \circ C_{cT_1}(z).$$

Summarizing the results thus far gives the following fixed point result

$$0 \in (T_1 + T_2)(x) \qquad \Leftrightarrow \qquad C_{cT_2} \circ C_{cT_1}(z) = z, \quad x = J_{cT_1}(z). \tag{3.9}$$

This splitting method is called Peaceman-Rachford splitting. The solution of this splitting method is found in an iterative manner. To do so we can formulate the following iteration

$$z^{(k+1)} = C_{cT_2} \circ C_{cT_1}(z^{(k)}). \tag{3.10}$$

After this iteration reaches convergence, the optimal value for x can be computed by $x^* = J_{cT_1}(z^*)$. Now recall the Cayley operator defined by $C_{cT} = 2J_{cT} - I$. This can be used to rewrite (3.10) as

$$\begin{aligned} z^{(k+1)} &= C_{cT_2} \circ C_{cT_1}(z^{(k)}), \\ &= (2J_{cT_1} - I) \circ C_{cT_2}(z^{(k)}), \\ &= (2J_{cT_1} - I) \circ (2J_{cT_1} - I)(z^{(k)}). \end{aligned}$$

With this notion, the Peaceman-Rachford iteration can now be expressed as

$$\begin{aligned} x^{(k+1)} &= J_{cT_2}(z^{(k)}) \\ y^{(k+1)} &= J_{cT_1}(2x^{(k+1)} - z^{(k)}) \\ z^{(k+1)} &= 2y^{(k+1)} - 2x^{(k+1)} + z(k). \end{aligned} \tag{3.11}$$

Where the y-iterate is only an intermediate iterate. The iterates of intereset are the x-iterate and the z-iterate. The x-iterate is the zero of the original problem $0 \in T(x)$ and the z-iterate is the dual variable which needs to converge for the x-iterate to converge.

Recacll that the Cayley operator of a monotone operator is nonexpansive. This causes the composition $C_{cT_2} \circ C_{cT_1}$ to also be nonexpansive, which in turn means that the Peaceman-Rachford iteration is not guaranteed to converge. It is only guaranteed to converge if either C_{cT_1} or C_{cT_2} is contractive, which will cause the iteration to converge geometrically.

As stated when the Krasnosel'skii-Mann iteration was introduced in Section 3.2, the discussed splitting methods are specific cases of the Krasnosel'skii-Mann iteration. This iteration (3.6) was given by $z^{(k+1)} = (1 - \lambda^{(k)})z^k + \lambda^{(k)} T(z^{(k)})$. In the case that we substitute $\lambda^k = 1 \quad \forall k$, this iteration becomes the Peaceman-Rachford iteration. This confirms the relevance of the Krasnosel'skii-Mann iteration since it has a clear link with the Peaceman-Rachford splitting method.

3.3.2. Douglas-Rachford splitting method

A second splitting method is the Douglas-Rachford splitting method. This method is motivated by the fact that the Peaceman-Rachford method is not guaranteed to converge. The Krasnosel'skii-Mann theorem (Section 3.2) ensures that applying averaging to the Peaceman-Rachford iteration will result in a method that

converges to a fixed point of $C_{cT_2} \circ C_{cT_1}$. Therefore, the Douglas-Rachford splitting method averages the operator and has the advantage that it always converges to the fixed point when $0 \in T_1(x) + T_2(x)$ has a solution. Averaging the Peaceman-Rachford iteration with averaging factor $1/2$ leads to the Douglas-Rachford splitting method given by

$$0 \in (T_1 + T_2)(x) \qquad \Leftrightarrow \left[\frac{1}{2}I + \frac{1}{2}(C_{cT_2} \circ C_{cT_1})\right](z) = z, \quad x = J_{cT_2}(z). \tag{3.12}$$

Notice that substituting $\lambda^{(k)} = 1/2 \quad \forall k$ into the Krasnosel'skii-Mann iteration (3.6) results in the same update equation for z. This shows that the Douglas-Rachford splitting method is also a specific case of the Krasnosel'skii-Mann iteration.

In a similar fashion as we previously did for the Peaceman-Rachford splitting method, the iteration for the Douglas-Rachford splitting method can be written as

$$\begin{aligned} x^{(k+1)} &= J_{cT_2}(z^{(k)}), \\ y^{(k+1)} &= J_{cT_1}(2x^{(k+1)} - z^{(k)}), \\ z^{(k+1)} &= z^{(k)} + y^{(k+1)} - x^{(k+1)}. \end{aligned} \tag{3.13}$$

The only iterate that has changed in comparison to the iteration of the Peaceman-Rachford splitting method (3.11) is the z-iterate. The advantage of the Douglas-Rachford splitting method is that it is guaranteed to converge to an optimal point if one exists.

3.4. Primal-dual method of multipliers

The algorithm used to validate the theoretical results later on during the simulations in Chapter 5 is the primal-duel method of multipliers (PDMM) algorithm. This section will discuss the iteration of the PDMM algorithm. The complete derivation of the PDMM algorithm can be found in Appendix A.

As touched upon in the introduction, the objective of the PDMM algorithm is to minimize the sum of convex functions over a graphical model, $G = (V, E)$, with V the set of nodes in the network and E the edges or connections in the network. Furthermore the number of nodes is denoted by $n = |V|$ and the number of edges is given by $m = |E|$.

Consider the following general convex optimization problem:

$$\begin{aligned} \min_x \quad & \sum_{i \in V} f_i(x_i), \\ s.t. \quad & A_{ij}x_i + A_{ji}x_j = b_{ij}, \qquad \forall (i,j) \in E, \end{aligned} \tag{3.14}$$

with variable $x \in \mathbb{R}^n$, $f_i : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ closed convex and proper (CCP) functions, f_i the objective function of node i, $A \in \mathbb{R}^{m \times n}$ defining the connections and $b \in \mathbb{R}^{m \times 1}$ defining the constraints between the nodes. The A matrix is not necessarily full rank. However, the linearly dependent constraints defined by A and b can be removed if the matrix is not full rank. The newly constructed full rank matrix combined with the new vector can be substituted for the old A matrix and b vector, meaning that it can be assumed that A is full rank. In vector form this general problem becomes

$$\begin{aligned} \min_x \quad & f(x), \\ s.t. \quad & Ax = b. \end{aligned} \tag{3.15}$$

This minimization problem can be solved with the aid of the PDMM algorithm. The PDMM algorithm iteratively applies the following iteration in order to converge to the optimal solution

$$\begin{aligned} x^{(k+1)} &= \arg\min_x \left[f(x) + z^{(k)^T}(Cx - d) + \frac{c}{2}\|Cx - d\|_2^2 \right], \\ \lambda^{(k+1)} &= z^{(k)} + c(Cx^{(k+1)} - d), \\ y^{(k+1)} &= 2\lambda^{(k+1)} - z^{(k)}, \\ z^{(k+1)} &= Py^{(k+1)}. \end{aligned} \tag{3.16}$$

These are the iterates of the standard PDMM algorithm. The P matrix is a permutation matrix, exchanging the upper half and the lower half of the y-iterate with each other. Applying the permutation matrix means that two nodes connected by an edge exchange their values, thus the permutation matrix represents the transmission of data between neighbouring nodes. Notice that A and b, which respectively defined the connections and constraints between the nodes, have disappeared. These have been restructured into the C matrix and d vector to have the proper dimensions in order to make it possible to solve the minimization problem (3.15) in a distributed manner. The C matrix and d vector are constructed as follows:

- $C(l,i) = A_{ij}, \quad$ for $i < j$,
  $C(l+m,i) = A_{ij}, \quad$ for $i > j$,
- $d = \frac{1}{2}\begin{bmatrix} b \\ b \end{bmatrix}$

where $C \in \mathbb{R}^{2m \times n}$ and $d \in \mathbb{R}^{2m}$. After rewriting A and b into C and d, these can be substituted into the x-iterate of the PDMM algorithm (3.16) along with the objective function. After substituting, the PDMM algorithm can iteratively be applied to find the optimal solution to the minimization problem (3.15).

3.5. Inexact primal-dual method of multipliers

The previously discussed PDMM algorithm assumed that the minimization problem to update the x-iterate is solved in an exact manner, without error. However often times this update needs to be approximated, which will lead to an error. The next thing to do is look at the implications when the update of the x-iterate is approximated. In order to do this, the iterates of the PDMM iteration are slightly adjusted to incorporate the inexactness. This way it becomes clear what the influence is of updating the x-iterate in an inexact manner.

The minimization problem to update the x-iterate can be solved by either using an algebraic expression if this exists or by using an iterative method to approximate the x value for which the argument is minimized. Using an algebraic expression will give the exact solution without an error, which means that this results in the standard PDMM algorithm. However, implementations of an iterative method will use a finite number of iterations. This causes the approximation of the x-iterate to be inexact. This is where the focus of this thesis specifically lies, the influence of the inexactness when the x-iterate is approximated.

At this point it is necessary to define some notation. The notation that will be used throughout this analysis is that ϵ is introduced as the error caused by the limited iterations. The inexact version of an iterate is represented by a tilde $\tilde{(\cdot)}$ and an exact update, which after the first iteration is based on an inexact iterate from the previous iteration is represented by a hat $\hat{(\cdot)}$.

The approach will be to look at each of the iterates of the PDMM algorithm one by one to see how an error that is introduced in the x-iterate propagates throughout the iteration. We start off by slightly adjusting the x-iterate to incorporate this error term as

$$\begin{aligned}\hat{x}^{(k+1)} &= \arg\min_x \left[f(x) + \hat{z}^{(k)^T}(Cx-d) + \frac{c}{2}\|Cx-d\|_2^2 \right] + \epsilon^{(k+1)}, \\ &= \tilde{x}^{(k+1)} + \epsilon^{(k+1)}.\end{aligned}$$

This means that the inexact update, $\hat{x}$, consists of the exact update given by $\tilde{x}$, with an error added that is given by $\epsilon^{(k+1)}$. The added error represents the inexactness caused by the approximation. The second iterate is the λ-iterate, which now becomes

$$\begin{aligned}\hat{\lambda}^{(k+1)} &= \hat{z}^{(k)} + c(C\hat{x}^{(k+1)} - d), \\ &= \hat{z}^{(k)} + c(C\tilde{x}^{(k+1)} - d) + cC\epsilon^{(k+1)}, \\ &= \tilde{\lambda}^{(k+1)} + cC\epsilon^{(k+1)}.\end{aligned}$$

The error introduced in the x-iterate, $\epsilon^{(k+1)}$, is multiplied by the optimization constant c and the matrix C. The optimization constant c is empirically optimized for each problem and influences the convergence rate of the PDMM algorithm. However, this shows that the optimization variable not only influences the convergence rate but also has an effect on the error since the error is multiplied by the optimization constant. The

matrix C represents the connections between the nodes and is filled with zeros and ones. This matrix will therefore not have an influence on the magnitude of the error.

The third iterate is the y-iterate. Substituting $\hat{\lambda}^{(k+1)}$ into this iterate gives

$$\begin{aligned}\hat{y}^{(k+1)} &= 2\hat{\lambda}^{(k+1)} - \hat{z}^{(k)},\\ &= 2(\hat{z}^{(k)} + c(C\tilde{x}^{(k+1)} - d) + cC\epsilon^{(k+1)}) - \hat{z}^{(k)},\\ &= \hat{z}^{(k)} + 2c(C\tilde{x}^{(k+1)} - d) + 2cC\epsilon^{(k+1)},\\ &= \tilde{y}^{(k+1)} + 2cC\epsilon^{(k+1)}.\end{aligned}$$

At this point the error is multiplied by $2cC$, this means that it is doubled when compared to the error in the λ-iterate. The final iterate to look at is the z-iterate. Substituting $\hat{y}^{(k+1)}$ gives

$$\begin{aligned}\hat{z}^{(k+1)} &= P\hat{y}^{k+1},\\ &= P(\hat{z}^{(k)} + 2c(C\tilde{x}^{(k+1)} - d) + 2cC\epsilon^{(k+1)}),\\ &= P\hat{z}^{(k)} + 2cP(C\tilde{x}^{(k+1)} - d) + 2cPC\epsilon^{(k+1)},\\ &= \tilde{z}^{(k+1)} + 2cPC\epsilon^{(k+1)}.\end{aligned}$$

The error is multiplied with the permutation matrix P when comparing to the previous iterate. The permutation matrix represents the transmission of data by permuting the upper half of the y-iterate with the lower half. This causes the permutation matrix to be filled with ones and zeros and will not affect the magnitude of the error. The influence of the permutation matrix is that an error caused by a node has an effect on the value of the z-iterate of its neighbour. To summarize, the full inexact PDMM iteration is given by

$$\begin{aligned}\hat{x}^{(k+1)} &= \tilde{x}^{(k+1)} + \epsilon^{(k+1)},\\ \hat{\lambda}^{(k+1)} &= \tilde{\lambda}^{(k+1)} + cC\epsilon^{(k+1)},\\ \hat{y}^{(k+1)} &= \tilde{y}^{(k+1)} + 2cC\epsilon^{(k+1)},\\ \hat{z}^{(k+1)} &= \tilde{z}^{(k+1)} + 2cPC\epsilon^{(k+1)}.\end{aligned} \tag{3.17}$$

Since the P and C matrices do not have an influence on the magnitude of the error, the only thing that can be taken into consideration when implementing the PDMM algorithm is the optimization constant c. As stated before, this will be a trade off between optimizing the convergence rate and not choosing c too large since the error will be multiplied by this optimization constant.

The mathematical proof of the convergence of inexact PDMM in Chapter 4 will use properties of monotone operators. This analysis later on will look at the convergence of the z-iterate, since this iterate directly influences the convergence of the x-iterate. Therefore the inexact z-iterate in operator notation is also required. The standard z-iterate (A.9) can be adjusted to include the inexactness term as

$$\hat{z}^{(k+1)} = C_{cT2} \circ C_{cT1}(\tilde{z}^{(k)} + 2cPC\epsilon^{(k)}) + 2cPC\epsilon^{(k+1)}, \tag{3.18}$$

$$= C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) + 2cPC\epsilon^{(k+1)}. \tag{3.19}$$

This is the notation that will be used throughout the convergence analysis in Chapter 4, since the properties of monotone operators can be applied to this.

3.6. Determining the optimal z

In order to show the convergence of the algorithm later on, the optimal values for the x and z iterates are needed beforehand. The way to determine the optimal value for x will be problem specific and will be discussed for each problem separately during the derivation of the problems in Appendix B. However, it is possible to formulate a general method to compute the optimal value for z, z^*, for all problems beforehand. This is used later on to compute the difference between the value of z at a certain iteration denoted as $z^{(k)}$ and z^*, to graphically demonstrate the convergence of z.

At this point it is necessary to discuss the behavior of the z-iterate. The first important thing to realize is that the C matrix is a tall matrix. This causes C as well as PC to have a rank of n. By inspection of the PDMM iteration (3.17), we see that these two matrices are the only matrices that have an influence on z, which has a length of $2m$. This means that only a part of z, determined by the column space of C, will be altered during the PDMM iterations since the matrices have a lower rank than the dimension of z. The part of z that will be altered will be called the controllable part of z from now on. This inherently means that there will be a part of z that remains unchanged and is called the uncontrollable part of z. However, z is multiplied by C in the minimization to update the x-iterate. This means that the uncontrollable part of z will not have an influence on x and therefore does not matter. The magnitude of the uncontrollable part is determined by the initialization. If z is initialized as the all zero vector, there is no uncontrollable part. However, if z is initialized with anything other than all zeros, the uncontrollable part will most likely not be zero.

Recall that the P matrix was a permutation matrix that interchanges the upper half of z with the lower half of z. This means that after two consecutive iterations, z is permuted twice so the uncontrollable part will be exactly the same again. This causes the algorithm to have two optimal values for z, that both have an identical controllable part, but a permuted uncontrollable part. Both these optimal values lead to the same x^*, since this uncontrollable part has no influence on x. We define the two optimal values as z_1^* and z_2^*. This realization can be used to derive a method to compute z^*. From the standard PDMM iterates (A.16), at convergence we have

$$\begin{aligned} z_2^* &= P(z_{*,1} + 2cCx_*), \\ z_1^* &= P(z_{*,2} + 2cCx_*). \end{aligned} \tag{3.20}$$

Recall that the x-iterate of the standard PDMM algorithm (A.16) is given by

$$x^{(k+1)} = \arg\min_x \left[f(x) + z^{(k)^T}(Cx - d) + \frac{c}{2}\|Cx - d\|_2^2 \right].$$

We know that at convergence the derivative of the x-iterate vanishes, since a fixed point is reached and an additional iteration will not change the value of x anymore. We start of by computing the derivative of the x-iterate of the standard PDMM algorithm, which results in

$$\frac{dx^{k+1}}{dx} = \frac{df(x)}{dx} + C^T z + cC^T Cx. \tag{3.21}$$

For both optimal points, the derivative of the x-iterate (3.21) should be equal to zero since an optimal point x^* has been reached. Writing this out for both optimal points results in

$$\frac{df(x)}{dx} = -C^T z_1^* - cC^T Cx^*, \tag{3.22}$$

$$\frac{df(x)}{dx} = -C^T z_2^* - cC^T Cx^*. \tag{3.23}$$

Recall that we started off by writing out an expression for z_2^*. Now this expression (3.20) can be substituted into (3.23) resulting in

$$\begin{aligned} \frac{df(x)}{dx} &= -C^T P(z_1^* + 2cCx^*) - cC^T Cx^*, \\ &= -C^T P z_1^* - 2cC^T PCx^* - cC^T Cx^*, \\ &= -C^T P z_1^* - cC^T PCx^* - cC^T(PCx^* + Cx^*), \\ &= -C^T P z_1^* - cC^T PCx^*. \end{aligned} \tag{3.24}$$

This expression and 3.22 can be rearranged in order to find an expression for the optimal value of z. First these expressions can be rearranged into

$$
\begin{aligned}
-C^T z_1^* - \overbrace{(\frac{\partial f(x^*)}{\partial x} + cC^T Cx^*)}^{g_1} = 0, \\
-C^T P z_1^* - \underbrace{(\frac{\partial f(x^*)}{\partial x} + cC^T PCx^*)}_{g_2} = 0.
\end{aligned}
$$

By defining g_1 and g_2 as indicated, these equations can more compactly be written in matrix form as

$$
-\begin{bmatrix} C^T \\ PC^T \end{bmatrix} z_1^* = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}.
$$

Finally this is rewritten to find an expression to compute the optimal value for z as

$$
z_1^* = -\begin{bmatrix} C^T \\ PC^T \end{bmatrix}^\dagger \begin{bmatrix} g_1 \\ g_2 \end{bmatrix},
$$

where † stands for the Moore-Penrose pseudoinverse [22]. The consequence of using the pseudoinverse of $[C^T PC^T]^T$ is that the result will only consist of the controllable part, hence $z_1^* = z_2^* = z^*$. This means that we will end up with the optimal value for z. This value for z^* can now be used can now be used to determine the error $\| z^{(k)} - z^* \|$ at each iteration in order to investigate the convergence.

3.7. Asynchronous updates and packet loss

There are two ways in which the PDMM algorithm can be used, synchronous and asynchronous updating of the nodes. Synchronous updating means that all the nodes in the network do their local computations and transmit at the exact same moment. Realistically it is not desirable to have all the nodes do their local computations based on a global clock since this goes against the idea of distributed processing. The asynchronous PDMM algorithm does not have the disadvantage of having all the nodes work on a global clock. Therefore, it is interesting to also implement the asynchronous version of PDMM during the simulations and see whether this gives similar results to the synchronous updates.

The implementation of the asynchronous PDMM algorithm goes as follows:

1. Randomly select a node i, with $i \in V$.
2. Compute the x-iterate and z-iterate for the selected node i.
3. Determine the neighbours of the selected node i.
4. Update the indices of the z vector that are affected by transmitting the data of the selected node i to its neighbors.

So instead of updating the complete z vector at a single iteration, only the part of the z vector that is affected by sending the data of node i to its neighbors is updated.

Another thing to take into consideration is packet loss. In a realistic scenario there is always a possibility that a transmission from a node to one or more of its neighbors fails. Therefore simulations will be presented that include packet loss. This is implemented in the asynchronous case by updating each neighbour of the selected node with a specified probability. To clearly demonstrate effect of packet loss, a packet loss probability of 0.5 is used throughout the simulations.

4

Convergence analysis

Recall from the introduction that the aim of this thesis is to investigate the influence of inexact local updates on the convergence of distributed optimization schemes. The algorithm that will have our particular interest is the PDMM algorithm since this algorithm is chosen to be used during the simulations later on. The inexact PDMM iteration in which the inexactness is incorporated was derived in Section 3.5 and can now be used to investigate the mathematical properties in the case of inexact updates. More specifically, it is important to investigate if the algorithm still converges and what the influence of an error is. Consider the general inexact update equation for the z-iterate (3.18) given by

$$\hat{z}^{(k+1)} = C_{cT_2} \circ C_{cT_1}(\tilde{z}^{(k)} + 2cPC\epsilon^{(k)}) + 2cPC\epsilon^{(k+1)}. \tag{4.1}$$

This is the exact update (based on an inexact previous iterate) given by $C_{cT2} \circ C_{cT1}(\tilde{z}^{(k)} + 2cPC\epsilon^{(k)})$ with the error caused by the approximation during the current iteration given by $2cPC\epsilon^{(k+1)}$ added. Notice that the exact update is based on an inexact variable from the previous iteration. This is why the error of the previous iteration, given by $2cPC\epsilon^{(k)}$, occurs within the argument of the Cayley operator. Whether an error is the inexactness at the current iteration or still belongs to the previous iteration can be determined by looking at the difference in indices in the update equation. The term $\tilde{z}^{(k)} + 2cPC\epsilon^{(k)}$ represents the inexact z-iterate of the previous iteration, with the error $\epsilon^{(k)}$ also caused during the previous iteration. The other term $\epsilon^{(k+1)}$ is the error caused by approximating the x-iterate during the current iteration.

Realize that without the inexactness, the update equation (4.1) is the z-iterate of any problem to which the Peaceman-Rachford splitting method is applied. The only thing that could be different when the Peaceman-Rachford method is applied to a different problem are the resulting operators T_1 and T_2. Also the $2cPC$ term, which is where the error is multiplied by, arises due to the monotone operators that resulted after applying Peaceman-Rachford splitting in order to derive the PDMM iteration. This means that this term should be replaced by the appropriate error term when the Peaceman-Rachford splitting method is applied to a different monotone operator. However, replacing this term would not change anything to the derivations in the remainder of this chapter, meaning that all the following proofs hold for Peaceman-Rachford splitting applied to any monotonic inclusion problem.

The proofs presented in this chapter assume that the operator $C_{cT_2} \circ C_{cT_1}$ is either nonexpansive or Lipschitz continuous with parameter $0 < L \leq 1$. However, the Cayley operator of a monotone operator is always a nonexpansive function [24, p. 22] and if the operator $C_{cT_2} \circ C_{cT_1}$ is nonexpansive, the averaged operator $(1/2I + 1/2C_{cT_2} \circ C_{cT_1})$ will be nonexpansive as well. This averaged operator is the update equation for the z-iterate of the Douglas-Rachford splitting method (3.12). This means that replacing the operator by its averaged operator will not change this property and that the presented proofs are also valid for problems to which Douglas-Rachford splitting is applied. Realize that Douglas-Rachford splitting will most likely result in a different Lipschitz parameter and error term, but this has no influence on the presented proofs. To conclude, this means that the analysis presented in the remainder of this chapter do not only hold for the specific case (PDMM) to which it is shown but holds for any problem to which either Douglas-Rachford or Peaceman-Rachford splitting is applied as long as the resulting operator $C_{cT_2} \circ C_{cT_1}$ satisfies the assumptions that are clearly stated at each theorem.

From here on the analysis will be done by making use of the inexact PDMM iteration since this algorithm is used during the simulations. This inexact update equation for the z-iterate 4.1 shows that an error introduced in iteration k will be included within the operator in iteration $k+1$. The inexact update of the z-iterate can also be written as (3.19)

$$\hat{z}^{(k+1)} = C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) + 2cPC\epsilon^{(k+1)},$$

where $\hat{z}^{(k)}$ the inexact update of the previous iteration. This is again the z-iterate of the standard Peaceman-Rachford iteration, where the $2cPC\epsilon^{(k+1)}$ term can be replaced in the case that splitting is applied to a different monotonic inclusion problem.

In the case that the iteration converges we have $C_{cT2} \circ C_{cT1}(\hat{z}^{(k+1)}) = \hat{z}^{(k+1)}$ at convergence, since a fixed point is reached and the operator maps this point to itself. Therefore we start off by looking at the difference between the exact update and the inexact input variable of this update, so $\left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k+1)}) - \hat{z}^{(k+1)} \right\|$. This norm should go to zero after sufficient iterations if there exists a fixed point of the operator $C_{cT2} \circ C_{cT1}(\hat{z}^{(k+1)})$.

The first theorem of this chapter makes use of the fact that the operator $C_{cT2} \circ C_{cT1}$ is nonexpansive 3.2.

To prevent any confusion, we will first quickly repeat the definition of a nonexpansive operator. Recall that an operator is Lipschitz continuous with constant L if the following holds (2.2)

$$\left\| T(y) - T(x) \right\| \leq L \left\| y - x \right\|.$$

A nonexpansive operator has Lipschitz constant $L = 1$, meaning that we then have $\left\| T(y) - T(x) \right\| \leq \left\| y - x \right\|$. By making use of this property we can now proof the first theorem, which is given by

Theorem 1. *For the nonexpansive operator $C_{cT2} \circ C_{cT1}$, the following holds:*

$$\left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k+1)}) - \hat{z}^{(k+1)} \right\| \leq \left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) - \hat{z}^{(k)} \right\| + 2 \left\| 2cPC\epsilon^{(k+1)} \right\|.$$

Proof. We have

$$\begin{aligned}
\left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k+1)}) - \hat{z}^{(k+1)} \right\| &= \left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k+1)}) - C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) - 2cPC\epsilon^{(k+1)} \right\|, \\
&\overset{(a)}{\leq} \left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k+1)}) - C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) \right\| + \left\| 2cPC\epsilon^{(k+1)} \right\|, \\
&\overset{(b)}{\leq} \left\| \hat{z}^{(k+1)} - \hat{z}^{(k)} \right\| + \left\| 2cPC\epsilon^{(k+1)} \right\|, \\
&= \left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) - \hat{z}^{(k)} + 2cPC\epsilon^{(k+1)} \right\| + \left\| 2cPC\epsilon^{(k+1)} \right\|, \\
&\overset{(c)}{\leq} \left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) - \hat{z}^{(k)} \right\| + 2 \left\| 2cPC\epsilon^{(k+1)} \right\|,
\end{aligned}$$

□

where (a) uses the triangle inequality, (b) uses nonexpansiveness and at (c) the triangle inequality is applied again. Theorem 1 analyses the behavior of the distance between the exact update, $C_{cT2} \circ C_{cT1}(\hat{z})$, and $\hat{z}$. If the error $\epsilon^{(k+1)}$ is a finitely summable sequence, we can conclude that this distance will be a nonexpansive series.

By looking at the first theorem, it becomes clear that $\|2cPC\epsilon\|$ is a determining factor in the rate at which the fixed point is reached. For a given node $2cPC$ is constant over iterations, meaning that the error ϵ caused by the inexactness will determine the magnitude of the factor $\|2cPC\epsilon\|$ at a certain iteration. Therefore it is interesting to look at the behavior of this error. For this, Lipschitz continuity will be used again. The error including $2cPC$ can be investigated with the aid of Lipschitz continuity, which leads to the second theorem.

Theorem 2. *Suppose a fixed point z^* of the operator $C_{cT2} \circ C_{cT1}$ exists and assume that the operator $C_{cT2} \circ C_{cT1}$ is Lipschitz continuous with parameter $0 < L \leq 1$. For geometric convergence with parameter p, we have*

$$\left\| 2cPC\epsilon^{(k+1)} \right\| \leq (p-L) \left\| \hat{z}^{(k)} - z^* \right\|,$$

where $p \geq L$ since $p - L$ has to be non-negative.

Proof.

$$\begin{aligned}
\left\| \hat{z}^{(k+1)} - z^* \right\| &= \left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) + 2cPC\epsilon^{(k+1)} - C_{cT2} \circ C_{cT1}(z^*) \right\|, \\
&= \left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) - C_{cT2} \circ C_{cT1}(z^*) + 2cPC\epsilon^{(k+1)} \right\|, \\
&\overset{(a)}{\leq} \left\| C_{cT2} \circ C_{cT1}(\hat{z}^{(k)}) - C_{cT2} \circ C_{cT1}(z^*) \right\| + \left\| 2cPC\epsilon^{(k+1)} \right\|, \\
&\overset{(b)}{\leq} L \left\| \hat{z}^{(k)} - z^* \right\| + \left\| 2cPC\epsilon^{(k+1)} \right\|, \qquad (4.2)
\end{aligned}$$

where (a) uses the triangle inequality and (b) uses Lipschitz continuity. For geometric convergence with parameter p, $\left\| \hat{z}^{(k+1)} - z^* \right\| \leq p \left\| \hat{z}^{(k)} - z^* \right\|$ must hold, which leads to

$$\begin{aligned}
L \left\| \hat{z}^{(k)} - z^* \right\| + \left\| 2cPC\epsilon^{(k+1)} \right\| &\leq p \left\| \hat{z}^{(k)} - z^* \right\|, \\
\left\| 2cPC\epsilon^{(k+1)} \right\| &\leq (p-L) \left\| \hat{z}^{(k)} - z^* \right\|,
\end{aligned}$$

where $p \geq L$ since $p - L$ has to be non-negative.

□

Notice the intermediate result given by $\left\| \hat{z}^{(k+1)} - z^* \right\| \leq L \left\| \hat{z}^{(k)} - z^* \right\| + \left\| 2cPC\epsilon^{(k+1)} \right\|$. For the interpretation of this intermediate result we assume that the error $\epsilon^{(k+1)}$ is a finitely summable sequence. This means that the error becomes zero at some point and does not have an influence from that point on. If this is the case, we will end up with $\left\| \hat{z}^{(k+1)} - z^* \right\| \leq L \left\| \hat{z}^{(k)} - z^* \right\|$ after sufficient iterations. This means that the distance between $\hat{z}$ and the optimal z will not increase over iterations. In the case that the operator $C_{cT2} \circ C_{cT1}$ is Lipschitz continuous with parameter $0 < L < 1$, the distance between $\hat{z}$ and the optimal z will decrease over iterations and an inexact distributed optimization scheme is guaranteed to reach a fixed point.

Theorem 2 shows that there are three things that determine the rate at which the error decreases: the Lipschitz parameter L, the geometric convergence parameter p and the norm $\left\| \hat{z}^{(k)} - z^* \right\|$. The Lipschitz parameter is problem dependent, since this is a bound on the mapping of the operator $C_{cT_2} \circ C_{cT_1}$. This means that the Lipschitz parameter will be constant over iterations. The parameter p is determined by the geometric convergence, thus that parameter is also constant over iterations. However, the error at a certain iteration between the current value of z and the fixed point z^* given by $\left\| \hat{z}^{(k)} - z^* \right\|$ does change over iterations. Therefore the final thing to investigate is the behaviour of this error. Examining this norm leads to the third and final theorem given by

Theorem 3. *Suppose a fixed point z^* of operator $C_{cT2} \circ C_{cT1}$ exists and assume that the operator $C_{cT2} \circ C_{cT1}$ is Lipschitz continuous with parameter $0 < L \leq 1$. For the convergence of z, the following holds*

$$\left\| \hat{z}^{(k+1)} - z^* \right\| \leq L^{k+1} \left\| z^{(0)} - z^{(*)} \right\| + \sum_{i=0}^{k} L^i \left\| 2cPC\epsilon^{(k+1-i)} \right\|.$$

Proof. Consider (4.2) given by

$$\left\| \hat{z}^{(k+1)} - z^* \right\| \leq L \left\| \hat{z}^{(k)} - z^* \right\| + \left\| 2cPC\epsilon^{(k+1)} \right\|.$$

Repeatedly applying this expression results in

$$\left\| \hat{z}^{(k+1)} - z^* \right\| \leq L^{k+1} \left\| z^{(0)} - z^{(*)} \right\| + \sum_{i=0}^{k} L^i \left\| 2cPC\epsilon^{(k+1-i)} \right\|.$$

□

Theorem 3 shows that an error at a certain iteration as well as the error made by the initialization of z becomes less influential as more iterations pass. This is caused by the fact that at each iteration previous errors are multiplied by the Lipschitz parameter L, which is strictly smaller than 1 if the operator is contractive. This means that the simulation presented later in chapter 5 should confirm that in the case that the error decreases, the algorithm will converge to the optimal point z^* or the two optimal points z_1^*, z_2^* in the case that z is initialized with non-zeros and the uncontrollable subspace is thereby non-empty.

5

Results

To validate the theoretical results derived in the previous chapters, simulations are presented in this chapter. The first thing to verify is the correctness of the derivation of the inexact PDMM iteration (3.17). The inexact PDMM iteration shows that the inexactness also depends on the optimization constant ρ. Therefore the second thing to look at is to determine whether this dependence also holds during the simulations.

After that, three problems are solved under various conditions. The problems are derived and casted into the PDMM iteration in appendix B. The first thing to differentiate between during the simulations is solving the problems in a synchronous and asynchronous manner, where packet loss is taken into consideration for the latter. Another aspect that is varied during the simulations is the difference in the way the local variable, the x-iterate, is computed by each node. In the case that an analytic expression is known, this is used first. The inexactness is simulated by adding a random Gaussian error to this exact update. The Gaussian error is zero-mean with a varying variance. This gives full control over the error, which makes it possible to investigate the convergence of the algorithm under accurately specified conditions. A second option to approximate the x-iterate is to either use a built-in MATLAB solver or implement a solver. As previously discussed, the Newton-Raphson method is implemented to use when this is chosen over a built-in solver.

Using a solver instead of the analytic expression is motivated by the fact that the standard PDMM algorithm can be used when an analytic expression exists since everything can be computed exact. However by using a solver, the iterate is approximated and inexactness is introduced, which leads to the previously derived inexact PDMM algorithm. The results when using a solver are then compared to the first results, where a Gaussian error was added to the exact update, to investigate if these results are similar.

Within the minimization problem to update the x-iterate of the PDMM algorithm is the optimization constant ρ. This constant is empirically optimized throughout all simulations.

5.1. Distributed average consensus

The first problem that is solved is the distributed averaging problem, of which the derivation can be found in Appendix B.1. This optimization problem assigns a value to each node and the aim is to compute the average of all values. This is achieved by solving the following minimization problem

$$\begin{aligned} \min_{x} \quad & \sum_{i \in V} \frac{1}{2}(x_i - x_i^{(0)})^2, \\ s.t. \quad & x_i - x_j = 0, \quad \forall (i,j) \in E \end{aligned}$$

where $x_i^{(0)}$ is the initial value of node i. A predefined topology is used to have consistent results and to be able to fairly compare the different results. A network is generated and is used for these simulations. The chosen topology has the following characteristics:

- Square 30×30 grid.
- 100 Nodes randomly placed.
- Node transmission range of 5.

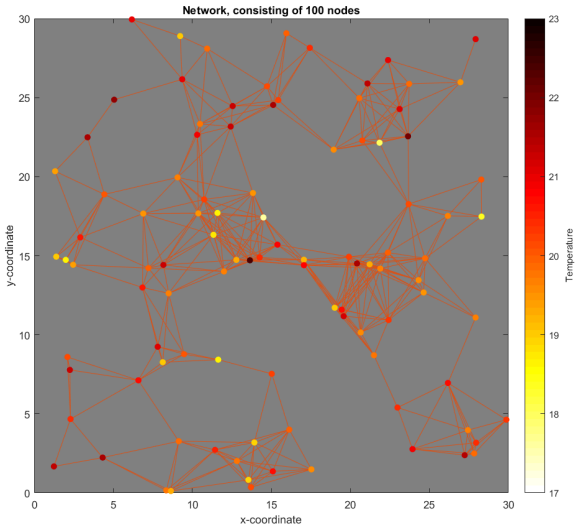

Figure 5.1: Topology used to solve the averaging problem

Throughout the results for the averaging problem it is assumed that we are dealing with a sensor network where each node measures the temperature and the goal is to compute the average of all of these temperatures in a distributed manner. For each node a temperature value is generated. These temperatures are normally distributed with mean $\mu = 20°C$ and variance $\sigma^2 = 1°C$. The topology including the temperature values can be found in Figure 5.1.

5.1.1. Comparison between domains

The first thing to start the simulations off with is the comparison between both domains. This is important since the convergence analysis in Chapter 4 has completely been conducted in the dual domain while we are interested in the optimal value in the primal domain. The comparison presented here is done by looking at the convergence in the primal domain, which is the x-iterate and the convergence in the dual domain, the z-iterate. This simulation is also ran in order to show and give a better understanding of the uncontrollable part of the z-iterate, which was discussed in Section 3.6. The PDMM algorithm permutes the uncontrollable part of the z-iterate. This means that after two iterations, the uncontrollable part is exactly the same again. Therefore it is chosen to plot separate curves of the even and odd iterations in the dual domain such that each curve has its own uncontrollable part. The resulting plots of the convergence in the primal and dual domain can be found in Figure 5.2.

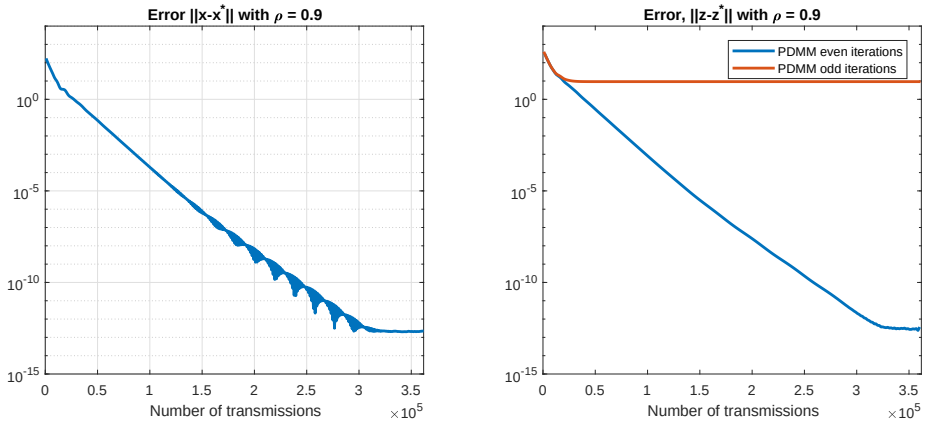

Figure 5.2: Comparison between primal and dual convergence of the PDMM algorithm. The difference between the even and odd iterations in the dual domain is caused by the uncontrollable part of the z-iterate.

The plot on the left hand side shows the convergence in the primal domain. It shows the error between the current value of the iterate and the optimal value on the y-axis versus the total number of transmissions that has currently been sent within the network on the x-axis. Thus a decreasing curve means that the x-iterate converges over the number of transmissions that has been sent. The curve flattening out at approximately $10^{-12.5}$ implies that convergence has been reached. This same reasoning holds for the plot of the convergence in the duel domain on the right hand side. The difference between both curves in the dual domain is solely caused by the uncontrollable part of the z-iterate, which does not change in magnitude but permutes each iteration. However, this simulation shows that the uncontrollable subspace does not have an influence on the convergence in the primal domain. This means that the uncontrollable subspace does not matter since the x-iterate is the variable of interest. In the remainder of this chapter only plots of the convergence in the primal domain will be presented since this is the variable of interest.

5.1.2. Inexact PDMM iteration

The first thing to verify is the inexact PDMM iteration. Recall the inexact PDMM iteration (3.17) given by

$$\begin{aligned}
\hat{x}^{(k+1)} &= \tilde{x}^{(k+1)} + \epsilon^{(k+1)} \\
\hat{\lambda}^{(k+1)} &= \tilde{\lambda}^{(k+1)} + \rho C \epsilon^{(k+1)} \\
\hat{y}^{(k+1)} &= \tilde{y}^{(k+1)} + 2\rho C \epsilon^{(k+1)} \\
\hat{z}^{(k+1)} &= \tilde{z}^{(k+1)} + 2\rho PC \epsilon^{(k+1)}.
\end{aligned} \tag{5.1}$$

These can be verified during the simulations since an algebraic expression for the x-iterate is known and a random zero-mean generated Gaussian error (ϵ) is added to the exact update ($\tilde{x}$) to get the inexact x-iterate($\hat{x}$). This means that both the exact update as well as the error are known. The exact and inexact variant of the x-iterate are both used to compute an exact and inexact variant of the z-iterate in parallel. Adding $2\rho PC\epsilon$ to the exact variant should give the same result as the inexact variant. The process to verify this correctness is shown in Figure 5.3 and the resulting plot of the difference $\|\hat{z}_1 - \hat{z}_2\|$ can be found in Figure 5.4.

Adding $2\rho PC\epsilon$ to the exact z-iterate gives a similar result as the inexact z-iterate, which was computed by using the inexact x-iterate. The slight difference that can be observed between $\hat{z}_1$ and $\hat{z}_2$ is caused by the finite precision of MATLAB . This means that the correctness of the inexact PDMM iteration is confirmed.

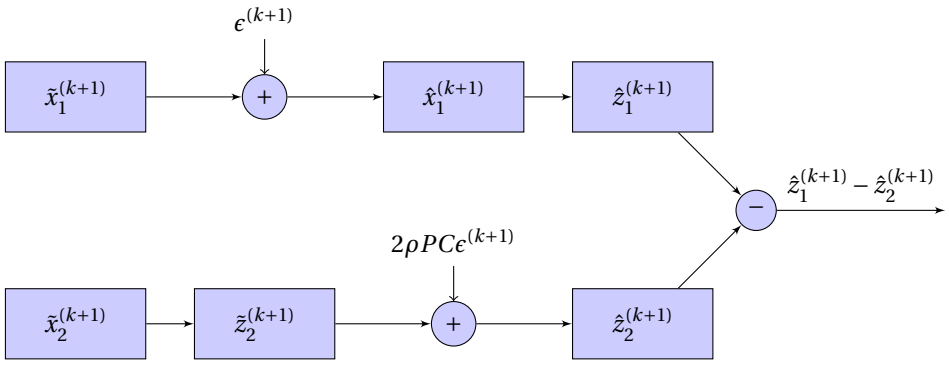

Figure 5.3: Block diagram to show the method to verify the correctness of the inexact PDMM algorithm. The inexact update of the z-iterate is computed in two ways, after which the difference is computed.

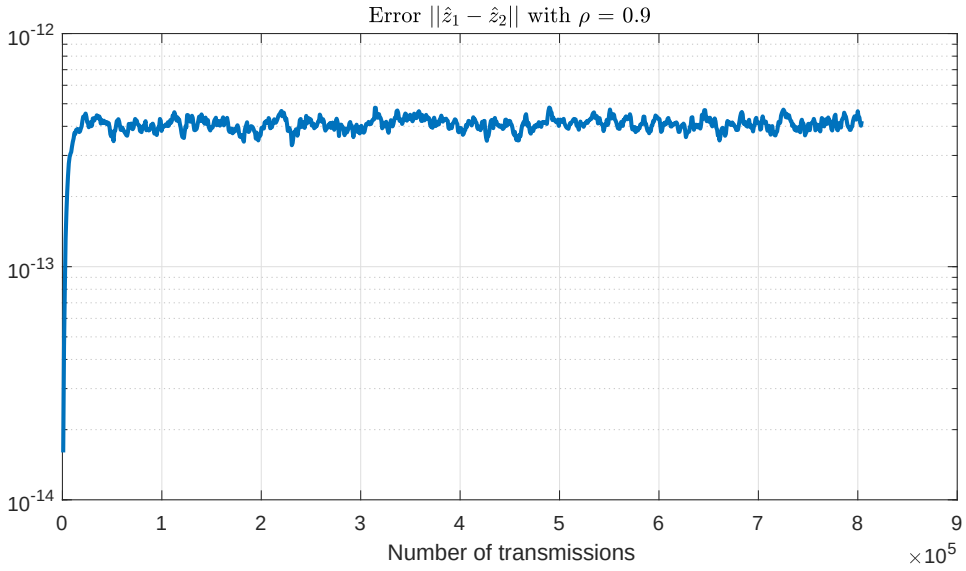

Figure 5.4: Error in the dual domain to show the correctness of the inexact PDMM iteration.

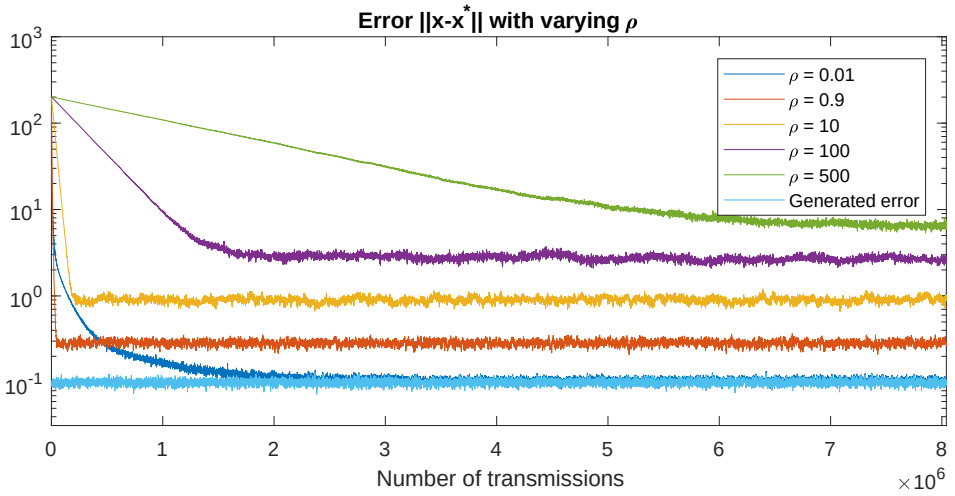

Figure 5.5: Primal convergence of the synchronous PDMM algorithm for different values of the optimization constant ρ. This shows the influence of the optimization constant on the error of the inexact PDMM algorithm.

Another thing that the inexact PDMM iteration (5.1) shows is that the inexactness also depends on the optimization constant ρ. The algorithm should be more accurate for a lower optimization optimization constant and thus converge further, which is verified in Figure 5.5.

5.1.3. Synchronous

After verifying the inexact PDMM iteration, we can now look at the influence of the inexactness on the convergence of the algorithm. The first step is to solve the averaging problem in a synchronous manner. This means that all nodes are updated simultaneously at a single iteration of the PDMM algorithm.

There exists an algebraic expression for the update equation of the x-iterate (B.3) for the averaging problem. This expression can be used to compute the update of the x-iterate in an exact manner. The inexactness is then simulated by adding a random Gaussian error to the exact value. Since the error is generated, the magnitude of the error can easily be altered. This is done by keeping the mean of the Gaussian error zero while adjusting the variance. Two different types of the random Gaussian error are considered, a step error where the variance of the error suddenly significantly changes and an error where the variance of the error decreases geometrically. The resulting plot can be found in Figure 5.6.

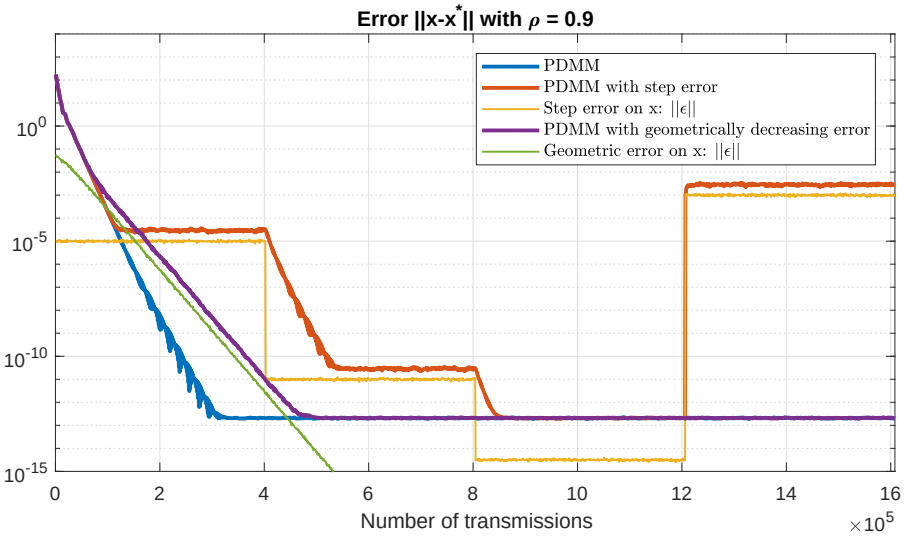

Figure 5.6: Primal convergence of the synchronous PDMM algorithm with a random Gaussian error added to the x-iterate. The $2\rho PC$ distance between the PDMM algorithm and the error is visible as well as the adjusted convergence rate in the case of a geometrically decreasing error.

The algorithm converges up until a certain distance from the error level, where the distance is determined by the term $2\rho PC\epsilon$. This means that if a certain convergence accuracy is desired, this can be used to determine how accurate the solver to compute the x-iterate should be since the approximation of the solver results in the inexactness ϵ. Another thing to note is that when the error suddenly decreases, which was shown by adding the step error to the exact update, the PDMM algorithm starts to converge further at the rate of the standard PDMM algorithm. In the case that the error on the x-iterate is increased, the error of the PDMM algorithm instantly increases up until a distance of $2\rho PC\epsilon$ from the new error. In the case of a geometrically decreasing error, the PDMM algorithm converges with a rate equal to the rate at which the error decreases as long as the rate of error of the error is lower than the rate of the standard PDMM algorithm.

The previous results used the algebraic expression of the x-iterate. In practice the algebraic expression is often not known and the x-iterate is computed by using an iterative solver. Therefore the next step is to use a solver to approximate the x-iterate investigate the behavior of the PDMM algorithm. As previously discussed, it is expected that the error of the solver determines how far the algorithm converges. The plots where the x-iterate is computed with the aid of a solver can be found in Figure 5.7.

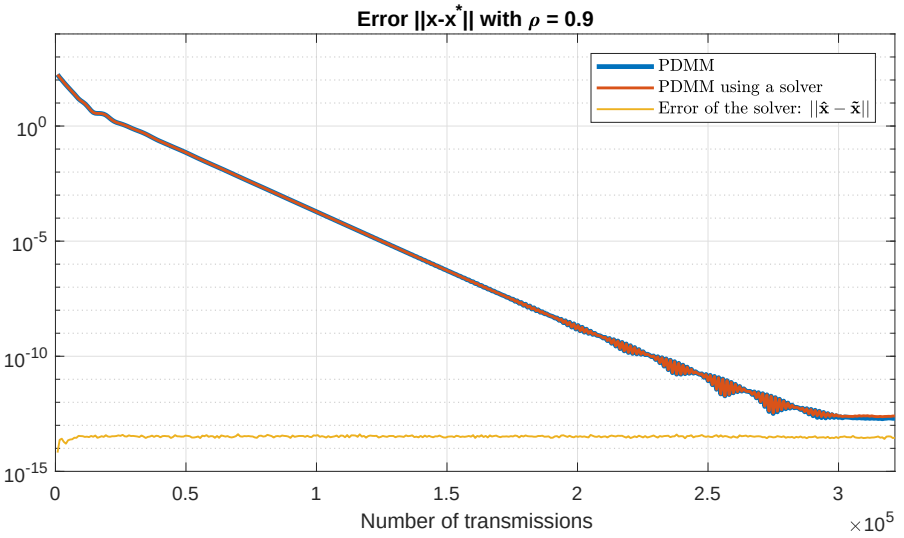

Figure 5.7: Primal convergence of the synchronous PDMM algorithm when a solver is used to approximate the x-iterate.

What stands out when inspecting this figure is the consistently low error of the solver. This is caused by the fact that the build-in MATLAB solver 'fminunc' is used, which uses the quasi-newton method. This method finds the minimum by locally approximating the objective function as a quadratic function. For the averaging problem this will give the exact solution in one step since the objective function is quadratic. The only remaining error is caused by the finite precision of MATLAB . This causes the inexact PDMM algorithm that uses the solver to reduce to the standard PDMM algorithm without inexactness.

5.1.4. Asynchronous

Up until this point the simulations have been in a synchronous manner. Synchronous means that all the nodes in the network do their local computations and transmit at the exact same moment. Realistically it is not desirable to have all the nodes do their local computations based on a global clock. Asynchronous PDMM does not have this disadvantage. Therefore it is interesting to investigate if the previously simulated convergence of the inexact PDMM algorithm also holds in the asynchronous case, where packet loss is also taken into consideration.

We start off by again by adding a random Gaussian error to the exact update. This simulations is done to investigate the behavior of the asynchronous PDMM algorithm with a Gaussian error and packet loss. The error to simulate the inexactness is again generated from a zero-mean Gaussian distribution with varying variance. In the synchronous case, at a single iteration of the PDMM algorithm, 100 random error values are generated. Plotting the norm of these 100 values at each iteration gave a reasonably constant line. However, in the asynchronous case only a single error value is generated since only one node is updated. This means that it is only possible to take the absolute value of the error which would result in a heavily fluctuating plot, even when the variance is kept constant. Therefore plotting the absolute value of the error makes it impossible to see the main results of the simulation. To overcome this, the mean and maximum value of the error is plotted over the parts of the simulation where the variance of the noise is constant. Finally, the simulation can be compared to the synchronous case to determine if the results are similar. The resulting plot of this simulation can be found in Figure 5.8.

The results are similar to the results in the synchronous case. The PDMM algorithm took a slightly smaller number of transmission to converge. However, this changes each time an asynchronous simulation is run. During the asynchronous simulations, every iteration a single node is randomly chosen which does its local computations and broadcasts to its neighbors. Some nodes in the network can have a larger impact on the convergence, for example selecting the same node coincidentally over and over again will lead to a decreased convergence rate. This means that randomly selecting these nodes more often will lead to a decreased convergence rate.

It becomes clear that the PDMM algorithm with packet loss converges slower, but still converges to the

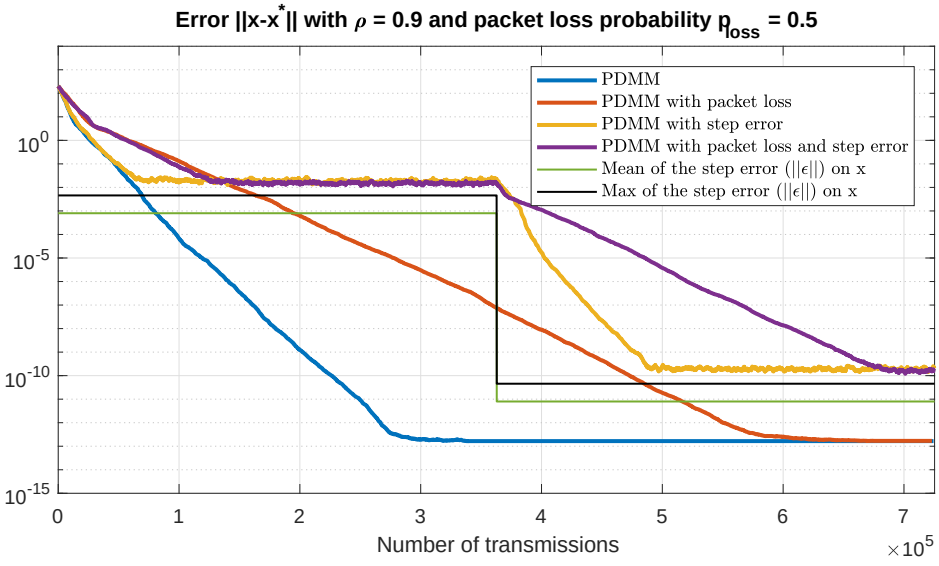

Figure 5.8: Primal convergence of the asynchronous PDMM algorithm with the inexactness simulated by adding a random Gaussian error to the exact update. It converges up until a certain distance from the error and packet influences the convergence rate.

same point as the standard PDMM algorithm. This makes sense since on average only half of the transmitted packages are received. The PDMM algorithm also starts converging further when the variance of the noise is suddenly lowered. So the only difference between the standard PDMM algorithm and the PDMM algorithm with packet loss is the rate of convergence. Another thing to note is that the PDMM algorithm again converges up until the same distance from, in this case, the maximum of the error. Note that the lines of the PDMM algorithms always lie above the maximum error of the solver.

Recall the synchronous simulations, where the x-iterate was updated with the aid of a solver. It was shown during these simulations that the solver will accurately approximate this update since the objective function is quadratic. Showing asynchronous simulations that make use of a solver would therefore not add anything useful and hence these simulations are left out.

5.2. P-norm optimization

The simulations up until now were all solving the averaging problem, where each node got a value assigned and the goal was to compute the average of all of these values. Now the same simulations are done, but now with the p-norm problem derived in Appendix B.2. This solution to this problem is found by minimizing the following minimization problem

$$\begin{aligned} \min_{x} \quad & \|x-a\|_p^p, \\ s.t. \quad & x_i - x_j = 0, \qquad \forall (i,j) \in E, \end{aligned} \tag{5.2}$$

where $x, a \in \mathbb{R}^n$ with n the number of nodes and p an integer restricted by $2 < p < \infty$. The vector a consists of (different) constant values, one for each node. For this problem a new graph is generated over which the p-norm problem is solved. The network can be found in Figure 5.9 and has the following characteristics

- Square 4×4 grid.
- 14 Nodes randomly placed.
- Node transmission range of 1.5.

5.2.1. Synchronous

First the p-norm optimization problem is solved by updating the nodes in a synchronous manner. For the first simulation a value of three is chosen for p, which defines the order of the problem. The iterative method

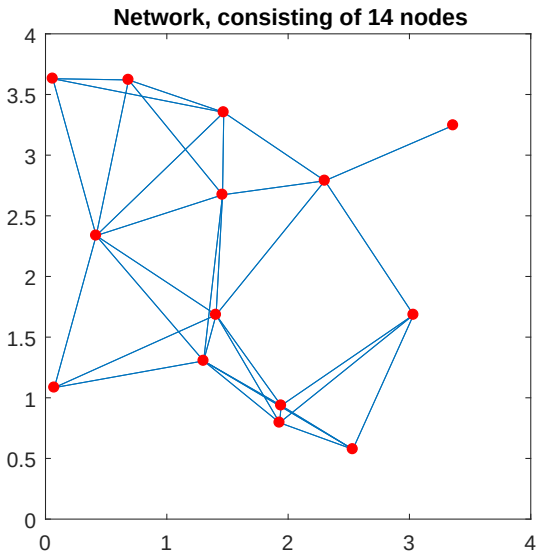

Figure 5.9: Topology used to solve the p-norm problem.

to locally solve the minimization is the Newton-Raphson method. The number of Newton steps that a node is allowed to take is varied with the aim to introduce inexactness. The resulting simulation can be found in Figure 5.10.

The simulation shows that only the curve where each node is limited to a single iteration is slightly different in comparison to the rest. This is caused by the fact that the objective function is still fairly close to being quadratic by keeping the order of the problem low. One way to overcome this is by increasing the order of the problem from three to five. The result of this change can be found in Figure 5.11.

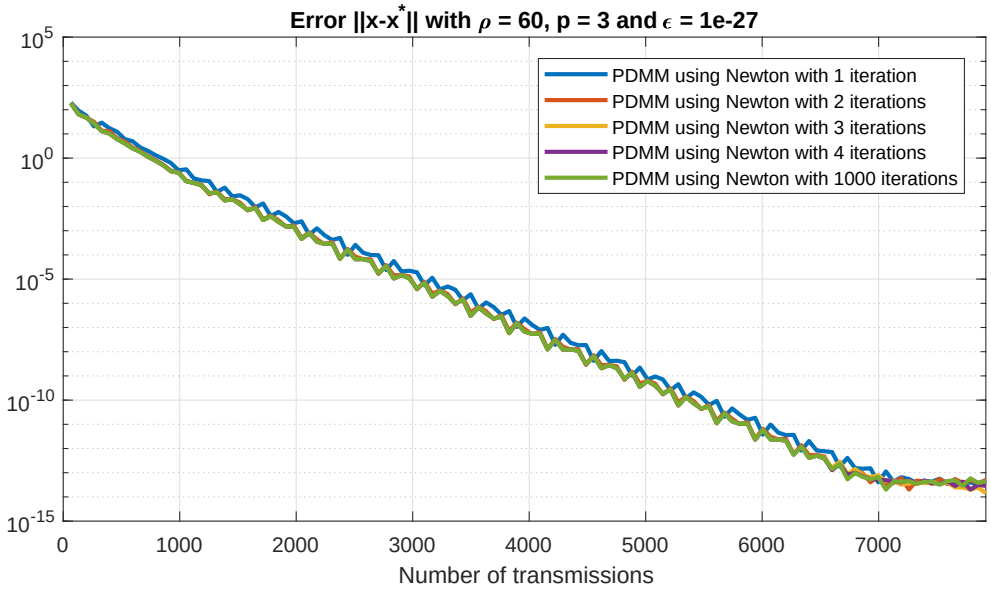

Figure 5.10: Primal convergence of synchronous PDMM of an objective function of order three and the number of Newton-Raphson iterations varied. The only curve that slightly differs from the others is when the Newton-Raphson is limited to a single iteration.

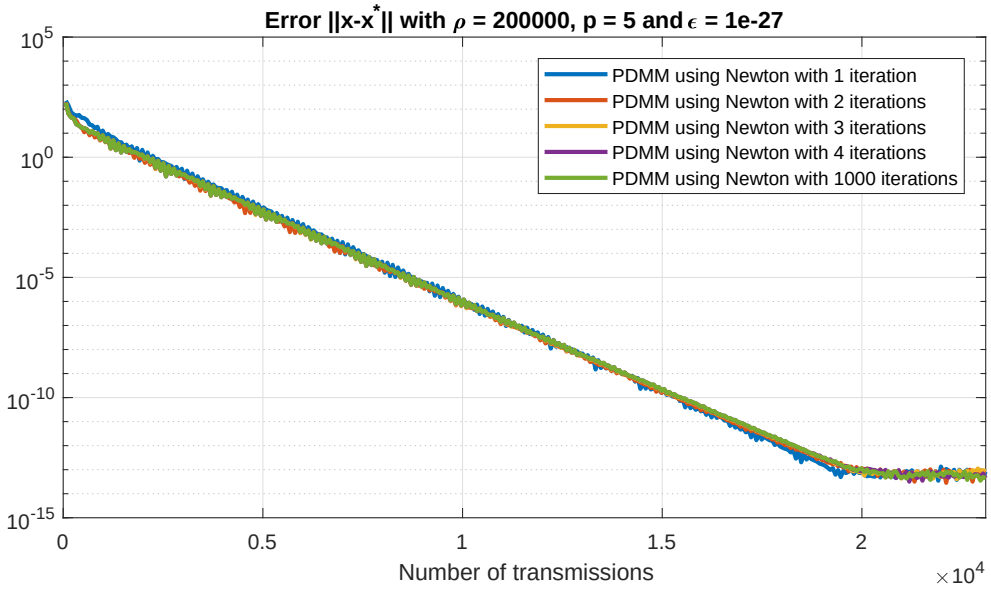

Figure 5.11: Primal convergence of synchronous PDMM of an objective function of order five and with the number of Newton-Raphson iterations varied. Notice the high value of the optimization constant.

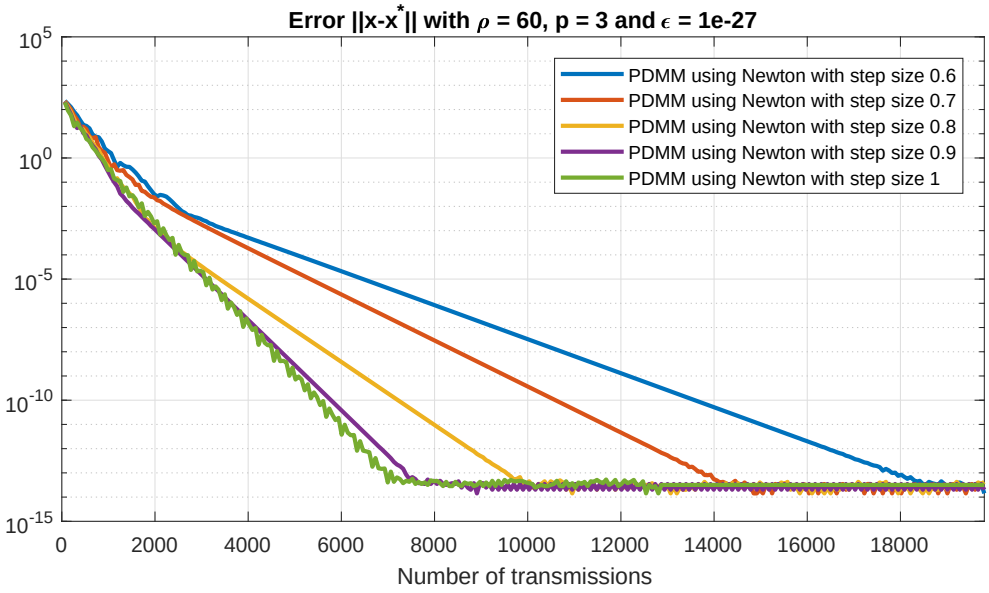

Figure 5.12: Primal convergence of synchronous PDMM of an objective function of order three and where the step size of the Newton-Raphson method is varied while keeping the number of Newton-Raphson iterations limited to one. The artificially introduced inexactness influences the convergence rate.

The two things that stand out from this simulation are the severe increase of the optimization constant and the increased number of transmissions needed in order to reach convergence. Even after increasing the order of the minimization problem, the simulation where the Newton-Raphson method is limited to a single iteration still gives the same result as the simulations where the Newton-Raphson method is allowed to use upto a 1000 iterations. This means that the convergence rate of the PDMM algorithm is lower than the rate at which the inexactness decreases.

The final option is to artificially introduce inexactness. One way to do so is by keeping the number of Newton-Raphson iterations limited to a single iteration and by varying the step size. This means that instead of taking a full step in the right direction, only a specified fractions of this step is taken. The resulting simulation can be found in Figure 5.12.

Artificially introducing inexactness results in significantly different curves. The only thing affected by this inexactness is the convergence rate of the inexact PDMM algorithm. All simulations converge, even though it does take a different number of transmissions before convergence is reached.

5.2.2. Asynchronous

After investigating the convergence in synchronous operation we now look at solving the p-norm minimization problem while updating in a asynchronous manner. So a single node is selected and, local computations are conducted and the data is transmitted to its neighbor. Besides this packet loss is taken into account again with a probability of 0.5, meaning that on average only half of the sent packages is received. The first thing to look at is the convergence with a low order of $p = 3$. The resulting plot can be found in Figure 5.13.

This simulation yields similar results in comparison to synchronous operation. The number of allowed iterations for the Newton-Raphson method has no influence on the convergence rate. The reason that the three curves where packet loss is kept constant slightly differ is that every iteration of the PDMM algorithm, a node is selected at random. This means that it is possible to select either a node that has a high influence on the convergence rate or a node that has a low influence. Another thing to note is that in this case taking packet loss into consideration will only have an influence on the convergence rate. However, since the number of allowed iterations for the Newton-Raphson method has no influence on the inexactness this will also be the case when the order is raised since this also leads to a decrease in the convergence rate of the PDMM algorithm. Therefore one way to simulate inexactness is by varying the size of the Newton-Raphson step again. This will show the effect of inexactness even though it is artificially generated. The result when varying the step size can be found in Figure 5.14.

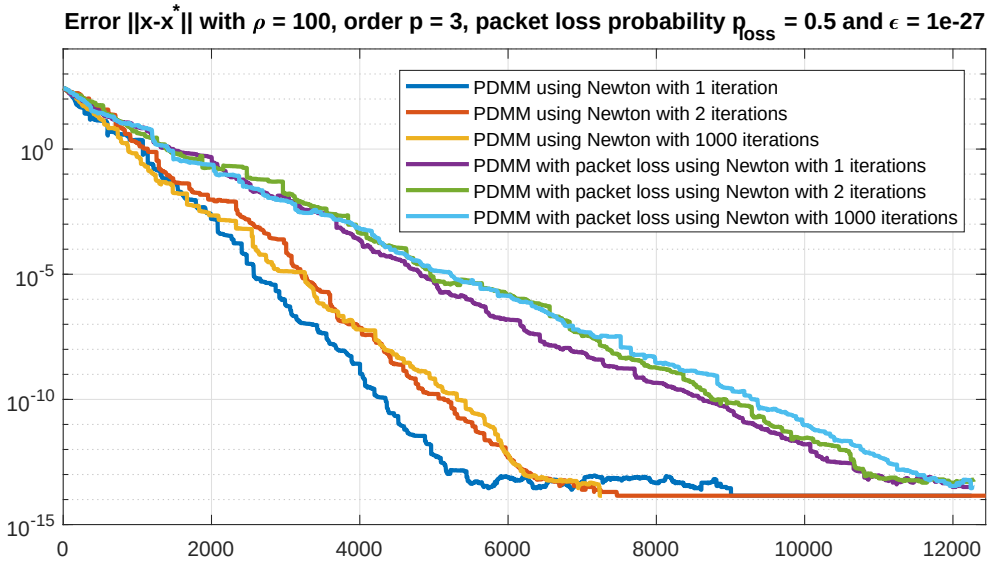

Figure 5.13: Primal convergence of asynchronous PDMM of an objective function of order three and the number of Newton-Raphson iterations varied. All curves show a similar convergence rate

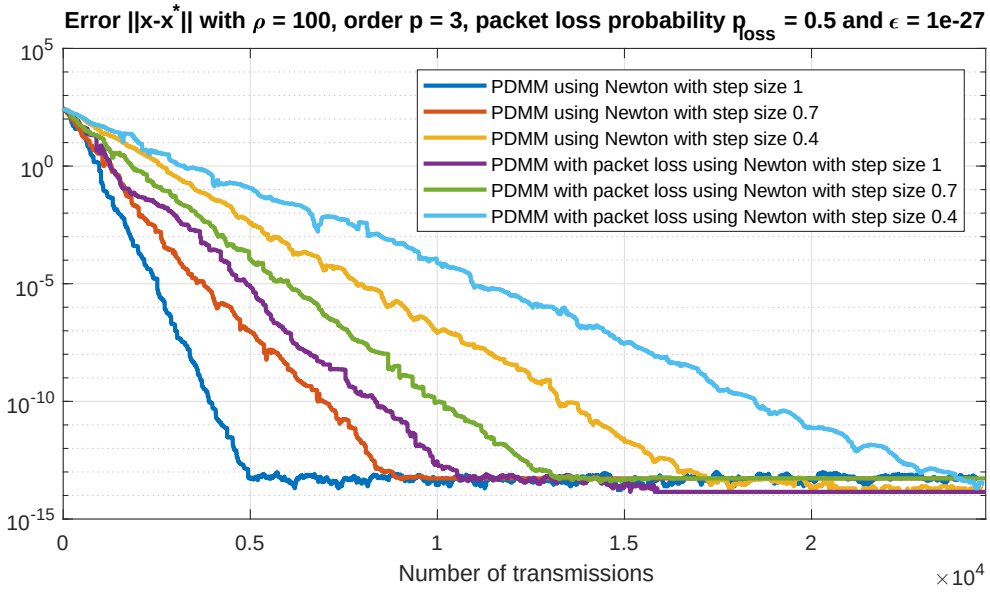

Figure 5.14: Primal convergence of asynchronous PDMM of an objective function of order three and where the step size of the Newton-Raphson method is varied while keeping the number of Newton-Raphson iterations limited to one. The artificially introduced inexactness influences the convergence rate.

The simulations show that both packet loss as well as an artificially generated error only have an effect on the convergence rate of the inexact PDMM iteration. What does stand out is that the curve where the x-iterate is approximated by using the Newton-Raphson method with a single iteration and a step size of 0.4 has a lower convergence rate than the curve where a step size of 0.7 is used and packet loss with a probability of 0.5 is taken into account. Furthermore we can conclude that inexactness have the same effect on both problems thus far. The inexact PDMM algorithm will continue to converge as long as the error decreases.

5.3. Channel capacity problem

The final problem we consider is the channel capacity problem. The objective of this problem is to distribute the available power among channels in a way that maximizes the capacity of all channels combined. The following minimization problem is solved in order to find the solution to the channel capacity problem

$$\begin{aligned} &\min_x -\sum_{i\in V} \log(\sigma_i^2 + x_i), \\ &s.t. \quad x \succeq 0, \qquad \mathbf{1}^T x = 1, \end{aligned}$$

where x_i the power allotted to channel i and σ_i^2 the noise variance of channel i. A new random network is generated for these simulations. For this problem, each of the nodes in the network represents a channel that is characterized by the noise variance of its channel. The chosen noise variances including the optimal power distribution can be found in Figure 5.15 while the generated network for this problem can be found in Figure 5.16 and has the following characteristics:

- Square 12×12 grid.
- 15 Nodes randomly placed.
- Node transmission range of 5.

In this case the PDMM algorithm finds values for the μ and λ iterates, which are used to compute the x-iterate. This means that the error caused by the solver is introduced on the μ and λ iterates. The analysis in Appendix B.3.1 shows the impact that errors on the μ and λ iterates have on the x-iterate. A simplified expression was derived for the case that $|\mu_i + \lambda_i| \gg |\epsilon_{\mu_i} + \epsilon_{\lambda_i}|$, which is given by

$$\left\| \epsilon_{x_i}^{(k)} \right\| = \left\| \frac{\epsilon_{\mu_i}^{(k)} + \epsilon_{\lambda_i}^{(k)}}{(\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)})^2} \right\|. \tag{5.3}$$

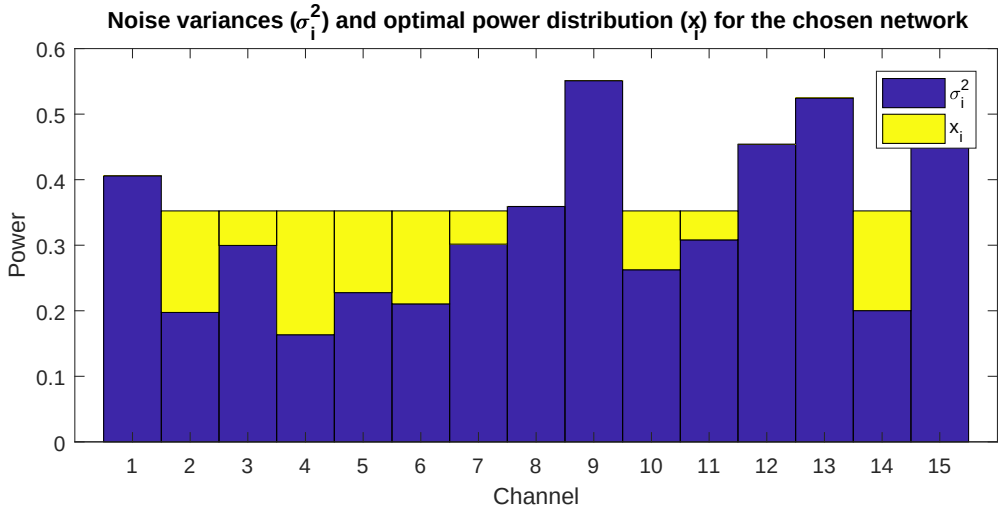

Figure 5.15: Chosen noise variances and optimal power distribution for the simulations of the channel capacity problem

Table 5.1: The optimal values for each channel

Channel	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
μ_i^*	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84
λ_i^*	0.375	0	0	0	0	0	0	0.053	1.023	0	0	0.635	0.932	0	0.929
$\mu_i^* + \lambda_i^*$	-2.46	-2.84	-2.84	-2.84	-2.84	-2.84	-2.84	-2.78	-1.81	-2.84	-2.84	-2.20	-1.91	-2.84	-1.91

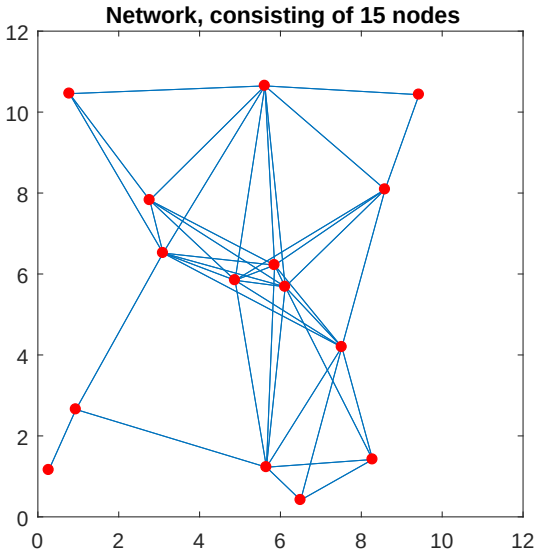

Figure 5.16: Topology used to solve the channel capacity problem.

It is important to first determine the values for μ_i and λ_i to see if the simplification is justified. To demonstrate this, the optimal values for the channels are used and are summarized in Table 5.1.

From the table it becomes clear that $\min(|\mu_i^* + \lambda_i^*|) = 1.81$, and therefore the simplification is justified for iteration numbers close to convergence. Recall that the impact that an error on μ and λ has on x is in the logarithmic domain given by given by

$$\log\left\|\epsilon_{x_i}^{(k)}\right\| = \log\left\|\epsilon_{\mu_i}^{(k)} + \epsilon_{\lambda_i}^{(k)}\right\| - \log\left\|(\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)})^2\right\|.$$

Thus an error introduced on the μ and λ iterates results in a lower error on the x iterate in the logarithmic domain. Therefore the convergence of μ and λ should show the same distance between the error and the point to which it converges, namely $2\rho PC\epsilon$. However, the x-iterate should converge further, which is determined by the term $-\log\left\|(\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)})^2\right\|$.

5.3.1. Synchronous

For the channel capacity problem exist analytic expressions to compute the μ and λ iterates. These analytic expressions can be used to compute the updates for these iterates in an exact manner after which the error can be simulated by adding a random Gaussian error. We therefore start off by adding a random Gaussian error to the exact update to see what the influence of the error is. The first thing to confirm is whether the x-iterate indeed converges further than the μ and λ iterates. The result of this simulation is displayed in Figure 5.17.

From the figure it becomes clear that the x-iterate converges further than the generated error, whereas the μ and λ iterates converge up until a distance $2\rho PC\epsilon$ of the error. The curve of the error on the μ-iterate is barely visible because this curve is nearly the same as the curve for the error on the λ-iterate. Also notice that the distance between the error of the μ and λ iterates and the randomly generated error is decreased. This distance was determined by $2\rho PC\epsilon$ and for this problem and the defined topology the optimal optimization constant is significantly lower than for the distributed averaging problem. Therefore this decreased constant also causes a decrease in distance between the error of the PDMM iterates and the generated error.

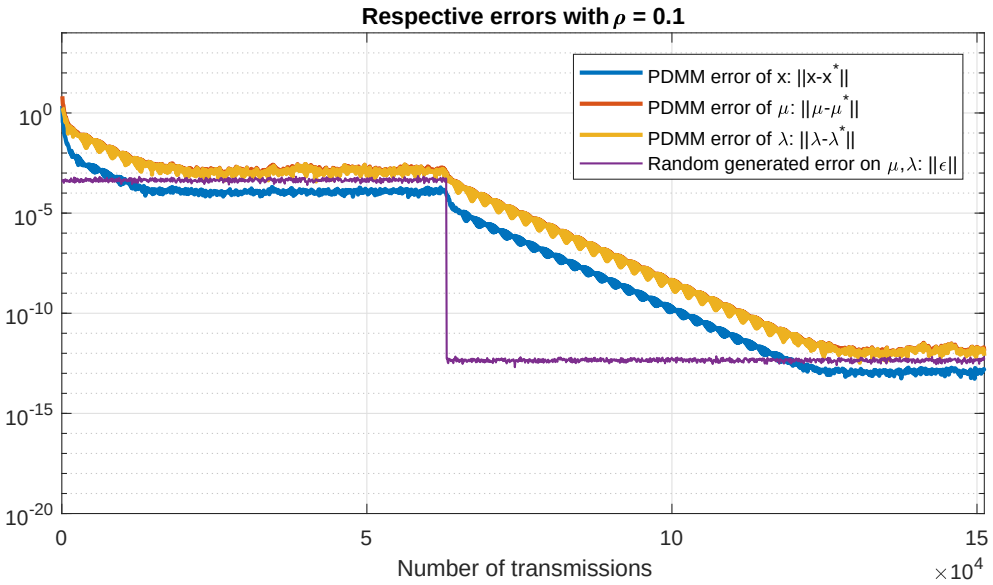

Figure 5.17: Convergence of the different iterates of the synchronous PDMM algorithm with a Gaussian error added to the exact update of the μ and λ iterates

In the remainder of this section, plots that only show the error of the x-iterate are presented since this is the variable of interest.

The next thing to simulate is the behavior of the algorithm when the μ and λ iterates are approximated by using a solver. The MATLAB solver fmincon is used for the channel capacity problem since the constraints must be taken into account. Using a solver results in the simulation shown in Figure 5.18

Even though the updates are done in a synchronous matter, the error of the solver deviates quite significantly over the transmissions. This is caused by the fact that the network for the channel capacity problem contains fewer nodes than the network for the previous problems. The network of the previous problems contained 100 nodes whereas the newly generated network for the channel capacity problem contains 15 notes. Computing the norm of 15 values at a single iteration results in a heavier fluctuating curve than the norm of 100 values. However, this simulation does show the expected result in comparison to the simulation

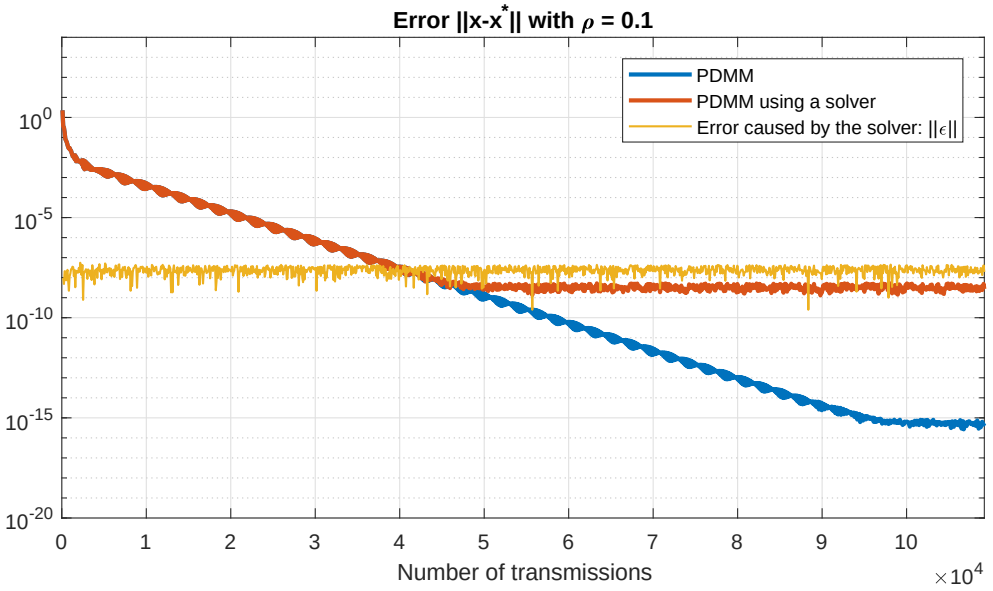

Figure 5.18: Primal convergence of the synchronous PDMM algorithm that uses a solver to approximate the μ and λ iterates

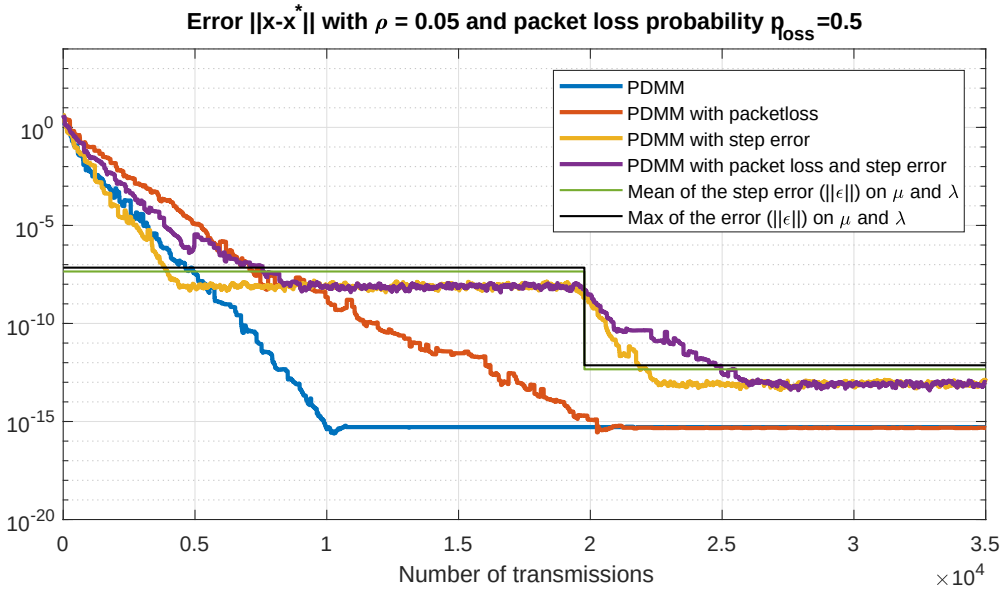

Figure 5.19: Primal convergence of the asynchronous PDMM algorithm with a Gaussian error added to the exact updates for the μ and λ iterates.

where the x-iterate converges further than the error.

5.3.2. Asynchronous

The simulations of the channel capacity problem thus far have all been in a synchronous manner. As previously explained, doing the local computations at the nodes and updating the nodes asynchronously is more realistic. The implementation of the asynchronous version of the channel capacity problem, with packet loss taken into consideration, is done in a similar manner as the previous problems.

First a Gaussian error is added to the exact update of the μ and λ iterates. From (5.3) it is clear that the x-iterate is equivalently affected by an error on either the μ or λ iterate. Therefore the error can be characterized by computing the norm of both these errors. However, the same issue occurs as in the case of the previous problems. Plotting the absolute value at each iteration would give a heavily fluctuating curve. Therefore the mean and maximum value of the error is plotted over the iterations where the noise variance is constant. The resulting plot can be found in Figure 5.19.

The standard inexact PDMM algorithm and the inexact PDMM algorithm with packet loss both converge in the same way as their exact variants up until the distance determined by (5.3). The PDMM algorithm where packet loss is taken into consideration shows a slower convergence rate. Also, the algorithms converge further when the variance of the generated error suddenly lowers.

Finally asynchronous inexact PDMM and approximating the μ and λ iterates with the aid of a solver are combined. The mean and maximum value of the error is again used in order to have a plot where it is still possible to distinguish the different curves. The resulting plot is presented in Figure 5.20.

The behavior is under these conditions still unchanged. The inexact PDMM algorithm with packet loss taken into consideration will inherently converge at a slower rate than the standard PDMM algorithm since on average only half of the sent packages are received. Also introducing an error causes the algorithms to converge up until a certain point from the error.

If a certain accuracy is desired, the inexact update equations of the standard PDMM algorithm (3.17) can be used to determine the errors on the μ and λ iterates. Then these errors in combination with the equation of the analysis of the inexact channel capacity problem (5.3) can be combined to find the link between the needed accuracy of the solver and the desired accuracy of the convergence of the x-iterate.

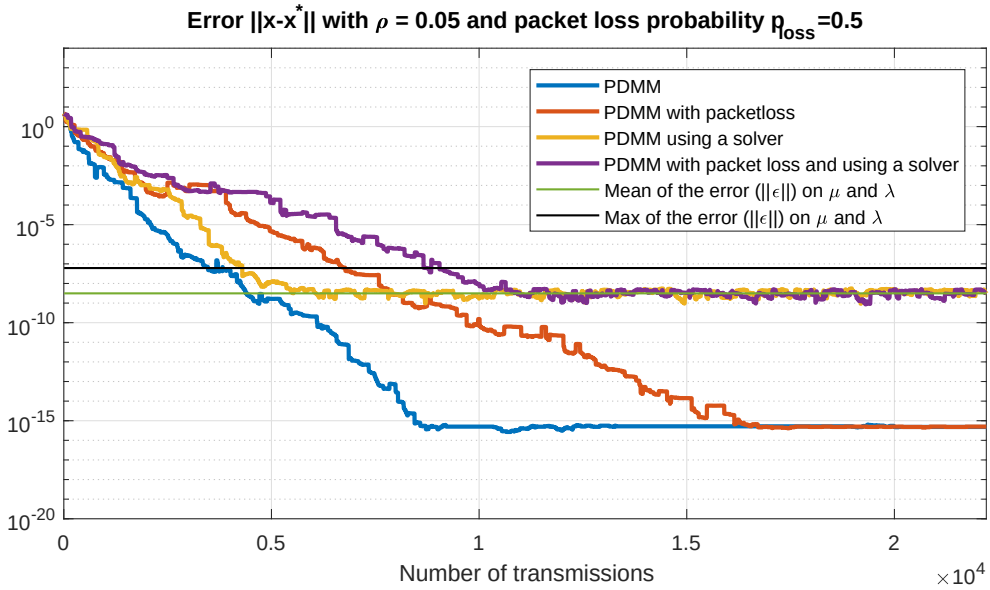

Figure 5.20: Primal convergence of the asynchronous PDMM algorithm that uses a solver to approximate the μ and λ iterates.

6

Conclusion and Future Work

The final chapter of this thesis focuses on conclusions that can be drawn from the research throughout this work and future work that could be done in order to investigate the topic even further.

6.1. Conclusion

To draw a conclusion we look at the research question and in particular we will see how the research question can be answered. The research question formulated in the introduction was:

What is the influence of inexact local updates on the convergence of distributed optimization schemes? Will the effects be different in asynchronous operations in comparison to synchronous operation and does packet loss have an influence on the convergence?

First two chapters were dedicated to a mathematical background and monotone operator theory in order to give a better understanding of the topic. The monotone operator theory also showed the link between the Krasnosel'skii-Mann iteration and operator splitting methods. These chapters were followed by the convergence analysis. This analysis provided mathematical proof for the convergence of the inexact PDMM algorithm. The chapter after that provided simulation results to validate the mathematical proofs of the convergence analysis.

The conclusion will be split up into two parts, each part addressing one of the parts of the research question. The first part of the research question is given by:

What is the influence of inexact local updates on the convergence of distributed optimization schemes?

Answering this part of the research question will be done in three parts. First we look at the derivation of the inexact PDMM iteration, after that the question will be addressed from the perspective of the convergence analysis after which it is answered from the point of view of the simulation results.

The chapter dedicated to monotone operator theory contained a section that introduced the inexact PDMM algorithm. This showed how an error introduced on the primal variable propagates to the dual variable. This derivation also showed that the error of the inexact PDMM iteration depends on the optimization variable. The next thing to discuss is the convergence analysis. The first important thing to realize is the fact that the convergence analysis was an analysis that holds for many distributed optimization schemes. Even though the PDMM algorithm was used for the simulation, the analysis was not limited to this algorithm. The foundation for this lies in the chapter about monotone operator theory and more specifically the sections about the Krasnosel'skii-Mann iteration and the splitting methods. The first thing shown by the convergence analysis was that the inexact PDMM algorithm still converges to a fixed point even if the primal variable is updated in an inexact manner as long as the error is finitely summable. Another thing shown by the convergence analysis is that the current error as well as all errors from previous iterations are multiplied by the Lipschitz parameter. This meant that an error has a decreasing influence as iterations pass.

With the convergence analysis treated it is time to move on to the simulations and discuss if the results confirmed convergence analysis. The first thing to verify was the correctness of the inexact PDMM iteration.

This was done by computing the inexact z-iterate in two ways and determining the difference between the both. This gave the expected result meaning that the inexact PDMM algorithm is correct. The second thing the inexact PDMM algorithm showed was the dependence on the optimization constant, which was verified by keeping the error constant and varying the optimization constant. After treating the inexact PDMM iteration, simulations were presented to investigate the mathematical proofs from the convergence analysis. The main conclusion about this is that the error only influences how far the algorithm converges, thus the magnitude of the resulting error when the inexact PDMM algorithm has reached its fixed point. If the error is suddenly lowered, the inexact PDMM algorithm starts converging further again at the convergence rate of the standard PDMM algorithm. When the error suddenly increases, the error of the inexact PDMM algorithm also instantly increases. If a geometrically decreasing error decreases at a rate lower than that of the standard PDMM algorithm, the inexact PDMM algorithm converges at the rate that the error decreases. The final thing that has to be kept in mind is that the inexact PDMM iteration only holds for variable that is directly optimized by the PDMM algorithm. The variable of interest can be different, as has been shown for the channel capacity optimization problem. However, the derivation of the channel capacity problem in Appendix B.3.1 shows how the inexact PDMM influences the variable of interest for this problem. The same approach can be used when one desires to solve other problems where the distributed optimization scheme does not directly solve for the variable of interest.

After treating the first part of the research question, we will now take a look at the second part of the research question. This part was formulated as:

Will the effects be different in asynchronous operations in comparison to synchronous operation and does packet loss have an influence on the convergence?

The conclusion of this part can be done in a concise manner since the results of synchronous and asynchronous updating are very similar. The behavior previously discussed for the first part of the research question is similar when the iterates are updated in an asynchronous matter. There is one thing that stands out during the asynchronous simulations. Recall that the inexact PDMM iteration shows what the distance between the error and the convergence of an iterate is. When asynchronously updating it turned out that it then becomes the distance between the maximum of the error and the convergence of an iterate instead of the mean of the error. The introduction of packet loss only results in a decreased convergence rate. The relevance of asynchronous updating could be underestimated since the results are similar to synchronous updating. However, it is important to realize that in a practical setup it is not desirable to update in a synchronous matter. The reason for this is that in synchronous operation all nodes still have to operate on a global clock. This makes it relevant to confirm that the convergence analysis also hold for asynchronous operation.

6.2. Future work

It is always important to determine the scope of a thesis by estimating what is feasible to achieve within the available time. This also means that there will always be things that are left out due to time constraints. This section will discuss things that could be interesting to further investigate.

To start off the PDMM algorithm was chosen as the distributed optimization scheme to use for the simulation. This is the reason that the inexact PDMM algorithm was derived. This was specifically focused on the PDMM algorithm. If one desires to use a different distributed optimization scheme, this needs to be derived again. Therefore it could be interesting to derive the inexact iteration for other popular algorithms.

Optimization problems were needed in order to conduct the simulations. In this thesis, three different problems were solved during the simulations. The first two problems gave either a low or no error when a solver was used. The third problem optimized over two variables where constraints needed to be taken into account. Therefore a built-in MATLAB solver was chosen which had a constant error. It could be interesting to run simulations of a distributed optimization scheme to solve different problems.

Another thing that was kept constant during the simulations was the distributed optimization scheme. All simulations made use of the PDMM algorithm. To validate that the results are not algorithm dependent it could be interesting to run simulations that use a different distributed optimization scheme for which the convergence analysis also holds.

Another property that could be useful is self-concordance [6, p. 496]. This property is used to determine a practical upper bound on the maximum number of Newton-Raphson iterations. This could be helpful since one requirement is that the error has to be a finitely summable sequence. One approach could be to start off with a low number of allowed iterations and increase this when the inexact PDMM algorithm stops with converging. Making use of self-concordance then gives the maximum number of Newton-Raphons iterations to which it has to increase.

The final thing to address is that one of the advantages of distributed processing is the ease to alter the topology. During the simulations the topology has been kept fixed. Therefore a change in topology could be implemented to investigate whether the behavior of the inexact PDMM iteration changes. However, since asynchronous updating and packet loss have been implemented a change in topology will most likely not have an effect on the inexact PDMM algorithm.

A

Primal-dual method of multipliers

The algorithm used to validate the theoretical results during the simulations in Chapter 5 is the primal-duel method of multipliers (PDMM) algorithm. This appendix will treat the derivation of the PDMM algorithm.

As touched upon in the introduction, the objective of the PDMM algorithm is to minimize the sum of convex functions over a graphical model, $G = (V, E)$, with V the set of nodes in the network and E the edges or connections in the network. Furthermore the number of nodes is denoted by $n = |V|$ and the number of edges is given by $m = |E|$.

A.1. General PDMM

The first thing to do is to derive the standard PDMM iteration. The following steps are taken in order to achieve this. We start off by formulating a general convex optimization problem and finding the dual function of this problem. However, this dual problem still has all the nodes dependent on each other. Since the aim is to solve problems in a distributed manner, the dual function is reformulated to remove the global dependencies. Peaceman-Rachford splitting is applied to the reformulated dual problem in order to find the solution to the optimality condition of the dual problem. The iterates of the Peaceman-Rachford splitting method are then rewritten to end up with the PDMM iteration.

Consider the following general convex optimization problem:

$$\begin{aligned} \min_{x} \quad & \sum_{i \in V} f_i(x_i), \\ s.t. \quad & A_{ij}x_i + A_{ji}x_j = b_{ij}, \qquad \forall (i,j) \in E, \end{aligned} \tag{A.1}$$

with variable $x \in \mathbb{R}^n$, $f_i : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ closed convex and proper (CCP) functions, f_i the objective function of node i, $A \in \mathbb{R}^{m \times n}$ defining the connections and $b \in \mathbb{R}^{m \times 1}$ defining the constraints between the nodes. The A matrix is not necessarily full rank. However, the linearly dependent constraints defined by A and b can be removed if the matrix is not full rank. The newly constructed full rank matrix combined with the new vector can be substituted for the old A matrix and b vector, meaning that it can be assumed that A is full rank. In vector form this general problem becomes

$$\begin{aligned} \min_{x} \quad & f(x), \\ s.t. \quad & Ax = b. \end{aligned} \tag{A.2}$$

In order to find the dual problem, we first derive the Lagrangian. The Lagrangian of this general problem is given by

$$\mathcal{L}(x, \delta) = f(x) + \delta^T (Ax - b), \tag{A.3}$$

where $\delta \in \mathbb{R}^m$ denotes the Lagrange multiplier. The dual function is defined as the infimum of the Lagrangian over x. Therefore the dual function that belongs to the general problem (A.2) is given by

$$g(\delta) = \inf_x \left[f(x) + \delta^T (Ax - b) \right].$$

Recall from the background in Chapter 2 that the conjugate of a function is defined as (2.3)

$$f^*(y) = \sup_{x \in \text{dom} f} \left[y^T x - f(x) \right].$$

This definition is used to replace the infimum of the dual function with the conjugate function. Doing this results in the following dual function

$$\begin{aligned} g(\delta) &= \inf_x \left[f(x) + \delta^T (Ax - b) \right], \\ &= -\sup_x \left[-A^T \delta x - f(x) + b^T \delta \right], \\ &= -f^*(-A^T \delta) - b^T \delta, \end{aligned}$$

where the final step uses the definition of the conjugate function. This dual function is now used to formulate the dual problem. The dual problem is defined as the maximum of the dual function over δ. This results in the following dual problem

$$\max_\delta \quad -f^*(-A^T \delta) - b^T \delta.$$

Which can equivalently be written as

$$\min_\delta f^*(-A^T \delta) + b^T \delta. \tag{A.4}$$

At the optimal point, the gradient of the dual problem should vanish. This is called the optimality condition. For the derived dual problem (A.4) the optimality condition is given by

$$0 \in -A \partial f^*(-A^T \delta) + b$$

Alternatively the optimality condition can be written in element-wise notation as

$$b_{ij} \in -\partial_{\delta_{ij}} f_i^*(-a_i^T \delta) - \partial_{\delta_{ij}} f_j^*(-a_j^T \delta), \qquad \forall (i,j) \in E.$$

This means that both f_i^* and f_j^* both depend on the same edge variable δ_{ij}. Therefore all nodes are still dependent on each other and at this point it is not possible to find the solution to the dual problem (A.4) in a distributed manner. Hence, auxiliary variables are introduced to remove these dependencies. Two node variables, $\lambda_{i|j}$ and $\lambda_{j|i}$, are created for each edge $(i,j) \in E$ with the constraint $\lambda_{i|j} = \lambda_{j|i} = \delta_{ij}$. All these auxiliary variables, $\lambda_{i|j}$, are stacked in the λ vector. For an arbitrary edge number l that connects nodes i and j, with $(i,j) \in E$ and $i < j$, we have $\lambda(l) = \lambda_{i|j}$. Similarly when $i > j$ with $(i,j) \in E$, this becomes $\lambda(l+m) = \lambda_{i|j}$, with m still the number of edges. This means that the top half of λ should be identical to the bottom half of λ.

The dimensions of the A matrix and the newly introduced λ vector do not match. In order to overcome this problem we introduce the matrix C that matches the dimension of the λ vector. To achieve this, the top $m \times n$ part of the C matrix is filled with the positive values of the A matrix and the lower $m \times n$ part of the C matrix is filled with the negative values of the A matrix.

Finally we define a vector $d \in \mathbb{R}^{2n}$ in order to convert the b vector to the appropriate dimension. This d vector is defined as $d^T = 0.5[b^T, b^T]$.

To summarize, the λ vector, C matrix and d vector are constructed as follows:

- $\left.\begin{array}{ll}\lambda(l)=\lambda_{i|j}, & \text{for } i<j,\\ \lambda(l+m)=\lambda_{i|j}, & \text{for } i>j,\end{array}\right\}$ At convergence: $\lambda(l)=\lambda(l+m)=\delta_{ij}$
- $\begin{array}{ll}C(l,i)=A_{ij}, & \text{for } i<j,\\ C(l+m,i)=A_{ij}, & \text{for } i>j,\end{array}$
- $d=\frac{1}{2}\begin{bmatrix}b\\ b\end{bmatrix}$,

where $\lambda\in\mathbb{R}^{2m}$, $C\in\mathbb{R}^{2m\times n}$ and $d\in\mathbb{R}^{2m}$. The introduction of the λ vector, C matrix, and d vector was motivated by the fact that it was not possible to solve the derived dual problem (A.4) in a distributed manner. The final thing to introduce such that it is possible to rewrite the dual problem into a distributed problem is a permutation matrix P. This matrix exchanges the upper m rows with the lower m rows of either a vector or a matrix. This means that the top and bottom half of a vector or matrix are exchanged. The physical meaning behind the permutation matrix is the transmission of data along an edge. Using this permutation matrix the following equivalence holds

$$Cx+PCx=2d \quad \Leftrightarrow \quad Ax=b. \tag{A.5}$$

This means that solving with the constraint $Cx=d$ is equivalent to solving with the constraint $Ax=b$, since we have $C+PC=(A^TA^T)^T$. Thus replacing the constraint of the general problem (A.2) with $Cx=d$ will keep the problem exactly the same. After replacing the constraint, the dual problem becomes

$$\min_{\lambda} \quad f^*(-C^T\lambda)+d^T\lambda+I_\Lambda(\lambda), \tag{A.6}$$

where I_Λ the indicator function. Recall that the indicator function is defined as

$$I_\Lambda\begin{cases}0, & \lambda\in\Lambda,\\ \infty, & \lambda\notin\Lambda.\end{cases}$$

with Λ the set defined by $\Lambda=\{\lambda\in\mathbb{R}^{2m}|\lambda=P\lambda\}$ and P the discussed permutation matrix. The reason that the indicator function is added in the dual problem is to ensure that the solution of the minimization lies in the set Λ. This set causes the upper half of the vector λ to be equal to the lower half, which is the way λ was defined.

In order to find the optimal solution for the rewritten dual problem (A.6), the optimality condition can be derived again. At the optimal point, the gradient should vanish. This results in the following optimality condition

$$0\in -C\partial f^*(-C^T\lambda)+d+N_\Lambda(\lambda), \tag{A.7}$$

where $N_\Lambda(\lambda)$ the normal cone operator. This is the gradient of the indicator function and is defined as

$$N_\Lambda(\lambda)=\begin{cases}g|g^T(y-\lambda)\le 0,\forall y\in\Lambda, & \lambda\in\Lambda,\\ \emptyset, & \lambda\notin\Lambda.\end{cases}$$

Recall the Peaceman-Rachford splitting method that was discussed in Section 3.3.1. This splitting method can be applied to the optimality condition (A.7). The optimality condition can be split up into the two operators T_1 and T_2 as

$$0\in \overbrace{-C\partial f^*(-C^T\lambda)+d}^{T_1(\lambda)}+\overbrace{N_\Lambda(\lambda)}^{T_2(\lambda)}. \tag{A.8}$$

The solution to this monotone inclusion problem is iteratively approximated by the Peaceman-Rachford iteration, which is given by

$$\begin{aligned} z^{(k+1)} &= C_{cT_2} \circ C_{cT_1}(z^{(k)}), \\ \lambda^{(k+1)} &= J_{cT_1}(z^{(k)}), \end{aligned} \tag{A.9}$$

where the Cayley operator was defined as (3.3) $C_{cT} = 2J_{cT} - I$. Therefore a closer look is taken at the resolvents of T_1 and T_2, respectively J_{T_1} and J_{T_2}, in order to simplify the iterates. This is where the proximal operator (3.5) from Chapter 2 comes in useful. The proximal operator is the resolvent of the subdifferential of a function, defined as

$$J_{c\partial f}(x) = \text{prox}_{cf}(x) = \arg\min_u \left[f(u) + \frac{1}{2c} \|u - x\|_2^2 \right].$$

This is used to rewrite the resolvent of the operator $T_2(\lambda)$ (A.8) as $J_{cT2}(y) = J_{cN_\Lambda}(y) = J_{c\partial I_\Lambda}(y)$. Recognize that the latter is the resolvent of the subdifferential of a function, which was the definition of the proximal operator. Rewriting the resolvent J_{cT_2} by making use of the proximal operator gives

$$\begin{aligned} J_{cT_2}(y) &= \arg\min_u \left[I_\Lambda(u) + \frac{1}{2c} \left\| u - y \right\|_2^2 \right], \\ &= \arg\min_{u \in \Lambda} \left\| u - y \right\|_2^2, \\ &= \Pi_\Lambda(y), \end{aligned} \tag{A.10}$$

where $\Pi_\Lambda(y)$ the projection of y onto Λ, which will from now on be written as u^+. This projection can be further simplified. The Lagrangian and optimality conditions are derived from (A.10) to obtain an alternative expression for u^+. We start of by deriving the Lagrangian as

$$\mathscr{L}(y, u, v) = \left\| u - y \right\|_2^2 + v^T (Pu - u).$$

This Lagrangian leads to two different different partial derivatives, one with respect to u and the other with respect to v. Setting both these derivatives equal to zero results in the optimality conditions as

$$\frac{\partial \mathscr{L}}{\partial u} = 0 \quad \Rightarrow \quad 2(u^+ - y) + (P - I)^T v = 0, \tag{A.11}$$

$$\frac{\partial \mathscr{L}}{\partial v} = 0 \quad \Rightarrow \quad Pu^+ - u^+ = 0. \tag{A.12}$$

These optimality conditions are combined in order to arrive at an expression for u^+ as

$$\begin{aligned} u^+ &= \frac{1}{2}\left(u^+ + u^+\right), \\ &\overset{(A.12)}{=} \frac{1}{2}\left(u^+ + Pu^+\right), \\ &\overset{(A.11)}{=} \frac{1}{2}\left(y + Py\right), \\ &= \frac{1}{2}\left(I + P\right) y. \end{aligned}$$

This means that we now have $J_{cT_2} = 1/2(I + P)$. This can be used to rewrite the Cayley operator on T_2 as

$$\begin{aligned} C_{cT2} &= 2J_{cT_2} - I, \\ &= 2\left[\frac{1}{2}(I + P)\right] - I, \\ &= P. \end{aligned}$$

So the Cayley operator on operator T_2 can be replaced by the permutation matrix P. This observation can be used to simplify the Peaceman-Rachford iteration (A.9) as

$$\begin{aligned} \lambda^{(k+1)} &= J_{cT_1}(z^{(k)}), \\ y^{(k+1)} &= 2\lambda^{(k+1)} - z^{(k)}, \\ z^{(k+1)} &= P y^{(k+1)}. \end{aligned} \tag{A.13}$$

Notice that the y-iterate is simply the Cayley operator on operator T_1 written out as $C_{cT_1} = 2J_{cT_1} - I$, which means that the z-iterate is still $C_{cT_2} \circ C_{cT_1}(z^{(k)})$.

At this point the λ-iterate has not changed when compared to the standard Peaceman-Rachford iteration (A.9). Therefore we will now take a closer look at this iterate in order to simplify the expression. The definition of the resolvent is needed in order to find an equivalent expression for $J_{cT_1}(z)$ and was given by $J_{cT} = (I + cT)^{-1}$. The λ-iterate of the Peaceman-Rachford iteration can be used to find an alternative expression for $\lambda^+ = J_{cT_1}(z)$ (A.13) as

$$\begin{aligned} \lambda^+ &= J_{cT_1}(z), \\ \lambda^+ &= (I + cT_1)^{-1}(z), \\ \lambda^+ + cT_1(\lambda^+) &= z, \\ \lambda^+ &\in z - cT_1(\lambda^+), \end{aligned} \tag{A.14}$$

where the operator T_1 is given by $T_1(\lambda^+) = -C\partial f^*(-C^T\lambda) + d$ (A.8). This operator can be substituted into (A.14) resulting in

$$\lambda^+ = z - c\left(-Cx^+ + d\right), \tag{A.15}$$

where $x^+ \in \partial f^*(-C^T\lambda^+)$. Recall the relation between the derivative of the conjugate function and the derivative of a standard function, which is given by $\partial f^* = (\partial f)^{-1}$. This property can be applied to x^+ as

$$\begin{aligned} x^+ &\in \partial f^*(-C^T\lambda^+), \\ \partial f(x^+) &= -C^T\lambda^+, \\ 0 &\in \partial f(x^+) + C^T\lambda^+. \end{aligned}$$

Substituting the previously derived expression for λ^+ (A.15) results in

$$\partial f(x^+) + C^T z + cC^T(Cx^+ - d) = 0.$$

Which means that we end up with the following expression for x^+

$$x^+ = \arg\min_x \left[f(x) + z^T(Cx - d) + \frac{c}{2}\|Cx - d\|_2^2 \right].$$

With this expression for the x-iterate, the full iteration (A.13) now becomes

$$\begin{aligned} x^{(k+1)} &= \arg\min_x \left[f(x) + z^{(k)^T}(Cx - d) + \frac{c}{2}\|Cx - d\|_2^2 \right], \\ \lambda^{(k+1)} &= z^{(k)} + c(Cx^{(k+1)} - d), \\ y^{(k+1)} &= 2\lambda^{(k+1)} - z^{(k)}, \\ z^{(k+1)} &= P y^{(k+1)}. \end{aligned} \tag{A.16}$$

These are the iterates of the standard PDMM algorithm. The P matrix is still the permutation matrix, exchanging the upper half and the lower half of the y-iterate with each other. Recall that λ was constructed in a way that at convergence $\lambda(l) = \lambda(l + m)$, with l an arbitrary edge number. Applying the permutation matrix means that two nodes connected by an edge exchange their value, thus the permutation matrix represents the transmission of data between neighbouring nodes.

B

Distributed optimization problems

During the simulations in Chapter 5, the PDMM algorithm is applied to three different problems. This eliminates the possibility that the convergence rate of the inexact PDMM algorithm depends on the optimized objective function. Also, as mentioned in the introduction, the problem defines the objective function that is minimized locally and will therefore have an influence on the inexactness. The three different problems that are simulated are: A quadratic problem, a problem of a higher order and a problem with a logarithm in the objective function. The first problem is the distributed average consensus problems where a distinct value is assigned to each node and the aim is to find the average of the values of all nodes in the network. The second problem minimizes a p-norm to the power p, where the order of the problem is determined by the parameter p and is chosen as $2 < p < \infty$. Lastly, the third problem is the Gaussian channel capacity problem, where the objective is to allocate power among different channels in a way that optimizes the capacity of all channels combined. In this appendix, the three problems are formulated, rewritten if necessary and finally cast in the previously derived PDMM iteration (A.16).

B.1. Distributed average consensus

The first problem is the distributed average consensus problem, where each node has a distinct different value assigned and the goal is to compute the average of all values. Consider a network in which each node $i \in V$ has an initial value $x_i^{(0)}$. The objective is to reach consensus at the average of all these initial values. The distributed consensus problem is an important problem and has been studied extensively on its own [7, 10, 17, 27, 28]. It is also used in various other applications such as distributed sensor fusion [25, 26], load balancing among processors in parallel computers [2, 5, 10] and the distributed coordination of autonomous moving agents [14, 20].

The first step in deriving the PDMM iteration for the distributed averaging problem is to state the problem as a minimization problem. This minimization is then compared to the general optimization problem (A.1) to see how it can be cast into the PDMM algorithm. The distributed averaging problem is given by

$$\begin{aligned} \min_x \quad & \sum_{i\in V} \frac{1}{2}(x_i - x_i^{(0)})^2, \\ s.t. \quad & x_i - x_j = 0, \quad \forall (i,j) \in E \end{aligned} \tag{B.1}$$

where $x_i^{(0)}$ is the initial value of node i. Setting the derivative of the objective function (B.1) equal to zero results in the average value, which verifies that this objective function leads to the desired value. Recognize that (B.1) is already of the same form as the general convex optimization problem (A.1). Therefore this can be cast directly in the PDMM iteration (A.16) with $f(x) = \sum_{i\in V} \frac{1}{2}(x_i - x_i^{(0)})^2$ and $d = 0$, since $b = 0$. This results in the following PDMM iteration to solve the distributed averaging problem

$$
\begin{aligned}
x^{(k+1)} &= \arg\min_x \left[\sum_{\epsilon \in V} \left(\frac{1}{2}(x_i - x_i^{(0)})^2 \right) + z^{(k)^T} Cx + \frac{\rho}{2} \|Cx\|_2^2 \right], \\
\lambda^{(k+1)} &= z^{(k)} + \rho C x^{(k+1)}, \\
y^{(k+1)} &= 2\lambda^{(k+1)} - z^{(k)}, \\
z^{(k+1)} &= P y^{(k+1)}.
\end{aligned} \tag{B.2}
$$

Notice that the constant c of the PDMM iteration is replaced by ρ. The ρ is exactly the same thing, but this way the difference between the matrix C and the optimization constant ρ is clearer. From now on ρ will consistently be used when referring to the optimization variable.

The next thing to do is to derive an analytic expression for the x-iterate. This is used for the simulations in two ways. Firstly the analytic expression of the update equation for the x-iterate is used to simulate the exact PDMM algorithm, which is used to compare the other results to. The second way the analytic expression is used is that the minimization to compute the x-iterate is also simulated by computing the x-iterate exact and adding a randomly generated Gaussian error to it. This way it is possible to accurately control the variance of the error, which directly impacts the magnitude of the error. The first step is to write the x-iterate vector based as

$$
x^{(k+1)} = \arg\min_x \left[\frac{1}{2}(x - x^{(0)})^T (x - x^{(0)}) + z^{(k)^T} Cx + \frac{\rho}{2} \|Cx\|_2^2 \right],
$$

where $x, x^{(0)} \in \mathbb{R}^n$, with n the number of nodes in the network and $x^{(0)}$ a vector with all the initial values stacked. The optimal value for this update equation is found by setting the derivative equal to zero which yields

$$
\begin{aligned}
&\frac{\partial x^{(k+1)}}{\partial x} = x - x^{(0)} + C^T z + cC^T Cx, \\
\Rightarrow \quad &x = (I + cC^T C)^{-1}(x^{(0)} - C^T z).
\end{aligned} \tag{B.3}
$$

This is the analytic expression that is used during the simulations to find the exact value for the x-iterate.

B.2. P-norm optimization

The previous average consensus problem had a quadratic objective function (B.1), which can accurately be approximated by the Newton-Raphson method in one step since this method uses a second order approximation. The next thing to investigate is the behaviour of the inexact PDMM algorithm with an objective function of higher powers than two. This higher power will result in an inexact approximation when the Newton-Raphson method is used. This problem is mainly theoretically motivated in an attempt to test the inexact PDMM algorithm by optimizing various different problems. We start off by formulating the p-norm optimization problem as

$$\begin{aligned} \min_x \quad & \|x-a\|_p^p, \\ s.t. \quad & x_i - x_j = 0, \qquad \forall (i,j) \in E, \end{aligned} \tag{B.4}$$

where $x, a \in \mathbb{R}^n$ with n the number of nodes and p an integer restricted by $2 < p < \infty$. The vector a consists of (different) constant values, one for each node. If this vector was not subtracted, the optimal x would always be a vector containing all zeros. The optimization problem (B.4) can equivalently be written as

$$\begin{aligned} \min_x \quad & \sum_{i \in V} |x_i - a_i|^p \\ s.t. \quad & x_i = x_j, \qquad \forall (i,j) \in E. \end{aligned}$$

The problem can now be compared to the general optimization problem (A.1) to investigate how this can be solved by using the PDMM algorithm. The p-norm optimization problem is already of the same form as the general optimization problem and can therefore directly be cast into the PDMM iteration (A.16), which results in

$$\begin{aligned} x^{(k+1)} &= \arg\min_x \left[\sum_{i \in V} \left(|x_i - a_i|^p \right) + z^{(k)^T} Cx + \frac{\rho}{2} \|Cx\|_2^2 \right], \\ \beta^{(k+1)} &= z^{(k)} + \rho C x^{(k+1)}, \\ y^{(k+1)} &= 2\beta^{(k+1)} - z^{(k)}, \\ z^{(k+1)} &= P y^{(k+1)}. \end{aligned} \tag{B.5}$$

Notice that the absolute value in the objective function causes it to be defined as

$$f_i(x) = \begin{cases} x_i - a_i & x_i \geq a_i \\ -x_i + a_i & x_i < a_i \end{cases}$$

The obvious thing to do is finding an algebraic expression for the x-iterate in order to have a way to compute the iterate exact. The derivative of the x-iterate (B.5) results in

$$\frac{\partial x^{(k+1)}}{\partial x_i} = \begin{cases} p(x_i - a_i)^{p-1} + C_i^T z^{(k)} + \rho C_i^T C_i x_i, & x_i \geq a_i, \\ -p(-x_i + a_i)^{p-1} + C_i^T z^{(k)} + \rho C_i^T C_i x_i, & x_i < a_i, \end{cases}$$

where the two different derivatives are caused by the absolute value. For $2 < p < \infty$, the derivative has an order of at least two, which means that it is not possible to compute the x-iterate algebraically while keeping p variable. The only way to compute the x-iterate is by using a solver. The two options are to either use a built-in solver from MATLAB or implement a solver. The latter is chosen to have as much control as possible over the solver. The algorithm that is implemented is the Newton-Raphson method. This method iteratively approximates the minimization by using the first and second order derivatives to take a step in the direction where the value of the objective function decreases. Interested readers are referred to [6, p. 484] for a detailed explanation on the Newton-Raphson method and its implementation. Besides the previously derived first order derivatives, the second order derivatives are also needed for the implementation of the Newton-Raphson method, which are given by

$$\frac{\partial^2 x^{(k+1)}}{\partial x_i^2} = \begin{cases} p(p-1)(x_i - a_i)^{p-2} + \rho C_i^T C_i, & x_i \geq a_i, \\ p(p-1)(-x_i + a_i)^{p-2} + \rho C_i^T C_i, & x_i < a_i. \end{cases}$$

With both the first and second order derivatives derived, the Newton-Raphson method can be implemented. The disadvantage of not having an algebraic expression is that it is impossible to compute the solution of the x-iterate exact and that the Newton-Raphson method will always be an approximation. However by carefully choosing the stopping criterion and allowing the Newton-Raphson method to have a sufficient number of iterations, the error of the approximation becomes small enough for trustworthy simulations.

B.3. Channel capacity optimization

The third optimization problem that will be solved in a distributed manner by using the PDMM algorithm is the channel capacity problem. Consider k different parallel Gaussian channels, each characterized by its noise variance σ_i^2 with $1 \leq i \leq k$, where the channels have a shared power constraint. This power constraint specifies the total power that is available to distribute among the channels. The objective is to distribute the total power P_{tot} in a way that optimizes the capacity of all channels combined. This is a well-known problem in the information theorem which results is the water-filling solution [6, 9]. The idea behind water-filling is to allot the available power to the channels with the lowest noise. This principle is illustrated in Figure B.1, where we can see that the available power is allotted to the three channels with the lowest noise and the channel with the highest noise does not get any power assigned, since the total available power was already distributed before the level of the noise variance of the fourth channel was reached.

The standard channel capacity problem is an iterative and centralized approach. However, a decentralized version of this approach can be derived. The centralized channel capacity problem is rewritten as a consensus problem such that it is possible to solve the problem in a distributed manner by making use of the PDMM algorithm. The following steps are taken to rewrite the problem:

1. Formulate the channel capacity problem as a minimization problem.
2. Construct the Lagrangian to eliminate the global constraint.
3. Compute the Karush-Kuhn-Tucker conditions, which are necessary to ensure that the computed solution is optimal.
4. Find the dual problem and rewrite this into a consensus problem.
5. Cast the dual problem into the PDMM algorithm.
6. The constrained minimization problem can now be solved in a distributed manner.

The first step is to formulate the channel capacity problem as a convex optimization problem. We consider the problem

$$\begin{aligned} &\min_x -\sum_{i\in V} \log(\sigma_i^2 + x_i), \\ &s.t. \quad x \succeq 0, \qquad \mathbf{1}^T x = 1, \end{aligned} \tag{B.6}$$

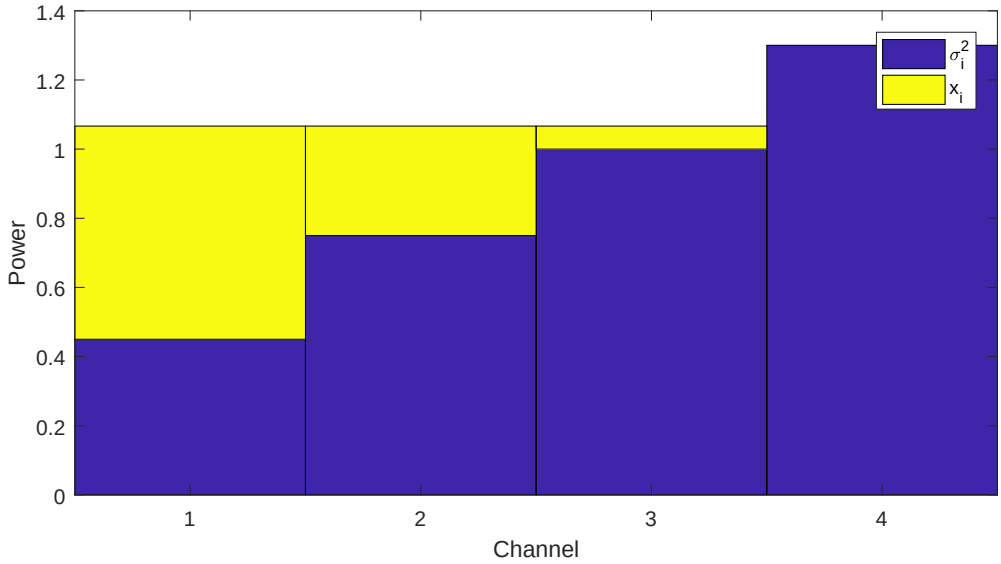

Figure B.1: Waterfilling

where x_i the power allotted to channel i and σ_i^2 the noise variance of channel i. Without loss of generality it is assumed that $P_{tot} = 1$, otherwise the constraint $\mathbf{1}^T x = 1$ can simply be altered to a desired $\mathbf{1}^T x = P_{tot}$ with P_{tot} the total available power. This is a convex optimization problem with an equality and an inequality constraint, where the equality constraint is a global constraint. To derive a distributed approach, both constraints have to be local such that the constraints can be checked at the nodes. To achieve this, the Lagrangian is first derived as

$$\begin{aligned} &\mathscr{L}(x,\mu,\lambda) = \sum_{i\in V}\left[-\log(\sigma_i^2 + x_i)\right] - \mu(\mathbf{1}^T x - 1) - \lambda^T x, \\ &s.t. \quad \lambda \geq 0, \end{aligned} \tag{B.7}$$

where $\mu \in \mathbb{R}, \lambda\mathbb{R}^{n\times 1}$ the Lagrange multipliers. To ensure that the found solution is optimal, the Karush-Kuhn-Tucker (KKT) conditions for the channel capacity problem are derived as

$$\begin{aligned} x^* &\geq 0, \\ \mathbf{1}^T x^* &= 1, \\ \lambda^* &\geq 0, \\ \lambda_i^* x_i^* &= 0 \qquad \forall i \in V, \end{aligned} \tag{B.8}$$

$$\frac{-1}{\sigma_i^2 + x_i^*} - \mu^* - \lambda_i^* = 0 \qquad \forall i \in V. \tag{B.9}$$

The last KKT-condition can be used to derive an expression for the optimal value of x. This expression can be used in Appendix B.3.1 to investigate the effect of inexact updates on the channel capacity problem. By making use of the last KKT-condition, the expression for the optimal value of x becomes

$$x_i^* = \frac{-1}{\mu^* + \lambda^*} - \sigma_i^2. \tag{B.10}$$

The originally stated problem (B.6) had a global constraint. Therefore the dual function is now derived, which leads to a dual problem that can be solved in a distributed manner. The Lagrangian (B.7) and the analytic expression for x_i (B.10) are combined to end up with the following dual function

$$g(\lambda,\mu) = \sum_{i\in V}\left[-\log\left(\frac{-1}{\mu_i + \lambda_i}\right) + 1 + \mu\sigma_i^2 + \lambda_i\sigma_i^2\right] + \mu. \tag{B.11}$$

The logarithm in the dual function results in an additional constraint on the Lagrange multipliers, namely $\mu_i + \lambda_i < 0$. The dual problem with the dual function (B.11) and the additional constraint becomes

$$\begin{aligned} \min_{\lambda,\mu} \quad & \sum_{i\in V}\left[\log\left(\frac{-1}{\mu_i + \lambda_i}\right) - \mu_i\sigma_i^2 - \lambda_i\sigma_i^2 - \frac{\mu_i}{N}\right], \\ s.t. \quad & \mu_i = \mu_j && \forall (i,j) \in V, \\ & \mu_i + \lambda_i < 0 && \forall i \in V, \\ & \lambda_i \geq 0 && \forall i \in V. \end{aligned} \tag{B.12}$$

At this point the problem is rewritten as a consensus problem and can now be solved by using using the PDMM algorithm. The dual problem (B.12) can be cast into the PDMM iteration (A.16), which gives the following iteration

$$\begin{aligned}
\lambda^{(k+1)},\mu^{(k+1)} &= \arg\min_{\mu,\lambda}\left(\sum_{i\in V}\left[\log\left(\frac{-1}{\mu_i+\lambda_i}\right)-\mu_i\sigma_i^2-\lambda_i\sigma_i^2-\frac{\mu_i}{N}\right]+z^{(k)^T}C\mu+\frac{\rho}{2}\left\|C\mu\right\|_2^2\right),\\
&\qquad s.t. \qquad \mu_i+\lambda_i<0, \qquad \lambda_i\geq 0 \qquad \forall i\in V,\\
\beta^{(k+1)} &= z^{(k)}+\rho C\mu^{(k+1)},\\
y^{(k+1)} &= 2\beta^{(k+1)}-z^{(k)},\\
z^{(k+1)} &= Py^{(k+1)}.
\end{aligned} \tag{B.13}$$

This is equivalent to the iteration for both previous problems except for the constraints that are now present at the update equation for the x-iterate and the fact that the x-iterate is computed by optimizing over two variables. To solve this constrained problem, the following steps are taken:

1. Solve the minimization as if the constraints are not present.
2. Check if the computed solutions fulfill the constraints.
3. If the constraints are not fulfilled, we have $\lambda_i<0$. The reason for this will be explained when the constraints are discussed in detail. To find the optimal solution set $\lambda_i=0$, which is the closest point to the optimal value, and solve for μ_i again.

First, in order to solve the optimization, the update equation for the λ and μ iterates is rewritten to include everything within the summation as

$$\lambda^{(k+1)},\mu^{(k+1)} = \arg\min_{\lambda,\mu}\left[\sum_{i\in V}\log\left(\frac{-1}{\mu_i+\lambda_i}\right)-\mu_i\sigma_i^2-\lambda_i\sigma_i^2-\frac{\mu_i}{N}+z^{(k)^T}C_i\mu_i+\frac{\rho}{2}C_i^TC_i\mu_i^2\right], \tag{B.14}$$

where C_i the i-th column of C. In order to find an analytic expression for μ_i and λ_i, the partial derivatives are derived as

$$\frac{\partial\left(\lambda^{(k+1)},\mu^{(k+1)}\right)}{\partial\mu_i} = \frac{-1}{\mu_i+\lambda_i}-\sigma_i^2-\frac{1}{N}+C_i^Tz+\rho C_i^TC_i\mu_i, \tag{B.15}$$

$$\frac{\partial\left(\lambda^{(k+1)},\mu^{(k+1)}\right)}{\partial\lambda_i} = \frac{-1}{\mu_i+\lambda_i}-\sigma_i^2. \tag{B.16}$$

These partial derivatives can be used to find algebraic expressions for the λ and μ iterates. These partial derivatives vanish at the optimal points. First (B.16) is set to zero in order to find an expression for λ_i, which is given by

$$\lambda_i = -\mu_i-\frac{1}{\sigma_i^2}. \tag{B.17}$$

This expression for λ_i is substituted into (B.15) in order to obtain an expression for μ_i as

$$\mu_i = \frac{1/N-C_i^Tz}{\rho C_i^TC_i}. \tag{B.18}$$

These expressions are used to solve the minimization without taking the constraints into considerations, which was the first step. The second step is to check whether the computed μ_i and λ_i fulfill the constraints. The two constraints that have to be fulfilled are

1. $\mu_i+\lambda_i<0$,
2. $\lambda_i\geq 0$.

By inspection of the constraint on λ_i (B.17), we see that this can be rewritten as

$$\lambda_i + \mu_i = -\frac{1}{\sigma_i^2},$$

where the σ_i^2 values are positive since these represent the noise variances of the channels. This means that the first constraint is automatically fulfilled, so only checking the second constraint is sufficient. If $\lambda_i < 0$, the second constraint is not fulfilled. In this case, the previously derived expressions for λ_i (B.17) and μ_i (B.18) do not hold anymore. The way to solve the problem then is to set $\lambda_i = 0$ and substitute this into (B.15) to find an alternative expression for μ_i. This results in the following expression

$$\begin{aligned} & \frac{-1}{\mu_i} - \sigma_i^2 - \frac{1}{N} + C_i^T z + \rho C_i^T C_i \mu_i = 0, \\ \Rightarrow \quad & \underbrace{\rho C_i^T C_i}_{a} \mu_i^2 + \underbrace{\left[-\sigma_i^2 - \frac{1}{N} + C_i^T z \right]}_{b} \mu_i \underbrace{-1}_{c} = 0. \end{aligned} \tag{B.19}$$

Equation (B.19) is solved to find an expression for μ_i by using the standard quadratic formula as

$$\mu_i = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{B.20}$$

Now the expression for μ_i has changed, the constraints have to be checked again. However, since $\lambda_i = 0$, the constraints can be simplified. The constraint $\lambda_i \geq 0$ will automatically be satisfied and the constraint $\mu_i + \lambda_i < 0$ becomes $\mu_i < 0$. We start by investigating a, b and c from the quadratic formula independently to see if μ_i satisfies this constraint. Recall that the C matrix is constructed by using the A matrix, which defines the edges (connections) in the network. Since the A and C matrix both contain an equivalent number of non-zero values in each column, we have $C_i^T C_i = A_i^T A_i$, which is the number of neighbors of node i. Since the network is connected, each node has at least one neighbour and we have $C_i^T C_i > 0$. The optimization constant ρ will also be larger than zero. Combining this means that for a of the quadratic formula (B.20), $a > 0$ holds.

Since $c = -1$ and $a > 0$, we have $-4ac = 4a > 0$. Evaluating the argument of the square root of the quadratic formula and using $-4ac > 0$ gives $b^2 - 4ac > b^2$. To summarize for both of the cases of the quadratic formula we have:

$$\begin{aligned} \mu_{i,1} &= \frac{-b + \sqrt{b^2 - 4ac}}{2b}, \\ &> \frac{-b + \sqrt{b^2}}{2b}, \\ &= \frac{-b + |b|}{2b}. \end{aligned} \qquad \left| \qquad \begin{aligned} \mu_{i,2} &= \frac{-b - \sqrt{b^2 - 4ac}}{2b}, \\ &< \frac{-b - \sqrt{b^2}}{2b}, \\ &= \frac{-b - |b|}{2b}. \end{aligned} \right.$$

This means that there will always be exactly one solution to the quadratic formula for which $\mu_i < 0$ holds and therefore fulfills the constraint.

The final thing to discuss is the computation of the optimal values of the power distribution (x_i^*) as well as the Lagrange multipliers (λ_i^* and μ_i^*). The optimal values are needed to compute the error at each iteration, since the results will be presented as plots showing the error. Later on in Appendix B.3.1 the dependency between the errors caused by the inexact PDMM algorithm on μ_i and λ_i, and the variable that we are interested in, x_i, will be derived. Therefore the optimal values of all three these variables are needed, to see if this dependency is also confirmed by the simulations.

We start off by discussing an approach to obtain the optimal value of x. Since this is standard water-filling, the optimal value can easily be computed in a non-distributed way. Without loss of generality assume that the channels are sorted with the first channel having the lowest noise variance and the last channel the highest noise variance, thus $\sigma_1^2 \leq \sigma_2^2 \leq \ldots \leq \sigma_n^2$. Then the optimal power distribution can be computed as follows:

1. Begin by allotting power (x_i) to the first channel.
2. Keep adding power to the first channel until either $x_1 = P_{tot}$ or $\sigma_1^2 + x_1 = \sigma_2^2$
3. Evenly add power to both channels until either $x_1 + x_2 = P_{tot}$ or $\sigma_1^2 + x_1 = \sigma_3^2$
4. Keep repeating this process of distributing the available power untill the noise variance of the next channel is reached or until the total amount of power, P_{tot}, is used.

In these steps, P_{tot} is the total available power and x_i is the power allotted to channel i. Recall that the total available power was assumed to be 1 (B.6). This process was also graphically shown in the beginning of this section in fig. B.1. With this method, the optimal power distribution can be computed.

Next, the optimal values for μ and λ are also needed as reference. At this point it is important to realize that the channel capacity problem was rewritten as a consensus problem (B.12), where the consensus was defined by $\mu_i = \mu_j, \forall (i,j) \in V$. This means that at convergence, all nodes ends up with the exact same value for μ. The consequence is that computing μ_i^* for one of the nodes in the network gives μ^* for all nodes in the network. Recall the complementary slackness condition (B.8) given by $\lambda_i^* x_i^* = 0$. This means that for each node either λ^* or the x^* must be equal to zero. The interpretations for the channel capacity problem is that for the channels to which power is allotted ($x_i^* \neq 0$), have $\lambda_i^* = 0$. Since the optimal values for x have previously been computed, it can easily be checked which ones are non-zero. The nodes for which $x_i^* \neq 0$ holds, have $\lambda_i^* = 0$. By choosing one of the nodes for which $\lambda_i^* = 0$ holds, the expression for the optimal value of x (B.10) can be rewritten and $\lambda_i^* = 0$ is substituted to find an expression for μ^* as

$$\mu^* = \frac{-1}{x_i^* + \sigma_i^2}, \qquad \text{when } \lambda_i^* = 0.$$

At this point it is possible to find reference values for x^* and μ^*. As previously discussed, $\lambda_i^* = 0$ if $x_i^* \neq 0$, so only λ_i^* needs to be calculated for nodes for which $x_i^* = 0$ holds. For the optimal value of λ, λ_i^*, (B.17) can be used, which results in

$$\lambda_i^* = -\mu_i^* - \frac{1}{\sigma_i^2}.$$

B.3.1. Inexact Channel capacity problem

The PDMM iteration to solve the channel capacity problem has just been derived. This was all under the assumption that all the iterates are computed exact. However, by inspection of the iteration (B.13) it becomes clear that the λ and μ iterates are approximated by solving a minimization. This inherently means that this approximation will introduce inexactness again. Therefore it is interesting to determine the effect of the inexact PDMM algorithm on the exactness of the resulting x-iterate for the channel capacity problem, since this is the variable of interest. The error of an iterate is defined as the difference between the exact update and the inexact update as

$$\left\| \epsilon_{x_i}^{(k)} \right\| = \left\| \hat{x}_i^{(k)} - \tilde{x}_i^{(k)} \right\|, \tag{B.21}$$

where the hat represents the inexact update and the tilde stands for the exact update, both based on the inexact variables from the previous iteration. To clarify in what way the update is based on an inexact variable, the minimization (B.14) to compute λ and μ is based on the inexact z from the previous iteration. The exact update of the x-iterate is defined as (B.10)

$$\tilde{x}_i^{(k)} = \frac{-1}{\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)}} - \sigma_i^2.$$

The link between the PDMM algorithm and the x-iterate is that the PDMM algorithm finds values for $\mu_i^{(k)}$ and $\lambda_i^{(k)}$, which are substituted into the expression for the x-iterate. Therefore $\epsilon_{\mu_i}^{(k)}$ and $\epsilon_{\lambda_i}^{(k)}$ are introduced as

respectively the error on the μ-iterate and the error on the λ-iterate at iteration k. Now the inexact update for the x-iterate can be defined with these errors explicitly incorporated as

$$\begin{aligned}\hat{x}_i^{(k)} &= \frac{-1}{\hat{\mu}_i^{(k)} + \hat{\lambda}_i^{(k)}} - \sigma_i^2, \\ &= \frac{-1}{\tilde{\mu}_i^{(k)} + \epsilon_{\mu_i}^{(k)} + \tilde{\lambda}_i^{(k+1)} + \epsilon_{\lambda_i}^{(k)}} - \sigma_i^2.\end{aligned}$$

The difference between the exact update and the inexact update is that the errors caused by the PDMM algorithm appear in the inexact expression. With both the exact and inexact update equations derived, these can be substituted and the error on the x-iterate (B.21) becomes

$$\begin{aligned}\left\|\epsilon_{x_i}^{(k)}\right\| &= \left\|\hat{x}_i^{(k)} - \tilde{x}_i^{(k)}\right\|, \\ &= \left\|\left(\frac{-1}{\tilde{\mu}_i^{(k)} + \epsilon_{\mu_i}^{(k)} + \tilde{\lambda}_i^{(k)} + \epsilon_{\lambda_i}^{(k)}} - \sigma_i^2\right) - \left(\frac{-1}{\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)}} - \sigma_i^2\right)\right\|, \\ &= \left\|\frac{1}{\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)}} - \frac{1}{\tilde{\mu}_i^{(k)} + \epsilon_{\mu_i}^{(k)} + \tilde{\lambda}_i^{(k)} + \epsilon_{\lambda_i}^{(k)}}\right\|, \\ &= \left\|\frac{\epsilon_{\mu_i}^{(k)} + \epsilon_{\lambda_i}^{(k)}}{(\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)})(\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k+1)} + \epsilon_{\mu_i}^{(k)} + \epsilon_{\lambda_i}^{(k)})}\right\|.\end{aligned}$$

This is the influence errors on the λ-iterate and μ-iterate, which are caused by using the PDMM algorithm with a limited number of iterations, have on the x-iterate. This expression shows that the error on the x-iterate does not only depends on the errors introduced by the inexactness of the PDMM algorithm, ϵ_{μ_i} and ϵ_{λ_i}, but also on the actual vales of μ_i^* and λ_i^*. This because we know that if the algorithm converges, $\tilde{\mu}_i^{(k+1)}$ and $\tilde{\lambda}_i^{(k+1)}$ will get closer to the optimal values μ_i^* and λ_i^* at each successive iteration. Now recall the constraint given by $\mu_i + \lambda_i < 0$. This means that the sum of both will most likely not be zero and therefore it is safe to assume that if the algorithm converges, $|\mu_i + \lambda_i| \gg |\epsilon_{\mu_i} + \epsilon_{\lambda_i}|$, which makes the errors in the denominator negligible. Applying this observation results in

$$\left\|\epsilon_{x_i}^{(k)}\right\| = \left\|\frac{\epsilon_{\mu_i}^{(k)} + \epsilon_{\lambda_i}^{(k)}}{(\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)})^2}\right\|, \tag{B.22}$$

where the tilde on a variable still means that it represents the exact update. The plots presented in chapter 5 will be on a logarithmic scale since the PDMM algorithm has the property that it converges at a geometric rate [34]. Therefore it is interesting to investigate what (B.22) implies in the logarithmic domain. This results in

$$\log\left\|\epsilon_{x_i}^{(k)}\right\| = \log\left\|\epsilon_{\mu_i}^{(k)} + \epsilon_{\lambda_i}^{(k)}\right\| - \log\left\|(\tilde{\mu}_i^{(k)} + \tilde{\lambda}_i^{(k)})^2\right\|.$$

This expression implies that, when shown on a logarithmic scale, the error on x-iterate is the errors caused by the PDMM algorithm on the μ and λ-iterates with the sum of the actual values of μ and λ squared and subtracted from that. Note that this expression is problem specific for the channel capacity problem and will not hold for the other problems.

Bibliography

[1] I. F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, Aug 2002. ISSN 0163-6804. doi: 10.1109/MCOM.2002.1024422.

[2] Amnon Barak and Amnon Shiloh. A distributed load-balancing policy for a multicomputer. *Software: Practice and Experience*, 15(9):901–913, 9 1985. ISSN 1097-024X. doi: 10.1002/spe.4380150905. URL `http:https://doi.org/10.1002/spe.4380150905`.

[3] Viorel Barbu and Teodor Precupanu. *Convexity and optimization in Banach spaces*. Springer Science & Business Media, 2012.

[4] Jacques F Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische mathematik*, 4(1):238–252, 1962.

[5] J. E. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2 (4):289–313, 12 1990. ISSN 1096-9128. doi: 10.1002/cpe.4330020403. URL `http:https://doi.org/10.1002/cpe.4330020403`.

[6] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[7] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE/ACM Transactions on Networking (TON)*, 14(SI):2508–2530, 2006.

[8] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[9] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[10] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[11] Daniel Gabay and Bertrand Mercier. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & Mathematics with Applications*, 2(1):17–40, 1976.

[12] Herman Heine Goldstine. *A History of Numerical Analysis from the 16th through the 19th Century*, volume 2. Springer Science & Business Media, 2012.

[13] Ralph E Gomory. An algorithm for integer solutions to linear programs. *Recent advances in mathematical programming*, 64:260–302, 1963.

[14] A. Jadbabaie, Jie Lin, and A. S. Morse. Coordination of groups of mobile autonomous agents using nearest neighbor rules. *IEEE Transactions on Automatic Control*, 48(6):988–1001, June 2003. ISSN 0018-9286. doi: 10.1109/TAC.2003.812781.

[15] Mark Aleksandrovich Krasnosel'skii. Two remarks on the method of successive approximations. *Uspekhi Matematicheskikh Nauk*, 10(1):123–127, 1955.

[16] Jingwei Liang, Jalal Fadili, and Gabriel Peyré. Convergence rates with inexact non-expansive operators. *Mathematical Programming*, 159(1-2):403–434, 2016.

[17] Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[18] W Robert Mann. Mean value methods in iteration. *Proceedings of the American Mathematical Society*, 4 (3):506–510, 1953.

[19] USDA Press Office. Forest service wildland fire suppression costs exceed $ 2 billion. URL https://www.usda.gov/media/press-releases/2017/09/14/forest-service-wildland-fire-suppression-costs-exceed-2-billion.

[20] R. Olfati-Saber and R. M. Murray. Consensus problems in networks of agents with switching topology and time-delays. *IEEE Transactions on Automatic Control*, 49(9):1520–1533, Sept 2004. ISSN 0018-9286. doi: 10.1109/TAC.2004.834113.

[21] Judea Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles, 1982.

[22] Roger Penrose. A generalized inverse for matrices. In *Mathematical proceedings of the Cambridge philosophical society*, volume 51, pages 406–413. Cambridge University Press, 1955.

[23] Ralph Tyrell Rockafellar. *Convex analysis*. Princeton university press, 2015.

[24] Ernest K Ryu and Stephen Boyd. Primer on monotone operator methods. *Appl. Comput. Math*, 15(1): 3–43, 2016.

[25] Demetri P Spanos, Reza Olfati-Saber, and Richard M Murray. Distributed sensor fusion using dynamic consensus. In *IFAC World Congress*. Prague Czech Republic, 2005.

[26] L. Xiao, S. Boyd, and S. Lall. A scheme for robust distributed sensor fusion based on average consensus. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 63–70, April 2005. doi: 10.1109/IPSN.2005.1440896.

[27] Lin Xiao and Stephen Boyd. Fast linear iterations for distributed averaging. *Systems & Control Letters*, 53 (1):65–78, 2004.

[28] Lin Xiao, Stephen Boyd, and Seung-Jean Kim. Distributed average consensus with least-mean-square deviation. *Journal of Parallel and Distributed Computing*, 67(1):33 – 46, 2007. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2006.08.010. URL http://www.sciencedirect.com/science/article/pii/S0743731506001808.

[29] G. Zhang and R. Heusdens. Linear coordinate-descent message-passing for quadratic optimization. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2005–2008, March 2012. doi: 10.1109/ICASSP.2012.6288301.

[30] Guoqiang Zhang and Richard Heusdens. Bi-alternating direction method of multipliers. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3317–3321. IEEE, 2013.

[31] Guoqiang Zhang and Richard Heusdens. Bi-alternating direction method of multipliers over graphs. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 3571–3575. IEEE, 2015.

[32] Guoqiang Zhang and Richard Heusdens. On simplifying the primal-dual method of multipliers. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 4826–4830. IEEE, 2016.

[33] Guoqiang Zhang and Richard Heusdens. Distributed optimization using the primal-dual method of multipliers. *IEEE Transactions on Signal and Information Processing over Networks*, 2017.

[34] Guoqiang Zhang, Richard Heusdens, and W Bastiaan Kleijn. On the convergence rate of the bi-alternating direction method of multipliers. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 3869–3873. IEEE, 2014.