NTNU

Norwegian University of
Science and Technology

Introducing a MATLAB Toolbox for F-Lipschitz Optimization

Maren Åshild Leithe

Master of Science in Engineering Cybernetics
Submission date: June 2011
Supervisor: Bjarne Anton Foss, ITK
Co-supervisor: Carlo Fischione, Department of Automatic Cotrol, KTH

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

The theory of mathematical optimization is useful within a wide range of disciplines such as science, engineering, economics and industry. Application areas have been growing steadily, driving forward the development of new effective methods. Inspired by the need for fast computational schemes in wireless sensor networks, a new optimization theory, called Fast Lipschitz, has emerged to provide effective algorithms both for distributed and centralized computations. An important property of these algorithms is that a *globally* optimal solution is always guaranteed. In this master thesis project, a new MATLAB toolbox is developed to check wether an optimization problem is F-Lipschitz and to solve it efficiently. The difficulty is posed in verifying that a given problem is in fact F-Lipschitz. However, it is shown that under certain circumstances, this operation has a computational complexity of $O(n^2)$ for a problem with n decision variables. The toolbox provides both a graphical interface as well as inline functions. A user guide is presented, explaining the functionalities by discussions and illustrations of example problems. Among others, a convex optimization problem of *distributed detection* is considered, as well as a non-convex *radio power allocation* problem. The novel toolbox presented in this thesis may be of considerable utility in solving optimization problems and studying their characteristics.

Acknowledgements

I would like to thank my supervisor, Ass. professor Carlo Fischione with the Department for Automatic Control at KTH, Stockholm, for providing excellent guidance, for all of the interesting discussions and for always taking the time to answer my questions.

Contents

1 Introduction

The continuous development of intelligent sensors steadily opens up new areas in which wireless sensor networks (WSNs) can be applied. For example, we may want to keep track of temperature and humidity in order to monitor the environment, or maybe we would like to monitor a patient to collect data for medical diagnostics. These are just a few possible applications, see [1] for more examples. There are many different aspects to WSNs, see e.g. [2], [3], [4], but the focus of this thesis is on *optimization*. It is a widely applied mathematical tool used for decision-making, improving efficiency and reducing development costs. Particularly, the different aspects of *distributed optimization* is researched. In the discussions which follows, it will become clear that this area of optimization has a particular importance within WSNs.

Wireless sensors have their own processing unit and data storage, but the computational capacity is limited, as is also the memory. Hence, in order to take advantage of the *combined* processing power in a sensor network, nodes have to collaborate with each other through message passing. In some situations the best option may be to apply a central coordination unit which receives information from all the sensors nodes in the network. This in turn enables it to solve some global optimization problem before returning the solution to the nodes. To limit the amount of information which is transmitted, each node may do simple computations and hence send data which is already partially processed.

However, in many situations it is difficult or even impossible to apply such a central coordination. In these cases it is necessary to apply *distributed* algorithms, i.e., each node runs local computations based on its own data and the information it receives from its neighboring nodes in order to solve an optimization problem which is global for the network. Consequently, the nodes will cooperate in parallel and, in effect, one might say that the network in itself becomes the coordination unit. We know for certain that, compared to a corresponding centralized setup, the distributed optimization algorithm will inevitably be slower. Safe to say then, the motives behind decentralizing will always be of a different nature. As mentioned, it may very well be the one alternative which is possible.

1.1 Motivation

Either distributed or centralized, *convex optimization* has over all been the most widely applied optimization strategy. Most problems are not convex, but many can be approximated convex in order to use Lagrangian methods which is a well established theory. The area of these methods which concerns distributed optimization is referred to as *Lagrangian decomposition methods*. Not surprising, there are however plenty of situations where convex approximation is not possible. For solving nonlinear optimization problems, which may be either convex or non-convex, a method referred to

as *iterative contraction mappings* has been developed within the *interference function optimization* theory. This may be applied in both centralized and in distributed settings, though we are most interested in the latter case here. The theory is not particularly general however, hence the range of problems it may be applied to is limited.

The decomposition methods are so far the ruling strategy for general distributed optimization. However, convergence is not particularly *fast*, which is of course always desirable. For the application of WSN, one might go so far as to say that fast algorithms are even a crucial requirement. As the wireless network easily picks up noise, the communication channels may have fast changing dynamics, and so the optimal solution must be calculated and applied quickly before it is already outdated. What is worse, the decomposition methods require a lot of message exchanging between nodes, which of course consumes power. Since these sensors are wireless, they have a limited power source, and hence it must be used efficiently. Yet another good reason as to why the optimization algorithm needs to converge fast; to minimize the amount of message passing.

1.2 Problem definition

Motivated by the issues discussed above, a new method for solving a class of nonlinear optimization problems has recently been developed by C. Fischione, see [5]. It has been named *Fast Lipschitz* (F-Lipschitz) optimization, which indicates that the method is particularly fast and that it is based on *Lipschitz contractive* constraints. The theory is general and may be applied to both centralized and distributed optimization. Considering the last application, it has been found that the method is faster and simpler than what has previously been available, not to mention that it is completely asynchronous, which is highly appealing for WSNs. At the same time it limits the amount of messages which are sent between nodes.

It has been shown that a feasible F-Lipschitz problem has a unique optimal solution which is given by the constraints at the equality. This opens up for a very efficient centralized algorithm in which the objective is to find the solution to a system of equations. The key to the fast distributed algorithm lies in the constraints being contractive. Basically, this means that the constraint function may be used to map the iterate into itself, which eventually will result in the iterate converging to the solution. Like always, there is a drawback to this wonderful new theory. To verify that a problem is in fact a feasible F-Lipschitz problem is potentially difficult. However, one might remark that this is nothing new, as the same can be said for problems which are convex.

The focus of this master thesis is to investigate F-Lipschitz optimization theory and compare its performance to other possible methods within *distributed optimization*. A literature survey on this subject is a natural part of the task. MATLAB® (developed by MathWorks) already includes an Op-

timization Toolbox^{TM}, which provides methods for the traditional convex optimization problems. This thesis includes an implementation of a new MATLAB toolbox for F-Lipschitz problems. It is designed to be able to verify if a problem is F-Lipschitz, as well as solving it in both a distributed and centralized setting. A natural part of this is to thoroughly investigate the properties implying that a problem is F-Lipschitz.

1.3 Outline of this work

The subsequent chapters are organized as follows. Chapter 2 summarizes a literature study that is focused on distributed optimization. In Chapter 3, the necessary background theory of F-Lipschitz optimization is presented, as well as a general comparison to the methods and techniques of Chapter 2. Chapter 4 revolves around all the different aspect related to the design of the toolbox. Among other things, it contains an extensive discussion on the computational complexity of verifying that a problem is F-Lipschitz. Also, we consider how the computations may be simplified by assuming certain problem structures or certain properties. A user guide of the toolbox is presented in Chapter 5, which includes examples of typical optimization problems within WSNs.

2 Previous Work

We initiate this section by a short introduction to the basics of optimization theory. Consider a real-valued function $f(x)$, where $x \in \mathbb{R}^n$, defined on a set $\mathcal{F}$, that is, $f(x) : \mathcal{F} \to \mathbb{R}$. The goal of optimization is to determine a $\hat{x} \in \mathcal{F}$ that minimizes $f(x)$, specifically, $f(\hat{x}) \leq f(x)$, $\forall x \in \mathcal{F}$. Formally, the *optimization problem* is defined as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in \mathcal{F} \end{aligned}$$

where the function f is referred to as the *objective function* and $\mathcal{F}$ as the *feasible set*, which is usually defined by a set of *constraints*.

2.1 Parallelization of iterative methods

For more extensive reading on the theory that is summarized in this section and the next, refer to [6] by Bertsekas and Tsitsiklis. The way of solving optimization problems, in general, is through applying iterative methods on the form

$$x(k+1) = f\left(x(k)\right), \qquad k = 0, 1, \ldots, \tag{2.1.1}$$

where each $x(k)$ is an n-dimensional vector, and f is some function mapping $\mathbb{R}^n$ into itself, i.e. $f : \mathbb{R}^n \to \mathbb{R}^n$. We denote the sequence generated by Eq. (2.1.1) as $\{x(k)\}$. If this sequence converges towards a limit x^*, and the function f is continuous, then we say that x^* is a *fixed point* of f and it satisfies $x^* = f(x^*)$.

It is quite straightforward to parallelize the execution of such an algorithm. If we denote the ith component of $x(k)$, $x_i(k)$, and let f_i denote the ith component of f, then Eq. (2.1.1) can be rewritten as

$$x_i(k+1) = f_i\left(x_1(k), \ldots, x_n(k)\right), \qquad i = 1, \ldots, n \tag{2.1.2}$$

Parallelization is then accomplished by letting each agent in a network of n nodes update one component of $x(k)$. For each iteration, the ith agent must know the values of all the components for which f_i is dependent on. Hence, when the iterate is updated, each agent must communicate its value to the nodes that need it.

Naturally, it is not necessary and not always the most efficient to parallelize the execution completely. Instead of decomposing $x(k)$ into scalar components, we may partition into larger pieces, namely vectors. These partitions will be called *block-components*. Formally, the vector space $\mathbb{R}^n$ will be decomposed into a Cartesian product of lower dimensional subspaces $\mathbb{R}^{n_j}$, where $j = 1, \ldots, p$ and $\sum_{j=1}^{p} n_j = n$. Correspondingly, the vector $x \in \mathbb{R}^n$

becomes $x = (x_1, \ldots, x_j, \ldots, x_p)$, where each x_j is an n_j-dimensional vector. Now we can write Eq. (2.1.1) as

$$x_j(k+1) = f_j\left(x(k)\right), \qquad j = 1, \ldots, p \tag{2.1.3}$$

where each $f_j : \mathbb{R}^n \to \mathbb{R}^{n_j}$. Instead of a network of n nodes, we now only need p agents, each of which is assigned the updating of a block-component of x. The resulting algorithm is then said to be *block-parallelized*.

This type of parallelization creates more flexibility in the system. For example, there may be a limited amount of agents and so it might be necessary to assign more than just one component to each of them. Also, it is not unlikely that some functions f_i involve common computations, hence it would be more efficient to group them together. Naturally, this also reduces the amount of communication which is necessary between nodes.

In addition to parallelizing the iterations, i.e. computing all the components of x simultaneously, there is yet another way of distributing an iterative algorithm. Namely, by letting the components of x be updated one at a time. This type of iteration is called *Gauss-Seidel*, while the formerly explained parallelized type, Eq. (2.1.2), is sometimes called *Jacobi*. Assuming that the components are updated in the order of its index i, a Gauss-Seidel iteration has the following form

$$x_i(k+1) = f_i\left(x_1(k+1), \ldots, x_{i-1}(k+1), x_i(k), \ldots, x_n(k)\right) \tag{2.1.4}$$

where $i = 1, \ldots, n$ as before. Notice that the update is done using the most recently computed values of each component of x, i.e. the algorithm always uses the most updated information. This has been known to sometimes have a positive effect on the convergence. For this reason, Gauss-Seidel algorithms are often preferred before the Jacobi-type algorithms, see [6] (Section 1.2).

It may not come as a surprise that it is also sometimes possible to parallelize a Gauss-Seidel iteration, in effect applying a combination of the two types. How much can be parallelized is dependent on the problem formulation. If for example every function f_i depends on all the components of x, then parallelization is not possible at all. Also, the chosen order of which the components are updated might be such that it limits the level of parallelization compared to a different ordering. It is often natural to choose the ordering which maximizes the parallelization. However, one must keep in mind that a different order in fact corresponds to a different algorithm, hence the result will in general be different.

In the following section we explore different types of formulations of the function f in Eq. (2.1.1). Several general algorithms will be given as well as a discussion on how they are to be distributed and potentially parallelized.

2.2 Convex optimization algorithms

The amount of optimization algorithms which have been developed over the years are immense. No attempt will be made to summarize all of these, but the most standard algorithms for general convex optimization will be shortly explained in the following. We now consider the optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & F(x) \\ \text{subject to} \quad & x \in X \end{aligned} \tag{2.2.1}$$

where $F : \mathbb{R}^n \to \mathbb{R}$ is convex and the set $X \subseteq \mathbb{R}^n$ is convex and closed. Note that the following algorithms are general and can be applied for non-convex problems as well. The difference is that in such cases there is no guarantee that the optimal point is actually the *global* optimal point, it is merely a local solution. While for convex cases it is guaranteed that the solution is globally optimal.

2.2.1 Gradient and subgradient methods

As we are to find the minimum of a function, we need to know in which direction the function decreases, i.e. we must consider the function's *gradient*, $\nabla F(x)$. This is of course dependent upon F being differentiable. To show that an iterative algorithm actually ends up at a minimizing point, it is desirable that the cost function decreases for each step. This is otherwise known as iterating in a *descent direction*. Formally, a direction s is descent if it fulfills

$$s^T \nabla F(x) < 0 \tag{2.2.2}$$

where ^{T} denotes the transpose operator. If a step size, γ, is sufficiently small and positive, this strategy yields $F(x + \gamma s) < F(x)$, which obviously translates into a decrease for each step. An algorithm that applies this principal is called a *descent algorithm*.

One of the most usual types of descent algorithms is the gradient algorithm. The idea is to try to decrease the cost function as much as possible for each iteration, this is done by iterating in the opposite (negative) direction of the gradient of F. Because of this, the gradient algorithm is often called the *steepest descent algorithm*. For the unconstrained case, i.e. for $X = \mathbb{R}^n$, it is defined as follows

$$x(k+1) = x(k) - \gamma \nabla F(x(k)) \tag{2.2.3}$$

where γ is the step size as was previously mentioned. A quite straightforward algorithm, but not so much if the problem happens to be constrained. As long as the iterate keeps inside of the set X, then it's just as easy as before. However, if we were to take a step which takes the iterate outside of the feasible region, then we have to *project* the iterate back onto the set. Letting

$P_X[\cdot]$ denote the orthogonal projection on the convex and closed set X, we define the *projected gradient algorithm* as follows

$$x(k+1) = P_X\left[x(k) - \gamma \nabla F(x(k))\right] \tag{2.2.4}$$

Projecting basically means that we must find the point in X which is closest to the iterate. Note that, as previously indicated, if the iterate is inside the set, then the closest point is the iterate itself. The orthogonal projection of a point $x_0 \in \mathbb{R}^n$ on X can be formally stated as

$$P_X[x_0] = \arg\min_{z \in X} ||z - x_0||_2 \tag{2.2.5}$$

where $||\cdot||_2$ is the Euclidian norm. Notice now that the projection may be interpreted as an optimization problem in itself. We want to minimize the distance between z and x_0, which is the objective, subject to z being inside X. For the algorithm (2.2.4), to be well defined, there must be a unique $z \in X$ which minimizes $||z - x_0||_2$. Since both the Euclidian norm and X are convex, uniqueness can be proven, see e.g. [6] (Proposition 3.2, Projection Theorem).

Now consider the case where the cost function, F, is not differentiable. Then, the corresponding alternative is to use a *subgradient* instead of the gradient. We use the following definition from [7] (Definition 2.1.4):

Definition 2.1. *A vector $a \in \mathbb{R}^n$ is a subgradient of a convex function $F : \mathbb{R}^n \to \mathbb{R}$ at a point $x \in \mathbb{R}^n$ if*

$$F(y) \geq F(x) + a^T(y - x), \quad \forall y \in \mathbb{R}^n \tag{2.2.6}$$

The set of all subgradients of a convex function F at $x \in \mathbb{R}^n$ is called the subdifferential of F at x, and is denoted by $\partial F(x)$:

$$\partial F(x) = \{a \in \mathbb{R}^n | F(y) \geq F(x) + a^T(y - x), \forall y \in \mathbb{R}^n\} \tag{2.2.7}$$

Note that if the subdifferential at x only contains one element, then F is differential at x with $\nabla F(x) = \partial F(x)$. The subgradient algorithm is identical to the projected gradient algorithm given by Eq. (2.2.4), except that the gradient is replaced by a subgradient. However, contrary to the gradient algorithm, the cost function will not necessarily decrease for every iteration. Here, the arguments for convergence are in stead based on the decrease of the distance between the iterate and the optimal solution.

Comments on convergence
If we apply a fixed step size γ to the subgradient algorithm, then the best-case scenario is convergence to an *area around* the optimal point. Formally, if the subgradients are bounded by some constant φ, then the algorithm converges to a ball around the optimal point, x^*, given by

$$\liminf_{k \to \infty} \quad F(x(k)) \leq F(x^*) + \frac{\gamma \varphi^2}{2} \tag{2.2.8}$$

See e.g. [8] (Proposition 8.2.2) for proof. To make sure that the algorithm actually converges, it will be necessary to apply a time-varying step size $\gamma(k)$ which diminishes over time. If the step size is chosen to fulfill

$$\gamma(k) \geq 0, \quad \sum_{k=1}^{\infty} \gamma(k) = \infty, \quad \sum_{k=1}^{\infty} \gamma(k)^2 < \infty, \tag{2.2.9}$$

and the subgradients are bounded, then the algorithm will converge to the optimal point. See e.g. [8] (Proposition 8.2.6) for proof.

Luckily, the analysis for the gradient algorithm gives a slightly more satisfying result. If $\nabla F(x)$ is Lipschitz continuous with Lipschitz constant, L, i.e.

$$||\nabla F(x) - \nabla F(y)|| \leq L||x - y||, \quad \forall x, y \in X \tag{2.2.10}$$

then the algorithm will converge to an optimal point using a constant step size, $\gamma \in (0, \frac{L}{2})$. See e.g. [6] (Proposition 3.4) for proof. If the Lipschitz property does not hold, another alternative is to, for each step, find the step size which results in the largest possible reduction of the cost function. That is, we always choose the γ which minimizes $F(x - \gamma \nabla F(x))$. This operation will however require global coordination, hence it is not a suitable alternative in the realm of distributed optimization.

2.2.2 Scaled gradient methods

Sometimes it may be possible to achieve an improved direction of iteration by *scaling* the gradient algorithm. We then get the following update

$$x(k+1) = P_X \left[x(k) - \gamma D(k)^{-1} \nabla F(x(k)) \right] \tag{2.2.11}$$

where $D(k)$ is the scaling matrix, which naturally must be invertible. This task is very easy if the matrix is chosen to be diagonal. A special case of this algorithm is the projected Jacobi method, where $D(k)$ is chosen to be diagonal with its elements the same as the diagonal of the Hessian matrix $\nabla^2 F(x(k))$.

Another important method emerges if we set $D(k)$ equal to the Hessian, namely *Newton's algorithm*. For quadratic problems, i.e. problems on the form $F(x) = \frac{1}{2} x^T A x - x^T b$, it can be shown that the algorithm converges in one step. Not to mention, it has been established (see e.g. [6]) that the algorithm in general converges much faster than other similar algorithms like the ones previously introduced.

The drawback is the potentially heavy computations required to invert the Hessian, especially since it must be done for each iteration. Also, we

must consider that the Hessian might not even be invertible. It won't even exist unless we require that F is twice differentiable at x. So the trick is typically to let $D(k)$ be an easy to invert approximation of the Hessian, which is the case for the mentioned Jacobi algorithm.

For the unconstrained case, i.e. we can leave out the projection $P_X[\cdot]$, Eq. (2.2.11) in general converges to the optimal point. However, for the constrained case the algorithm in general *fails* to converge. This is because the sequence $\{x(k)\}$ generated by the algorithm does not have the minimizing point x^* as a fixed point. In order to obtain convergence it is necessary to define a different kind of projection, namely we must replace the Euclidian norm with a norm which is determined by $D(k)$. We define the norm

$$||x||_{D(k)} = \left(x^T D(k) x\right)^{1/2} \tag{2.2.12}$$

and apply this to obtain the new projection

$$P_X[x_0]_{D(k)} = \quad \arg\min_{z \in X} ||z - x_0||_{D(k)} \tag{2.2.13}$$

Under certain conditions of the matrix $D(k)$, it can be shown that the minimum of Eq. (2.2.13) is attained at a unique element of X, see e.g. [6] (Proposition 3.6).

2.2.3 Distribution and parallelization

The algorithms which have been presented in this section can all be distributed in a network as it was made clear in Section 2.1. The ith agent updates the ith component of x according to the algorithm of choice, and the update is communicated to the agents which require it. Dependency is determined by the objective function F. Agent i must receive the current x_j if $\nabla_i F$ depends on it or, as in the case of e.g. the Jacobi algorithm, if $\nabla^2_{ii} F$ depends on it.

Each of the algorithms, as they are stated without the projection, can then be parallelized according to the Jacobi algorithm (2.1.2). However, as was previously explained, the projection poses an optimization problem in itself which involves all the components of x. In other words, a projection algorithm is not amenable to be parallelized as such. There is however an important special case which allows for parallelization. Namely, if $X = [x_{1,\min}, x_{1,\max}] \times [x_{2,\min}, x_{2,\max}] \dots [x_{n,\min}, x_{n,\max}]$, otherwise known as a box, the projection can be done independently for each component. The ith component of x is simply projected onto the interval given by $[x_{i,\min}, x_{i,\max}]$.

Alternatively, we state the more general case in which the set X is a Cartesian product of lower dimensional subsets X_i and we decompose x into block-components. Each X_i is a closed convex subset of $\mathbb{R}^{n_i}$ and $n_1 + n_2 + \cdots + n_p = n$. Then the projection of x on X is equal to the vector

$\big[P_{X_1}[x_1], \ldots, P_{X_p}[x_p]\big]$, where $P_{X_i}[x_i]$ is the projection of x_i onto X_i. This also opens up for a Gauss-Seidel type iteration of the projected algorithm. See [6] (Proposition 3.8) for proof of convergence of the Gauss-Seidel gradient projection algorithm.

2.3 Decomposition methods

We now somewhat switch our focus. In this section we will be discussing how the actual *problem structure* creates possibilities for distributed computations, as compared to the *method structure* which was previously discussed. The techniques presented here are based on *strong duality*, i.e. the optimal values of the dual problem are the same as for the primal problem. This can often be shown to apply for problems that are convex. It can however also be proven for non-convex problems, though it is more difficult. Refer to [9] for the theory which is summarized here.

The idea behind decomposition methods is that we can split the original optimization problem into several *subproblems*. We can often achieve totally independent subproblems, but they need to be coordinated in order to find a solution to the original problem. For this reason, we define a *master problem* which is equivalent to the original problem. See Figure 1 for an illustration of the decomposition.

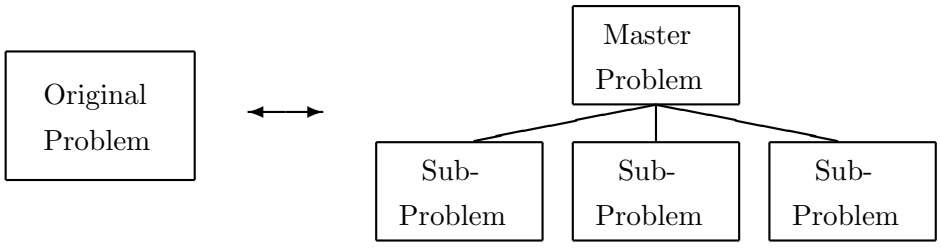

Figure 1: The original problem is decomposed into a master problem and several subproblems.

The methods will be explained through simple examples before we conclude with a more general summary. Though the results might be obvious, we start with the most basic form of decomposition. This is known as the *separable* problem, which may e.g. have the following structure

$$\begin{aligned} &\text{minimize} && f_1(x_1) + f_2(x_2) \\ &\text{subject to} && x_1 \in \mathcal{C}_1, \quad x_2 \in \mathcal{C}_2 \end{aligned} \tag{2.3.1}$$

where $x_1 \in \mathbb{R}^{n_1}$ and $x_2 \in \mathbb{R}^{n_2}$. The objective is so called *block separable* in x_1 and x_2. This means that f_1 and f_2 can be solved separately, either in parallel or sequentially. Though the example is quite trivial, it illustrates quite simply what we strive to achieve. In the subsequent sections we will introduce two types of changes to the structure of the problem which will

inevitably complicate the decomposition. In other words, it will be necessary to apply some techniques in order to transform the initially non-separable problem into a structure which is separable.

2.3.1 Complicating variable

We introduce a variable which complicates the problem in the sense that, if we fix it, the problem becomes separable. It can also be referred to as the *coupling* variable. For simplicity, let us consider the unconstrained problem

$$\text{minimize} \quad f(x) = f_1(x_1, y) + f_2(x_2, y) \tag{2.3.2}$$

where $x = (x_1, x_2, y)$, $x_1 \in \mathbb{R}^{n_1}$, $x_2 \in \mathbb{R}^{n_2}$ and $y \in \mathbb{R}^{n_y}$. Here, y is a vector of complicating variables. We can think of x_1 and x_2 as *private* or *local* variables, while y can be considered as *public* or *interface* variables between the two subproblems f_1 and f_2. There are generally two ways of decomposing problem (2.3.2), namely through *primal* or *dual* decomposition.

Primal decomposition
The discussion which follows also applies when the problem is extended with separable constraints, i.e. constraints on the form $x_1 \in \mathcal{C}_1$, $x_2 \in \mathcal{C}_2$. We fix y and define

$$\begin{aligned} &\text{subproblem 1:} \quad \underset{x_1}{\text{minimize}} \quad f_1(x_1, y) \\ &\text{subproblem 2:} \quad \underset{x_2}{\text{minimize}} \quad f_2(x_2, y) \end{aligned} \tag{2.3.3}$$

with the respective optimal values $\phi_1(y)$ and $\phi_2(y)$. Further, we define the master problem such that it is equivalent to the original problem, as follows

$$\underset{y}{\text{minimize}} \quad \phi(y) = \phi_1(y) + \phi_2(y). \tag{2.3.4}$$

This problem may also be referred to as the *primal problem*. Notice that it is defined by the complicating variables of the original problem. If $\phi(y)$ is differentiable, the master problem can be solved using e.g. the gradient or Newton method, if not then a subgradient algorithm must be applied. See Algorithm 1 for a subgradient version.

Algorithm 1 Primal

loop
1. Solve the subproblems.
   Find x_1 that minimize $f_1(x_1, y)$ and a subgradient $g_1 \in \partial\phi_1(y)$
   Find x_2 that minimize $f_2(x_2, y)$ and a subgradient $g_2 \in \partial\phi_2(y)$
2. Update complicating variable
   $y_{k+1} := y_k - \gamma(k)(g_1 + g_2)$

end loop

For each iteration of the master problem, we fix y and solve the two subproblems independently. We find the subgradients g_1 and g_2 of f_1 and f_2 which, quite naturally, also corresponds to subgradients of ϕ_1 and ϕ_2 at y. A subgradient of ϕ at y is then given by $g_1 + g_2$. If $\phi_1(y)$ and $\phi_2(y)$ are differentiable, then the subgradients are simply given by the gradient of the optimal value of the subproblems with respect to y.

Dual decomposition
We introduce two auxiliary variables y_1 and y_2 and reformulate the problem (2.3.2) as follows

$$\begin{aligned} \underset{y}{\text{minimize}} \quad & f_1(x_1, y_1) + f_2(x_2, y_2) \\ \text{subject to} \quad & y_1 = y_2 \end{aligned} \tag{2.3.5}$$

The idea is to apply local versions y_1 and y_2 of the complicating variable y and at the same time ensure consistency through the additional constraint. Now, clearly this problem is still not separable, but the dual problem is. The Lagrangian is defined as

$$L(x_1, y_1, x_2, y_2, \lambda) = f_1(x_1, y_1) + f_2(x_2, y_2) + \lambda^T(y_1 - y_2) \tag{2.3.6}$$

where λ is the Lagrange multipliers. Then, the dual problem is given by the maximization of the Lagrangian with respect to λ. As we can see, the problem is now separable, i.e. we can minimize over (x_1, y_1) and (x_2, y_2) separately. We define

$$\begin{aligned} \text{subproblem 1:} \quad & \underset{x_1, y_1}{\text{infimum}} \quad \left\{ f_1(x_1, y_1) + \lambda^T y_1 \right\} \\ \text{subproblem 2:} \quad & \underset{x_2, y_2}{\text{infimum}} \quad \left\{ f_2(x_2, y_2) - \lambda^T y_2 \right\} \end{aligned} \tag{2.3.7}$$

with optimal values $g_1(\lambda)$ and $g_2(\lambda)$. The master problem then becomes

$$\underset{\lambda}{\text{maximize}} \quad g(\lambda) = g_1(\lambda) + g_2(\lambda) \tag{2.3.8}$$

which is equivalent to the dual problem and $g(\lambda)$ is called the *dual function*. As before, the choice of algorithm depends on the differentiability of $g(\lambda)$.

Some useful results will now be stated, see e.g. [8] (Section 8.1), which will accompany us in deriving a subgradient algorithm for solving the master problem. If we find $\bar{x}_1$ and $\bar{y}_1$ minimizing $g_1(\lambda)$, then a subgradient of $g_1(\lambda)$ at λ is given by $\bar{y}_1$. Correspondingly, a subgradient of $g_2(\lambda)$ at λ is given by $-\bar{y}_2$. Thus, for minimization of the negative dual function $-g(\lambda)$ (which naturally is the same as maximizing the positive dual function) we can use the subgradient $\bar{y}_2 - \bar{y}_1$. See Algorithm 2 for the resulting dual decomposition algorithm.

Algorithm 2 Dual

loop

1. Solve the subproblems.
   Find x_1, y_1 that minimize $f_1(x_1, y_1) + \lambda^T y_1$
   Find x_2, y_2 that minimize $f_2(x_2, y_2) - \lambda^T y_2$
2. Update dual variable
   $\lambda_{k+1} := \lambda_k - \gamma(k)(y_2 - y_1)$

end loop

Of course, the iterates y_1 and y_2 will in general not be feasible for the original problem (2.3.5), i.e. $y_1 \neq y_2$. Observing the algorithm, we see that feasibility will occur only at the maximum of $g(\lambda)$. A reasonable guess of a feasible point $(x_1, x_2, \bar{y})$ can be found by letting $\bar{y} = (y_1 + y_2)/2$, i.e. the average. This can be interpreted as the projection of y_1 and y_2 onto the set $y_1 = y_2$. An even better point can be found by taking this averaged value and calculate the two primal subproblems (2.3.3), thus finding the x_1 and x_2 which corresponds to $\bar{y}$.

2.3.2 Complicating constraints

We now consider the case where the two subproblems are coupled via constraints, known as complicating constraints. Consider the following problem

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & f_1(x_1) + f_2(x_2) \\ \text{subject to} \quad & x_1 \in \mathcal{C}_1, \quad x_2 \in \mathcal{C}_2 \\ & h_1(x_1) + h_2(x_2) \preceq 0 \end{aligned} \tag{2.3.9}$$

where $\mathcal{C}_1$ and $\mathcal{C}_2$ are the feasible sets of the subproblems. Let h_1 and h_2 have dimension p, then we have a set of p complicating constraints, i.e. constraints which involve both x_1 and x_2. Again, we consider both primal and dual decomposition as means for distributing the problem.

Primal decomposition

We can decompose the complicating constraints by defining a variable $t \in \mathbb{R}^p$ and the two subproblems as

$$\begin{aligned} \text{subproblem 1:} \quad & \underset{x_1}{\text{minimize}} \quad f_1(x_1) \\ & \text{subject to} \quad x_1 \in \mathcal{C}_1, \quad h_1(x_1) \preceq t \\ \text{subproblem 2:} \quad & \underset{x_2}{\text{minimize}} \quad f_2(x_2) \\ & \text{subject to} \quad x_2 \in \mathcal{C}_2, \quad h_2(x_2) \preceq -t \end{aligned} \tag{2.3.10}$$

We can interpret t as the amount of resources which is allocated to the first subproblem. Then, to be consistent with the original problem, the amount of resources allocated to the second subproblem must be $-t$. If we fix t, then

these two subproblems can be solved independently. We denote the optimal values $\phi_1(t)$ and $\phi_2(t)$ and the master problem, which is the equivalent of problem (2.3.9), is defined as

$$\underset{t}{\text{minimize}} \quad \phi(t) = \phi_1(t) + \phi_2(t) \tag{2.3.11}$$

As before, to apply a subgradient master algorithm, we must first find a subgradient for each of the two subproblems. See e.g. [8] (Section 6.5.3) for proof of the following results. Consider the convex optimization problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & x \in X, \quad h(x) \preceq z \end{array} \tag{2.3.12}$$

and let $p(z)$ denote the optimal value of the problem. If λ is an optimal dual variable for the constraint $h(x) \preceq z$, then a subgradient of $p(z)$ at z is given by $-\lambda$.

Hence, to find a subgradient for $\phi(t)$, we solve the two subproblems finding the optimal x_1 and x_2, as well as the optimal dual variables λ_1 and λ_2 associated with each of the constraints; $h_1(x_1) \preceq t$ and $h_2(x_2) \preceq -t$. Then, not suprising, a subgradient of $\phi(t)$ at t is given by $\lambda_2 - \lambda_1$. See Algorithm 3 for the resulting master algorithm.

Algorithm 3 Primal

loop

1. Solve the subproblems.
   Solve subproblem 1, finding x_1 and λ_1
   Solve subproblem 2, finding x_2 and λ_2
2. Update dual variable
   $t_{k+1} := t_k - \gamma(k)(\lambda_2 - \lambda_1)$

end loop

Dual decomposition

As for the problem with the complicating variable, it is quite easy to separate problem (2.3.9) through the Langrian, which is given as

$$L(x_1, x_2, \lambda) = f_1(x_1) + f_2(x_2) + \lambda^T(h_1(x_1) + h_2(x_2)) \tag{2.3.13}$$

For a fixed λ we can define the following subproblems

$$\begin{aligned}
&\text{subproblem 1:} && \underset{x_1}{\text{minimize}} && f_1(x_1) + \lambda^T h_1(x_1) \\
& && \text{subject to} && x_1 \in \mathcal{C}_1 \\
&\text{subproblem 2:} && \underset{x_2}{\text{minimize}} && f_2(x_2) + \lambda^T h_2(x_2) \\
& && \text{subject to} && x_2 \in \mathcal{C}_2
\end{aligned} \tag{2.3.14}$$

and given the respective optimal values $g_1(\lambda)$ and $g_2(\lambda)$ of the two subproblems, the master problem once again becomes

$$\underset{\lambda}{\text{maximize}} \quad g(\lambda) = g_1(\lambda) + g_2(\lambda) \tag{2.3.15}$$

We use the same results as before to derive a subgradient algorithm for the master problem. Let $\bar{x}_1$ and $\bar{x}_2$ minimize $g_1(\lambda)$ and $g_2(\lambda)$, respectively. Then a subgradient for $g_1(\lambda)$ at λ is given by $h_1(\bar{x}_1)$ and the corresponding for $g_2(\lambda)$ is given by $h_2(\bar{x}_2)$. Thus, the minimization of $-g(\lambda)$ can be done using the subgradient $-h_1(\bar{x}_1) - h_2(\bar{x}_2)$. See Algorithm 4 for the resulting algorithm.

Algorithm 4 Dual

loop

1. Solve the subproblems.
   Find $\bar{x}_1$ which minimizes subproblem 1
   Find $\bar{x}_2$ which minimizes subproblem 2
2. Update dual variable
   $\lambda_{k+1} := P_{\mathcal{A}}[\lambda_k + \gamma(k)(h_1(\bar{x}_1) + h_2(\bar{x}_2))]$

end loop

Note that we must apply a projected subgradient algorithm in this case since a Lagrangian multiplier associated with an inequality constraint must be positive. In other words, we must project onto the set $\mathcal{A} = \{\lambda : \lambda \geq 0\}$. Similar to the previous example, the optimal x_1 and x_2 of the subproblems are not necessarily feasible according to the original problem, i.e. we may have $h_1(x_1) + h_2(x_2) \not\leq 0$. If this is the case, we can find a feasible set of variables by letting $t = (h_1(x_1) - h_2(x_2))/2$ and solve the primal subproblems (2.3.10) in order to find the corresponding x_1 and x_2.

2.3.3 Decomposition in general

Considering primal and dual decomposition for problems with coupling variables and for problems with coupling constraints, it is clear that the techniques are actually quite similar. The main differences comes down to details on how to compute the necessary subgradients. This similarity is no coincidence, in fact there is a standard way of representing the coupling which applies for both cases, namely through the *consistency constraint*. Take e.g.

the problem (2.3.9) with complicating constraints. If we introduce two new variables y_1 and y_2 we can reformulate the problem as

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & f_1(x_1) + f_2(x_2) \\ \text{subject to} \quad & x_1 \in \mathcal{C}_1, \quad h_1(x_1) \preceq y_1 \\ & x_2 \in \mathcal{C}_2, \quad h_2(x_2) \preceq -y_2 \\ & y_1 = y_2 \end{aligned} \tag{2.3.16}$$

Now we have a problem which can be divided into two subproblems, only connected through the consistency constraint. This sort of reformulation can be done with any two subproblems which are coupled together through variables or constraints.

The standard form provides us with a general way of describing more complicated decomposition structures than what we have considered so far. We may represent a problem through a *hypergraph*, where each node represents a subproblem involving the local variables, objectives and constraints. Each *hyperedge* is then associated with a consistency constraint. Note that a hyperedge may connect more than just two nodes, in which case the associated consistency constraint enforces equality between more than two variables. Like the examples we have been discussing, the simplest decomposition structure consist of only two subsystems, as shown in Figure 2. A more complex structure is shown in Figure 3 which consists of 5 subproblems and 4 consistency constraints.

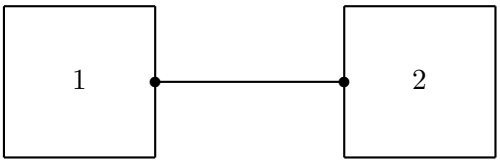

Figure 2: The simplest decomposition structure.

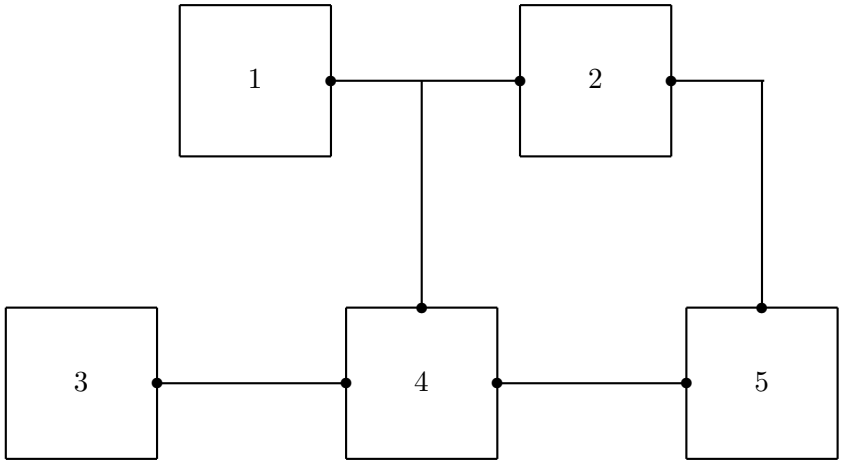

Figure 3: A hypergraph with 5 subsystems and 4 consistency constraints.

Considering the hypegraphs, we can now shortly explain how primal and dual decomposition works in general. For the first case, we let each hyperedge be associated with one public variable. Every subsystem solves its optimization problem independently using the associated public variable values. Then, each system must produce a subgradient for each hyperedge it is connected to. Combining these subgradients, the public variables can be updated, hopefully in such a fashion which will result in convergence.

For the dual decomposition, each subsystem has its own copy of the public variable associated with a hyperedge, as well as a *price vector*. Observing the examples discussed above, we see that this vector is represented by the Lagrangian multipliers, λ. Each subsystem optimizes its problem with respect to local variables, including the public variable copies. Then, the copies associated with a hyperedge can be compared with each other, followed by an updating of the prices in order to drive the copies towards equality, i.e. towards optimality.

An example where dual decomposition is applied to decentralize a specific type of problem, namely a geometric program, will be presented in the following section.

2.4 Geometric programs

We dedicate a section to geometric programs, GPs, as this type of optimization problem has received some attention lately. Efficient and reliable solution methods have been developed during the recent years and numerous applications, particularly in WSNs and circuit design, have been found to be equivalent to this type of problem. A GP is characterized by the special form of its objective function and constraints. Initially, the problem is not convex, but we will see that it can be mechanically converted into a convex problem. Refer to [10] for an extensive tutorial on geometric programming.

2.4.1 Problem formulation

To formulate a GP we must first introduce *monomial* and *posynomial* functions. Given a vector $x = (x_1, \ldots, x_n)$, a monomial function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$f(x) = c\, x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n} \tag{2.4.1}$$

where $c > 0$ and $a_i \in \mathbb{R}$, $i = 1, \ldots, n$. Note that, by the definition, any positive constant as well as any variable is a monomial. The division or multiplication of two monomials is also a monomial, additionally we may raise a monomial to any power. The sum of one or more monomials is called a posynomial and we define it as

$$f(x) = \sum_{k=1}^{K} c_k\, x_1^{a_{1_k}} x_2^{a_{2_k}} \cdots x_n^{a_{n_k}} \tag{2.4.2}$$

where $c_k > 0$ and $a_{i_k} \in \mathbb{R}$, $k = 1, \ldots, K$. Because we can have $K = 1$, a monomial is also a posynomial by definition. Two posynomials may be added together or multipied with the result being a posynomial. We may also divide a posynomial with a monomial.

Now, for the actual formulation of a geometric program. We define the following as a GP in *standard form*

$$\begin{array}{lll} \underset{x}{\text{minimize}} & f_0(x) & \\ \text{subject to} & f_i(x) \leq 1, & i = 1, \ldots, m \\ & h_l(x) = 1, & l = 1, \ldots, p \end{array} \tag{2.4.3}$$

where f_i, $i = 0, 1, \ldots, m$, are posynomials and h_l, $l = 1, \ldots, p$, are monomials. The standard form is not convex, but the trick is that we can convert problem (2.4.3) into a nonlinear convex problem by a logarithmic transformation of the objective and constraint functions. Let us define the variables $y_i = \log x_i$ and minimize the logarithm of the objective, $\log f_0$, subject to the logarithm of the constraints

$$\begin{array}{lll} \underset{y}{\text{minimize}} & \log f_0(e^y) & \\ \text{subject to} & \log f_i(e^y) \leq 0, & i = 1, \ldots, m \\ & \log h_l(e^y) = 0, & l = 1, \ldots, p \end{array} \tag{2.4.4}$$

where $y = (y_1, \ldots, y_n)$ and the notation e^y means componentwise exponentiation. This is a GP in *convex form*. For a problem to be convex, the equality constraints must be linear. Just to make a point of how simple and smart this formulation is, we will prove that this applies for problem (2.4.4). We have that $h(x)$ is monomial

$$h(x) = c\, x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n} \tag{2.4.5}$$

and so the transformation above results in

$$\begin{aligned} \log h(e^y) &= \log c + a_1 \log x_1 + \cdots + a_n \log x_n \\ &= \log c + a_1\, y_1 + \cdots + a_n\, y_n \end{aligned} \tag{2.4.6}$$

which is an affine function, i.e. a linear function plus a constant. Now, since we have that $\log h(e^y) = 0$, we get the following linear equality constraint

$$a_1\, y_1 + \cdots + a_n\, y_n = -\log c \tag{2.4.7}$$

2.4.2 Distributed algorithms

As the problem becomes convex, we know that there exists efficient and robust algorithms which can be applied in a central coordinated system. However, there has also been attempts on developing distributed algorithms for applications such as power control in wireless networks [11]. Basically, the goal is to minimize the radio power each node use to transmit their signals. The obvious reason for this is to save energy and to reduce interference on the transmissions of other nodes. At the same time, the signals must be strong enough to be successfully received at their destination. This problem will be discussed in the following.

We consider a network with n transmitter-receiver pairs and denote the transmit powers as P_i, $i = 1, \ldots, n$. An important term within power control is the Signal-to-Interference Ratio, SIR, which can be defined somewhat differently depending on the network dynamics. In [11] the SIR for receiver i is modeled as

$$SIR_i = \frac{P_i G_{ii} F_{ii}}{\sum_{j \neq i}^{N} P_j G_{ij} F_{ij} + n_i} \tag{2.4.8}$$

where G_{ij} is the channel gain from transmitter j to receiver i, F_{ij} models Rayleigh fading and n_i is the noise power for receiver i. As can be seen, the denominator is a posynomial and the numerator is a monomial, hence we know that $1/SIR$ is a posynomial. Considering this, there is in fact a large variety of different problems involving e.g. data rates, delays or outage probabilities which can be formed as GPs.

We are presented with a strategy for distributed implementation which is based on the decomposition methods formerly explained. Namely, dual decomposition is applied in order to partition the GP into smaller subproblems. Note that this is only made possible by transforming the GP into a *convex* problem. We consider the following standard form GP

$$\min \quad \sum_i f_i(x_i, \{x_j\}_{j \in I(i)}) \tag{2.4.9}$$

where $I(i)$ is the set of nodes which is coupled with node i, x_i and x_j

denotes the local variables and f_i is either a monomial or a posynomial. Note that, although the problem is unconstrained, the analysis will be similar for problems with constraints which are local to each node, something which we can expect in wireless networks. As before, we do a variable change, $y_i = \log x_i$, $\forall i$, to transform the problem to convex form

$$\min \quad \sum_i f_i(e^{y_i}, \{e^{y_j}\}_{j \in I(i)}) \tag{2.4.10}$$

The problem can now be decomposed by introducing auxiliary variables y_{ij} and consistency constraints

$$\begin{aligned} \min \quad & \sum_i f_i(e^{y_i}, \{e^{y_{ij}}\}_{j \in I(i)}) \\ \text{s.t.} \quad & y_{ij} = y_j, \ \ \forall j \in I(i), \ \forall i \end{aligned} \tag{2.4.11}$$

and defining the Langrangian as

$$\begin{aligned} & L(\{y_i\}, \{y_{ij}\}; \{\gamma_{ij}\}) \\ = & \sum_i f_i(e^{y_i}, \{e^{y_{ij}}\}_{j \in I(i)}) + \sum_i \sum_{j \in I(i)} \gamma_{ij}(y_j - y_{ij}) \\ = & \sum_i L_i(y_i, \{y_{ij}\}; \{\gamma_{ij}\}) \end{aligned} \tag{2.4.12}$$

where

$$\begin{aligned} & L_i(y_i, \{y_{ij}\}; \{\gamma_{ij}\}) \\ = & \sum_i f_i(e^{y_i}, \{e^{y_{ij}}\}_{j \in I(i)}) + \left(\sum_{j : i \in I(j)} \gamma_{ji} \right) y_i - \sum_{j \in I(i)} \gamma_{ij} y_{ij} \end{aligned} \tag{2.4.13}$$

Note that the second term of (2.4.13) is just the sum of all terms y_i in (2.4.12). This is exactly the same procedure as was applied for the simple example with the complicating variable in Section 2.3. Obviously, the minimization of each L_i can now be computed distributively by each node in parallel. The problem only contains local variables, except the dual variables $\{\gamma_{ji}, j : i \in I(j)\}$ which must be received through message passing. Last but not least; in order to obtain the optimal dual variables $\{\gamma_{ij}\}$ the master dual problem is defined as

$$\max_{\{\gamma_{ij}\}} \quad g(\{\gamma_{ij}\}) \tag{2.4.14}$$

where

$$g(\{\gamma_{ij}\}) = \sum_i \min_{y_i, \{y_{ij}\}} L_i(y_i, \{y_{ij}\}; \{\gamma_{ij}\}) \tag{2.4.15}$$

The following subgradient algorithm is suggested for solving the maximization given by (2.4.14)

$$\gamma_{ij}(t+1) = \gamma_{ij}(t) + \delta(t)(y_j(t) - y_{ij}(t)) \tag{2.4.16}$$

where $\delta(t)$ is the step size. As was explained in Section 2.2, the step size must diminish over time in order to obtain convergence. An appropriate choice could be $\delta(t) = \delta_0/t$ for some constant $\delta_0 > 0$.

Let us now shortly summarize the overall algorithm. Node i receives the dual variables $\{\gamma_{ji}, j : i \in I(j)\}$ and minimizes L_i to find y_i and $\{y_{ij}\}$. The value of y_i must then be passed to the nodes which need it, i.e. the nodes j where $i \in I(j)$. Receiving $\{y_j, j \in I(i)\}$, node i can update the local dual variables $\{\gamma_{ij}, j \in I(j)\}$ using (2.4.16). Finally, the newly updated dual variables are passed to the nodes which are coupled with node i.

Considering this procedure, it becomes very clear why decomposition methods in general demands a lot of message passing. Both the dual variables as well as the local variable y_i must be broadcasted to other nodes. If this method is to be applied directly for a power control problem, which will include $SIRs$ given by Eq. (2.4.8), then it is not enough to receive information about the interfering transmit powers P_j. Each node must even gain knowledge about the interfering channels, such as the channel gain G_{ij}. Consequently, this method would require a truly unreasonably large amount of message passing. The authors of [11] do however argue that it is possible to reduce the amount of messages quite substantially by taking advantage of the problem structure. Specifically, they weave the parameters together into one variable; $P_{ij}^R = G_{ij}P_j$, which they refer to as the *effective received power*. A simulation of this problem with three transmitter-receiver pairs resulted in convergence after 100 - 200 iterations.

2.5 Interference function theory

The methods which will be described here have a quite specific application area, namely *power control* in wireless networks, which was shortly discussed in the previous section. It will however become clear that this theory has a close relation to the more general theory of F-Lipschitz optimization. We refer to [12] for an excellent and thorough description of the interference function theory.

2.5.1 Problem formulation

The objective is to make sure that each node in the network has an acceptable connection. This can be ensured by regulating the transmitted powers such that the interference caused by other nodes is limited. In other words, we have to make sure that the Signal-to-Interference ratio (SIR), is accept-

able. In a broad class of power controlled systems this can be described with constraints on the following form

$$\boldsymbol{p} \geq \mathbf{I}(\boldsymbol{p}) \tag{2.5.1}$$

where $\boldsymbol{p} = [p_1, \ldots, p_n]$, p_i denotes the transmitter power of node i, $\mathbf{I}(\boldsymbol{p}) = [I_1(\boldsymbol{p}), \ldots, I_n(\boldsymbol{p})]$ and $I_i(\boldsymbol{p})$ denotes the effective interference of other nodes which node i must overcome. In the following, Eq. (2.5.1) will be referred to as *interference constraints*.

The interference function $I_i(\boldsymbol{p})$ will have different forms according to how a receiver is assigned to each node. There may be a fixed assignment, in which a node always transmits to the same base station, or there may be a new assignment for each iteration. Either way, the interference will always depend on the SIR of node i at base station k which is here defined as

$$p_i \, \mu_{ki}(\boldsymbol{p}) = p_i \, \frac{h_{ki}}{\sum_{j \neq i} h_{kj} p_j + \sigma_k} \tag{2.5.2}$$

where h_{ki} denotes the gain from node i to base k and σ_k denotes the receiver noise power at base station k. Let us derive the interference function for a fixed assignment. If we let a_i be the assigned base of node i, then the SIR of node i at base a_i must fulfill $p_i \mu_{a_i i}(\boldsymbol{p}) \geq \gamma_i$ for some minimum required ratio $\gamma_i > 0$. Then, we can define the following interference constraint

$$p_i \geq I_i^{FA}(\boldsymbol{p}) = \frac{\gamma_i}{\mu_{a_i i}(\boldsymbol{p})} \tag{2.5.3}$$

For more examples of different kinds of interference functions, see [12].

2.5.2 Standard power control algoritm

The author of [12] presents us with the following iterative algorithm, which is referred to as the *power control algorithm*

$$\boldsymbol{p}(t+1) = \mathbf{I}(\boldsymbol{p}(t)) \tag{2.5.4}$$

It is proven that the algorithm (2.5.4) converges both synchronously and totally asynchronously given that $\mathbf{I}(\boldsymbol{p})$ satisfies certain properties. We here state the definition of a *standard* interference function [12]:

Definition 2.2. *Interference function* $\mathbf{I}(\boldsymbol{p})$ *is standard if for all* $\boldsymbol{p} \geq 0$ *the following properties are satisfied*

- *Positivity* $\mathbf{I}(\boldsymbol{p}) > 0$
- *Monotonicity* If $\boldsymbol{p} \geq \boldsymbol{p}'$, then $\mathbf{I}(\boldsymbol{p}) \geq \mathbf{I}(\boldsymbol{p}')$
- *Scalability* For all $\alpha > 1$, $\alpha \mathbf{I}(\boldsymbol{p}) > \mathbf{I}(\alpha \boldsymbol{p})$

When $\mathbf{I}(\boldsymbol{p})$ is a standard interference function, Eq. (2.5.4) is referred to as the *standard power control algorithm*. It is shown that, for any initial power vector $\boldsymbol{p}$, the sequence generated by this algorithm converges to a fixed point $\boldsymbol{p}^*$ which is unique. Also, we have that $\boldsymbol{p}^* \leq \boldsymbol{p}$ for any feasible vector $\boldsymbol{p}$, i.e. the point $\boldsymbol{p}^*$ is the solution of the interference constraints (2.5.1) which corresponds to the minimum total transmitted power. This is particularly desirable in wireless networks since the nodes are very likely to run on battery power.

The obvious way of distributing the power control algorithm is to let each node i update its transmit power according to

$$p_i(t+1) = I_i(\boldsymbol{p}(t)). \tag{2.5.5}$$

Considering the definition of the SIR given by Eq. (2.5.2), it seems that each node needs knowledge about all the channel gains and transmit powers of the other nodes. However, this can be simplified by letting

$$\mu_{ki}(\boldsymbol{p}) = \frac{h_{ki}}{R_k(\boldsymbol{p}) - h_{ki}p_i} \tag{2.5.6}$$

where $R_k(\boldsymbol{p}) = \sum_i h_{ki}p_i + \sigma_k$ denotes the total received power at base k. In other words, each node only needs knowledge of its own channel gain and the total received power at the base station.

As mentioned, we are presented with a result which proves asynchronous convergence, something which gives a good indication on the robustness of the algorithm. There are substantial benefits in using an asynchronous algorithm for this particular application. It allows for more flexibility in that some nodes can execute more iterations than others and hence perform power adjustments more frequently. Not to mention that it allows the nodes to perform their updates using outdated information. Wireless communication is not particularly reliable and delays or loss of information is likely to occur. See [13] for further analysis of this asynchronous distributed algorithm.

3 F-Lipschitz Optimization

Regarding the methods which were discussed in the previous chapter, it is clear that to accomplish distributed optimization is most often not a trivial matter. None of the methods are satisfactory for the general application of wireless networks, particularly when we consider the amount of information which must be transmitted between nodes. Until now, alternative methods for problems which are *non-convex* have also been lacking.

F-Lipschitz optimization can be performed distributively with only local computations and a limited amount of message passing. The realm of problem formulations which can be found to be of type F-Lipschitz includes both non-convex and convex problems. Not to mention that this type of problem seems to be particularly pervasive in wireless sensor networks. The distributed algorithm is based on contractive constraints, which makes it very simple and efficient. We will see that the difficult part is actually to verify that the problem is F-Lipschitz.

3.1 Notation

The notation $\mathbb{R}^+$ denotes the set of strictly positive valued real numbers. By $|\cdot|$ we denote the absolute value of a real number. For a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ we define the 1-norm $||\cdot||_1$ and the ∞-norm $||\cdot||_\infty$ as

$$||\mathbf{A}||_1 = \max_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}| \quad \text{and} \quad ||\mathbf{A}||_\infty = \max_{j=1}^{n} \sum_{i=1}^{n} |a_{ij}|$$

Note that we take the sum of all the elements on each *row* when calculating the 1-norm, while for the ∞-norm we sum each *column*. By $\boldsymbol{a} \preceq \boldsymbol{b}$ or $\boldsymbol{a} \succeq \boldsymbol{b}$ we denote the element-wise inequalities between the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. The notation $\mathbf{1}$ is used to denote the vector $(1,\ldots,1)^T$ whose dimension is clear from the context.

We will denote $\mathscr{D}$ as a set defining the bound constraints of the vector $\boldsymbol{x} \in \mathbb{R}^n$, i.e. $\mathscr{D} = [x_{1,\min}, x_{1,\max}] \times [x_{2,\min}, x_{2,\max}] \ldots [x_{n,\min}, x_{n,\max}] \in \mathbb{R}^n$, where $-\infty < x_{i,\min} < x_{i,\max} < \infty$, for $i = 1,\ldots,n$. The notation $[x_i]^{\mathscr{D}}$ is used to denote the orthogonal projection with respect to the Euclidian norm of the i-th component of the vector $\boldsymbol{x}$ onto the i-th component of the closed set $\mathscr{D}$. Specifically,

$$[x_i]^{\mathscr{D}} = \begin{cases} x_i & \text{if } x_i \in [x_{i,\min}, x_{i,\max}] \\ x_{i,\min} & \text{if } x_i < x_{i,\min} \\ x_{i,\max} & \text{if } x_i > x_{i,\max}. \end{cases}$$

As is custom, ∇ denotes the gradient operator. Given a vector function $\mathbf{F}(\boldsymbol{x})$: $\mathbb{R}^n \to \mathbb{R}^n$, we use the gradient matrix definition, $\nabla\mathbf{F}(\boldsymbol{x}) = [\nabla F_1(\boldsymbol{x}) \ldots \nabla F_n(\boldsymbol{x})]$, which is the transpose of the Jacobian matrix.

3.2 Background theory

It is important to note that the properties given here is a generalization in that we are only considering the 1-norm and ∞-norm. In other words, there are F-Lipschitz problems which will not be verified by these properties because the constraints are contractive according to some other norm. See [14] for more general properties.

3.2.1 Problem formulation

We here state Definition 3.1 [5] of an F-Lipschitz problem:

Definition 3.1 (**F-Lipschitz optimization**). *An F-Lipschitz optimization problem is defined as*

$$\begin{aligned}
\max_{\boldsymbol{x}} \quad & f_0(\boldsymbol{x}) && && \text{(3.1a)}\\
\text{s.t.} \quad & x_i \leq f_i(\boldsymbol{x}), && i = 1, \ldots, l && \text{(3.1b)}\\
& x_i = h_i(\boldsymbol{x}), && i = l+1, \ldots, n && \text{(3.1c)}\\
& \boldsymbol{x} \in \mathscr{D}, &&
\end{aligned}$$

where $\mathscr{D} \subset \mathbb{R}^n$ is a non-empty, convex and compact set, $l \leq n$, with objective and constraints being continuous differentiable functions such that

$$\begin{aligned}
& f_0(\boldsymbol{x}) : \mathscr{D} \to \mathbb{R}^m, && m \geq 1\\
& f_i(\boldsymbol{x}) : \mathscr{D} \to \mathbb{R}, && i = 1, \ldots, l\\
& h_i(\boldsymbol{x}) : \mathscr{D} \to \mathbb{R}, && i = l+1, \ldots, n
\end{aligned}$$

Let $\mathbf{f}(\boldsymbol{x}) = [f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_l(\boldsymbol{x})]^T$, $\mathbf{h}(\boldsymbol{x}) = [h_{l+1}(\boldsymbol{x}), h_{l+2}(\boldsymbol{x}), \ldots, h_n(\boldsymbol{x})]^T$, and $\mathbf{F}(\boldsymbol{x}) = [F_i(\boldsymbol{x})] = [\mathbf{f}(\boldsymbol{x})^T \ \mathbf{h}(\boldsymbol{x})^T]^T$. The following properties must be verified:

1.a $\nabla f_0(\boldsymbol{x}) \succ 0$, *i.e.* $f_0(\boldsymbol{x})$ *is strictly increasing,* (3.2a)

1.b $|\nabla \mathbf{F}(\boldsymbol{x})|_\infty < 1$, (3.2b)

and either

2.a $\nabla_j F_i(\boldsymbol{x}) \geq 0 \quad \forall i, j$, (3.2c)

or

3.a $f_0(\boldsymbol{x}) = c\mathbf{1}^T\boldsymbol{x}, \quad c \in \mathbb{R}^+$, (3.2d)

3.b $\nabla_j F_i(\boldsymbol{x}) \leq 0 \quad \forall i, j$, (3.2e)

3.c $|\nabla \mathbf{F}(\boldsymbol{x})|_1 < 1$, (3.2f)

or

4.a $f_0(\boldsymbol{x}) \in \mathbb{R}$, (3.2g)

$$4.b \quad |\nabla \mathbf{F}(\boldsymbol{x})|_1 < \frac{\delta}{\delta + \Delta}, \tag{3.2h}$$

$$\delta = \min_{i, \boldsymbol{x} \in \mathscr{D}} \nabla_i f_0(\boldsymbol{x}), \tag{3.2i}$$

$$\Delta = \max_{i, \boldsymbol{x} \in \mathscr{D}} \nabla_i f_0(\boldsymbol{x}). \tag{3.2j}$$

The properties (3.2a) - (3.2h) will in the following be referred to as the *qualifying properties* of an F-Lipschitz optimization problem. Note that it is only required that these properties apply for all $\boldsymbol{x} \in \mathscr{D}$, in other words we need only consider a specific interval of values for $\boldsymbol{x}$. Also note that $\nabla F(\boldsymbol{x})$ is here the transpose of the Jacobian, i.e. the summation of each column for the ∞-norm translates into the summation over all partial derivatives of $F_i(\boldsymbol{x})$.

Problem (3.1) can be applied in both centralized and distributed settings, the latter of course being the main focus here. As is common in wireless sensor networks, each node i is associated with its own decision variable x_i. Correspondingly, constraint i belongs to node i. In other words, there has to be just as many constraints as there are decision variables. Note that it is quite possible to have only inequality constraints, in which case $l = n$, or only equality constraints, which means that $l = 0$.

F-Lipschitz optimization allows for a multi-objective, which means that the objective function is allowed to be a vector in $\mathbb{R}^m$. Of course we can also have $m = 1$ which results in scalar optimization. Examples of objective functions are

$$f_0(\boldsymbol{x}) = \boldsymbol{x} \in \mathbb{R}^n$$
$$f_0(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x}, \qquad \boldsymbol{c} \in \mathbb{R}^n, \quad \boldsymbol{c} \succ 0.$$

Note however that in order to verify property (3.2a) it is often necessary to do a scalarization of the objective function. This is the case for the examples given here where the corresponding scalarized functions are

$$f_0(\boldsymbol{x}) = \sum_{i=1}^{n} x_i, \qquad \boldsymbol{x} \in \mathbb{R}^n$$
$$f_0(\boldsymbol{x}) = \sum_{i=1}^{n} c_i x_i, \quad \boldsymbol{c} \in \mathbb{R}^n, \quad \boldsymbol{c} \succ 0.$$

3.2.2 Distributed algorithm

If we can verify the qualifying properties for a given problem, then we know for certain that a unique solution exists and that it satisfies the constraints at the equality, see Theorem 3.3 [5]. The beauty of this is that we get the distributed algorithm directly. Since each node is associated with *one* local constraint, it is a natural choice of algorithm for updating the local variable. Convergence is certified as the constraint i is Lipschitz contractive with x_i^* as a fixed point. We here state Proposition 3.9 [5] defining the algorithm:

Proposition 3.2. *Let $\boldsymbol{x}(0) \in \mathbb{R}^n$ be an initial guess of the optimal solution to a feasible F-Lipschitz problem* (3.1)*. Let $\boldsymbol{x}^i(k) = [x_1(\tau_1^i(k)), x_2(\tau_2^i(k)), \ldots, x_n(\tau_n^i(k))]$ be the vector of decision variables available at node i at time $k \in \mathbb{N}_+$, where $\tau_j^i(k)$ is the delay with which the decision variable of node j reaches node i. Then the following iterative algorithm converges to the optimal solution:*

$$\begin{aligned} x_i(k+1) &= [f_i(\boldsymbol{x}^i(k))]^{\mathscr{D}} \quad i = 1, \ldots, l \\ x_i(k+1) &= [h_i(\boldsymbol{x}^i(k))]^{\mathscr{D}} \quad i = l+1, \ldots, n \end{aligned} \tag{3.3}$$

where $k \in \mathbb{N}_+$ is an integer associated to the iterations.

Since the optimization problem is to be solved in a wireless sensor network, we can expect the communication to be somewhat unreliable and delays and losses will most certainly occur. Observe that Proposition 3.2 accounts for this fact and the resulting algorithm is *asynchronous*. Compared to the Lagrangian decomposition methods, the amount of messages exchanged between nodes are substantially reduced. Here, each node only has to broadcast its own decision variable to the other nodes. Also, if $f_i(\boldsymbol{x})$ or $h_i(\boldsymbol{x})$ only depends on decision variables of the neighboring nodes, a not unlikely scenario, then the transmission of these variables will be fast and practical.

3.2.3 Problems in canonical form

We have defined an F-Lipschitz problem by the special form (3.1), which will in the following be referred to as the *standard form*. In general optimization literature it is common to define a problem on the *canonical form*, as follows

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & g_0(\boldsymbol{x}) && && \text{(3.4a)} \\ \text{s.t.} \quad & g_i(\boldsymbol{x}) \leq 0, && i = 1, \ldots, l && \text{(3.4b)} \\ & p_i(\boldsymbol{x}) = 0, && i = l+1, \ldots, n && \text{(3.4c)} \\ & \boldsymbol{x} \in \mathscr{D}, \end{aligned}$$

where

$$\begin{aligned} g_0(\boldsymbol{x}) &: \mathscr{D} \to \mathbb{R}^m, \quad m \leq n \\ g_i(\boldsymbol{x}) &: \mathscr{D} \to \mathbb{R}, \quad i = 1, \ldots, l \\ p_i(\boldsymbol{x}) &: \mathscr{D} \to \mathbb{R}, \quad i = l+1, \ldots, n \end{aligned}$$

Problem (3.4) can be converted into the standard form by the following transformations

$$\begin{aligned}
\max_{\boldsymbol{x}} \quad & f_0(\boldsymbol{x}) && && \text{(3.5a)}\\
\text{s.t.} \quad & x_i \leq f_i(\boldsymbol{x}), && i = 1, \ldots, l && \text{(3.5b)}\\
& x_i = h_i(\boldsymbol{x}), && i = l+1, \ldots, n && \text{(3.5c)}\\
& \boldsymbol{x} \in \mathscr{D}, &&
\end{aligned}$$

where

$$\begin{aligned}
f_0(\boldsymbol{x}) &= -g_0(\boldsymbol{x}), && && \text{(3.6)}\\
f_i(\boldsymbol{x}) &= x_i - \gamma_i g_i(\boldsymbol{x}), && i = 1, \ldots, l, && \text{(3.7)}\\
h_i(\boldsymbol{x}) &= x_i - \mu_i p_i(\boldsymbol{x}), && i = l+1, \ldots, n && \text{(3.8)}
\end{aligned}$$

with $\gamma_i > 0$, $i = 1, \ldots, l$, and $\mu_i \in \mathbb{R}$, $i = l+1, \ldots, n$. Problem (3.5) has the same optimal solution as problem (3.4) since, given that $\gamma_i > 0$ and $\mu_i \neq 0$, the constraints hold only if the constraints of (3.4) hold. It has been established what is required of the canonical form problem to be of type F-Lipschitz, namely when the problem (3.5) satisfies the qualifying properties (3.2a) - (3.2h). Similar as before, we let $\mathbf{G}(\boldsymbol{x}) = [g_1(\boldsymbol{x}), \ldots, g_l(\boldsymbol{x}), p_{l+1}(\boldsymbol{x}), \ldots,$
$p_n(\boldsymbol{x})]^T$ and here state Theorem 3.7 [5] defining the qualifying properties for the canonical form problem:

Theorem 3.3. *Consider the optimization problems* (3.4) *and* (3.5)*. Suppose that* $\forall\ \boldsymbol{x} \in \mathscr{D}$

$$\textit{1.a} \quad \nabla g_0(\boldsymbol{x}) \prec 0, \tag{3.9a}$$

$$\textit{1.b} \quad \nabla_i G_i(\boldsymbol{x}) < 0 \quad \forall i, \tag{3.9b}$$

and either

$$\textit{2.a} \quad \nabla_j G_i(\boldsymbol{x}) \leq 0 \quad \forall j \neq i, \tag{3.9c}$$

$$\textit{2.b} \quad \nabla_i G_i(\boldsymbol{x}) > \sum_{j \neq i} |\nabla_j G_i(\boldsymbol{x})| \quad \forall i, \tag{3.9d}$$

or

$$\textit{3.a} \quad g_0(\boldsymbol{x}) = -c\mathbf{1}^T\boldsymbol{x}, \quad c \in \mathbb{R}^+, \tag{3.9e}$$

$$\textit{3.b} \quad \nabla_j G_i(\boldsymbol{x}) \geq 0 \quad \forall j \neq i, \tag{3.9f}$$

$$\textit{3.c} \quad \nabla_i G_i(\boldsymbol{x}) > \sum_{j \neq i} |\nabla_i G_j(x)| \quad \forall i, \tag{3.9g}$$

or

$$\textit{4.a} \quad g_0(\boldsymbol{x}) \in \mathbb{R}, \tag{3.9h}$$

$$4.b \quad \frac{\delta}{\delta+\Delta}\nabla_i G_i(\boldsymbol{x}) > \sum_{j\neq i} |\nabla_i G_j(x)| \quad \forall i, \tag{3.9i}$$

where δ and Δ are defined in Eqs. (3.2i) *and* (3.2j)*. Then, problem* (3.5) *is F-Lipschitz.*

The proof of Theorem 3.3 states how to choose γ_i and μ_i such that the properties (3.9a) - (3.9i) implies contractive constraints. Namely, the following inequality must be fulfilled for $\boldsymbol{x} \in \mathscr{D}$

$$1 - \gamma_i \nabla_i g_i(\boldsymbol{x}) \geq 0 \tag{3.10}$$

and correspondingly for μ_i.

3.3 Relation to previous work

Comparing the F-Lipschitz theory to the interference function theory discussed in Section 2.5, there are obvious similarities. Both theories presents distributed asynchronous algorithms which are based on contractive constraints. In fact, it can be shown that the standard form problem (3.1) along with the qualifying properties (3.2) is a much more general case of the *standard* interference function as it is defined in Definition 2.2. Refer to [14] for the proof of this.

As opposed to interference function theory, geometric programs are not specifically designed to be optimized in a distributed fashion. However, it has been shown that this may be accomplished using the Lagrangian decomposition methods of Section 2.3. That said, the problem might turn out to be optimized more efficiently using F-Lipschitz theory. Particularly, there is a class of GPs that can be shown to fulfill the conditions of an F-Lipschitz problem, hence the fast distributed algorithm may be applied in stead. Because of the special structure of the GP, there are some particular conditions which can be checked to more easily verify if the problem is feasible or not, see [14] for details. A nice illustration of how F-Lipschitz optimization relates to other areas of optimization theory can be seen in Figure 4.

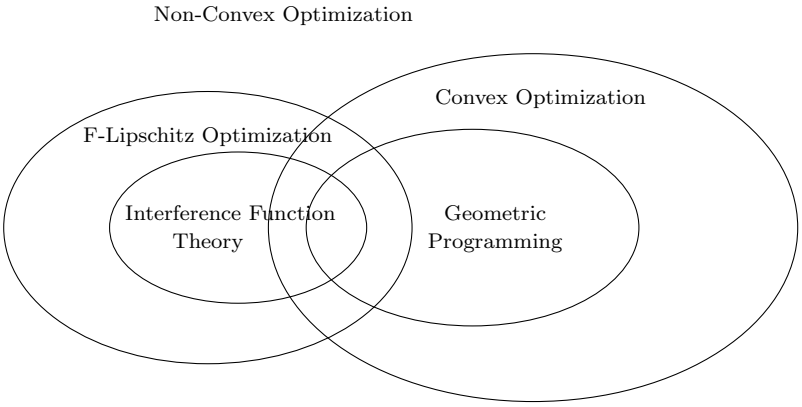

Figure 4: The relation of F-Lipschitz optimization theory compared to other areas of optimization.

We have yet to make a comparison with the Lagrangian decomposition methods, even though indications have been made that these methods require extensive message passing and is therefore a less efficient and less desirable alternative when F-Lipschitz optimization may be applied in stead. This argument has been more thoroughly backed up in [5]. It is shown that strong duality always applies to the F-lipschitz problem, hence both Lagrangian iterative methods for centralized optimization as well as decomposition methods may be applied. The conclusion is that the F-Lipschitz problem is always less computationally expensive to solve, be it in a centralized or in a distributed setup. However, the superiority is greater for the distributed case precisely because the F-Lipschitz algorithm requires such a small amount of information exchange.

4 Implementation Design

The work done here includes the implementation of a toolbox for MATLAB®, which focuses on F-Lipschitz optimization problems. Like the classical Optimization Toolbox™, it was desirable to provide both a graphical interface as well as inline functions. Also, inspiration was taken from *Disciplined convex programming* [15], developed by Grant, Boyd and Ye. This theory involves a framework which provides function libraries and rulesets such that the user can easily define problems which are known for certain to be convex.

That said, it was a goal to provide flexibility in how the optimization problem could be defined. The user should be able to investigate a problem with an arbitrary structure, as long as the requirements set by the problem definition (3.1) are fulfilled. At the same time, the toolbox should provide for special cases of the problem structure, like e.g. linear or quadratic constraints. Not to mention support for both the standard and the canonical problem definition. Much focus will be laid on the computation of the qualifying properties for different kinds of problem structures, as this is an issue which is yet to be thoroughly investigated.

4.1 Qualifying properties - computational complexity

It has been shown that the problem (3.1) can be solved with an effective and robust algorithm in a distributed setup. However, we must first determine that the problem is in fact F-Lipschitz by investigating the qualifying properties. Thus, it is interesting to determine how computationally heavy this operation actually is. We will consider the qualifying properties for both the standard form and the canonical form, as well as discuss possible simplifications when problem (3.1) or problem (3.4) has a certain structure. The discussion which follows is mainly based on a theoretical aspect. There will however be some occasional comments on how the theory translates into the programming environment, i.e. MATLAB.

4.1.1 Newton's method applied to unconstrained optimization

Considering the qualifying properties (3.2a) - (3.2h), we observe that most of them can be confirmed or rejected by finding the minimum or maximum of a function. E.g. property (3.2a), stating that $f_0(\boldsymbol{x})$ should be strictly increasing, can be verified by finding

$$s_i = \min_{\boldsymbol{x} \in \mathscr{D}} \nabla_i f_0(\boldsymbol{x}), \quad \forall i, \tag{4.1}$$

for each of the m objectives and checking that $s_i > 0$, $\forall i$. Observe that the problem is in fact constrained as we require that $x \in \mathscr{D}$. However, as these constitutes simple bound constraints, we can assume that problem (4.1) can be solved with the same computational complexity as an unconstrained optimization problem. The same conclusion can be drawn for the other properties where a minimum or maximum must be found.

MATLAB's Optimization Toolbox^{TM} provides us with a minimization function, *fmincon*, which will be applied in order to solve such problems as the one described above. Specifically, *fmincon* may be used with several algorithms, but the method called *active set* will be the one applied here. This is simply because it is one of the fastest algorithms and at the same time it is quite reliable. If it fails, the *interior point* algorithm is the back-up plan. *fmincon* supports both linear and nonlinear constraints, but we will only make use of the possibility to set bound constraints. It is the only minimization function which supports this.

Being the most effective first order algorithm, we consider the computational complexity of Newton's method for unconstrained optimization to help us on the way of finding the total complexity of the qualifying properties. Of course, this is not the algorithm which is actually used, but it is generally a difficult task to find an overall computational complexity for an optimization method. A nice convergence analysis has however been introduced for the well established Newton's method, [16] (Section 9.5), and the argument is that *fmincon* with only bound constraints should have a similar overall complexity. Newton's method using backtracking line search to choose the step size t is here analyzed. The following assumptions are made

- f is twice continuously differentiable.
- f is strongly convex with constant m, i.e. $\nabla^2 f(x) \succeq mI$.[1]
- The Hessian of f is Lipschitz continuous with constant L, i.e.

$$||\nabla^2 f(x) - \nabla^2 f(y)||_2 \leq L||x - y||_2, \quad \forall x, y \tag{4.2}$$

where $f(x)$ is the function we wish to minimize. Given these assumptions, it is proven that there exists an η and a γ, where $0 < \eta \leq m^2/L$ and $\gamma > 0$, such that the following hold

- If $||\nabla f(x^{(k)})||_2 \geq \eta$, then

$$f(x^{(k+1)}) - f(x^{(k)}) \leq -\gamma \tag{4.3}$$

- If $||\nabla f(x^{(k)})||_2 < \eta$, then the backtracking line search selects $t^{(k)} = 1$ and

$$\frac{L}{2m^2}||\nabla f(x^{(k+1)})||_2 \leq \left(\frac{L}{2m^2}||\nabla f(x^{(k)})||_2\right)^2 \tag{4.4}$$

We show here that if the second condition is satisfied at iteration k, i.e. $||\nabla f(x^{(k)})||_2 < \eta$, then it is also satisfied for all future iterates $l \geq k$. Using the inequality (4.4) and the fact that $\eta \leq m^2/L$ we derive the following result for iteration $k + 1$:

[1]Note that $\succeq$ here implies positive definiteness and not element-wise inequality as was the notation in Chapter 3.

$$||\nabla f(x^{(k+1)})||_2 \leq \frac{L}{2m^2}||\nabla f(x^{(k)})||_2^2 < \frac{L}{2m^2}\eta^2 \leq \frac{L}{2m^2}\frac{m^2}{L}\eta < \eta$$

The same result can be found for $k+2$, $k+3$, ..., hence the second condition is satisfied for all iterations $l \geq k$. This means that the iterations of Newton's method can be divided into two stages; the first for when the inequality, $||\nabla f(x)||_2 \geq \eta$, holds and the second when we have $||\nabla f(x)||_2 < \eta$. The first stage is referred to as the *damped Newton phase* as the step size t can be chosen to be smaller than one, while the second stage is called the *pure Newton phase* since $t = 1$ and the entire length of the Newton step is applied.

We can now estimate the total complexity by deriving an upper bound on the number of iterations for each of the two stages. Adopting the notation of [16], we let p^* denote the optimal value $f(x^*)$, where x^* is the optimal point. Considering the inequality (4.3), which is valid for the damped Newton phase, we observe that f decreases with at least γ for each step. An upper bound on the number of damped Newton steps is therefore given by

$$\frac{f(x^{(0)}) - p^*}{\gamma}$$

This is evident because a single step exceeding this bound would reduce f below the value of p^*, which is impossible.

To find an upper bound for the pure Newton phase is a bit more tricky. We consider the inequality (4.4) and apply it recursively for $l \geq k$

$$\begin{aligned}
\frac{L}{2m^2}||\nabla f(x^{(l)})||_2 &\leq \left(\frac{L}{2m^2}||\nabla f(x^{(l-1)})||_2\right)^2 \\
&\leq \left(\frac{L}{2m^2}||\nabla f(x^{(l-2)})||_2\right)^4 \\
&\vdots \\
&\leq \left(\frac{L}{2m^2}||\nabla f(x^{(k)})||_2\right)^{2^{l-k}} \\
&\leq \left(\frac{1}{2}\right)^{2^{l-k}}
\end{aligned}$$

where we again used that $\eta \leq m^2/L$ for the last inequality. Now, strong convexity implies that $f(x^{(l)}) - p^* \leq \frac{1}{2m}||\nabla f(x^{(l)})||_2^2$. See [16] (Section 9.1) for proof of this inequality. Finally, we can derive the following result

$$
\begin{aligned}
f(x^{(l)}) - p^* &\leq \frac{1}{2m}||\nabla f(x^{(l)})||_2^2 \\
&\leq \frac{2m^3}{L^2}\left(\frac{L}{2m^2}||\nabla f(x^{(k)})||_2\right)^{2^{l-k+1}} \\
&\leq \frac{2m^3}{L^2}\left(\frac{1}{2}\right)^{2^{l-k+1}}
\end{aligned}
$$

which indicates that convergence is very fast once the algorithm has entered this stage. Obviously, the exponentiated term will become very small just after a few iterations. This is called *quadratic convergence*. Letting ϵ denote the accuracy of the solution and $\epsilon_0 = 2m^3/L^2$, the number of pure Newton steps can be found as

$$
\begin{aligned}
\epsilon &= \epsilon_0\left(\frac{1}{2}\right)^{2^{l-k+1}} \\
\log_2\left(\frac{\epsilon}{\epsilon_0}\right) &= \log_2\left(\frac{1}{2}\right)\ 2^{l-k+1} \\
-\log_2\left(\frac{\epsilon}{\epsilon_0}\right) &= 2^{l-k+1} \\
l - k + 1 &= \log_2\log_2\left(\frac{\epsilon_0}{\epsilon}\right)
\end{aligned}
$$

Conclusively, the total number of iterations until $f(x) - p^* \leq \epsilon$ is bounded above by

$$
\frac{f(x^{(0)}) - p^*}{\gamma} + \log_2\log_2(\epsilon_0/\epsilon).
$$

The last term grows extremely slowly with the required accuracy ϵ, hence it can be considered as a constant for practical purposes. Letting $\log_2\log_2(\epsilon_0/\epsilon) = 6$ translates into an accuracy of about $\epsilon \approx 5 \cdot 10^{-20}\epsilon_0$. Now for the final conclusion; the number of iterations of Newton's method required to minimize f is bounded above by

$$
\frac{f(x^{(0)}) - p^*}{\gamma} + 6. \tag{4.5}
$$

Clearly, it is in order to discuss how we can find this γ and what it depends on. We introduce the variable M, which is implied by the assumption that f is strongly convex. It is shown that, if $\nabla^2 f(x) \succeq mI$, there also exists an $M > 0$ such that $\nabla^2 f(x) \preceq MI$, see [16] (Section 9.1). The proof of inequality (4.3) derives the following expression

$$
\gamma = \alpha\beta\eta^2\frac{m}{M^2}.
$$

The two constants α and β are parameters chosen for the backtracking line search, where $0 < \alpha < 0.5$ and $0 < \beta < 1$. It is also shown that η depends only on α, m and L. In other words; knowing the structure of f, we can find these parameters and calculate the estimate (4.5).

Naturally, we can't presume that our estimate is a particularly *tight* upper bound. First, we base the approximation on an extremely small error and secondly, since we can't know beforehand what p^* is, we must assume some range in which the solution lies. Though not particularly accurate and based on certain assumptions, these results gives an impression of what we can expect of Newton's method. The most important issue for our continuous discussion is that the number of iterations does not appear to be dependent on the problem size, i.e. the number of decision variables. In fact, it has been shown that the method has similar performance for problems in $\mathbb{R}^{10000}$ as for problems in $\mathbb{R}^{10}$.

4.1.2 Standard form

In this section we will discuss the complexity of verifying the qualifying properties (3.2a) - (3.2h) assuming that we have no knowledge about the problem structure. Of course, this seems quite unlikely for a practical application, but it is an interesting case for the toolbox if the user is to be allowed to define an arbitrarily structured problem. Also, it is useful for comparison when we later investigate the complexity for known structures.

We have learned that solving an unconstrained optimization problem using Newton's method will require a number of iterations which is not dependent on the problem size. For simplicity, we will therefore denote the amount of iterations required for an unconstrained minimization as a constant number s. Table 1 shows the complexity in O-notation for each of the properties. Here follows a short explanation on how these results were derived:

i) As previously explained, see problem (4.1); to check property (3.2a) (1.a), we must find the minimum of $\nabla_i f_0(\boldsymbol{x})$, $\forall i$, for each of the m objectives. In other words, it will take $O(mns)$ iterations.

ii) To check condition 1.b, we have to investigate if the sum of each column of $\nabla \mathbf{F}(\boldsymbol{x})$ is always less than one. To do this we define a help function

$$h_j(\boldsymbol{x}) = \sum_{i=1}^{n} |\nabla_i F_j(\boldsymbol{x})| \tag{4.6}$$

and solve the maximization problem

$$\max_{x \in \mathscr{D}} \; h_j(\boldsymbol{x}) \tag{4.7}$$

for all $j = 1, \ldots, n$. The maximum values must then all be less than one. As the summation in (4.6) takes n iterations and solving the problem (4.7) takes s iterations, the total complexity becomes $O(n^2 s)$, i.e.,

assuming that $h_j(\boldsymbol{x})$ is evaluated for each iteration of the maximization. These same arguments can also be used for deriving the complexity of properties 3.c and 4.b.

iii) To check condition 2.a, $\nabla_j F_i(\boldsymbol{x}) \geq 0$, $\forall i, j$, we have no choice but to minimize $\nabla_j F_i(\boldsymbol{x})$ for each $i = 1, \ldots, n$, for each $j = 1, \ldots, n$, investigating if the minimum is positive. This translates into $O(n^2 s)$ iterations. The same argument can be used for property 3.b.

iv) Obviously, to check condition 3.a would be trivial if we knew how the objective function looked like. Regardless, we will shortly discuss the complexity in the case that we don't know, for the sake of consistency. In order to determine that the objective function is linear, we can evaluate $\nabla f_0(\boldsymbol{x})$ at several different values of $\boldsymbol{x}$, checking that the resulting vector is always the same. Finally, we just need to investigate if all the elements of the vector are the same. As both the operations requires iterating through a vector with length n, the complexity is $O(n)$.

	Property	Complexity
1.a	$\nabla f_0 \prec 0$	$O(mns)$
1.b	$\lvert\nabla \mathbf{F}(\boldsymbol{x})\rvert_\infty < 1$	$O(n^2 s)$
2.a	$\nabla_j F_i(\boldsymbol{x}) \geq 0 \quad \forall i, j$	$O(n^2 s)$
3.a	$f_0(\boldsymbol{x}) = c\mathbf{1}^T \boldsymbol{x}, \quad c \in \mathbb{R}^+$	$O(n)$
3.b	$\nabla_j F_i(\boldsymbol{x}) \leq 0 \quad \forall i, j$	$O(n^2 s)$
3.c	$\lvert\nabla \mathbf{F}(\boldsymbol{x})\rvert_1 < 1$	$O(n^2 s)$
4.a	$f_0(\boldsymbol{x}) \in \mathbb{R}$	$O(1)$
4.b	$\lvert\nabla \mathbf{F}(\boldsymbol{x})\rvert_1 < \frac{\delta}{\delta+\Delta}$	$O(n^2 s)$

Table 1: Computational complexities of the qualifying properties.

The attentive reader perhaps noticed that the computation of δ and Δ, Eq. (3.2i) and (3.2j) respectively, which are required for condition 4.b, has yet to be considered. Actually, δ can easily be found in the process of checking condition 1.a. Since we are already minimizing $\nabla_i f_0(\boldsymbol{x})$ for each i, all that remains is to find the smallest of these minimums, which is then of course equal to δ. Correspondingly, Δ is found by maximizing $\nabla_i f_0(\boldsymbol{x})$ for each i, which translates into a computational complexity of $O(ns)$ since $m = 1$. This means that, since $O(ns)$ is obviously less than $O(n^2 s)$, the complexity of condition 4.b remains the same.

Since we have $m \leq n$, the overall dominant term is $O(n^2 s)$. We may also consider the matter of best- and worst-case scenarios in terms of amount of iterations before we can determine the one or the other. If the problem is F-Lipschitz, the best-case would obviously be that the problem fulfills condition 2.a, because this is the first alternative we check. In the worst-case,

the problem fulfills conditions 3.a, 3b and 3.c. One would think that the most amount of iterations would be required if the third and last alternative applied. This is however not the case since the failing of condition 3.c automatically implies that 4.b cannot be fulfilled. Thus, if it is the last alternative which applies, then at worst condition 3.b will fail, which means less or equal amounts of iterations as compared to the mentioned worst-case.

When applied in a practical implementation, we can consider additional means of time optimizing the algorithms. As an example, let us consider the discussion under ii). Defining the help function (4.6) in MATLAB, we can perform the summation using vector multiplication since $\nabla \mathbf{F}$ is provided as a matrix. In other words, $h_j(\boldsymbol{x})$ is calculated as $\mathbf{1}^T \times |\nabla F_j(\boldsymbol{x})|$, $\mathbf{1} \in \mathbb{R}^n$, which is a lot more efficient operation than summation through a simple loop. This becomes particularly important when the function is to be minimized, as *fmincon* will most likely evaluate it not only once, but several times per iteration.

4.1.3 Canonical form

For comparison, let us shortly summarize the computational complexity of verifying the qualifying properties (3.9a) - (3.9i) for the canonical form, which is shown in Table 2. Evidently, the overall complexity is the same for the canonical form as for the standard form, namely $O(n^2 s)$. An explanation for how the complexity was derived for some of the properties has already been given in the previous section.

The reasoning behind conditions 2.b, 3.c and 4.b is similar as for the corresponding conditions above. We create a help function

$$h_i(\boldsymbol{x}) = \nabla_i G_i(\boldsymbol{x}) - \sum_{j \neq i} |\nabla_j G_i(\boldsymbol{x})| \tag{4.8}$$

and find the minimum of $h_i(\boldsymbol{x})$ for all $i = 1, \ldots, n$, which should be larger than zero. Since the sum in (4.8) does not include i, we get $O(n(n-1)s) = O(n^2 s - ns)$ iterations. However, it is logical to simplify this to $O(n^2 s)$ since the quadratic term is dominant as n gets large.

	Property	Complexity
1.a	$\nabla g_0 \succ 0$	$O(mns)$
1.b	$\nabla_i G_i(\boldsymbol{x}) > 0 \quad \forall i$	$O(ns)$
2.a	$\nabla_j G_i(\boldsymbol{x}) \leq 0 \quad \forall j \neq i$	$O(n^2 s)$
2.b	$\nabla_i G_i(\boldsymbol{x}) > \sum_{j \neq i} \lvert\nabla_j G_i(\boldsymbol{x})\rvert \quad \forall i$	$O(n^2 s)$
3.a	$g_0(\boldsymbol{x}) = -c\mathbf{1}^T\boldsymbol{x}, \quad c \in \mathbb{R}^+$	$O(n)$
3.b	$\nabla_j G_i(\boldsymbol{x}) \geq 0 \quad \forall j \neq i$	$O(n^2 s)$
3.c	$\nabla_i G_i(\boldsymbol{x}) > \sum_{j \neq i} \lvert\nabla_i G_j(\boldsymbol{x})\rvert \quad \forall i$	$O(n^2 s)$
4.a	$g_0(\boldsymbol{x}) \in \mathbb{R}$	$O(1)$
4.b	$\frac{\delta}{\delta+\Delta}\nabla_i G_i(\boldsymbol{x}) > \sum_{j \neq i} \lvert\nabla_i G_j(\boldsymbol{x})\rvert \quad \forall i$	$O(n^2 s)$

Table 2: Computational complexities of the qualifying properties for the canonical form.

4.1.4 Linear or quadratic constraints

Now that we have derived the complexity for a problem which is entirely unknown, let us investigate the consequences if we make certain assumptions. Obviously, we stand the most to gain if we can avoid running n^2 optimization problems for nearly every property we investigate. We will see that this is made possible if the problem happens to have linear or quadratic constraints. Here, both the standard and the canonical form will be considered simultaneously as the consequences are roughly the same. See Table 3 for a summary of the computational complexity for both the standard and canonical form properties when the constraints are either linear or quadratic.

In the case of linear constraints, we can write $\mathbf{F}(\boldsymbol{x})$ compactly as

$$\mathbf{F}(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x} + \boldsymbol{b} \tag{4.9}$$

where $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{b} \in \mathbb{R}^n$ and $\nabla\mathbf{F}(\boldsymbol{x}) = \mathbf{A}$. Since the constraint gradient does not depend on $\boldsymbol{x}$, we can just ignore the bound constraints and it becomes extremely easy to check most of the properties. For example, to determine if $|\nabla\mathbf{F}(\boldsymbol{x})|_\infty < 1$, we only need to find the absolute sum of each column of $\mathbf{A}$ and check if it is less than one. Correspondingly, to check that $\nabla_i G_i(\boldsymbol{x}) > \sum_{j \neq i} |\nabla_j G_i(\boldsymbol{x})|$, $\forall i$, we only have to determine if each diagonal element of $\mathbf{A}$ is larger than the absolute sum of the other elements on the same column. This still requires $O(n^2)$ iterations, but we also have to keep in mind that the computations which are necessary per iteration are in this case minimal.

Now, for the quadratical constraints. We define each constraint $F_i(\boldsymbol{x})$ as

$$F_i(\boldsymbol{x}) = \boldsymbol{x}^T\mathbf{Q}_i\boldsymbol{x} + \boldsymbol{r}_i^T\boldsymbol{x} + c_i \tag{4.10}$$

where $\mathbf{Q}_i \in \mathbb{R}^{n\times n}$, $\boldsymbol{r}_i \in \mathbb{R}^n$ and $c_i \in \mathbb{R}$. This time we obviously have to account for the bound constraints as the gradient depends on $\boldsymbol{x}$. However, we do not have to consider the whole range of values. It is sufficient to investigate the properties at the extremes, namely at $\boldsymbol{x}_{\min}$ and $\boldsymbol{x}_{\max}$. The question is: for each term of x_i, which of $x_{i,\min}$ and $x_{i,\max}$ gives the minimum value and which gives the maximum value? Let us consider the partial derivative of Eq. (4.10)

$$\nabla_j F_i(\boldsymbol{x}) = \sum_{h=1}^{n} (Q_{i,j,h} + Q_{i,h,j}) x_h + r_{i,j} \tag{4.11}$$

Clearly, we will find the minimum value if we use $x_{h,\min}$ whenever $(Q_{i,j,h} + Q_{i,h,j})$ is positive and $x_{h,\max}$ when it is negative. Vice versa, of course, for the maximum value. So for example, if we are to determine if $\nabla_j F_i(\boldsymbol{x}) \geq 0$, $\forall i, j$, we have to iterate through each partial derivative of each $F_i(\boldsymbol{x})$ investigating each coefficient and evaluating, resulting in a total of n^3 iterations. It is, after all, quite logical since we have n constraints, each consisting of an $n \times n$ matrix.

On the other hand, if we want to determine if $|\nabla \mathbf{F}(\boldsymbol{x})|_\infty < 1$, we cannot use different inputs of $\boldsymbol{x}$ for each partial derivative, it must be the same for all of them. Since the ∞-norm involves the sum of all the partial derivatives of $F_i(\boldsymbol{x})$, the solution is to sum all the coefficients of a given x_h as

$$s_h = \sum_{j=1}^{n} (Q_{i,j,h} + Q_{i,h,j}) \tag{4.12}$$

Then, the polarity of s_h will determine which of $x_{h,\min}$ and $x_{h,\max}$ will contribute to the minimum value and which to the maximum value. s_h can otherwise be found as the sum of all the elements of the h-th column of $(\mathbf{Q}_i + \mathbf{Q}_i^T)$. By investigating (4.12) for $h = 1, \ldots, n$ we put together two vectors representing each of the extremes and denote them as $\boldsymbol{x}_{\text{lower}}$ and $\boldsymbol{x}_{\text{upper}}$. Finally, all we have to do is calculate

$$\begin{aligned} \boldsymbol{v}_{i,\min} &= |(\mathbf{Q}_i + \mathbf{Q}_i^T)\boldsymbol{x}_{\text{lower}} + \boldsymbol{r}_i| \\ \boldsymbol{v}_{i,\max} &= |(\mathbf{Q}_i + \mathbf{Q}_i^T)\boldsymbol{x}_{\text{upper}} + \boldsymbol{r}_i| \end{aligned} \tag{4.13}$$

for all $i = 1, \ldots, n$, where the j-th element of either $\boldsymbol{v}_{i,\min}$ or of $\boldsymbol{v}_{i,\max}$ will represent the largest *absolute* value of $\nabla_j F_i(\boldsymbol{x})$. Then the sum of all the elements of each of the two vectors should separately be less than one. This entire operation has complexity $O(n^2)$, i.e. the same as for the linear constraints.

So, is there any way that we can simplify this? Yes, if we know that the matrix $\mathbf{Q}_i$ is either positive or negative, then we can simply replace $\boldsymbol{x}_{\text{lower}}$ and $\boldsymbol{x}_{\text{upper}}$ in (4.13) with $\boldsymbol{x}_{\min}$ and $\boldsymbol{x}_{\max}$ respectively, or vice versa for the

latter case. What is even better, though, is problems where $\boldsymbol{x}$ is bounded to only negative or only positive values, which is in fact quite common. Then $\mathbf{Q}_i$ can contain whatever and we can still use $\boldsymbol{x}_{\min}$ and $\boldsymbol{x}_{\max}$ as the extreme vectors. Consequently, the complexity of determining $\nabla_j F_i(\boldsymbol{x}) \geq 0$ is reduced to $O(n^2)$ since we do not need to iterate through every element of each $\mathbf{Q}_i$. The same conclusions can be drawn for problems with any kind of $\mathbf{Q}_i$ and any kind of bounds as long as the minimizing and maximizing input vectors for each partial derivate are known. It is, after all, not unlikely as the problem itself is known to the one who defines it.

Finally, it is worth mentioning that if we in the worst case need to do the n^3 iterations for condition 2.a, then at least it will not be necessary for condition 3.b as the resulting vectors may be stored. All of these conclusions can also be made for the canonical form properties. Note that determining if $\nabla_i G_i(\boldsymbol{x}) > 0$, $\forall i$, must take $O(n^2)$ by the same logic.

	Standard	Canonical
1.a	$O(mns)$	$O(mns)$
1.b	$O(n^2)$	$O(n), O(n^2)$
2.a	$O(n^2), O(n^3)$	$O(n^2), O(n^3)$
2.b	-	$O(n^2)$
3.a	$O(n)$	$O(n)$
3.b	$O(n^2)$	$O(n^2)$
3.c	$O(n^2)$	$O(n^2)$
4.a	$O(1)$	$O(1)$
4.b	$O(n^2)$	$O(n^2)$

Table 3: Computational complexities of the qualifying properties for both the standard and canonical form when the constraints are either linear or quadratic. Where there are two options, only the first applies to linear constraints, while for quadratic constraints the first is best-case and the second is worst-case.

The reasoning behind the n^3 iterations comes down to how the constraints are defined. Obviously, it might be that many of the elements of $\mathbf{Q}_i$ are actually equal to zero and so we could have spared us a lot of iterations. We can now draw an important conclusion. The key as to why it is simple to investigate the properties for quadratic constraints is the knowledge that the gradient is monotonic. In other words, if we know that the constraint gradient, $\nabla\mathbf{F}$ or $\nabla\mathbf{G}$, is increasing or decreasing over $\boldsymbol{x} \in \mathscr{D}$, then we also know that its maximum and minimum values are at the extremes, $\boldsymbol{x}_{\text{lower}}$ and $\boldsymbol{x}_{\text{upper}}$, which may be different for each partial derivative of $F_i(\boldsymbol{x})$.

Then, it is just a matter of determining if the appropriate inequality holds at these values and we automatically know that it holds for all $\boldsymbol{x} \in \mathscr{D}$. Most often, one of these extreme vectors can be set equal to $\boldsymbol{x}_{\min}$ and the other to $\boldsymbol{x}_{\max}$. Otherwise they are known as combinations of the elements of $\boldsymbol{x}_{\min}$ and $\boldsymbol{x}_{\max}$ which are evident from the problem structure. To summarize, the claim is that a problem which has monotonic constraint gradients may be verified F-Lipschitz with the same computational complexity as problems with linear constrains, see Table 3. Obviously, there are more than just quadratic constraints which falls in this category. Other examples may be cubic or exponential constraints.

4.1.5 Further improvements

As mentioned at the beginning of this discussion in Section 4.1.2, it seems unlikely that the objective function is completely unknown in the case of practical applications. So, let us discuss how we can simplify the computational complexity if we assume some knowledge about the objective. Perhaps we already know that the objective is strictly increasing, or strictly decreasing in the case of the canonical form. Then condition 1.a is verified in constant time. However, if it becomes necessary to check condition 4.b, then we are back to square one as we have to run the same calculations anyway in order to find δ. If we on the other hand know that the gradient of the objective is increasing or decreasing, we can determine condition 1.a (and find δ and Δ) in $O(mn)$.

Considering the constraints, we have already discussed the benefits of knowing that the gradient is increasing or decreasing. Also, it is useful to know if the constraints themselves are all increasing or all decreasing as conditions 2.a and 3.b can be determined in constant time. Not to mention that the vectors $\boldsymbol{x}_{\text{lower}}$ and $\boldsymbol{x}_{\text{upper}}$ can then be found very easily. For the canonical case it really only makes sense to have increasing constraints as condition 1.b demands that all diagonal elements $\nabla \mathbf{G}(\boldsymbol{x})$ have to be larger than zero.

4.2 Library of objective functions

In the spirit of Disciplined convex programming [15], it was desirable to provide the user with some possible objective functions for an F-Lipschitz problem. The idea is that if a problem is defined using this library, we should only be able to enter an objective function which is valid, i.e. strictly increasing for the standard form and strictly decreasing for the canonical form. An obvious reasoning here is that the objective function will be clearly defined, as compared to an arbitrary function input. Then we can avoid the possibly expensive *fmincon* by using our knowledge of the function's structure.

Some of these functions will be presented in the following. We will state the requirements for a function to be valid as well as comment on how δ and Δ can be found. For simplicity, the discussion will mostly revolve around

strictly increasing functions, which will be referred to as *valid standard functions*. Also note that the library will only allow for scalar functions, i.e. we always have $m = 1$.

Though it is trivial, the linear function $f_0(\boldsymbol{x}) = \boldsymbol{a}^T\boldsymbol{x}$, $\boldsymbol{a} \in \mathbb{R}^n$, is not uncommon as an objective. Clearly, this is a valid standard function only if $\boldsymbol{a} \succ 0$ and correspondingly $\boldsymbol{a} \prec 0$ for the canonical form. We may find δ and Δ simply as

$$\delta = \min_i \{a_i\} \quad \text{and} \quad \Delta = \max_i \{a_i\}.$$

The simple linear function can easily be expanded to something perhaps more interesting as

$$f_0(\boldsymbol{x}) = (\boldsymbol{a}^T\boldsymbol{x})^k, \tag{4.14}$$

where $k \in \mathbb{R}$. If k is positive, the function is valid standard only if $\boldsymbol{a} \succ 0$ and $\boldsymbol{x} \succ 0$, $\forall \boldsymbol{x} \in \mathscr{D}$. Correspondingly, if k is negative, then we must have $\boldsymbol{a} \prec 0$ and $\boldsymbol{x} \prec 0$. Notice that the base $(\boldsymbol{ax})$ is always positive.

Now the question is, how do we find δ and Δ? Or more specifically, when is $\nabla_i f_0(\boldsymbol{x}) = a_i k(\boldsymbol{ax})^{k-1}$ at its maximum and minimum over all i? Like for the linear function, δ involves the smallest *absolute* coefficient, $a_{\min}$, of $\boldsymbol{a}$ and Δ the largest, $a_{\max}$. Then, the answer to the question is

$$\delta = \begin{cases} a_{\min} k(\boldsymbol{a}^T\boldsymbol{x}_{\max})^{k-1} & \text{if } 0 < k < 1, \\ a_{\min} k(\boldsymbol{a}^T\boldsymbol{x}_{\min})^{k-1} & \text{if } k > 1 \text{ or } k < 0, \end{cases} \tag{4.15}$$

and

$$\Delta = \begin{cases} a_{\max} k(\boldsymbol{a}^T\boldsymbol{x}_{\min})^{k-1} & \text{if } 0 < k < 1, \\ a_{\max} k(\boldsymbol{a}^T\boldsymbol{x}_{\max})^{k-1} & \text{if } k > 1 \text{ or } k < 0. \end{cases} \tag{4.16}$$

We consider only δ and justify the results as follows. When $k < 0$, $\boldsymbol{x}$ is bounded to only negative numbers. Since the exponent is negative, we actually need the base to be as large as possible, which we acquire for $\boldsymbol{x}_{\min}$ as this will be the largest *absolute* value. The reasoning behind $k > 1$ should be obvious. When $0 < k < 1$, $\boldsymbol{x}$ is bounded to only positive numbers. However, the gradient will have a negative exponent. Thus, we again need the base to be as large as possible which we now acquire for $\boldsymbol{x}_{\max}$. When $k = 1$ we are of course back to the linear function.

The third type of objective is referred to as a *simple quadratic* function. This because it can only contain quadratic terms on the form x_i^2 and not $x_i x_j$. It is defined as follows

$$f_0(\boldsymbol{x}) = a_1 x_1^{b_1} + a_2 x_2^{b_2} \ldots + a_n x_n^{b_n} \tag{4.17}$$

where $b_i = \{1, 2\}$, $i = 1, \ldots, n$. In order to have a valid standard function, all coefficients a_i associated with a linear term must be positive. The coefficients associated with a quadratic term can be both negative and positive. However, the corresponding x_i must then be bounded to only negative numbers and only positive numbers respectively. Naturally, all elements of $\boldsymbol{a}$ must also be non-zero. δ and Δ can be found as follows

$$\delta = \min_i \begin{cases} a_i & \text{if } b_i = 1 \\ 2a_i x_{i,\min} & \text{if } b_i = 2 \text{ and } a_i > 0 \\ 2a_i x_{i,\max} & \text{if } b_i = 2 \text{ and } a_i < 0 \end{cases} \tag{4.18}$$

and

$$\Delta = \max_i \begin{cases} a_i & \text{if } b_i = 1 \\ 2a_i x_{i,\max} & \text{if } b_i = 2 \text{ and } a_i > 0 \\ 2a_i x_{i,\min} & \text{if } b_i = 2 \text{ and } a_i < 0 \end{cases} \tag{4.19}$$

The last type of function we will discuss is found from yet an exponent expansion. Namely, we define the following objective function

$$f_0(\boldsymbol{x}) = (a_1 x_1^{b_1} + a_2 x_2^{b_2} \ldots + a_n x_n^{b_n})^k \tag{4.20}$$

which is clearly based on Eq. (4.17). Now, it becomes a bit more tricky. Let us consider the gradient

$$\nabla_i f_0(\boldsymbol{x}) = \begin{cases} a_i k (a_1 x_1^{b_1} + a_2 x_2^{b_2} \ldots + a_n x_n^{b_n})^{k-1} & \text{if } b_i = 1 \\ 2a_i x_i k (a_1 x_1^{b_1} + a_2 x_2^{b_2} \ldots + a_n x_n^{b_n})^{k-1} & \text{if } b_i = 2 \end{cases} \tag{4.21}$$

where we now assume that $k \neq 1$ to avoid repeating any results. To be certain that we always have $\nabla f_0(\boldsymbol{x}) \succ 0$, we must restrict the coefficients and the values of $\boldsymbol{x}$ in such a way that the base is always positive. In other words; all the coefficients a_i associated with a quadratic term must be positive, while the linear term coefficients must have the same polarity as the associated decision variable. If $k < 0$, then clearly the linear term a_i must be negative and the corresponding x_i must be bounded negative. The quadratic x_i must be bounded to only negative values as well. Not surprising, the opposite applies for $k > 0$.

For calculating δ and Δ, we first need to find the smallest and the largest possible coefficient of Eq. (4.21), which are denoted $a_{c,\min}$ and $a_{c,\max}$. These can be found from the following equations

$$a_{c,\min} = \min_i \begin{cases} a_i k & \text{if } b_i = 1 \\ 2a_i x_{i,\min} k & \text{if } b_i = 2 \text{ and } k > 0 \\ 2a_i x_{i,\max} k & \text{if } b_i = 2 \text{ and } k < 0 \end{cases} \tag{4.22}$$

and

$$a_{c,\max} = \max_i \begin{cases} a_i k & \text{if } b_i = 1 \\ 2a_i x_{i,\max} k & \text{if } b_i = 2 \text{ and } k > 0 \\ 2a_i x_{i,\min} k & \text{if } b_i = 2 \text{ and } k < 0 \end{cases} \tag{4.23}$$

Secondly, we have to find the smallest and the largest possible value of the base and exponent of Eq. (4.21). Namely, we define $f_{g,\min}$ and $f_{g,\max}$ as

$$f_{g,\min} = \sum_i \begin{cases} a_i x_{i,\max} & \text{if } b_i = 1 \text{ and } 0 < k < 1 \\ a_i x_{i,\min} & \text{if } b_i = 1 \text{ and } k > 1 \text{ or } k < 0 \\ a_i x_{i,\max}^2 & \text{if } b_i = 2 \text{ and } 0 < k < 1 \\ a_i x_{i,\min}^2 & \text{if } b_i = 2 \text{ and } k > 1 \text{ or } k < 0 \end{cases} \tag{4.24}$$

and

$$f_{g,\max} = \sum_i \begin{cases} a_i x_{i,\min} & \text{if } b_i = 1 \text{ and } 0 < k < 1 \\ a_i x_{i,\max} & \text{if } b_i = 1 \text{ and } k > 1 \text{ or } k < 0 \\ a_i x_{i,\min}^2 & \text{if } b_i = 2 \text{ and } 0 < k < 1 \\ a_i x_{i,\max}^2 & \text{if } b_i = 2 \text{ and } k > 1 \text{ or } k < 0 \end{cases} \tag{4.25}$$

which can be justified using similar arguments as those given for Eq. (4.15) and (4.16). Finally, δ and Δ is simply found as

$$\delta = a_{c,\min}(f_{g,\min})^{k-1} \quad \text{and} \quad \Delta = a_{c,\max}(f_{g,\max})^{k-1}.$$

Clearly, all of these functions has gradients which are *monotonic*, since the maximum and minimum of a gradient can be found using the extreme values, $\boldsymbol{x}_{\min}$ and $\boldsymbol{x}_{\max}$. As was also concluded in Section 4.1.4, it seems that monotonicity is the key behind avoiding *fmincon*. In making this connection, notice that by introducing these types of possible objective functions, we have at the same time introduced new classes of possible constraints additional to the linear and quadratic which was previously discussed. After all, finding δ or Δ involves similar operations compared to what is required for verifying the qualifying properties of the constraints. That said, the definition of these new types of constraints may in general be more relaxed, as there is no requirement on them being *strictly* increasing, or strictly decreasing for that matter.

4.3 Solvers

Naturally, support for testing the distributed algorithm (3.3) must be implemented in the toolbox. It is very straightforward, particularly when the problem is on standard form. The canonical form algorithm is however more interesting, namely because it is possible to increase its efficiency. As previously stated, it is necessary to choose values for γ_i and μ_i such that the inequality (3.10) is fulfilled. For simplicity, the assembled vector of these values will be referred to as γ in the following. In stead of calculating a constant vector γ which ensures the inequality for all $\boldsymbol{x} \in \mathscr{D}$, it is possible to calculate new values for each iteration of the algorithm. This way, the vector γ becomes adaptive, ensuring that the inequality holds for the current iterate without being unnecessarily small. The result is increased efficiency as smaller values of γ implies smaller steps and hence a slower convergence. See Algorithm 5 for the pseudocode of this distributed algorithm.

Algorithm 5 Distributed algorithm with adaptive γ

Require: $\boldsymbol{x}_0 \in \mathscr{D}$

loop

1. Evaluate $\gamma_{i,k} = 1/\nabla_i g_i(\boldsymbol{x}_k)$.
2. Update variable
$$x_{i,k+1} := [x_{i,k} - \gamma_{i,k} g_i(\boldsymbol{x}_k)]^{\mathscr{D}}$$
3. Broadcast $x_{i,k+1}$ to other nodes.
4. Receive $x_{j,k+1}$, $\forall j \neq i$.

end loop

Of course, the the actual algorithm will not really be distributed and there will be no message passing, just variable assignment. Hence, no information will get lost and the algorithm will be executed synchronously as *all* of the the decision variables will be updated for each iteration. Consequently, the results generated by this algorithm should be considered as best-case. Unfortunately, there are no other readily available distributed algorithms which can be used for comparison. It would have been very useful to have a solver which applies the Lagrangian decomposition methods discussed in Section 2.3, but these are solemnly dependent on the problem structure, hence it is a challenging task to implement a general solution. At least, there are a few more options when it comes to computing the problem in a centralized fashion.

The centralized F-Lipschitz algorithm exploits that the solution lies at the equality of the constraints. In other words, it shall solve a system of equations which can be both linear and nonlinear. MATLAB provides this type of solver through the method *fsolve*. It needs a function $\mathbf{F}(\boldsymbol{x})$, which takes a vector $\boldsymbol{x}$ and returns a vector, as well as a starting point. So, the constraint function is used as input and *fsolve* will return a solution which satisfies $\mathbf{F}(\boldsymbol{x}) = 0$. The method actually has three types of algorithms to choose from, but here the default (trust region dogleg) will consequently be used, as this is the only algorithm which is specially designed to solve

nonlinear equations. Unfortunately, *fsolve* does not take bound constraints, hence it is not able to do the projection on $\mathscr{D}$ and the returned solution might very well be outside of bounds. Despite this, the algorithm will give a good indication on the performance of a centralized algorithm.

Finally, a third solver is introduced whose performance can be compared to that of the centralized F-Lipschitz solver. The previously mentioned function, *fmincon*, implements the well explored theory of Lagrangian methods, which one might say is the centralized counterpart to the decomposition methods. Linear equalities and inequalities may be defined through matrix inputs, while nonlinear constraints are defined through a function. As has already been mentioned, *fmincon* also takes upper and lower bounds on the decision variables. The method provides four algorithms, but the trust region reflective is here excluded for practical reasons as it requires specific combinations of inputs. MATLAB recommends trying the interior point algorithm first. Then, to potentially obtain more speed, one may try the SQP next and the active set last.

Both *fmincon* and *fsolve* returns information on the algorithm's performance. There is of course the amount of steps of the algorithm before convergence was reached, but also the total number of function evaluations. Very often, an algorithm will need to do certain computations before it is able to take a step, i.e. iterations within the iterate sort of speak. This is reflected by the number of function evaluations, which is the sum of *all* the internal iterations. In that sense, this number may be considered as the *actual* number of iterations which was necessary to achieve convergence.

5 F-Lipschitz Optimization Toolbox

Now that the ground work has been laid and the theoretical aspects have been considered, it is time to investigate how it all works in practice. Here follows a user guide of the F-Lipschitz Optimization Toolbox including examples ranging from the simple to the more advanced. Figure 5 shows an image of the toolbox. We will look at one panel at a time and explain the functionality[2] step-by-step by using examples. This chapter is only focused on the graphical interface of the toolbox, for the documentation of the inline functions, see Appendix A.

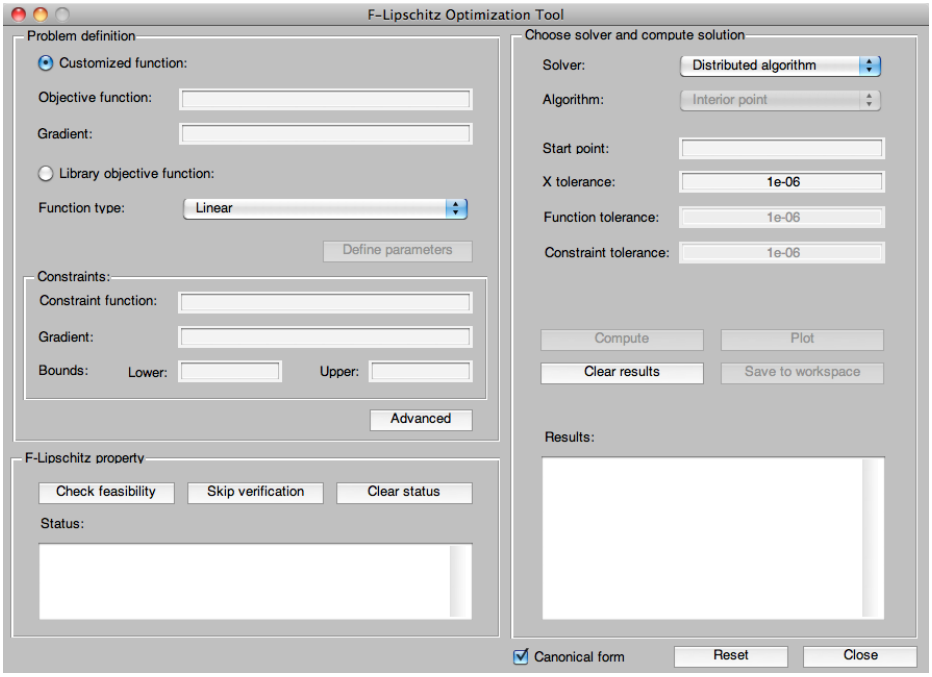

Figure 5: The F-Lipschitz Optimization Toolbox.

5.1 Simple example in $\mathbb{R}^2$

Consider the following example

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & (x_1^2 + x_2^2)^{-1} \\ \text{s.t.} \quad & x_1 - 0.5x_2 - 1 \leq 0 \\ & -x_1 + 2x_2 \leq 0 \\ & 0 < x_1, x_2 \leq 10 \end{aligned} \tag{5.1}$$

which is obviously on canonical form. In general, we will need to declare our objective and constraints in separate *m-file functions*. So we define the following four files:

[2]For simplicity, the discussions are devoid of the SQP algorithm as it failed for every problem.

```
function g = g_0(x)
    g = 1/(x(1)^2 + x(2)^2);
end

function g = nabla_g_0(x)
    g_ = zeros(1,2);
    g_(1,1) = -2*x(1)/(x(1)^2 + x(2)^2)^2;
    g_(1,2) = -2*x(2)/(x(1)^2 + x(2)^2)^2;
    g = g_;
end

function g = G(x)
    g_ = zeros(2,1);
    g_(1,1) = x(1) - 0.5*x(2) - 1;
    g_(2,1) = -x(1) + 2*x(2);
    g = g_;
end

function g = nabla_G(~)
    g_ = zeros(2,2);
    g_(1,1) = 1;
    g_(1,2) = -0.5;
    g_(2,1) = -1;
    g_(2,2) = 2;
    g = g_;
end
```

These should be named *g_0.m*, *nabla_g_0.m*, *G.m* and *nabla_G.m* respectively. Notice that the objective has dimension $m \times 1$, where $m = 1$ in this case, while the gradient, $\nabla g_0(\boldsymbol{x})$, has dimension $m \times n$. Correspondingly, the constraint function should return an $n \times 1$ vector and the gradient the $n \times n$ Jacobian matrix. Now we can enter the problem into the toolbox as shown in Figure 6. In stead of writing the bound vectors directly, which obviously is less practical when the amount of decision variables is large, it is possible to save them in the workspace and load them from there. E.g. if we have a variable $ub = [10, 10]$ in the workspace, then we just enter ub in the upper bound field of the toolbox.

Notice that the radio button for *Customized function* should be selected and not *Library objective function*. The latter option involves what was discussed in Section 4.2, which will be regarded later for this same problem. Now, if we wish, we can move on to checking the qualifying properties in order to determine if this is a feasible F-Lipschitz problem or not. See Figure 7 for the F-Lipschitz property panel. Before pressing the *Check feasibility* button, make sure that the checkbox named *Canonical form* located at the bottom of the window is checked. Not surprising, this is to let the toolbox know that we have defined our problem on the canonical form and not on the standard F-Lipschitz form.

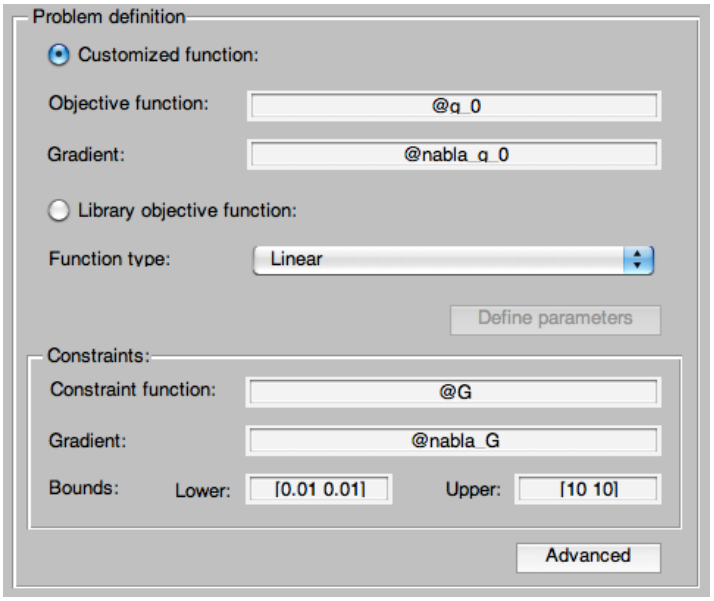

Figure 6: Problem definition of simple example.

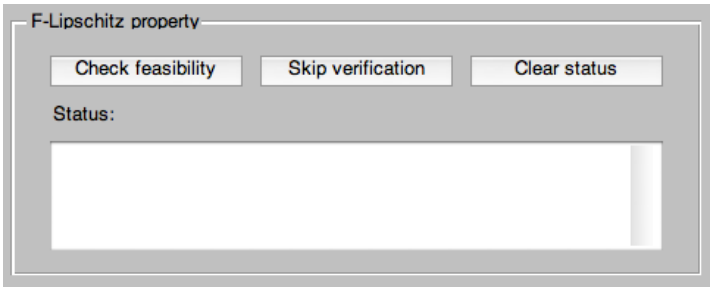

Figure 7: Verification of the qualifying properties of an F-Lipschitz problem.

If there is something wrong with the problem definition, like the dimensions of the constraints does not match or the m-file functions are not on the workspace path, there will be an error message in the status window. If everything is correctly defined, the status window will contain the following message:

```
-The objective function is strictly decreasing.
-The diagonal of the constraint gradient is strictly positive.

Alternatives:
1. -The off-diagonal elements of the constraint gradient are all
negative.
   -The diagonal of the constraint gradient is dominant.

The problem is F-Lipschitz.
Unconstrained fmincon used 10 iterations and 30 function
```

```
evaluations in total.
Total amount of iterations: 10
```

Observe that the number of iterations is quite small, but the amount of function evaluations is substantial. This is because, as previously stated, we don't apply our knowledge of the problem structure, thus *fmincon* must investigate the problem for us. Later on we will see that it is possible to reduce the amount of iterations substantially. Since the problem is feasible, the *Compute* button shown in Figure 8 is now enabled and the lower bound vector is automatically copied over to the start point field. Note that, also here, a workspace variable can be used in stead of writing the vector directly. If we want to skip the calculations of the qualifying properties and go straight to computing the solution, it is simply a matter of pressing the *Skip verification* button in Figure 7. Then, the toolbox will check all the inputs to see if the problem is correctly defined before enabling the *Compute* button.

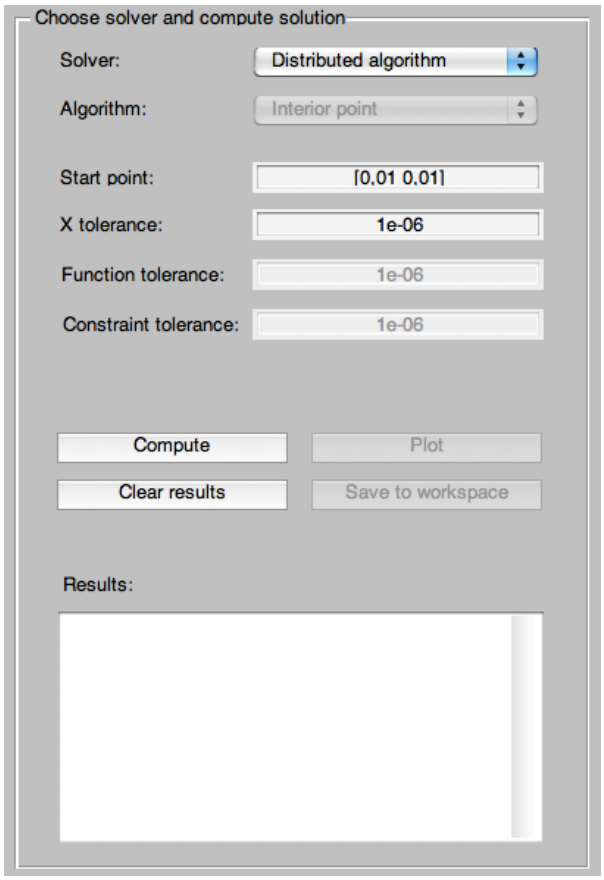

Figure 8: Computation of the optimal solution for the simple example.

The distributed algorithm is the default choice of solver. As can be seen, the *Algorithm* popup menu, the *Function tolerance* field and the *Constraint tolerance* field are disabled. All of these are enabled for the Lagrangian

solver, i.e. *fmincon*, while the function tolerance is enabled for the centralized algorithm, i.e. *fsolve*. Keep in mind, though, that "function" means the constraint function in the latter case and not the objective function. Finally, the distributed algorithm is executed by pressing the *Compute* button and the following message appears in the result window:

```
The algorithm stopped after 21 iterations.
There were 21 function evaluations.

The solution values are
x(1) = 1.333
x(2) = 0.667
```

The *Plot* and *Save to workspace* buttons now also becomes enabled. Pressing the first button, a new window pops up plotting the iterations versus the decision variables. In other words, the figure illustrates the algorithm's progression. The resulting plot is shown in Figure 9. Note however that the lowermost line was changed to dashed and the addition of the legend was done manually using the figure tools. We see that the decision variables change fast in the beginning, but then it takes a long time to reach the specified accuracy. The bright side is that we can reduce the amount of iterations quite substantially by reducing the required accuracy. For this example, reducing the accuracy to $1 \cdot 10^{-3}$ almost halves the amount of iterations to 11. The printed solution is changed in $x_2 = 0.666$, while x_1 remains the same.

Let us also try the centralized algorithm, i.e. *fsolve*. The same solution is returned after only 3 iterations and 9 function evaluations. Clearly, F-Lipschitz optimization is an important new theory not only for distributed settings, but for centralized optimization as well. Trying the last solver, the Lagrangian, we may choose between several algorithms, the interior point being the default. Pressing the *Compute* button, a popup window appears asking for the indexes of the constraint function which are to be equalities. This has not yet been defined as the information is actually unimportant for the F-Lipschitz theory. Since all the other constraints will be inequalities by default, we simply press *OK* without entering anything as we only have inequalities in this case. The interior point algorithm needs 11 iterations and 33 function evaluations to find the same solution as before. Clearly, *fsolve* is a lot more efficient algorithm, but if we also include the function evaluations from the verifying operation, we get a total of 39 evaluations versus the interior point method's 33. The active set algorithm is however the most efficient, using only 2 iterations and 6 function evaluations.

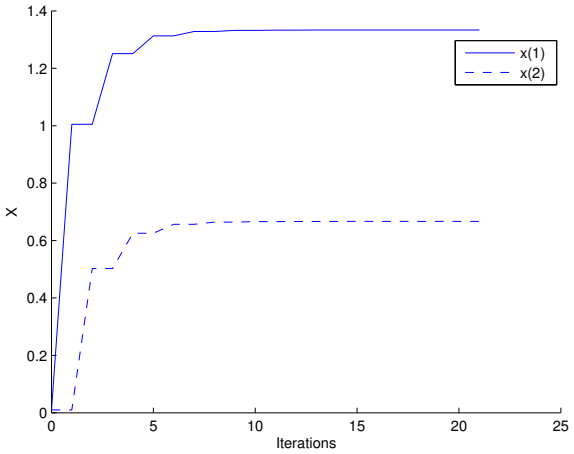

Figure 9: Iterations versus $\boldsymbol{x}$ for the distributed algorithm.

Finally, it is also possible to save the solution vector to the workspace by pressing the *Save to workspace* button. A new window will pop up asking for a variable name. Take care however, if a variable with the same name already exists in the workspace, it will simply be overwritten.

5.2 Simple example - revisited

As was mentioned, we will regard the problem (5.1) once again using one of the library objective functions. Also, we will have a look at what the *Advanced* button can offer. For a fresh start, press the *Reset* button and the toolbox will delete all data and go back to the default settings. Now, in stead of selecting the *Customized function* radio button, select *Library objective function* and choose the alternative called *Simple quadratic, exponent* from the function type menu. It is very important now to make sure that the *Canonical form* checkbox is checked, or else the toolbox will have you defining a strictly *increasing* function in stead of strictly *decreasing*. Pressing the *Define parameters* button, a window pops up where we can define our objective function. Figure 10 shows the window with the correct inputs.

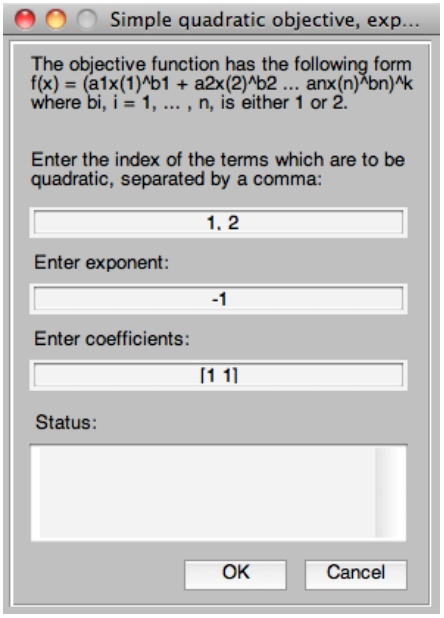

Figure 10: Defining the objective function for simple example.

If any of the input fields has an erroneous format or the given parameters are such that the function cannot be strictly decreasing, error messages will appear in the status window when pressing *OK*. In other words, the window will only close if the inputs are valid. Note, however, that the coefficients *must* be entered in order to register an objective function, while it is strictly speaking not necessary to fill in any indexes. If everything is correct, the window will close and the main window status field will in this case state:

```
Confirmation: The chosen objective function is strictly
decreasing as long as x is bounded positive.
```

which we of course already know is true from the bound constraints of the problem definition (5.1).

Now for the constraints. Pressing the *Advanced* button, a new window pops up which is shown in Figure 11. The purpose of the upper panel is the means for providing information about the objective function when we have to use a customized objective, i.e. the function type does not exist in the library. This will be reviewed for later examples. The lower panel contains the functionality for defining either linear or quadratic constraint. Select *Linear* from the function type menu and press the *Define* button. A new window pops up where we can input both linear inequality and equality constraints on matrix form. Figure 12 shows this window with the correct inputs for our problem. Note that, also here, the inputs can be given by variables in the workspace.

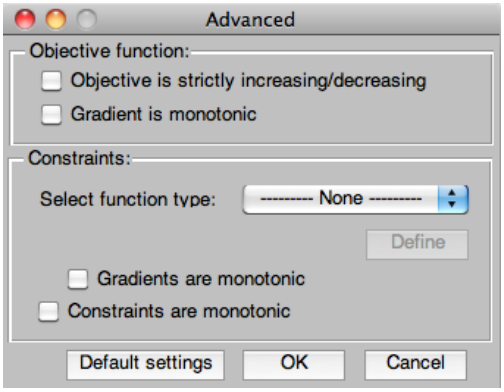

Figure 11: Advanced options.

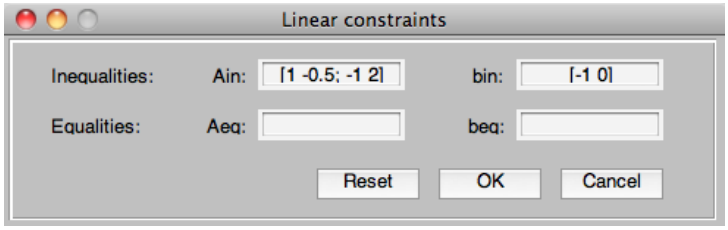

Figure 12: Defining the linear constraints of simple example.

Pressing *OK* twice, we are back to the main window and observe that the input fields for the constraint function and gradient have been disabled. All that remains now is to fill in the bounds and investigate if our problem is F-Lipschitz. This time, the status window contains the following message:

```
The combination of Ain and Aeq is referred to as the matrix A.
-The objective function is strictly decreasing.
-The diagonal of A is positive.

Alternatives:
1. -The off-diagonal elements of A are all negative.
   -Each element on the diagonal of A is larger than the sum of
the absolute values of the other elements on the same row.

The problem is F-Lipschitz.
Unconstrained fmincon used 0 iterations and 0 function
evaluations in total.
Total amount of iterations: 6
```

Clearly, we have managed to completely avoid *fmincon* and in doing so reducing the number of iterations from 10 to 6. Most importantly, the 30 function evaluations from before are now eliminated. This means that, with the total amount of 15 iterations/evaluations, the centralized algorithm outperforms the interior point method with good margin. Even more

astounding, the distributed algorithm is also superior, spending a total of 27 iterations versus the interior point method's 33 evaluations. The active set method is however still in the lead with its 6 evaluations. This simple problem may also be transformed to the standard form by finding

$$\gamma_1 = \frac{1}{\nabla_1 g_1(\boldsymbol{x})} = 1 \qquad \text{and} \qquad \gamma_2 = \frac{1}{\nabla_2 g_2(\boldsymbol{x})} = 0.5,$$

which are applied in redefining the problem according to (3.5). Using the objective function library, it is then important to remember that

$$\min_{\boldsymbol{x}} \ (x_1^2 + x_2^2)^{-1} \qquad \text{and} \qquad \max_{\boldsymbol{x}} \ x_1^2 + x_2^2$$

are in fact the same. While the first objective function is strictly *decreasing* for positive $\boldsymbol{x}$, the second is obviously strictly *increasing* for the same range of values. The qualifying properties will be verified for the standard form problem as well, but we will see that this is not always the case.

5.3 Example in $\mathbb{R}^2$ with quadratic constraint

Consider the following optimization problem

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & -x_1 - x_2 \\ \text{s.t.} \quad & 4x_1^2 + x_2^2 + x_1 - 2 \leq 0 \\ & x_1^2 + 3x_2^2 + x_1 x_2 - 3.5 \leq 0 \\ & 0.1 \leq x_1, x_2 \leq 1.5 \end{aligned} \tag{5.2}$$

which is on canonical form. The reason why we need the decision variables to be larger than zero, or actually just one of them, is because of the condition 1.b which demands $\nabla_2 g_2(\boldsymbol{x}) = x_1 + 6x_2 > 0$. Each constraint can be written on the form

$$G_i(\boldsymbol{x}) = \boldsymbol{x}^T \mathbf{Q}_i \boldsymbol{x} + \boldsymbol{r}_i^T \boldsymbol{x} + c_i$$

where

$$\mathbf{Q}_1 = \begin{bmatrix} 4 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \mathbf{r}_1 = \begin{bmatrix} 1 & 0 \end{bmatrix}^T, \qquad c_1 = -2,$$

and

$$\mathbf{Q}_2 = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 3 \end{bmatrix}, \qquad \mathbf{r}_2 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T, \qquad c_2 = -3.5.$$

Let us define the problem in the toolbox using a library objective function and the advanced settings. First, we store the constraint parameters given above as variables in the workspace. Here, we use the obvious naming *q1, r1, c1, q2, r2* and *c2*. Choose the *Linear* alternative in the function type menu and press the *Define parameters* button. A window pops up in which there is only one input field. Here, we enter the vector, $[-1, -1]$, and press *OK*. The constraints are defined by entering the advanced settings and choosing *Quadratic* from the function type menu. Pressing the *Define* button, the window shown in Figure 13 pops up, where the parameters of the first constraint have been entered. Notice the possibility of choosing if the constraint should be an inequality or an equality. Press *Next* and enter the corresponding inputs for the second constraint.

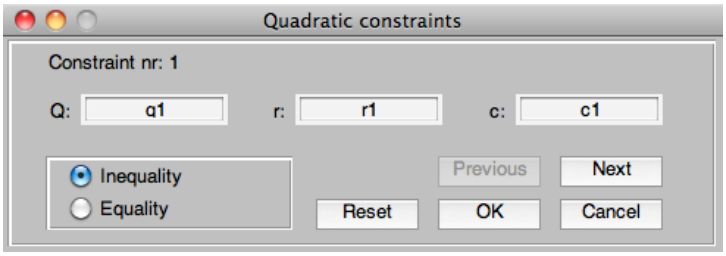

Figure 13: Defining the quadratic constraints.

Finally, the upper and lower bounds are set to $[0.1, 0.1]$ and $[1.5, 1.5]$, respectively. Pressing the *Check feasibility* button, the following message appears in the status window:

```
-The objective function is strictly decreasing.
-The diagonal of the constraint gradient is strictly positive.

Alternatives:
1. -The off-diagonal elements of the constraint gradient are not
all negative.
2. -The gradients of the objective are equal.
   -The off-diagonal elements of the constraint gradient are all
positive.
   -The diagonal elements are all larger than the absolute sum
of the same column.

The problem is F-Lipschitz.
Unconstrained fmincon used 0 iterations and 0 function
evaluations in total.
Total amount of iterations: 18
```

Thinking back to the discussion in Section 4.1.4, one might realize that it should be possible to reduce the number of iterations further. In fact, not only are $\mathbf{Q}_1$ and $\mathbf{Q}_2$ both positive matrices, but $\boldsymbol{x}$ is also bounded to only

positive numbers. By that note, we may open the advanced settings and check both the *Gradients are monotonic* and the *Constraints are monotonic* checkbox. In this case of quadratic constraints, the latter actually implies the first, but of course it does not harm to check both boxes. When checking feasibility once again, we now find that the toolbox only use 6 iterations, i.e. a third of what was previously found. If the problem is defined with m-file functions and we have to use *fmincon*, the verification takes 12 iterations and 36 function evaluations in total. Thus, either way we are a lot better off when the problem is defined through the utilities of the toolbox.

Running the distributed algorithm, we get the results shown below and the plot illustrating the progress of the algorithm can be seen in Figure 14.

```
The algorithm stopped after 15 iterations.
There were 15 function evaluations.

The solution values are
x(1) = 0.394
x(2) = 0.992
```

Once again we observe that the iterates changes quite fast in the beginning, followed by a slow convergence to the required accuracy. The centralized algorithm converges in 7 iterations and 21 function evaluations, which is faster than the interior point algorithm, using 9 iterations and 27 function evaluations. Notice that if the 6 iterations from the verification operation are included, these methods are equally good. Still, the distributed algorithm is more efficient than both of them, not to mention that it is almost just as good as the active set algorithm with its 6 iterations and 20 function evaluations.

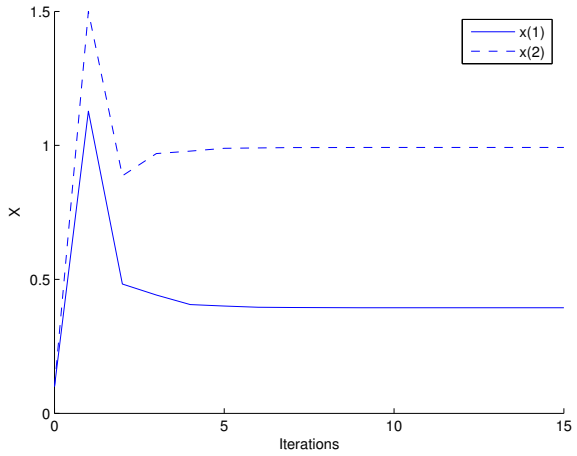

Figure 14: Iterations versus $\boldsymbol{x}$ for the distributed algorithm.

The problem shall now be transformed into the standard form. In order to do that, we need to find γ_1 and γ_2 such that the inequality (3.10) is fulfilled for all $x_{i,\min} \leq x_i \leq x_{i,\max}$, $\forall i$, i.e. for all $\boldsymbol{x} \in \mathscr{D}$. The best, or in other words the largest, γ_i can then be found by solving the optimization problem

$$\gamma_i = \min_{\boldsymbol{x} \in \mathscr{D}} \frac{1}{\nabla_i g_i(\boldsymbol{x})}$$

which results in $\gamma_1 = 0.0769$ and $\gamma_2 = 0.0952$. As stated in Section 3.2.3, we let

$$f_0(\boldsymbol{x}) = -g_0(\boldsymbol{x}) = x_1 + x_2$$

and using $f_i(\boldsymbol{x}) = x_i - \gamma_i g_i(\boldsymbol{x})$, we get the following constraint parameters

$$\mathbf{Q}_1 = \begin{bmatrix} -0.3077 & 0 \\ 0 & -0.0769 \end{bmatrix}, \qquad \mathbf{r}_1 = \begin{bmatrix} 0.9231 & 0 \end{bmatrix}, \qquad c_1 = 0.1538,$$

$$\mathbf{Q}_2 = \begin{bmatrix} -0.0952 & -0.0476 \\ -0.0476 & -0.2857 \end{bmatrix}, \qquad \mathbf{r}_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}, \qquad c_2 = 0.3333.$$

The problem may be defined as before, using the library objective function and the advanced settings. Remember that the *Canonical form* checkbox at the bottom of the main window should not be checked this time. Investigating the qualifying properties, the toolbox now returns the following message:

```
-The objective function is strictly increasing.
-The infinity norm is less than one.

Alternatives:
1. -The elements of the constraint gradient are not all positive.
2. -The gradients of the objective are equal.
   -The elements of the constraint gradient are not all negative.
3. -The objective function is scalar.
   -The one norm of the constraint gradient is not smaller than
the coefficient 0.500.

The problem is not F-Lipschitz.
Unconstrained fmincon used 0 iterations and 0 function
evaluations in total.
Total amount of iterations: 17
```

Clearly, it would have made sense if the problem should fulfill the conditions of the second alternative, since this was the case for the canonical form problem. As can be observed, the gradient of the constraints are however

not negative over all values of $\boldsymbol{x}$. This is because of the fairly large positive value of $\mathbf{r}_1$. The elements of $\mathbf{Q}_1$ are simply not "negative enough" to compensate for this. That said, by choosing the values of γ small enough, there will eventually be some limit where the conditions are verified. This will however result in the distributed algorithm being slower.

If we now press the *Skip verification* button and move on to the computation regardless, we will see that the distributed algorithm returns exactly the same result as before. In other words, the problem is in fact F-Lipschitz and it is proven that the qualifying properties (3.2a) - (3.2h) are only *sufficient* and not *necessary*. However, the distributed algorithm spend more than twice as many iterations as the canonical algorithm. This is because we had to choose constant values for γ when transforming the problem, thus excluding the opportunity of choosing the best values for each iteration.

5.4 Radio power allocation

Since power control has previously been discussed both regarding the interference function theory as well as for geometric programs with applied decomposition methods, we here present a similar example to which we apply the F-Lipschitz optimization theory. This application is also discussed in [5].

We consider a network of n transmitter nodes, where node i transmits using a radio power p_i, $i = 1, \ldots, n$. There are n receiver nodes, where node i receives the power $G_{ii}p_i$ from transmitter i, i.e. G_{ii} is the channel gain. The interference at receiver i is then given by $\sum_{k\neq i} G_{ik}p_k$. To make things a bit more interesting, we also introduce a nonlinear term, $M_{ij}p_i^2p_j^2$, which represents the intermodulation between the signals from transmitter i and j. This typically occurs when the amplifier of the receiver consists of somewhat unreliable components. The signal to interference plus noise ratio, $SINR$, of the i-th transmitter-receiver pair is then given by

$$SINR_i = \frac{G_{ii}p_i}{\sigma_i + \sum_{k\neq i} G_{ik}p_k + \sum_{k\neq i} M_{ik}p_i^2p_k^2} \tag{5.3}$$

where σ_i is the thermal noise. Note that the values M_{ik}, $k \neq i$, are smaller than G_{ik} and can be both positive and negative.

The problem can be written on the following form

$$\begin{aligned} \min_{\boldsymbol{p}} \quad & \boldsymbol{p} \\ \text{s.t.} \quad & SINR_i \geq S_{\min}, \quad i = 1, \ldots, n, \\ & p_{\min}\mathbf{1} < \boldsymbol{p} \leq p_{\max}\mathbf{1}, \end{aligned} \tag{5.4}$$

where $S_{\min}$ is the minimum required $SINR$ to guarantee that the signal is successfully received. The upper and lower bounds on $\boldsymbol{p}$ are quite naturally the smallest and the largest power level which are available to the transmitter. We let $x_i = -p_i$ and rewrite the problem on canonical form as

$$
\begin{aligned}
\min_{\boldsymbol{x}} \quad & -\boldsymbol{x} \\
\text{s.t.} \quad & G_{ii}x_i + S_{\min}\left(\sigma_i - \sum_{k\neq i} G_{ik}x_k + \sum_{k\neq i} M_{ik}x_i^2 x_k^2\right) \leq 0, \\
& i = 1, \ldots, n, \\
& -p_{\max}\mathbf{1} < \boldsymbol{p} \leq -p_{\min}\mathbf{1}.
\end{aligned} \tag{5.5}
$$

As is the usual notation, we denote the constraint function above as $g_i(\boldsymbol{x})$. Let us also consider the gradient of the function, which is given as

$$
\nabla_j g_i(\boldsymbol{x}) = \begin{cases} G_{ii} + 2S_{\min}\sum_{k\neq i} M_{ik}x_i x_k^2 & \text{if } j = i, \\ -S_{\min}G_{ij} + 2S_{\min}M_{ij}x_i^2 x_j & \text{otherwise.} \end{cases} \tag{5.6}
$$

Now that we have defined the problem, we may test it with the toolbox. The parameter values are shown in Table 4 and the MATLAB code defining the constraints and their gradients can be found in Appendix B. At a glance we see that the scalarized version of the objective function in problem (5.5) is clearly strictly decreasing. Thus, if the objective is defined using an m-file function, we may check the *Objective is strictly increasing/decreasing* checkbox in the advance settings, consequently saving these iterations. Otherwise, a library objective function may of course also be used with the appropriate coefficient input. More importantly, observing the constraint gradient (5.6), keeping in mind that the decision variables are bounded to only negative numbers, we conclude that it is monotonic for both of the cases. In other words, we can also check the *Gradients are monotonic* checkbox in the constraints panel of the advanced settings. Thus, we can completely avoid *fmincon* when investigating the qualifying properties. Note that this checkbox can only be used when the extreme vectors are equal to $x_{\min}$ and $x_{\max}$, see the discussion under Section 4.1.4.

Parameter	Value	
n	10	
$S_{\min}$	1	$\forall i$
G_{ij}	-90 dBm	$\forall i, j \neq i$
G_{ii}	-70 dBm	$\forall i$
M_{ij}	-120 dBm	$\forall i, j \neq i$
σ_i	-130 dBm	$\forall i$
$p_{\min}$	-25 dBm	
$p_{\max}$	0 dBm	

Table 4: Parameter values which were found in [5].

Investigating the qualifying properties, the toolbox returns the following status message:

```
-The objective function is strictly decreasing.
-The diagonal of the constraint gradient is strictly positive.

Alternatives:
1. -The off-diagonal elements of the constraint gradient are all
negative.
   -The diagonal of the constraint gradient is dominant.

The problem is F-Lipschitz.
Unconstrained fmincon used 0 iterations and 0 function
evaluations in total.
Total amount of iterations: 110
```

So, the problem is verified to be F-Lipschitz and we can safely move on to try the distributed algorithm. Observe that the number of iterations indicates that the toolbox did in fact use $O(n^2)$ iterations to determine that the problem was feasible. Had the checkboxes remained unchecked, *fmincon* would have used 120 iterations and 1320 function evaluations, which is clearly a large increase. Since the parameters has to be converted from dBm to watts, namely through

$$P = 10^{(d-30)/10}$$

where d is the power ratio in dBm, the numbers becomes very small. Therefore, we must take care to set the required accuracy sufficiently small. In stead of using the default, $1 \cdot 10^{-6}$, all of the tolerances should be set to $1 \cdot 10^{-24}$.

The distributed algorithm converges after only 4 iterations, returning the solution $x_i = -3.162 \cdot 10^{-6}$, $\forall i$, which is actually equal to the upper bound of $\boldsymbol{x}$, i.e. we get $p_i = -25$ dBm. The progress of the algorithm can be viewed in Figure 15. Testing the centralized algorithm, a solution is found after 2 iterations and 22 function evaluations. Unfortunately, it is the wrong solution, namely $x_i = -1.099 \cdot 10^{-6}$, which is clearly outside of the bounds. This is because, as previously mentioned, *fsolve* has no support for bound constraints. On the other hand, it can be interesting to know where the "actual" fixed point was located. Observe that for this problem with quite a large amount of decision variables, the distributed algorithm is a lot more efficient than the centralized.

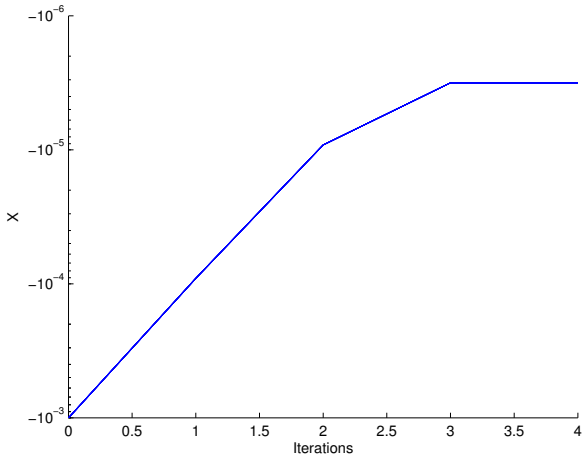

Figure 15: Iterations versus $\boldsymbol{x}$ for the distributed algorithm. Each of the lines for the n decision variables are on top of each other.

For comparison, let us also investigate how well the Lagrangian solver performs. The interior point algorithm uses as much as 17 iterations and 195 function evaluations, see Figure 16. On the other hand, the active set algorithm is much faster, using only 3 iterations and 33 function evaluations. They both find the globally best solution, i.e. the same as was found using the distributed algorithm. Even though *fsolve* went outside of the bounds, the F-Lipschitz theory was without a doubt superior in this case. Counting the iterations spent for the verifying operations, both the distributed and the centralized algorithm still outperforms the interior point method.

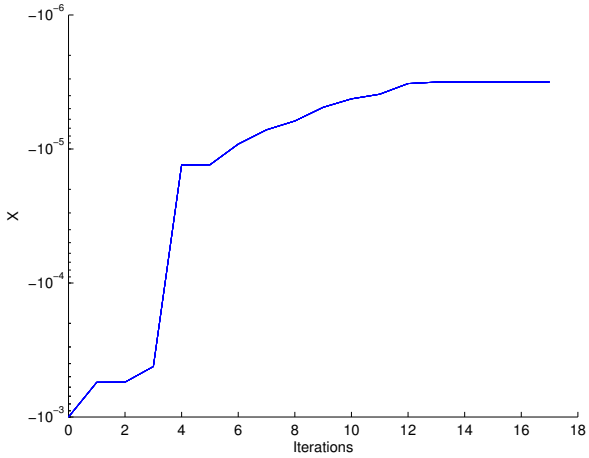

Figure 16: Iterations versus $\boldsymbol{x}$ for the interior point algorithm. Each of the lines for the n decision variables are on top of each other.

It is interesting to consider what other strategies we might have used for distributing the optimization of this problem. The fact is, because the intermodulation coefficient, M_{ij}, can be negative, it is not possible to use either geometric programming or the interference function theory. For the latter case, it is simply because the interference function, $I(\boldsymbol{p})$, will not fulfill the required conditions to be *standard*, see Definition 2.2 in Section 2.5. This leaves us with decomposition methods, which has already been confirmed to be a less satisfactory alternative. In other words, the distributed F-Lipschitz algorithm is without a doubt the best option in this case.

5.5 Computation of a norm

In wireless sensor networks, there are several cases where it is necessary to compute a stable matrix $\mathbf{K}$ with respect to some norm. This is common in e.g. distributed consensus where the goal is for all the nodes to "come to consensus" on a measurement value of some environmental parameter. Each row $\mathbf{k}_i$, $i = 1, \ldots, n$ of the matrix $\mathbf{K}$ belongs to a node. For calculating a suitable threshold on the norm of $\mathbf{k}_i$, we use the following proposition as it is stated in [14] (Proposition 2.1):

Proposition 5.1. *Let* $\mathbf{K} = [\mathbf{k}_i] \in \mathbb{R}^{n \times n}$*, where* $\mathbf{k}_i \in \mathbb{R}^{1 \times n}$*. Let* $0 < \gamma_{max} < 1$*. Suppose there exists a vector* $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T \succ 0$*, such that*

$$x_i + \sqrt{x_i} \sum_{j \in \Theta_i} \sqrt{x_j} \leq \gamma_{max}, \tag{5.7}$$

for all $i = 1, \ldots, n$*, where the set* Θ_i *is the collection of communicating nodes located at two-hops distance from the node* i*, plus the neighbors of* i*. If* $||\mathbf{k}_i||^2 \leq x_i$*,* $i = 1, \ldots, n$*, then* $||\mathbf{K}||_2 \leq \gamma_{max}$*.*

In other words, if we find a vector $\boldsymbol{x}$ which satisfies the inequality (5.7), we find a threshold for each of the norms of the rows $\mathbf{k}_i$. As long as the norms are below these thresholds, the matrix $\mathbf{K}$ will be stable. Naturally, we would like to maximize $\boldsymbol{x}$ in order to have as much freedom as possible in choosing the elements of each $\mathbf{k}_i$. The optimization problem can then be stated as

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & -\mathbf{1}^T \boldsymbol{x} \\ \text{s.t.} \quad & x_i + \sqrt{x_i} \sum_{j \in \Theta_i} \sqrt{x_j} \leq \gamma_{\max}, \quad i = 1, \ldots, n, \\ & \boldsymbol{x} \in \mathbb{R}^n_+ \end{aligned} \tag{5.8}$$

which is on canonical form. Clearly, it is necessary to find better bounds on $\boldsymbol{x}$ if the problem is to be tested with the toolbox. This can easily be derived by exploiting the formulation of the problem. Because we have $0 < \gamma_{\max} < 1$, the constraints implies that $0 \prec \boldsymbol{x} \prec 1$. It is safe to use the vector $\mathbf{1}^T$ as the upper bound since the constraints always ensures that $\boldsymbol{x}$ is less than this.

A better alternative must however be found for the lower bound. Consider the constraint function of problem (5.8) when it is at the equality and let $x_j \approx 1$, $\forall j \in \Theta_i$. Then, x_i must be at its minimum, i.e. the following must apply

$$x_{i,\min} + \sqrt{x_{i,\min}} \cdot |\Theta_i| = \gamma_{\max}.$$

A variable change, $x_{i,\min} = y^2$, leaves us with a regular quadratic equation, which results in the following solution for $x_{i,\min}$

$$x_{i,\min} = \left(\frac{-|\Theta_i| + \sqrt{|\Theta_i|^2 + 4\gamma_{\max}}}{2} \right)^2.$$

The network we will be considering is shown in Figure 17 and the problem shall be tested using $\gamma_{\max} = 0.75$. Like before, the constraint function is denoted $g_i(\boldsymbol{x})$. The gradient is given as

$$\nabla_j g_i(\boldsymbol{x}) = \begin{cases} 1 + \frac{1}{2\sqrt{x_i}} \sum_{j \in \Theta_i} \sqrt{x_j} & \text{if } j = i, \\ \sqrt{x_i} \cdot \frac{1}{2\sqrt{x_j}} & \text{if } j \in \Theta_i, \\ 0 & \text{otherwise.} \end{cases} \tag{5.9}$$

See Appendix B for the MATLAB code defining the constraints and their gradients. Like in the previous example, we immediately observe that the objective function is valid F-Lipschitz and we can use the corresponding checkbox of the advanced settings. This time, however, we cannot use the checkbox for monotonic constraint gradients, even though they actually are monotonic and $\boldsymbol{x}$ is also bounded to only positive numbers.

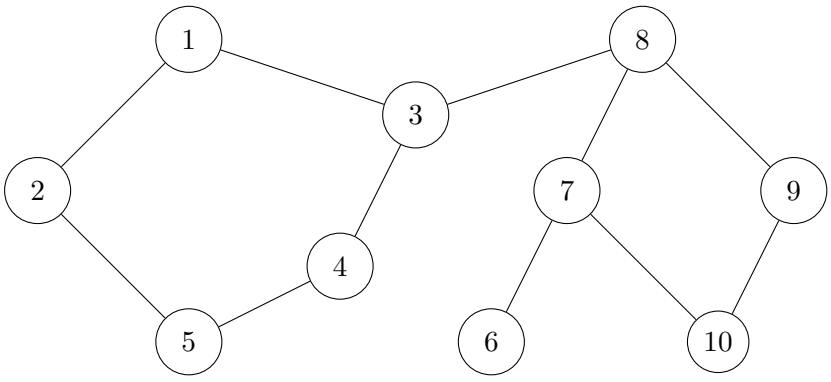

Figure 17: The network of nodes which should distributively optimize problem (5.8).

Observe that if Eq. (5.9) is not zero, there is always one decision variable with a negative exponent and one or more which has positive exponents. This means that the minimum value of each partial derivative must occur when the variable with the negative exponent is at the upper bound and the rest are at the lower bound. In other words, we can unfortunately not use $\boldsymbol{x}_{\min}$ and $\boldsymbol{x}_{\max}$ as the extreme vectors. Because of this, the toolbox requires quite a large amount of iterations to verify that the problem is F-Lipschitz, specifically it takes 180 iterations and 1870 function evaluations in total. Otherwise, it would have taken something in the range of a 100 iterations like for the previous esample. The problem was shown to fulfill the second alternative of the qualifying properties, namely (3.9e) - (3.9g).

With an accuracy of $1 \cdot 10^{-6}$, the distributed algorithm converges in 35 iterations. The solution values are shown in Table 5 and the plot of the algorithm's progress is shown in Figure 18. With a somewhat similar performance, the centralized algorithm converges in 4 iterations and 44 function evaluations, of course returning the same solution as the distributed algorithm. A plot of the iterations is shown in Figure 19.

i	1	2	3	4	5	6	7	8	9	10
x_i	0.131	0.171	0.069	0.131	0.171	0.253	0.117	0.066	0.196	0.151

Table 5: Solution found by the distributed algorithm.

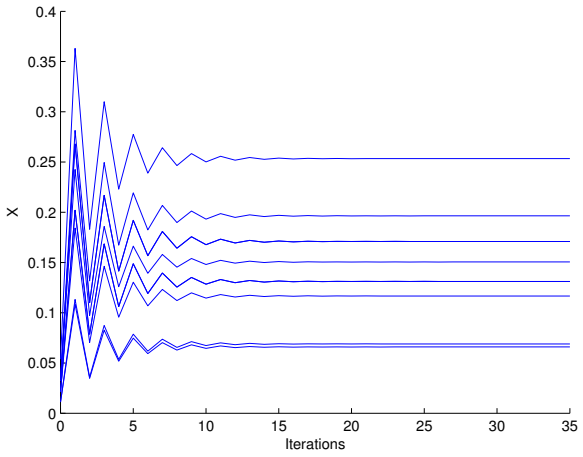

Figure 18: Iterations versus $\boldsymbol{x}$ for the distributed algorithm. Clearly, some of the lines are lying on top of each other, which is not surprising since there are solution values in Table 5 which are equal.

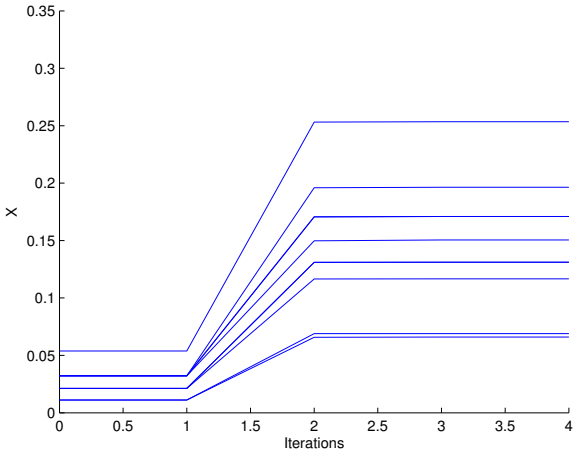

Figure 19: Iterations versus $\boldsymbol{x}$ for the centralized algorithm.

The Lagrangian solver seems to be doing particularly bad for this problem. Not only does the interior point algorithm use as much as 43 iterations and 473 function evaluations, but the returned solution is not the globally best point, which is shown in Table 5, it is merely a local solution. See Figure 20 for a plot of the iterations. Same goes for the active set algorithm. It is a lot faster with its 6 iterations and 66 function evaluations, but the solution is local. Had we been able to verify the problem as easily as for the power control example, then both the distributed and centralized F-Lipschitz algorithms would have been a lot more efficient than the interior point method. The active set is faster either way, but it is a major drawback that it cannot find the globally best solution.

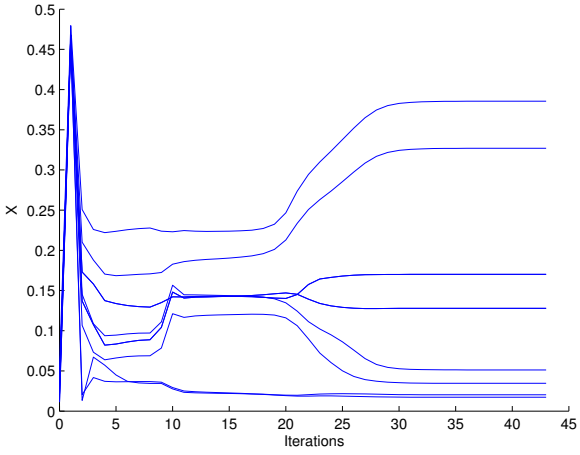

Figure 20: Iterations versus $\boldsymbol{x}$ for the interior point method.

5.6 Distributed detection

We consider yet another problem, also discussed in [5], which is typical within wireless sensor networks, namely event detection, also referred to as binary hypothesis testing. The idea is that each sensor node in a network should be able to detect if one out of two events has occurred. Denoting the first event H_0 and the second H_1, the detection of each event can be modeled by a Gaussian random variable $w_i(s)$ as follows

$$\Gamma_i(s) = \begin{cases} w_i(s) & \text{if } H_0, \\ E + w_i(s) & \text{if } H_1, \end{cases}$$

where $\Gamma_i(s)$ then denotes the outcome of the random variable at sample s and E is a signal level which separates H_1 from H_0. The mean of $w_i(s)$ is zero and its variance, σ^2, models the uncertainty of detection.

A logical solution is to let each node i take several samples, $s = 1, \ldots, S$ and average $\Gamma_i(s)$ over all the samples. This is classically known as the *Likelihood ratio test* and it is here defined as

$$T_i = \frac{1}{S} \sum_{s=1}^{S} \Gamma_i(s) \gtreqless x_i$$

where x_i is a detection threshold. This basically means that if $T_i \leq x_i$, node i will decide that H_0 was the occurring event and H_1 otherwise. Based on this test, we may also derive the probability of false alarm and of misdetection. The first implies the chances of H_1 being erroneously detected, while the second corresponds to the detection of H_0 when H_1 actually happened. These probabilities are defined as follows

$$P_{\text{fa}}^{(i)}(x_i) = \Pr[T_i > x_i[H_0]] = Q\left(\frac{x_i}{\sqrt{\frac{\sigma^2}{S}}}\right),$$

$$P_{\text{md}}^{(i)}(x_i) = \Pr[T_i \leq x_i[H_1]] = Q\left(\frac{E - x_i}{\sqrt{\frac{\sigma^2}{S}}}\right),$$

where $Q(x)$ is the complementary standard Gaussian distribution given by

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{t^2}{2}} dt. \tag{5.10}$$

Finally, we may derive the problem. The goal is to optimize the thresholds, x_i, so as to minimize the probability of false alarm. In order to avoid the probability of misdetection becoming unreasonably large, it must be

constrained with a maximum threshold, here denoted c_i. By taking advantage of the information held by other nodes, the overall performance may be improved. In other words, each node receives misdetection probabilities from neighboring nodes, accounting for their opinions by applying weighting factors. A global optimization problem corresponding to these descriptions may be defined as follows

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \sum_{i=1}^{n} P_{\text{fa}}^{(i)}(x_i) \\ \text{s.t.} \quad & \sum_{j=i}^{n} b_{i,j} P_{\text{md}}^{(j)}(x_j) \leq c_i, \quad i = 1, \ldots, n, \\ & 0 \preceq \boldsymbol{x} \preceq E\mathbf{1}, \end{aligned} \tag{5.11}$$

where the weighting factors, $b_{i,j} \leq 0$, $\forall i, j$, $b_{i,i} \neq 0$, $\sum_{j=i}^{n} b_{i,j} = 1$, and $b_{i,j} = 0$ when node j does not transmit its probability to node i. See Appendix B for the MATLAB code defining the objective and constraints, as well as their gradients.

We test the problem for a network of $n = 5$ nodes. Let $E = 2$, $\sigma = 1$, $S = 1$ and $c_i = 1/15$. Moreover, the weighting factors $b_{i,j}$, $\forall i, j$, are initially drawn randomly from the standard uniform distribution on the interval $[0, 1]$, then normalized. Additionally, one must ensure that the values $b_{i,i}$, $\forall i$, are sufficiently large compared to the other factors in order to have a feasible F-Lipschitz problem. This is also logically justified in that the node should weigh in its own opinion more than of the other nodes. The derivative of $P_{\text{fa}}^{(i)}(x_i)$ is given by

$$Q'\left(\frac{x_i}{\sqrt{\frac{\sigma^2}{S}}}\right) = -\frac{1}{\sqrt{2\pi\frac{\sigma^2}{S}}}\, e^{-\frac{x_i^2}{2\sigma^2/S}} \tag{5.12}$$

which is clearly always negative, hence we know that the objective function must be strictly decreasing. Since the derivative of the constraint function is given by $Q'(\cdot)$ with a *negative* argument, we may correspondingly conclude that the constraint gradient is strictly positive. Additionally, since both the constraint function and the objective function consists of positive combinations of exponential functions, it is also clear that all of the gradients are monotonic. Conclusively, both the checkboxes in the objective function panel of the advanced settings may be checked, as well as both the checkboxes for the constraints. This allows us to verify the feasibility of the problem, on the third alternative, in only 15 iterations compared to the 21 iterations and 126 function evaluations which would have otherwise been necessary.

The distributed algorithm converges in 10 iterations and finds the solution $x_i = 0.499$, $\forall i$, see Figure 21 for the algorithm's progression. One might think it strange that all the decision variables should have the same value

since the weighting factors, $b_{i,j}$, are all different. Keep in mind though, that the sum of the weights for each constraint are always equal to one, hence in this sense the constraints are the same. Logically, it is also reasonable that the threshold should be equal for each node since they all have the same uncertainty of detection. The centralized algorithm returns the same solution after 4 iterations and 24 function evaluations, see Figure 22.

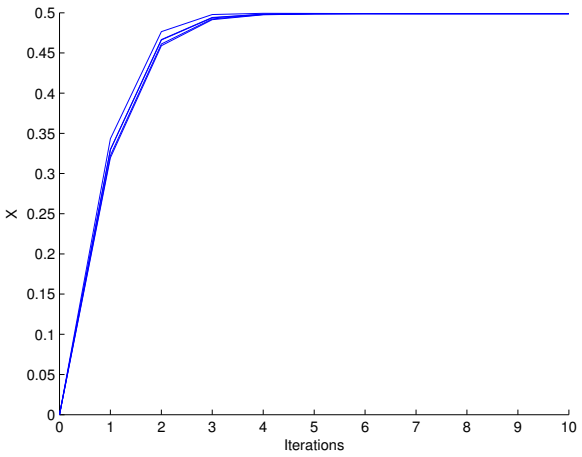

Figure 21: Iterations versus $\boldsymbol{x}$ for the distributed algorithm.

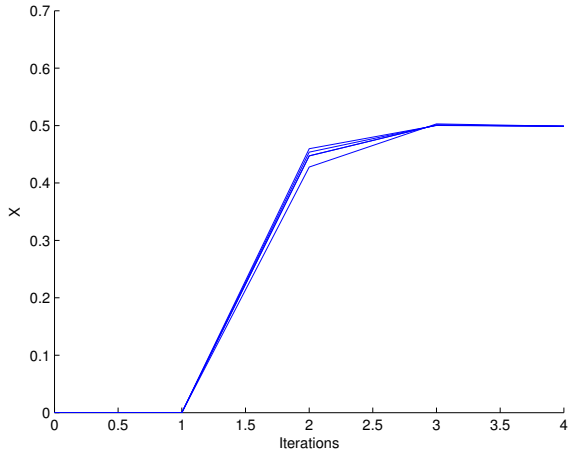

Figure 22: Iterations versus $\boldsymbol{x}$ for the centralized algorithm.

Testing the Lagrangian methods, we find that the interior point algorithm converges in 10 iterations and 61 function evaluations, see Figure 23, while the active set method converges in 5 iterations and 31 function evaluations, see Figure 24. Notice that the distributed method is this time the very best performer. With its total of 25 iterations for both the verifying

process and the algorithm itself, it beats the 31 evaluations of the active set method. Once again, both of the F-Lipschitz algorithms are faster than the interior point method.

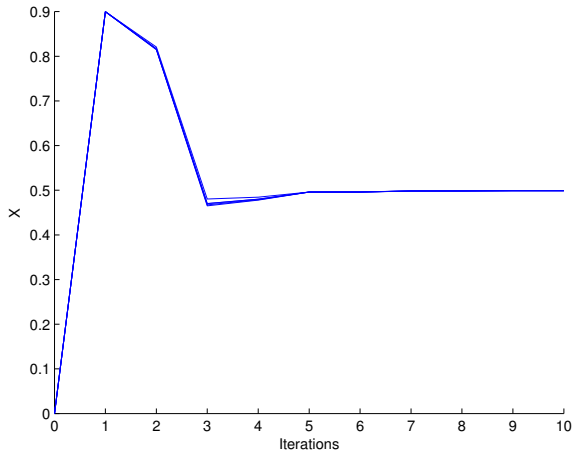

Figure 23: Iterations versus $\boldsymbol{x}$ for the interior point method.

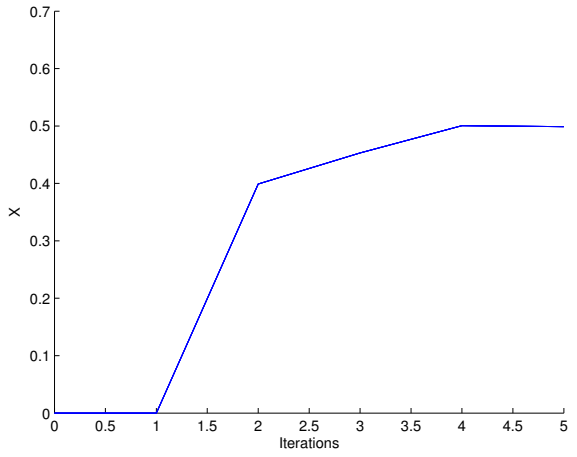

Figure 24: Iterations versus $\boldsymbol{x}$ for the active set method.

6 Discussion and Further Work

Table 6 summarizes the results of the previous chapter. Clearly, the active set method has quite often proven to be effective, hence it is interesting to consider what might be the cause of this. The strategy applied by this method is to iteratively solve the optimization problem defined by an *active set* of the constraints. In other words, the method will start looking for a solution where the constraints are at the equality. Since it is known that an F-Lipschitz problem has its optimal point when all of the constraints are active, this might be an explanation to the method's good performance. This theory is also supported by observing that its progression is quite alike that of *fsolve*, see Figures 22 and 24. Still, it is somewhat peculiar that the active set method should be faster than the centralized algorithm for the problems with linear and quadratic constraints. After all, *fsolve* do not even take the objective function into consideration, while the active set method do.

Analyzing the results, one may conclude that both the distributed and the centralized algorithms in themselves are superior when the decision variables increase. For the two problems with $n = 10$ it is clear that, if the verification process has a complexity of $O(n^2)$, both the F-Lipschitz algorithms are still better than the interior point method. As can be seen, the active set method remains the most efficient. Overall, the power control problem was solved faster than the norm computation problem. This may be justified by remembering that all of the constraints of the first problem were equal, while this was not the case for the norm computation. Even though the active set method was faster, it did not find the globally optimal point for the norm computation example, neither did the interior point method. Obviously, it is a major drawback that these methods cannot guarantee a global solution unless the problem is convex.

Example	Linear constraints	Quadratic constraints	Power control allocation	Norm computation	Distributed detection
n	2	2	10	10	5
Verify	6	6	110	180, 1870	15
Algorithms					
Distributed	21	15	4	35	10
Centralized	3, 9	7, 21	2, 22	4, 44	4, 24
Interior p.	11, 33	9, 27	17, 195	43, 473	10, 61
Active set	2, 6	6, 20	3, 33	6, 66	5, 31

Table 6: Summary of the performance of the different algorithms for each of the examples discussed in Chapter 5. Where there are two numbers, the first is the amount of iterations and the second is the number of function evaluations.

The effectiveness of the active set method throughout all of the examples is evident. It is however shown that the distributed algorithm is in fact faster for the distributed detection problem, even when including the iterations from the verifying process. This is perhaps explained by the somewhat more complicated objective function, consisting of a sum of exponential functions, as compared to the exceedingly simple objective of the two previous examples, namely $-\mathbf{1}^T\boldsymbol{x}$. It is interesting to note that the active set method is about twice as fast as the interior point method for this example. The fact is that the latter method should actually perform exceptionally well for problems which are convex, which is the case for this problem. It might be that the active set method is simply that much superior because of the favorable properties of the F-Lipschitz problem which was discussed above.

There are numerous of additions and changes which may be applied to improve the toolbox. Naturally, more function types can be added to the objective function library, like for example the monomial function

$$f(x) = cx_1^{a_1}x_2^{a_2}\cdots x_n^{a_n},$$

which is important for the definition of a geometric program. Given $\boldsymbol{x} \in \mathscr{D}$, the minimum and maximum of this function's gradient may be found by systematically checking the value of each exponent, a_i. Correspondingly, this function as well as others may also be added to the class of constraints in addition to the already defined linear and quadratic constraints.

It would have been beneficial to extend the verification of the qualifying properties in such a way that one could choose which of the three alternatives the toolbox should check. Quite often, one has some indication as to which alternative might apply, hence there is no need in wasting time checking the other properties. Of course, this would require that the potential user has some more knowledge of the theory. Either way, it is always better to have more options and increased flexibility. To avoid using *fmincon* for the verification, one could possibly implement Newton's method for this purpose. This would however mean that the user must also provide the Hessian of the objective and constraints, or else one would have to find a way of approximating it as is the case for *fmincon*. In order to support monotonic constraint and objective gradients which need customized extreme vectors, additional functionality can be implemented for the advanced settings. This should allow the user to input the combinations of $x_{i,\min}$ and $x_{i,\max}$, $\forall i$, which constitutes the extreme vectors for each of the partial derivatives. In this case, the toolbox would have been able to verify the F-Lipschitz property of the norm computation problem in approximately the same amount of iterations as for the power control problem.

Finally, it is always nice to extend the selection of solvers so as to have more to compare with, especially distributed computation schemes. As have been mentioned, it is challenging, if not impossible, to make a general solver

based on the Lagrangian decomposition methods. However, a solver could have been implemented for some specific problems, like e.g. the application examples of Chapter 5, which are typical for wireless sensor networks. Also, it would be very interesting to see how a second-order algorithm might perform solving an F-Lipschitz problem. An example of this is the heavy ball method which is applied in a distributed setup in [17].

7 Conclusion

This work presented a novel toolbox in which the characteristics of both convex and non-convex problems may be investigated. It was argued that problems with certain properties may be verified F-Lipschitz with a computational complexity of $O(n^2)$. This claim was supported by the simulation of several different examples and it was shown that these properties are not unusual for typical problems within wireless sensor networks. Moreover, the results indicated that this reduction of computational complexity is essential if the F-Lipschitz algorithms are to compete with the centralized Lagrangian methods.

It has however become clear that the F-Lipschitz distributed algorithm is far superior than the Lagrangian decomposition methods. A comparison can be made to the example referred to in Section 2.4 for which convergence was reached in 100 - 200 iterations for a problem with only 3 decision variables. In some situations, the structure and parameters of a problem may be of such nature that the F-Lipschitz property is always maintained. The verification process is in other words not necessary, thus in these cases both the F-Lipschitz algorithms are almost exclusively the better choice in terms of iterations. Remember that, in terms of time, the distributed algorithm is inevitably slower for each iteration since it is dependent on message passing between nodes.

It is evident that both of the algorithms scales quite well with problem size, particularly compared to the interior point method. One may conclude that the F-Lipschitz optimization has an advantage in avoiding the potentially complicated objective function in its algorithms. Though in that case it is often more of a challenge to show that the objective is valid, i.e., strictly increasing or decreasing. Evidently, the active set method has proven to be an exceedingly good solver for F-Lipschitz problems in centralized settings. However, we have seen that things can go very wrong when it is applied to more complicated non-convex problems, like for the norm computation example. Also, if it is faster only when the problem is F-Lipschitz, then we might argue that the verification process should be the precursor to this algorithm as well, which ultimately makes *fsolve* the superior algorithm. Conclusively, more testing is called for, both on problems which are F-Lipschitz as well as similar problems which are *not*.

References

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci. *"Wireless sensor networks: a survey"*, Computer Networks 38, 393-422, 2002.

[2] A. Speranzon, C. Fischione, K. H. Johansson, *"Distributed and collaborative estimation over wireless sensor networks"*, IEEE Conference on Decision and Control, 2006.

[3] A. Speranzon, C. Fischione, K. H. Johansson, A. Sangiovanni-Vincentelli, *"A Distributed Minimum Variance Estimator for Sensor Networks"*, IEEE Journal on Selected Areas in Communications, vol. 26, pp. 609 - 621, May 2008.

[4] A. Bonivento, C. Fischione, L. Necchi, F. Pianegiani, A. Sangiovanni-Vincentelli, *"System Level Design for Clustered Wireless Sensor Networks"*, IEEE Transactions on Industrial Informatics, vol. 3, no. 3, pp. 202 - 214, 2007.

[5] C. Fischione, U. Jönsson. *"Fast-Lipschitz Optimization with Wireless Sensor Network Applications"*, in Proc. of ACM/IEEE International Conference on Information Processing in Sensor Networks 2011, (ACM/IEEE IPSN 11), Chicago, IL, USA, April 2011.

[6] D. P. Bertsekas, J. N. Tsitsiklis. *"Parallel and Distributed Computation: Numerical Methods"*, Athena Scientific, 1997.

[7] B. Johansson. *"On distributed optimization in networked systems"*, Ph.D. dissertation, KTH, 2009.

[8] D. P. Bertsekas, A. Nedić, and A. E. Ozdaglar. *"Convex Analysis and Optimization"*, Athena Scientific, 2003.

[9] S. Boyd, L. Xiao, A. Mutapic, J. Mattingley. *"Notes on Decomposition Methods"*, Standford University, Notes for EE364B. [Online]. Available: http://www.stanford.edu/class/ee364b/lectures.html

[10] S. Boyd, S. J. Kim, L. Vandenberghe, A. Hassibi. *"A Tutorial on Geometric Programming"*, Optimization and Engineering, vol. 1, no. 1, p. 1, 2006.

[11] M. Chiang, C. W. Tan, D. P. Palomar, D. O'Neill and D. Julian. *"Power Control by Geometric Programming"*, IEEE Transactions on Wireless Communications, 2007.

[12] R.D Yates. *"A Framework for Uplink Power Control in Cellular Radio Systems"*, IEEE Journal on Selected Areas in Communications, vol. 13, pp. 1341 - 1347, September 1995.

[13] J. Herdtner, E. K. P. Chong. *"Analysis of a Class of Distributed Asynchronous Power Control Algorithms for Cellulcar Wireless Systems"*, IEEE Journal on Selected Areas in Communications, vol. 18, no. 3, pp. 436 - 446, March 2000.

[14] C. Fischione. *"Fast-Lipschitz Optimization with Wireless Sensor Network Applications"*, IEEE Transactions on Automatic Control, to Appear, 2011.

[15] M. Grant, S. Boyd, Y. Ye. *"Disciplined Convex Programming"*, Global Optimization: From Theory to Implementation, p. 155-210, Springer, 2006.

[16] S. Boyd, L, Vandenberghe. *"Convex Optimization"*, Published, 2004.

[17] R. Jäntti, S. L. Kim. *"Second-Order Power Control with Asymptotically Fast Convergence"*, IEEE Journal on Selected Areas in Communications, vol. 18, no. 3, March 2000.

A F-Lipschitz Optimization Toolbox

This appendix contains the documentation of all the inline functions provided by the F-Lipschitz Optimization Toolbox.

Functions for verifying the F-Lipschitz property of a standard form problem:

```
[f_lip, msg, tot_itr, func_count] =
isflip(handle_nabla_f_0, handle_nabla_F, m, n, lb, ub, data_adv)

[f_lip, msg, tot_itr, func_count] =
isfliplincon(handle_nabla_f_0, m, n, A, lb, ub, data_adv)

[f_lip, msg, tot_itr, func_count] =
isflipquadcon(handle_nabla_f_0, m, n, Q, r, lb, ub, data_adv)
```

The first function is for problems with a somewhat arbitrary structure, while the second and third are for problems with linear and quadratic constraints respectively. The first input is the function handle to the gradient of the objective. If your gradient function is called 'myfun', it is passed as '@myfun'. The first function also needs the function handle of the constraint gradient, *handle_nabla_F*, while the second function only needs the $n \times n$ matrix $\mathbf{A}$ which in itself represent the gradient of the linear constraints. For the last function, the constraint gradient is defined by the $n \times n \times n$ matrix $\mathbf{Q}$ and the $n \times n$ matrix $\mathbf{r}$. The matrix $\mathbf{Q}_i$ belonging to the constraint i is given by the following indexing, $\mathbf{Q}(:, :, i)$, while the vector r_i is correspondingly given as i-th *row* of $\mathbf{r}$.

The structure *data_adv* can be used to provide information about the problem, but it is not a necessary input. If used, it should have the following fields which should be equal to 1 if the statement is true and 0 otherwise

- data_adv.obj_strict - The objective is strictly increasing/decreasing.
- data_adv.obj_grad - The objective gradient is monotonic.
- data_adv.constr_grad - The constraint gradients are monotonic.
- data_adv.constr - The constraints are monotonic.

The remaining inputs n, m, lb and ub should be self-explanatory. Finally, the output *f_lip* is either 1 or 0 depending if the problem was found to be F-Lipschitz or not. *msg* is a report of the verification process, potentially an error message if something went wrong. The last two outputs is the total amount of iterations and function evaluations performed during the process.

Functions for verifying the F-Lipschitz property of a canonical form problem:

```
isflip_canonical
(handle_nabla_g_0, handle_nabla_G, m, n, lb, ub, data_adv)

isfliplincon_canonical(handle_nabla_g_0, m, n, A, lb, ub, data_adv)

isflipquadcon_canonical
(handle_nabla_g_0, m, n, Q, r, lb, ub, data_adv)
```

where the outputs have been omitted since they are the same as for the standard form functions. The inputs are the same as before, hence no further explanation is needed.

Functions for running the distributed algorithm:

```
[sol, itrs, x, done] = flipdistr(handle_F, x_0, lb, ub, ac)

[sol, itrs, x, done] = flipdistrlin(A, b, x_0, lb, ub, ac)

[sol, itrs, x, done] = flipdistrquad(Q, r, c, x_0, lb, ub, ac)

[sol, itrs, x, done] =
flipdistr_canonical(handle_G, handle_nabla_G, x_0, lb, ub, ac)

[sol, itrs, x, done] =
flipdistrlin_canonical(A, b, x_0, lb, ub, ac)

[sol, itrs, x, done] =
flipdistrquad_canonical(Q, r, c, x_0, lb, ub, ac)
```

Not surprising, the first three functions are algorithms for standard form problems, while the last three are for problems on canonical form. Also here, there are separate functions for problems with arbitrary structured constraints, as well as linear and quadratic constraints. The input *x_0* is the n-dimensional starting point, while *ac* is the desired accuracy which is not a necessary input since there is always a default accuracy of $1 \cdot 10^{-6}$. The remaining inputs all have the same meaning as above. Obviously, the output *sol* represents the solution values, *itrs* is a vector, $[1, 2, \ldots, t]$, where t is the amount of iterations, while x is an $n \times t$ matrix containing all the values of each x_i for each iterate of the algorithm. In other words, the last column of x also gives the solution values, i.e., if a solution was found. This is determined by the output *done*, which is 1 if the algorithm converged and 0 if not.

B MATLAB Code

Here follows the m-file functions used for the radio power allocation example discussed in Section 5.4:

```
function g = G(x)
    n = 10;
    G_ii = 10^((-70-30)/10);
    G_ij = 10^((-90-30)/10);
    M = 10^((-120-30)/10);
    S_min = 1;
    sigma = 10^((-130-30)/10);

    g_ = zeros(n,1);
    intf = G_ij.*x;
    intmod = M.*x.^2;
    for i = 1:n,
        sum1 = intf*ones(n,1) - G_ij*x(i);
        sum2 = x(i)^2*(intmod*ones(n,1)) - M*x(i)^4;
        g_(i,1) = G_ii*x(i) + S_min*(sigma - sum1 + sum2);
    end
    g = g_;
end

function g = nabla_G(x)
    n = 10;
    G_ii = 10^((-70-30)/10);
    G_ij = 10^((-90-30)/10);
    M = 10^((-120-30)/10);
    S_min = 1;

    g_ = zeros(n,n);
    intmod = M.*x.^2;
    for i = 1:n,
        for j = 1:n,
            if j == i
                sum = x(i)*(intmod*ones(n,1)) - M*x(i)^3;
                g_(i,i) = G_ii + 2*S_min*sum;
            else
                g_(i,j) = -S_min*G_ij + 2*S_min*M*x(i)^2*x(j);
            end
        end
    end
    g = g_;
end
```

Here follows the m-file functions used for the norm computation example discussed in Section 5.5:

```
function g = G(x)
    global THETA
    n = 10;
    gamma_max = 0.75;

    g_ = zeros(n,1);
    x_sqrt = sqrt(x);
    for i = 1:n
        g_(i,1) = x(i) + x_sqrt(i)*(THETA(i,:)*x_sqrt') - gamma_max;
    end
    g = g_;
end
```

```
function g = nabla_G(x)
    global THETA
    n = 10;

    g_ = zeros(n,n);
    x_sqrt = sqrt(x);
    for i = 1:n,
        for j = 1:n,
            if j == i
                g_(i,i) = 1 + (1/(2*x_sqrt(i)))*(THETA(i,:)*x_sqrt');
            else
                if THETA(i,j)
                    g_(i,j) = (1/(2*x_sqrt(j)))*x_sqrt(i);
                end
            end
        end
    end
    g = g_;
end
```

```
THETA =

     0     1     1     1     1     0     0     1     0     0
     1     0     1     1     1     0     0     0     0     0
     1     1     0     1     1     0     1     1     1     0
     1     1     1     0     1     0     0     1     0     0
     1     1     1     1     0     0     0     0     0     0
     0     0     0     0     0     0     1     1     0     1
     0     0     1     0     0     1     0     1     1     1
     1     0     1     1     0     1     1     0     1     1
     0     0     1     0     0     0     1     1     0     1
     0     0     0     0     0     1     1     1     1     0
```

Here follows the m-file functions used for the distributed detection example discussed in Section 5.6:

```
function g = g_0(x)
    n = 5;
    mu = 0;
    sigma = 1;
    S = 1;

    q = ones(1,n) - normcdf(x,mu,sigma/S);
    g = q*ones(n,1);
end

function g = nabla_g_0(x)
    sigma = 1;
    S = 1;

    v = (x.^2)./(2*sigma^2/S);
    exps = exp(-v);
    g = exps./-sqrt(2*pi*sigma^2/S);
end

function g = G(x)
    global WEIGHT
    n = 5;
    mu = 0;
    sigma = 1;
    S = 1;
    E = 2;
    c_i = 1/15;

    q = ones(1,n) - normcdf(ones(1,n).*E - x,mu,sigma/S);
    g = WEIGHT*q' - c_i*ones(n,1);
end
```

```
function g = nabla_G(x)
    global WEIGHT
    n = 5;
    sigma = 1;
    S = 1;
    E = 2;

    v = (ones(1,n).*E - (x.^2))./(2*sigma^2/S);
    exps = exp(-v);
    p_md = exps./sqrt(2*pi*sigma^2/S);

    g_ = zeros(n,n);
    for i = 1:n,
        g_(i,:) = WEIGHT(i,:).*p_md;
    end
    g = g_;
end
```

```
WEIGHT =

    0.9071    0.0228    0.0175    0.0526         0
    0.0543    0.8719    0.0354         0    0.0383
         0    0.0489    0.9078    0.0327    0.0106
    0.0032         0    0.0538    0.9282    0.0148
    0.0360    0.0130         0    0.0147    0.9363
```