Minimax Optimization without second order information

Mark Wrobel

LYNGBY 2003
EKSAMENSPROJEKT
NR. 6

IMM

Trykt af IMM, DTU

Preface

This M.Sc. thesis is the final requirement to obtaining the degree: *Master of Science in Engineering*. The work has been carried out in the period from the 1st of September 2002 to the 28th of February 2003 at the Numerical Analysis section at Informatics and Mathematical Modelling, Technical University of Denmark. The work has been supervised by Associate Professor Hans Bruun Nielsen and co-supervised by Professor, dr.techn. Kaj Madsen.

I wish to thank Hans Bruun Nielsen for many very useful and inspiring discussions, and for valuable feedback during this project. Also i wish to thank Kaj Madsen, for introducing me to the theory of minimax, and especially for the important comments regarding exact penalty functions.

I also wish to thank M.Sc. Engineering, Ph.D Jacob Søndergaard for interesting discussions about optimization in general and minimax in particular.

Finally i wish to thank my office colleges, Toke Koldborg Jensen and Harald C. Arnbak for making this time even more enjoyable, and for their daily support, despite their own heavy workload.

Last but not least, i wish to thank M.Sc. Ph.D student Michael Jacobsen for helping me proofreading.

Kgs. Lyngby, February 28th, 2003

Mark Wrobel c952453

Abstract

This thesis deals with the practical and theoretical issues regarding minimax optimization. Methods for large and sparse problems are investigated and analyzed. The algorithms are tested extensively and comparisons are made to Matlab's optimization toolbox. The theory of minimax optimization is thoroughly introduced, through examples and illustrations. Algorithms for minimax are trust region based, and different strategies regarding updates are given.

Exact penalty function are given an intense analysis, and theory for estimating the penalty factor is deduced.

Keywords*: Unconstrained and constrained minimax optimization, exact penalty functions, trust region methods, large scale optimization.*

Resumé

Denne afhandling omhander de praktiske og teoretiske emner vedrørende minimax optimering. Metoder for store og sparse problemer undersøges og analyseres. Algoritmerne gennemgår en grundig testning og sammenlignes med matlabs optimerings pakke. Teorien for minimax optimering introduceres igennem illustrationer og eksempler. Minimax optimeringsalgoritmer er ofte baseret på trust regioner og deres forskellige strategier for opdatering undersøges.

En grundig analyse af eksakte penalty funktioner gives og teorien for estimering af penalty faktoren σ udledes.

***Nøgleord**: Minimax optimering med og uden bibetingelser, eksakt penalty funktion, trust region metoder, optimering af store problemer.*

Contents

Chapter 1

Introduction

The work presented in this thesis, has its main focus on the theory behind minimax optimization. Further, outlines for algorithms to solve unconstrained and constrained minimax problems are given, that also are well suited for problems that are large and sparse.

Before we begin the theoretical introduction to minimax, let us look at the linear problem of finding the Fourier series expansion to fit some design specification. This is a problem that occur frequently in the realm of applied electrical engineering, and in this short introductory example we look at the different solutions that arise when using the ℓ_1, ℓ_2 and ℓ_∞ norm, i.e.

$$\begin{aligned} F(\mathbf{x}) &= \|\mathbf{f}(\mathbf{x})\|_1 = |f_1(\mathbf{x})| + \cdots + |f_m(\mathbf{x})| && (\ell_1) \\ F(\mathbf{x}) &= \|\mathbf{f}(\mathbf{x})\|_2^2 = f_1(\mathbf{x})^2 + \cdots + f_m(\mathbf{x})^2 && (\ell_2) \\ F(\mathbf{x}) &= \|\mathbf{f}(\mathbf{x})\|_\infty = \max\{|f_1(\mathbf{x})|, \ldots, |f_m(\mathbf{x})|\}\ , && (\ell_\infty) \end{aligned}$$

where $\mathbf{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$ is a vector function. For a description of the problem and its three different solutions the reader is referred to Appendix A. The solution to the Fourier problem, subject to the above three norms are shown in figure 1.1.

As shown in [MN02, Example 1.4], the three norms responds differently to "outliers" (points that has large errors), and without going into details, the ℓ_1 norm is said to be a robust norm, because the solution based on the ℓ_1 estimation is not affected by outliers. This behavior is also seen in figure 1.1, where the ℓ_1 solution fits the horizontal parts of the design specification (fat black line) quite well. The corresponding residual function shows that most residuals are near zero, except for some large residuals.

The ℓ_2 norm (least-squares) is a widely used and popular norm. It give rise to smooth optimization problems, and things tend to be simpler when using this norm. Unfortunately the solution based on the ℓ_2 estimation is not robust towards outliers. This is seen in figure 1.1 where the ℓ_2 solution shows ripples near the discontinuous parts of the design specification. The highest residuals are smaller than that of the ℓ_1 solution. This is because ℓ_2 also try to minimize the largest residuals, and hence the ℓ_2 norm is sensitive to outliers.

A development has been made, to create a norm that combines the smoothness of ℓ_2 with the robustness of ℓ_1. This norm is called the Huber norm and is described in [MN02, p. 41] and [Hub81].

The ℓ_∞ norm is called the Chebyshev norm, and minimizes the maximum distance between the data (design specification) and the approximating function, hence the name minimax approximation. The norm is not robust, and the lack of robustness is worse than that of ℓ_2. This is clearly shown in figure 1.1 were the ℓ_∞ solution shows large oscillations, however, the maximum residual is minimized and the residual functions shows no spikes as in the case of the ℓ_1 and ℓ_2 solutions.

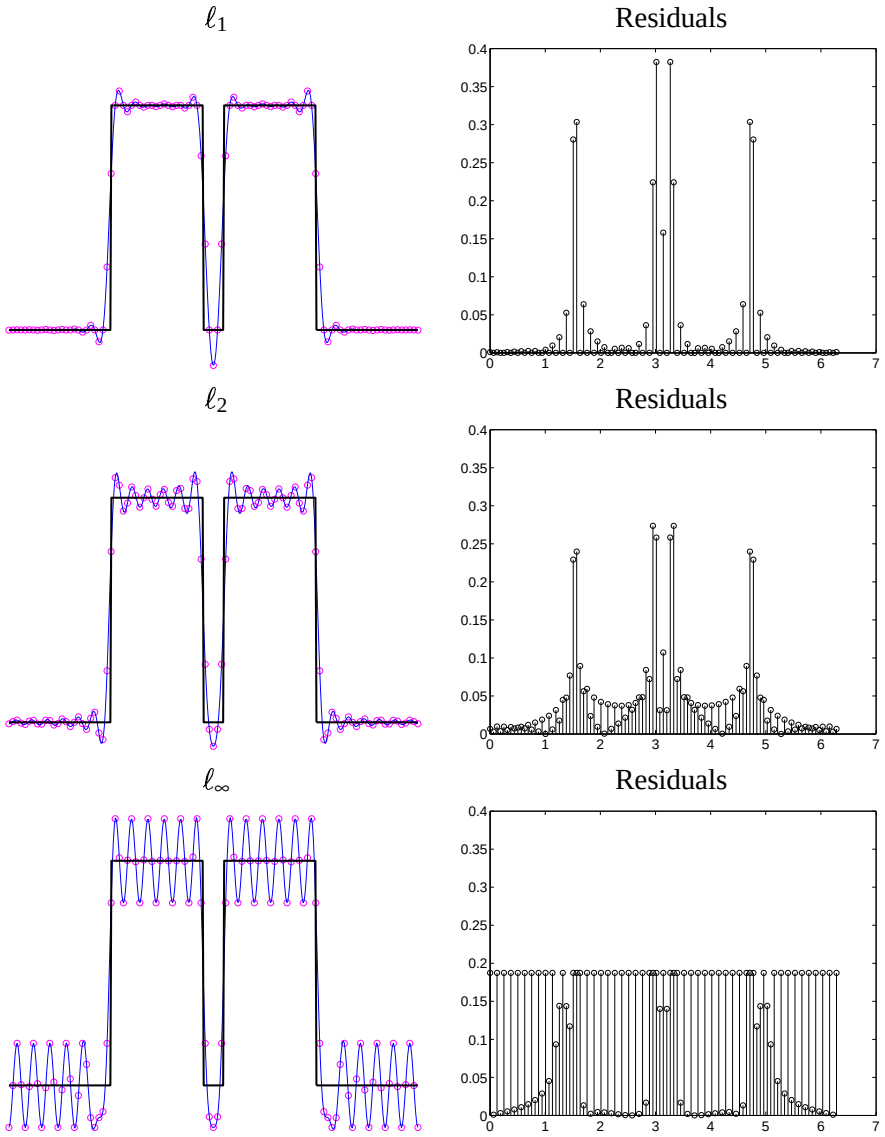

Figure 1.1: Left: Shows the different solution obtained by using an ℓ_1, ℓ_2 and ℓ_∞ estimator. Right: The corresponding residual functions. Only the ℓ_∞ case does not have any spikes in the residual function.

1.1 Outline

We start by providing the theoretical foundation for minimax optimiazation, by presenting *generalized gradients*, *directional derivatives* and fundamental propositions.

In chapter 3, the theoretical framework is applied to construct two algorithms that can solve unconstrained non-linear minimax problems. Further test results are discussed.

We present the theoretical basis for constrained minimax in chapter 4, which is somewhat similar to unconstrained minimax theory, but has a more complicated notation. In this chapter we also take a closer look at the exact penalty function, which is used to solve constrained problems.

Trust region strategies are discussed in chapter 5, where we also look at scaling issues.

Matlab's solver `linprog` is described in chapter 6, and various problems regarding `linprog`, encountered in the process of this work, is examined. Further large scale optimization, is also a topic for this chapter.

Chapter 2

The Minimax Problem

Several problems arise, where the solution is found by minimizing the maximum error. Such problems are generally referred to as minimax problems, and occur frequently in real life problems, spanning from circuit design to satellite antenna design. The minimax method finds its application on problems where estimates of model parameters, are determined with regard to minimizing the maximum difference between model output and design specification.

We start by introducing the minimax problem

$$\min_{\mathbf{x}} F(\mathbf{x}), \quad F(\mathbf{x}) \equiv \max_i f_i(\mathbf{x}), \quad i \in \{1,\ldots,m\}, \tag{2.1}$$

where $F : \mathbb{R}^m \mapsto \mathbb{R}$ is piecewise smooth, and $f_i(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}$ is assumed to be smooth and differentiable. A simple example of a minimax problem is shown in figure 2.1, where the solid line indicates $F(\mathbf{x})$.

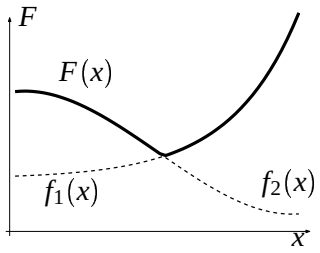

Figure 2.1: $F(x)$ (the solid line) is defined as $\max_i\{f_i(\mathbf{x})\}$. The dashed lines denote $f_i(\mathbf{x})$, for $i = 1,2$. The solution to the problem illustrated, lies at the kink between $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$.

By using the Chebyshev norm $\|\cdot\|_\infty$ a class of problems called Chebyshev approximation problems occur

$$\min_x F_\infty(\mathbf{x}), \quad F_\infty(\mathbf{x}) \equiv \|\mathbf{f}(\mathbf{x})\|_\infty, \tag{2.2}$$

where $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{f}(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$. We can easily rewrite (2.2) to a minimax problem

$$\min_x F_\infty(\mathbf{x}) \equiv \max\left\{ \max_i\{f_i(\mathbf{x})\}, \max_i\{-f_i(\mathbf{x})\} \right\}, \tag{2.3}$$

It is simple to see that (2.2) can be formulated as a minimax problem, but not vice versa. Therefore the following discussion will be based on the more general minimax formulation in (2.1). We will refer to the Chebyshev approximation problem as Chebyshev.

2.1 Introduction to Minimax

In the following we give a brief theoretical introduction to the minimax problem in the unconstrained case. To provide some "tools" to navigate with, in a minimax problem, we introduce terms like the *generalized gradient* and the *directional gradient*. At the end we formulate conditions for stationary points etc.

According to the definition (2.1) of minimax, F is not in general a differentiable function. Rather F will consist of piecewise differentiable sections, as seen in figure 2.1. Unfortunately the presence of kinks in F makes it impossible for us to define an optimum by using $F'(\mathbf{x}^*) = 0$ as we do in the 2-norm case, where the objective function is smooth.

In order to describe the generalized gradient, we need to look at all the functions that are active at $\mathbf{x}$. We say that those functions belongs to the *active set*

$$\mathcal{A} = \{ j : f_j(\mathbf{x}) = F(\mathbf{x}) \} \tag{2.4}$$

e.g. there are two active functions at the kink in figure 2.1, and everywhere else only one active function.

Because F is not always differentiable we use a different measure called the *generalized gradient* first introduced by [Cla75].

$$\begin{aligned} \partial F(\mathbf{x}) &= \mathtt{conv}\{ f_j'(\mathbf{x}) \mid j : f_j = F(\mathbf{x}) \} && (2.5) \\ &= \{ \sum_{j \in \mathcal{A}} \lambda_j f_j'(\mathbf{x}) \mid \sum_{j \in \mathcal{A}} \lambda_j = 1, \lambda_j \geq 0 \}, && (2.6) \end{aligned}$$

so $\partial F(\mathbf{x})$ defined by `conv` is the convex hull spanned by the gradients of the active inner functions. We see that (2.6) also has a simple geometric interpretation, as seen in figure 2.2 for the case $\mathbf{x} \in \mathbb{R}^2$.

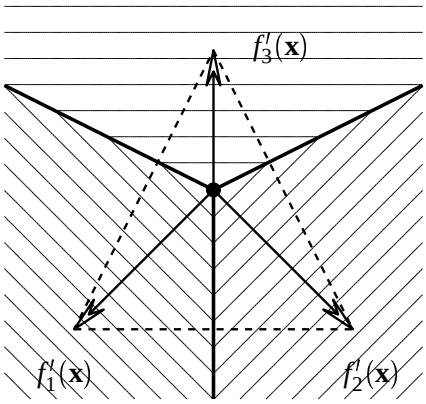

Figure 2.2: The contours indicate the minimax landscape of $F(\mathbf{x})$ whith the inner functions f_1, f_2 and f_3. The gradients of the functions are shown as arrows. The dashed lines show the border of the convex hull as defined in (2.6). This convex hull is also the generelized gradient $\partial F(\mathbf{x})$.

The formulation in (2.5) has no multipliers, but is still equivalent with (2.6) i.e. if $\mathbf{0}$ is not in the convex hull defined by (2.6) then $\mathbf{0}$ is also not in the convex hull of (2.5) and vice versa.

We get (2.6) by using the first order Kuhn-Tucker conditions for optimality. To show this, we first have to set up the minimax problem as a nonlinear programming problem.

$$\begin{array}{lrcl} \min_{x,\tau} & g(x,\tau) & = & \tau \\ \text{s.t.} & c_j(x,\tau) & \equiv & -f_j(\mathbf{x}) + \tau \geq 0 \end{array} \tag{2.7}$$

where $j = 1, \ldots, m$. One could imagine τ as being indicated by the thick line in figure 2.1. The constrains $f_j(\mathbf{x}) \leq \tau$ says that τ should be equal the largest function $f_j(\mathbf{x})$, which is in fact the minimax formulation $F(x) = g(x, \tau) = \tau$.

Then we formulate the Lagrangian function

$$\mathcal{L}(x, \tau, \lambda) = g(x, \tau) - \sum_{j=1}^{m} \lambda_j c_j(x, \tau) \tag{2.8}$$

By using the first order Kuhn-Tucker condition, we get

$$\mathcal{L}'(x, \tau, \lambda) = \mathbf{0} \;\Rightarrow\; \mathbf{g}'(x, \tau) = \sum_{j=1}^{m} \lambda_j \mathbf{c}_j'(x, \tau). \tag{2.9}$$

For the active constraints we have that $f_j = \tau$, and we say that those functions belong to the active set $\mathcal{A}$.

The inactive constraints are those for which $f_j < \tau$. From the theory of Lagrangian multipliers it is known that $\lambda_j = 0$ for $j \notin \mathcal{A}$. We can then rewrite (2.9) to the following system

$$\begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} = \sum_{j \in \mathcal{A}} \left(\lambda_j \begin{bmatrix} -\mathbf{f}_j'(\mathbf{x}) \\ 1 \end{bmatrix} \right) \tag{2.10}$$

which yield the following result

$$\sum_{j \in \mathcal{A}} \lambda_j f_j'(\mathbf{x}) = \mathbf{0} \quad , \quad \sum_{j \in \mathcal{A}} \lambda_j = 1 \; . \tag{2.11}$$

Further the Kuhn-Tucker conditions says that $\lambda_j \geq 0$. We have now explained the shift in formulation in (2.5) to (2.6).

Another kind of gradient that also comes in handy, is the *directional gradient*. In general it is defined as

$$g_{\mathbf{d}}'(\mathbf{x}) = \lim_{t \to 0} \{ \frac{g(\mathbf{x} + t\mathbf{d}) - g(\mathbf{x})}{t} \} \tag{2.12}$$

which for a minimax problem leads to

$$F_{\mathbf{d}}'(\mathbf{x}) = \max \{ \mathbf{f}_j'(\mathbf{x})^T \mathbf{d} \mid j \in \mathcal{A} \} \; . \tag{2.13}$$

This is a correct way to define the directional gradient, when we remember that $F(\mathbf{x})$ is a piecewise smooth function. Further, the directional gradient is a scalar. To illustrate this further, we need to introduce the theorem of strict separation which have a connection to convex hulls.

Theorem 2.1 *Let Γ and Λ be two nonempty convex sets in $\mathbb{R}^n$, with Γ compact and Λ closed. If Γ and Λ are disjoint, then there exists a plane*

$$\{ \mathbf{x} \mid x \in \mathbb{R}^n, \mathbf{d}^T \mathbf{x} = \alpha \} \; , \; \mathbf{d} \neq \mathbf{0} \; ,$$

which strictly separates them, and conversely. In other words

$$\Gamma \cap \Lambda = \emptyset \Leftrightarrow \begin{cases} \exists \mathbf{d} \text{ and } \alpha \\ \mathbf{x} \in \Gamma \Rightarrow \mathbf{d}^T \mathbf{x} < \alpha \\ \mathbf{x} \in \Lambda \Rightarrow \mathbf{d}^T \mathbf{x} > \alpha \end{cases}$$

Proof: [Man65, p. 50]

It follows from the proof to proposition 2.24 in [Mad86, p. 52], where the above theorem is used, that if $M(\mathbf{x}) \in \mathbb{R}^n$ is a set that is convex, compact and closed, and there exists a $\mathbf{d} \in \mathbb{R}^n$ and $\mathbf{v} \in M(\mathbf{x})$. Then $\mathbf{d}$ is separated from $\mathbf{v}$ if

$$\mathbf{d}^T\mathbf{v} < 0 \quad , \quad \forall \mathbf{v} \in M(\mathbf{x}) \ .$$

Without loss of generality $M(\mathbf{x})$ can be replaced with $\partial F(\mathbf{x})$. Remember that as a consequence of (2.6) $\mathbf{f}_j'(\mathbf{x}) \in \partial F(\mathbf{x})$, therefore $\mathbf{v}$ and $\mathbf{f}_j'(\mathbf{x})$ are interchangeable, so now an illustration of both the theorem and the generalized gradient is possible and can be seen in figure 2.3. The figure shows four situations where $\mathbf{d}$ can be interpreted as a decend direction if it fulfils the above theorem, that is $\mathbf{v}^T\mathbf{d} < 0$. The last plot on the figure shows a case where $\mathbf{d}$ can not be separated from M or $\partial F(\mathbf{x})$ for that matter, and we have a stationary point because $F_{\mathbf{d}}'(\mathbf{x}) \geq 0$ as a consequence of $\mathbf{d}^T\mathbf{v} \geq 0$, so there is no downhill direction.

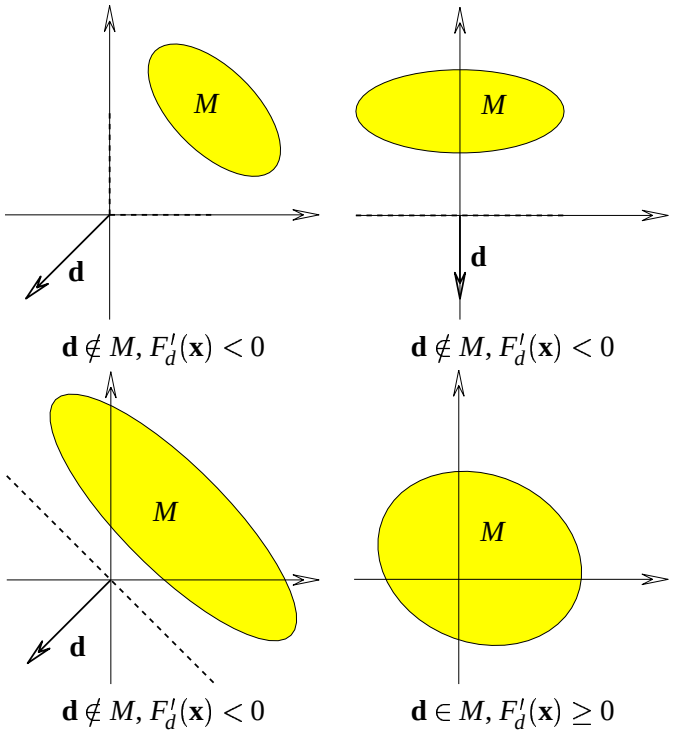

Figure 2.3: On the first three plots $\mathbf{d}$ is not in M. Further $\mathbf{d}$ can be interpreted as a descend direction, which leads to $F_d'(\mathbf{x}) < 0$. On the last plot, however, $\mathbf{0}$ is in M and $\mathbf{d}^T\mathbf{v} \geq 0$.

Figure 2.3 can be explained further. The vector $\mathbf{d}$ shown on the figure corresponds to a downhill direction. As described in detail in chapter 3, $\mathbf{d}$ is found by solving an LP problem, where the solver tries to find a downhill direction in order to reduce the cost function. In the following we describe when such a downhill direction does not exist.

The dashed lines in the figure corresponds to the extreme extent of the convex hull $M(\mathbf{x})$. That is, when M grows so it touches the dashed lines on the coordinate axes. We will now describe what happens when M has an extreme extent.

In the first plot (top left) there must be an active inner function where $\mathbf{f}'_j(\mathbf{x}) = \mathbf{0}$ if $M(\mathbf{x})$ is to have an extreme extend. This means that one of the active inner functions must have a local minimum (or maximum) at $\mathbf{x}$. By using the strict separation theorem 2.1 we see that $\mathbf{v}^T\mathbf{d} = 0$ in this case. Still however it must hold that $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$ because $F(\mathbf{x}) = \max\{\mathbf{f}_j(\mathbf{x})\}$ is a convex function. This is illustrated in figure 2.4, where the gray box indicate an area where $F(\mathbf{x})$ is constant.

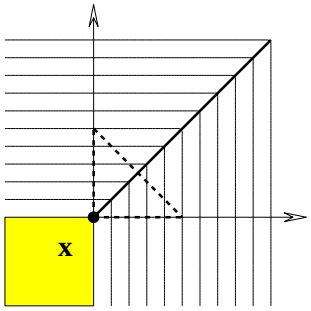

Figure 2.4: The case where one of the active functions has a zero gradient. Thus $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$. The dashed line indicate $\partial F(\mathbf{x})$. The gray box indicate an area where $F(\mathbf{x})$ is constant.

In the next two plots (top right) (bottom left), we see by using theorem 2.1 and the definition of the directional gradient, that $\mathbf{v}^T\mathbf{d} = 0$, and hence $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$, since $F(\mathbf{x})$ is a convex function. As shown in the following, we have a stationary point if $\mathbf{0} \in M(\mathbf{x})$.

In the last case in figure 2.3 (bottom right) we have that $\mathbf{v}^T\mathbf{d} \geq 0$ which leads to $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$. In this case $\mathbf{0}$ is in the *interior* of $M(\mathbf{x})$. As described later, this situations corresponds to $\mathbf{x}$ being a strongly unique minimum. Unique in the sense that the minimum can only be a point, and not a line as illustrated in figure 2.6, or a plane as seen in figure 2.4.

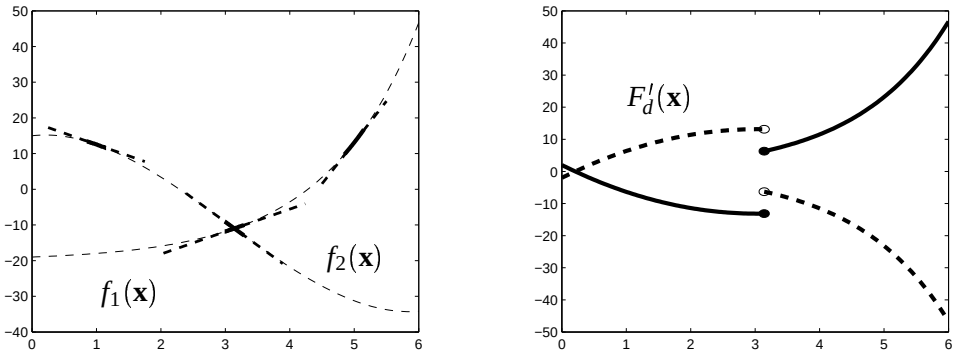

Figure 2.5: Left: Visualization of the gradients in three points. The gradient is ambiguous at the kink, but the generalized gradient is well defined. Right: The directional derivative for $\mathbf{d} = -1$ (dashed lines) and $\mathbf{d} = 1$ (solid lines).

We illustrate the directional gradient further with the example shown in figure 2.5, where the directional gradient is illustrated in the right plot for $\mathbf{d} = -1$ (dashed lines) and $\mathbf{d} = 1$ (solid lines).

Because the problem is one dimensonal, and $\|\mathbf{d}\| = 1$, it is simple to illustrate the relationship between $\partial F(\mathbf{x})$ and $F'_{\mathbf{d}}(\mathbf{x})$. We see that there are two stationary points at $x \approx 0.2$ and at $x \approx 3$. At $x \approx 0.2$ the the convex hull generalized gradient must be a point, because there is only one active function, but it still holds that $\mathbf{0} \in \partial F(\mathbf{x})$, so the point is stationary. In fact it

is a local maximum as seen on the left plot. Also there exists a $\mathbf{d}$ for which $F'_{\mathbf{d}}(\mathbf{x}) < 0$ and therefore there is a downhill direction.

At $x \approx 3$ the generalized gradient is an interval. The interval is illustrated by the black circles for $d = 1$ ($\partial F(\mathbf{x})$) and by the white circles for $d = -1$ ($-\partial F(\mathbf{x})$). It is seen that both the intervals have the property that $\mathbf{0} \in \partial F(\mathbf{x})$, so this is also a stationary point. Further it is also a local minimum as seen on the left plot, because it holds for all directions that $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$ illustrated by the top white and black circle. In fact, it is also a strongly unique local minimum because $\mathbf{0}$ is in the *interior* of the interval. All this is described and formalized in the following.

2.1.1 Stationary Points

Now we have the necessary tools to define a stationary point in a minimax context. An obvious definition of a stationary point in 2-norm problems would be that $F'(\mathbf{x}) = 0$. This however, is not correct for minimax problems, because we can have "kinks" in F like the one shown in figure 2.1. In other words F is only a piecewise differentiable function, so we need another criterion to define a stationary point.

Definition 2.1 $\mathbf{x}$ *is a stationary point if*

$$\mathbf{0} \in \partial F(\mathbf{x})$$

This means that if the null vector is inside *or* at the border of the convex hull of the generalized gradient $\partial F(\mathbf{x})$, then we have a stationary point. We see from figure 2.2 that the null vector is inside the convex hull. If we removed, say $f_3(\mathbf{x})$ from the problem shown on the figure, then the convex hull $\partial F(\mathbf{x})$ would be the line segment between $f'_1(\mathbf{x})$ and $f'_2(\mathbf{x})$. If we removed a function more from the problem, then $\partial F(\mathbf{x})$ would collapse to a point.

Proposition 2.1 *Let* $\mathbf{x} \in \mathbb{R}^n$. *If* $\mathbf{0} \in \partial F(\mathbf{x})$ *Then it follows that* $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$

Proof: See [Mad86, p. 29]

In other words the proposition says that there is no downhill directions from a stationary point. The definition give rise to an interesting example where the stationary point is in fact a line, as shown in Figure 2.6. We can say that at a stationary point it will always hold that

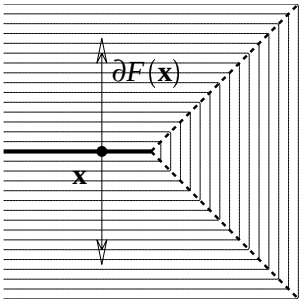

Figure 2.6: The contours show a minimax landscape, where the dashed lines show a kink between two functions, and the solid line indicate a line of stationary points. The arrows show the convex hull at a point on the line.

the directional derivative is zero or positive. A proper description has now been given about

stationary points. Still, however, there remains the issue about the connection between a local minimum of F and stationary points.

Proposition 2.2 *Every local minimum of F is a stationary point.*

Proof: See [Mad86] p.28.

This is a strong proposition and the proof uses the fact that F is a convex function.

As we have seen above and from figure 2.6, a minimum of $F(\mathbf{x})$ can be a line. Another class of stationary points have the property that they are unique, when only using first order derivatives. They can not be lines. These points are called *Strongly unique local minima*.

2.1.2 Strongly Unique Local Minima.

As described in the previous, a stationary point is not necessarily unique, when described only by first order derivatives. For algorithms that do not use second derivatives, this will lead to slow final convergence. If the algorithm uses second order information, we can expect fast (quadratic) convergence in the final stages of the iterations, even though this also to some extent depends on the problem.

But when is a stationary point unique in a first order sense? A strongly unique local minima can be characterized by only using first order derivatives. which gives rise to the following proposition.

Proposition 2.3 *For $\mathbf{x} \in \mathbb{R}^n$ we have a strongly unique local minimum if*

$$\mathbf{0} \in \text{int}\{\partial F(\mathbf{x})\}$$

Proof: [Mad86, p. 31]

The mathematical meaning of int$\{\cdot\}$ in the above definition, is that the null vector should be *interior* to the convex hull $\partial F(\mathbf{x})$. A situation where this criterion is fulfilled is shown in figure 2.2, while figure 2.6 shows a situation where the null vector is situated at the *border* of the convex hull. So this is *not* a strongly unique local minimum, because it is not interior to the convex hull.

Another thing that characterizes a point that is a strongly unique local minima is that the directional gradient is strictly uphill $F'_{\mathbf{d}}(\mathbf{x}) > 0$ for all directions $\mathbf{d}$.

If $C \in \mathbb{R}^n$ is a convex set, and we have a vector $\mathbf{z} \in \mathbb{R}^n$ then

$$\mathbf{z} \in \text{int}\{C\} \;\Leftrightarrow\; \mathbf{z}^T\mathbf{d} < \sup\{\mathbf{g}^T\mathbf{d} \mid \mathbf{g} \in C\}\,, \mathbf{d} \neq \mathbf{0}\,. \tag{2.14}$$

where, $\mathbf{d} \in \mathbb{R}^n$. The equation must be true, because there exists a $\mathbf{g}$ where $\|\mathbf{g}\| > \|\mathbf{z}\|$. This is illustrated in figure 2.7.

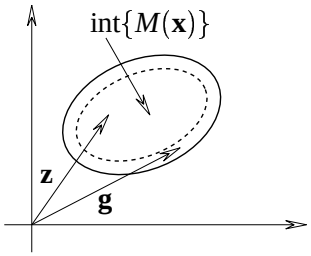

Figure 2.7: Because **z** is confined to the interior of M, **g** can be a longer vector than **z**. Then It must be the case that the maximum value of inner product $\mathbf{g}^T\mathbf{d}$ can be larger than $\mathbf{z}^T\mathbf{d}$.

If we now say that $\mathbf{z} = \mathbf{0}$, and $C = \partial F(\mathbf{x})$ then (2.14) implies that

$$\mathbf{0} \in \text{int}\{\partial F(\mathbf{x})\} \;\Leftrightarrow\; \mathbf{0} < \max\{\mathbf{g}^T\mathbf{d} \mid \mathbf{g} \in \partial F(\mathbf{x})\}\,, \mathbf{d} \neq \mathbf{0}\,. \tag{2.15}$$

We see that the definition of the directional derivative corresponds to

$$F'_{\mathbf{d}}(\mathbf{x}) = \max\{\mathbf{g}^T\mathbf{d} \mid \mathbf{g} \in \partial F(\mathbf{x})\} \;\Rightarrow\; F'_{\mathbf{d}}(\mathbf{x}) > 0\,. \tag{2.16}$$

If we look at the strongly unique local minimum illustrated in figure 2.2 and calculate $F'_{\mathbf{d}}(\mathbf{x})$ where $\|\mathbf{d}\| = 1$ for all directions, then we will see that $F'_d(\mathbf{x})$ is strictly positive as shown in figure 2.8.

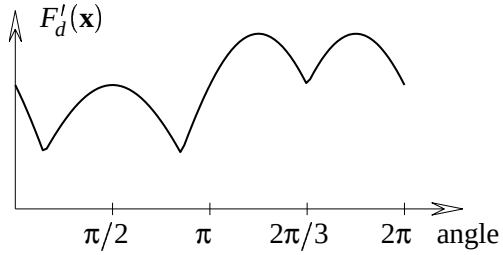

Figure 2.8: $F'_d(\mathbf{x})$ corresponding to the landscape in figure 2.2 for all directions **d** from 0 to 2π, where $\|\mathbf{d}\| = 1$.

From figure 2.8 we see that $F'_d(\mathbf{x})$ is a continuous function of **d** and that

$$F'_d(\mathbf{x}) > K\|\mathbf{d}\|\,,\ \text{where}\quad K = \inf\{\ F'_d(\mathbf{x}) \mid \|\mathbf{d}\| = 1\ \} > 0 \tag{2.17}$$

An interesting consequence of proposition 2.3 is that if $\partial F(\mathbf{x})$ is more than a point and $\mathbf{0} \in \partial F(\mathbf{x})$ then first order information will suffice from *some* directions to give final convergence. From other direction we will need second order information to obtain fast final convergence. This can be illustrated by figure 2.9. If the decent direction (d_1) towards the stationary line is perpendicular to the convex hull (indicated on the figure as the dashed line) then we need second order information. Otherwise we will have a kink in F which will give us a distinct indication of the minimum only by using first order information.

We illustrate this by using the parabola test function, where $\mathbf{x}^* = [0,0]^T$ is a minimizer, see figure 2.9. When using an SLP like algorithm[1] and a starting point $\mathbf{x}^{(0)} = [0,t]$ where $t \in \mathbb{R}$, then the minimum can be identified by a kink in F. Hence first order information will suffice to give us fast convergence for direction d_2. For direction d_1 there is no kink in F. If $\mathbf{x}^{(0)} = [t,0]$, hence a slow final convergence is obtained. If we do not start from $\mathbf{x}^{(0)} = [0,t]$

[1] Introduced in Chapter 3

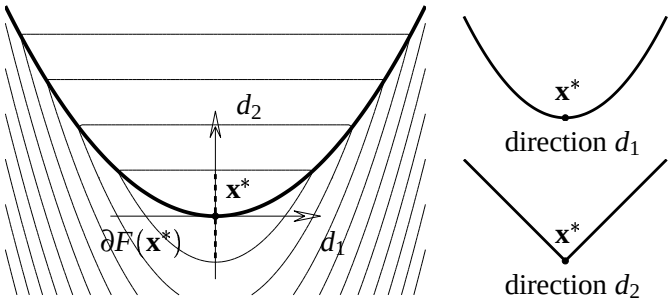

Figure 2.9: Left: A stationary point $\mathbf{x}^*$, where the convex hull $\partial F(\mathbf{x}^*)$ is indicated by the dashed line. Right: A neighbourhood of $\mathbf{x}^*$ viewed from direction d_1 and d_2. For direction d_1 there is no kink in F, while all other direction will have a kink in F.

then we will eventually get a decent direction that is parallel with the direction d_1 and get slow final convergence.

This property that the direction can have a influence on the convergence rate is stated in the following proposition. In order to understand the proposition we need to define what is meant by *relative interior*.

A vector $\mathbf{x} \in \mathbb{R}^n$ is said to be relative interior to the convex set $\partial F(\mathbf{x})$, if $\mathbf{x}$ is interior to the affine hull of $\partial F(\mathbf{x})$.

The Affine hull of ∂F is described by the following. If ∂F is a point, then aff$\{\partial F\}$ is also that point. If ∂F consists of two points then aff$\{\partial F\}$ is the line trough the two points. Finally if ∂F consists of three points, then aff$\{\partial F\}$ is the plane spanned by the three points. This can be formulated more formally by

$$\text{aff}\{\partial F(\mathbf{x})\} = \{\sum_{j=1}^{m} \lambda_j f_j'(\mathbf{x}) \mid \sum_{j=1}^{m} \lambda_j = 1\}. \tag{2.18}$$

Note that the only difference between (2.6) and (2.18) is that the constraint $\lambda_j \geq 0$ has been omitted in (2.18)

Definition 2.2 $\mathbf{z} \in \mathbb{R}^n$ *is relatively interior to the convex set* $S \in \mathbb{R}^n$ $\mathbf{z} \in \text{ri}\{S\}$ *if* $\mathbf{z} \in \text{int}\{\text{aff}\{S\}\}$.

[Mad86, p. 33]

If e.g. $S \in \mathbb{R}^n$ is a convex hull consisting of two points, then aff$\{S\}$ is a line. In this case $\mathbf{z} \in \mathbb{R}^n$ is said to be relatively interior to S if $\mathbf{z}$ is a point on that line.

Proposition 2.4 *For* $\mathbf{x} \in \mathbb{R}^n$ *we have*

$$\mathbf{0} \in \text{ri}\{\partial F(\mathbf{x})\} \Leftrightarrow \begin{cases} F_{\mathbf{d}}'(\mathbf{x}) = 0 & \text{if } \mathbf{d} \perp \partial F(\mathbf{x}) \\ F_{\mathbf{d}}'(\mathbf{x}) > 0 & \text{otherwise} \end{cases}, \tag{2.19}$$

Proof: [Mad86] p. 33.

The proposition says that every direction that is not perpendicular to ∂F will have a kink in F. A strongly unique local minimum can also be expressed in another way by looking at the *Haar* condition.

Definition 2.3 *Let F be a piecewise smooth function. Then it is said that the Haar condition is satisfied at $\mathbf{x} \in \mathbb{R}^n$ if any subset of*

$$\{ f_j'(\mathbf{x}) \mid j \in \mathcal{A} \}$$

has maximal rank.

By looking at figure 2.2 we see that for $\mathbf{x} \in \mathbb{R}^2$ it holds that we can only have a strongly unique local minimum $\mathbf{0} \in \text{int}\{\partial F(\mathbf{x})\}$ if at least three functions are active at $\mathbf{x}$. If this was not the case then the null vector could not be an interior point of the convex hull. This is stated in the following proposition.

Proposition 2.5 *Suppose that $F(\mathbf{x})$ is piecewise smooth near $\mathbf{x} \in \mathbb{R}^n$, and that the Haar condition holds at $\mathbf{x}$. Then if $\mathbf{x}$ is a stationary point, it follows that at least $n+1$ surfaces meet at $\mathbf{x}$. This means that $\mathbf{x}$ is a strongly unique local minimum.*

Proof: [Mad86] p. 35.

2.1.3 Strongly Active Functions.

To define what is meant by a degenerate stationary point, we first need to introduce the definition of a strongly active function. At a stationary point $\mathbf{x}$, the function $f_j(\mathbf{x})$ is said to be strongly active if

$$j \in \mathcal{A} \quad , \quad \mathbf{0} \notin \text{conv}\{ f_k'(\mathbf{x}) \mid k \in \mathcal{A}, k \neq j \}. \tag{2.20}$$

If $f_k(\mathbf{x})$ is a strongly active function at a stationary point and if we remove it, then $\mathbf{0} \notin \partial F(\mathbf{x})$. So by removing $f_k(\mathbf{x})$ the point $\mathbf{x}$ would no longer be a stationary point. This is illustrated in figure 2.10 for $\mathbf{x} \in \mathbb{R}^2$.

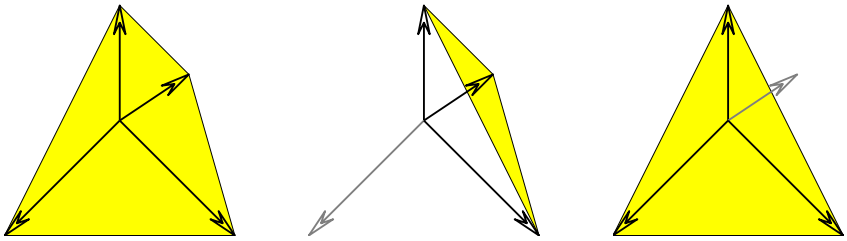

Figure 2.10: Left: The convex hull spanned by the gradients of four active functions. Middle: We have removed a strongly active function, so that $\mathbf{0} \notin \partial F(\mathbf{x})$. Right: An active function has been removed, still $\mathbf{0} \in \partial F(\mathbf{x})$.

If we remove a function $f_j(\mathbf{x})$ that is not strongly active, then it holds that $\mathbf{0} \in \partial F(\mathbf{x})$. In this case $\mathbf{x}$ is still a stationary point and therefore still a minimizer. If not every active function

at a stationary point is strongly active, then that stationary point is called degenerate. Figure 2.10 (left) is a degenerate stationary point.

The last topic we will cover here, is if the local minimizer $\mathbf{x}^*$ is located on a smooth function. In other words if $F(\mathbf{x})$ is differentiable at the local minimum, then $\mathbf{0} \in \partial F(\mathbf{x})$ is reduced to $\mathbf{0} = F'(\mathbf{x})$. In this case the convex hull collapses to a point, so there would be no way for $\mathbf{0} \in \text{int}\{\partial F(\mathbf{x})\}$. This means that such a stationary point can *not* be a strongly unique local minimizer, and hence we can not get fast final convergence towards such a point without using second order derivatives.

At this point we can say that the kinks in $F(\mathbf{x})$ help us. In the sense, that it is because of those kinks, that we can find a minimizer by only using first order information and still get quadratic final convergence.

Chapter 3

Methods for Unconstrained Minimax

In this chapter we will look at two methods that only use first order information to find a minimizer of an unconstrained minimax problem. The first method (SLP) is a simple trust region based method that uses sequential linear programming to find the steps toward a minimizer.

The second method (CSLP) is based upon SLP but further uses a corrective step based on first order information. This corrective step is expected to give a faster convergence towards the minimizer.

At the end of this chapter the two methods are compared on a set of test problems.

3.1 Sequential Linear Programming (SLP)

In its basic version, SLP solves the nonlinear programming problem (NP) in (2.7) by solving a sequence of linear programming problems (LP). That is, we find the minimax solution by only using first order information. The nonlinear constraints of the NP problem are approximated by a first order Taylor expansion

$$\mathbf{f}(\mathbf{x}+\mathbf{h}) \simeq \ell(\mathbf{h}) \equiv \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\mathbf{h} \;, \tag{3.1}$$

where $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^m$ and $\mathbf{J}(\mathbf{x}) \in \mathbb{R}^{m\times n}$. By combining the framework of the NP problem with the linearization in (3.1) and a trust region, we define the following LP subproblem

$$\begin{array}{lrcl} \min_{\mathbf{h},\alpha} & g(\mathbf{h},\alpha) \equiv \alpha & & \\ \text{s.t.} & \mathbf{f}+\mathbf{J}\mathbf{h} & \leq & \alpha \;, \\ & \|\mathbf{h}\|_\infty & \leq & \eta \end{array} \tag{3.2}$$

where $\mathbf{f}(\mathbf{x})$ and $\mathbf{J}(\mathbf{x})$ is substituted by $\mathbf{f}$ and $\mathbf{J}$. The last constraint in (3.2) might at first glance seem puzzling, because it has no direct connection to the NP problem. This is, however, easily explained. Because the LP problem only uses first order information, it is likely that the LP landscape will have no unique solution, like the situation shown in Figure 3.3 (middle). That is $\alpha \to -\infty$ for $\|\mathbf{h}\| \to \infty$. The introduction of a *trust region* eliminates this problem.

By having $\|\mathbf{h}\|_\infty \leq \eta$ we define a trust region that our solution $\mathbf{h}$ should be within. That is, we only trust the linarization up to a length η from $\mathbf{x}$. This is reasonable when remembering that the Taylor approximation is only good in some small neighbourhood of $\mathbf{x}$.

We use the Chebyshev norm ℓ_∞ to define the trust region, instead of the more intuitive Euclidean distance norm ℓ_2. That is because the Chebyshev norm is an easy norm to use in connection with LP problems, because we can implement it as simple bounds on $\mathbf{h}$. Figure 3.1 shows the trust region in $\mathbb{R}^2$ for the ℓ_1, ℓ_2 and the ℓ_∞ norm.

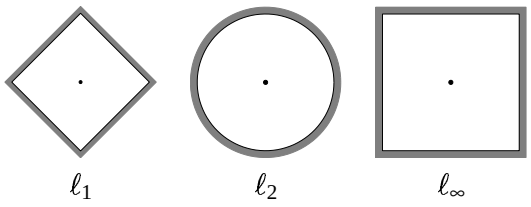

Figure 3.1: Three different trust regions, based on three different norms; ℓ_1, ℓ_2 and ℓ_∞. Only the latter norm can be implemented as bounds on the free variables in an LP problem.

Another thing that might seem puzzling in (3.2) is α. We can say that α is just the linearized equivalent to τ in the nonlinear programming problem (2.7). Equivalent to τ, the LP problem says that α should be equal to the largest of the linearized constraints $\alpha = \max\{\ell(\mathbf{h})\}$.

An illustration of $\alpha(\mathbf{h})$ is given in figure 3.2, where $\alpha(\mathbf{h})$ is the thick line. If the trust region is large enough, the solution to $\alpha(\mathbf{h})$ will lie at the kink of the thick line. But it is also seen that the kink is not the same as the real solution (the kink of the dashed lines, for the non-linear functions). This explains why we need to solve a sequence of LP subproblems in order to find the real solution.

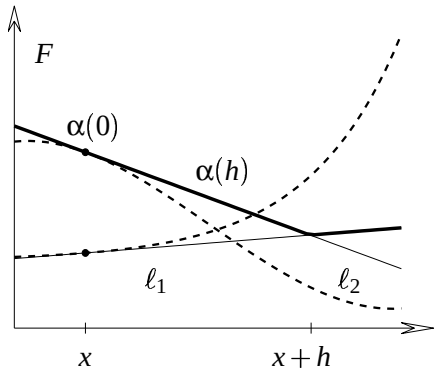

Figure 3.2: The objective is to minimize $\alpha(\mathbf{h})$ (the thick line). If $\|\mathbf{h}\|_\infty \leq \eta$, then the solution to $\alpha(\mathbf{h})$ is at the kink in the thick line. $\ell_1(\mathbf{h})$ and $\ell_2(\mathbf{h})$ are the thin lines.

For Chebyshev minimax problems $F_\infty = \max\{\mathbf{f}, -\mathbf{f}\}$, a similar strategy of sequentially solving LP subproblems can be used, just as in the minimax case. We can then write the Chebyshev LP subproblem as the following

$$\begin{array}{lrcl} \min_{\mathbf{h},\alpha} & g(\mathbf{h},\alpha) & \equiv & \alpha \\ \text{s.t.} & \mathbf{f}+\mathbf{J}_f\mathbf{h} & \leq & \alpha \\ & -\mathbf{f}-\mathbf{J}_f\mathbf{h} & \leq & \alpha \\ & \|\mathbf{h}\|_\infty & \leq & \eta \end{array}, \tag{3.3}$$

In section 2 we introduced two different minimax formulations (2.1) and (2.3). Here we will investigate the difference between them, seen from an LP perspective.

The Chebyshev formulation in (2.3) gives rise to a mirroring of the LP constraints as seen in (3.3) while minimax (2.1) does not, as seen in (3.2). This means that the two formulations have different LP landscapes as seen in figure 3.3. The LP landscapes are from the linearized parabola[1] test function evaluated in $\mathbf{x} = [-1.5,\ 9/8]^T$. For the parabola test function it holds that both the solution and the nonlinear landscape is the same, if we view it as a minimax or a Chebyshev problem. The LP landscape corresponding to the minimax problem have no unique minimizer, whereas the Chebyshev LP landscape does in fact have a unique minimizer.

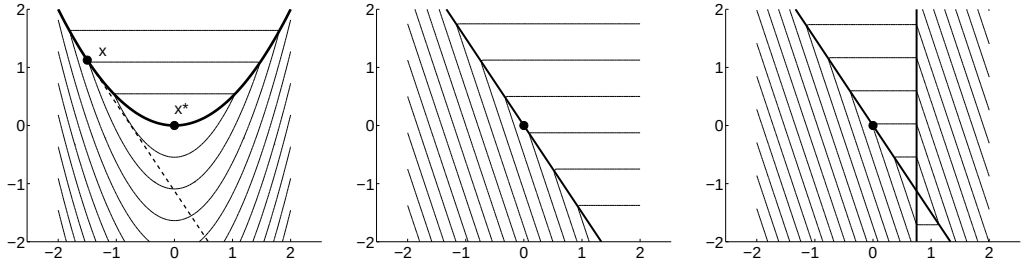

Figure 3.3: The two LP landscapes from the linarized parabola test function evaluated at $\mathbf{x} = [-1.5, 9/8]^T$. Left: The nonlinear landscape. Middle: The LP landscape of the minimax problem. We see that there is no solution. Right: For the Chebyshev problem we have a unique solution.

In this theoretical presentation of SLP we have not looked at the constrained case of minimax, where $F(\mathbf{x})$ is minimized subject to some nonlinear constraints. It is possible to give a formulation like (3.2) for the constrained case, where first order Taylor expansions of the constraints are taken into account. This is described in more detail in chapter 4.

3.1.1 Implementation of the SLP Algorithm

We will in the following present an SLP algorithm that solves the minimax problem, by approximating the nonlinear problem in (2.7) by sequential LP subproblems (3.2). As stated above, a key part of this algorithm, is an LP solver. We will in this text use Matlab's LP solver `linprog` `ver.1.22`, but still keep the description as general as possible.

In order to use `linprog`, the LP subproblem (3.2) has to be reformulated to the form

$$\min_{\hat{\mathbf{x}}} \mathbf{g}'(\mathbf{h}, \alpha)^T \hat{\mathbf{x}} \quad \text{s.t.} \quad \mathbf{A}\hat{\mathbf{x}} \leq \mathbf{b}\ . \tag{3.4}$$

This is a standard format used by most LP solvers. A further description of `linprog` and its various settings are given in Chapter 6. By comparing (3.2) and (3.4) we get

$$\mathbf{g}' = \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix},\ \mathbf{A} = [\mathbf{J}(\mathbf{x})\ \ -\mathbf{e}],\ \hat{\mathbf{x}} = \begin{bmatrix} \mathbf{h} \\ \alpha \end{bmatrix},\ \mathbf{b} = -\mathbf{f}(\mathbf{x})\ , \tag{3.5}$$

where $\hat{\mathbf{x}} \in \mathbb{R}^{n+1}$ and $\mathbf{e} \in \mathbb{R}^m$ is a column vector of all ones and the trust region in (3.2) is implemented as simple bounds on $\hat{\mathbf{x}}$.

[1] A description of the test functions are given in Appendix C.

We solve (3.4) with `linprog` by using the following setup.

$$\begin{array}{l} \mathbf{LB} := [-\eta\mathbf{e};\ -\infty]; \quad \mathbf{UB} := -\mathbf{LB}; \\ \text{Calculate: } \mathbf{g}' \text{ and } \mathbf{A}, \mathbf{b} \text{ by (3.5).} \\ [\ \hat{\mathbf{x}}, \ldots\ ] := \texttt{linprog}(\mathbf{g}', \mathbf{A}, \mathbf{b}, \mathbf{LB}, \mathbf{UB}, \ldots);\ , \end{array} \tag{3.6}$$

where $\mathbf{e} \in \mathbb{R}^n$ is a vector of ones.

If we were to solve the Chebyshev problem (2.2) we should change $\mathbf{A}$ and $\mathbf{b}$ in (3.5) to

$$\mathbf{A} = \begin{bmatrix} \mathbf{J}_f & -\mathbf{e} \\ -\mathbf{J}_f & -\mathbf{e} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} -\mathbf{f} \\ \mathbf{f} \end{bmatrix}, \tag{3.7}$$

The implementation of a minimax solver can be simplified somewhat, by only looking at minimax problems. One should realize that the Chebyshev problems can be solved in such a solver, by substituting $\mathbf{f}$ and $\mathbf{J}$ by $[\,\mathbf{f}\ -\mathbf{f}\,]$ and $[\,\mathbf{J}\ -\mathbf{J}\,]$.

An Acceptable step has the property that it is downhill and that the linarization is a good approximation to the non-linear functions. We can formulate this by by looking at the linearized minimax function

$$L(\mathbf{x};\mathbf{h}) \equiv \max_i\{\ell_i\}, \quad i \in \{1,\ldots,m\}. \tag{3.8}$$

By looking at figure 3.2 and the LP-problem in (3.2) it should easily be recognized that

$$\min_{\mathbf{h}} L(\mathbf{x},\mathbf{h}) \quad \text{s.t.} \quad \|\mathbf{h}\|_\infty \le \eta \tag{3.9}$$

is equivalent with the LP problem in (3.2).

From figure 3.2 we see that it must be the case that $L(\mathbf{x};\mathbf{h}) = \alpha$. The gain predicted by linear model can then be written as

$$\begin{array}{rcl} \Delta L(\mathbf{x};\mathbf{h}) & \equiv & L(\mathbf{x};\mathbf{0}) - L(\mathbf{x};\mathbf{h}) \\ & = & F(\mathbf{x}) - \alpha \end{array}\ . \tag{3.10}$$

The gain in the nonlinear objective function can be written in a similar way as

$$\Delta F(\mathbf{x};\mathbf{h}) \equiv F(\mathbf{x}) - F(\mathbf{x}+\mathbf{h})\ , \tag{3.11}$$

where F is defined either by (2.1) or (2.3). This leads to the formulation of the *gain factor* ρ

$$\rho = \frac{\Delta F(\mathbf{x};\mathbf{h})}{\Delta L(\mathbf{x};\mathbf{h})}. \tag{3.12}$$

The SLP algorithm uses ρ to determine if a step should be accepted or rejected. If $\rho > \varepsilon$ then the step is accepted. In [JM94] it is proposed that ε is chosen so that $0 \le \varepsilon \le 0.25$. In practice we could use $\varepsilon = 0$, and many do that, but for convergence proofs, however, we need $\varepsilon > 0$.

If the gain factor $\rho \approx 1$ or higher, then it indicates that linear subproblem is a good approximation to the nonlinear problem, and if $\rho < \varepsilon$, then the opposite argument can be made.

The gain factor is used in the update strategy for the trust region radius η because it, as stated above, gives an indication of the quality of linear approximation to the nonlinear problem. We now present an update strategy proposed in [JM94], where the expansion of the trust region is regulated by the term $\rho_{old} > \varepsilon$, i.e.,

$$\begin{aligned} &\texttt{if}(\rho > 0.75) \texttt{ and } (\rho_{old} > \varepsilon) \\ &\quad \eta = 2.5 * \eta; \\ &\texttt{elseif } \rho < 0.25 \\ &\quad \eta = \eta/2; \end{aligned} \tag{3.13}$$

The update strategy says that the trust region should be increased only when ρ indicates a good correspondence between the linear subproblem and the nonlinear problem. Further the criterion $\rho_{old} > \varepsilon$ prevents the trust region from oscillating, which could have a damaging effect on convergence. A poor correspondence is indicated by $\rho < 0.25$ and in that case the trust region is reduced.

The SLP algorithm does not use linesearch, but we may have to reduce the trust region several times to find an acceptable point. So this sequence of reductions replace the linesearch. We have tried other update strategies than the above. The description of those, and the results are given in chapter 5.

As a stopping criterion we would could use a criterion that stops when a certain precision is reached

$$\frac{F(\mathbf{x}) - F(\mathbf{x}^*)}{\max\{1, F(\mathbf{x}^*)\}} < \delta \tag{3.14}$$

where δ is a specified accuracy and $\mathbf{x}^*$ is the solution. This can however not be used in practice where we do not know the solution.

A stopping criterion that is activated when the algorithm stops to make progress e.g. $\|\mathbf{x}_k - \mathbf{x}_{k-1}\| < \varepsilon_1$, can not be used in the SLP algorithm, because $\mathbf{x}$ does not have to change in every iteration, e.g. the step $\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{h}$ is discarded if $\rho < \varepsilon$.

Instead we could exploit that the LP solver in every iteration calculates a basic step $\mathbf{h}$ so that

$$\|\mathbf{h}\| < \delta \,. \tag{3.15}$$

This implies that the algorithm should stop when the basic step is small, which happens either when the trust region $\eta < \delta$ or when a strongly unique local minimizer has been found. In the latter case the LP landscape has a unique solution as shown on figure 3.3 (right).

Another important stopping criteria that should be implemented is

$$\Delta L \leq 0 \,. \tag{3.16}$$

When the LP landscape has gradients close to zero, it can some times happen, due to rounding errors in the solver, that ΔL drops below zero. This is clearly an indication that a stationary point has been reached, and the algorithm should therefore stop. If this stopping criteria were absent, the algorithm would continue until the trust region radius was so small that the

Algorithm 3.1.1 SLP

```
begin
  k := 0;  x := x_0;  found := false;  ρ_old := 2ε;                    {1°}
  f := f(x); J := J_f(x);
  while (not found)
    calculate x̂ by solving (3.2) by using the setup in (3.6).
    α := x̂(n+1);    h := x̂(1:n);
    x_t := x + h;   f_t := f(x_t);
    ρ = ΔF(x;h)/ΔL(x;h)
    if ρ > ε                                                           {2°}
      x := x_t;   f := f_t;   J := J(x_t);
    Use the trust region update in (3.13)
    ρ_old = ρ;
    k = k + 1;
    if stop or k > k_max                                               {3°}
      found := true;
end
```

{1°} The user supplies the algorithm with an initial trust region radius η and ε which should be between $0 \leq \varepsilon \leq 0.25$. The stopping criteria takes k_{max} that sets the maximum number of iterations, and δ that sets the minimum step size.

{2°} If the gain factor ρ is small or even negative, then we should discard the step and reduce the trust region. On the other hand, if ρ is large then it indicate that $L(\mathbf{x};\mathbf{h})$ is a good approximation to $F(\mathbf{x}_t)$. The step should be accepted and the trustregion should be increased. When the step size $\|\mathbf{h}\| \to 0$ then its possible that $\Delta L < 0$ due to rounding errors.

{3°} The algorithm should stop, when a stopping criterion indicates that a solution has been reached. We recommend to use the stopping criteria in (3.15) and (3.16).

stopping criterion in (3.15) was activated. This would give more unnecessary iterations. If $\Delta L = 0$ then we must be at a local minimum.

The above leads to the basic SLP algorithm that will be presented in the following with comments to the pseudo code presented in algorithm 3.1.1.

One of the more crucial parameters in the SLP algorithm is ε. Lets take a closer look at the effect of changing this variable.

The condition $\Delta F > \varepsilon \Delta L$ should be satisfied in order to accept the new step. So when ε is large, then ΔF has to be somewhat large too. In the opposite case ε being small, ΔF can be small and still SLP will take a new step. In other words, a small value of ε will render the algorithm more biased to accepting new steps and increase the trust region - it becomes more optimistic, see (3.13).

In the opposite case when ε is large the algorithm will be more conservative and be more likely to reject steps and reduce the trust region. The effect of ε is investicated further in section 3.1.3.

Situations occur where the gain factor $\rho < 0$, which happens in three cases: When $\Delta F < 0$, which indicate that $\mathbf{h}$ is an uphill step, and when $\Delta L < 0$, which only happens when $\|\mathbf{h}\|$ is approaching the machine accuracy and is due to rounding errors. Finally both $\Delta L < 0$ and $\Delta F < 0$, which is caused by both an uphill step and rounding errors. Due to the stopping criteria in (3.16) the two last scenarios would stop the algorithm.

3.1.2 Convergence Rates for SLP

An algorithm like SLP will converge towards stationary points, and as shown in chapter 2 such points are also local minimizers of $F(\mathbf{x})$. The SLP method is similar to the method of Madsen (method 1) [Mad86, pp. 76–78], because they both are trust region based and use an LP-solver to find a basic step. The only difference is that the trust region update in method 1 does not have the regularizing term $\rho_{old} > \varepsilon$, as in (3.13).

When only using first order derivatives it is only possible to obtain fast (quadratic) final convergence if $\mathbf{x}^*$ is a strongly unique local minimum. In that case the solution is said to be regular. This is stated more formal in the following theorem.

Theorem 3.1 *Let $\{\mathbf{x}_k\}$ be generated by method 1 (SLP) where $\mathbf{h}_k$ is a solution to (3.2). Assume that we use (3.1). If $\{\mathbf{x}_k\}$ converges to a strongly unique local minimum of (2.7), then the rate of convergence is quadratic.*

Proof: [Mad86, p. 99]

We can also get quadratic convergence for some directions towards a stationary point. This happens when there is a kink in F from that direction. In this case first order derivatives will suffice to give fast final convergence. See proposition 2.4.

If $\mathbf{d} \perp \partial F(\mathbf{x})$ or $\mathbf{x}^*$ is not a strongly unique local minimum, then we will get slow (linear) convergence. Hence in that case we need second order information to get the faster quadratic final convergence.

3.1.3 Numerical Experiments

In this section we have tested the SLP algorithm with the Rosenbrock function with $w = 10$ and the Parabola test function, for two extreme values of ε. The SLP algorithm has also been tested with other test problems, and section 3.3.3, is dedicated to present those results.

SLP Tested on Rosenbrock's Function

Rosenbrock is a Chebyshev approximation problem and we tested the problem with two values of ε to investigate the effect of this parameter. The values chosen was $\varepsilon = 0.01$ and $\varepsilon = 0.25$. The results are presented in figure 3.4 (left) and (right). The Rosenbrock problem has a regular solution at $\mathbf{x}^* = [1 \;\; 1]^T$, and we can expect quadratic final convergence in both cases. In both cases the starting point is $\mathbf{x}_0 = [-1.2 \;\; 1]^T$.

We start by looking at the case $\varepsilon = 0.01$, where the following options are used

$$\text{opts} = [\, \eta,\ \varepsilon,\ k_{max},\ \delta \,] = [\, 1,\ 0.01,\ 100,\ 10^{-5} \,], \tag{3.17}$$

The SLP algorithm converged to the solution $\mathbf{x}^*$ in 20 iterations and stopped on $\|\mathbf{d}\| \leq \delta$. There where four active functions in the solution.

For $F(\mathbf{x})$ we see that the first 10 iterations show a stair step like decrease, after which $F(\mathbf{x})$ decreases more smoothly. At the last four iterations we have quadratic convergence as seen on figure 3.4 (left). This shows that SLP is able to get into the vicinity of the stationary point in the global part of the iterations. When close enough to the minimizer, we get quadratic convergence because Rosenbrock has a regular solution.

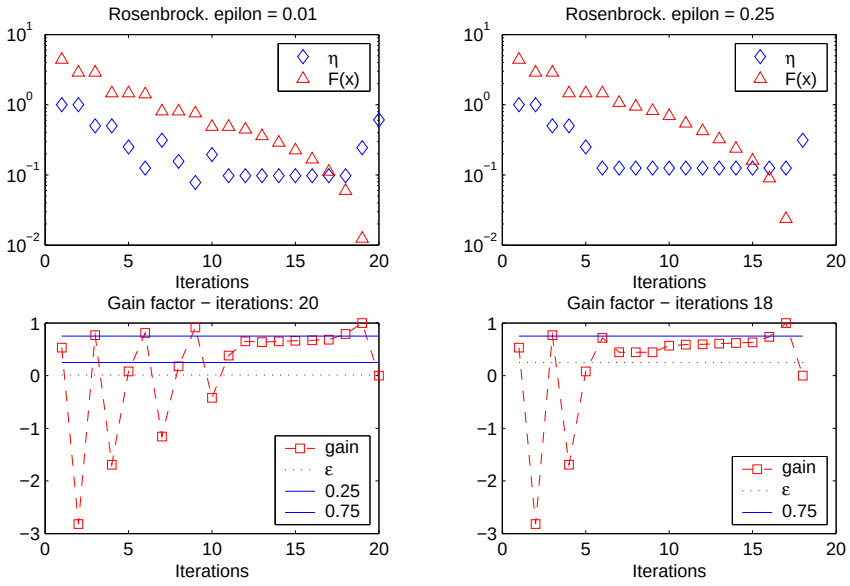

Figure 3.4: Performance of the SLP algorithm when using the Rosenbrock function. The acceptance area of a new step, lies above the dashed line denoting the value of ε.

The gain factor ρ oscillates in the first 11 iterations. This affects the trust region η as seen on the top plot of figure 3.4 (left). The trust region is also oscillating, but in a somewhat more dampened way, because of $\rho_{old} > \varepsilon$.

At the last iterations, when near the minimizer, the linear subproblem is in good correspondence with the nonlinear problem. This is indicated by the increase in ρ at the last 9 iterations. We end with $\rho = 0$ due to $\Delta F = 0$.

The trust region increases in the last 3 iterations, because ρ indicate that the linear subproblem is a good approximation to the nonlinear problem.

From the plots one can clearly see, that whenever a step is rejected $F(\mathbf{x})$ remains constant and the trust region is decreased, due to the updating in (3.13).

The SLP algorithm was also tested with $\varepsilon = 0.25$, with the same starting point $\mathbf{x} = [-1.2 \;\; 1]^T$. The following options were used

$$\text{opts} = [\,\eta,\ \varepsilon,\ k_{max},\ \delta\,] = [\,1,\ 0.25,\ 100,\ 10^{-5}\,]. \tag{3.18}$$

SLP found the minimizer $\mathbf{x}^*$ in 18 iterations and stopped on $\|\mathbf{d}\| \leq \delta$ and there were four active functions in the solution.

For $F(\mathbf{x})$ we again see a stair step like decrease in $F(\mathbf{x})$ in the first 6 iterations, because of the oscillations in ρ. The global part of the iterations gets us close to the minimizer, after which we start the local part of the iterations. This is indicated by ρ that increases steadily during the last 12 iterations. Again we get quadratic convergence because the solution is regular.

We see that by using $\varepsilon = 0.25$, the oscillations of the gain factor ρ is somewhat dampened in comparison with the case $\varepsilon = 0.01$. This affects the trust region, so it too does not oscillate. Again we see an increase of the trust region at the end of the iterations because of the good gain factor.

The numerical experiment shows that a small value of ε makes the algorithm more optimistic. It more easily increases the trust region, as we see on figure 3.4, so this makes the algorithm more biased towards increasing it's trust region, when ε is small.

The gain factor ρ for $\varepsilon = 0.01$ shows that four steps were rejected, so optimistic behaviour has its price in an increased amount of iterations, when compared to $\varepsilon = 0.25$.

For $\varepsilon = 0.25$ we see that only two steps were rejected, and that the trust region η decreases to a steady level sooner than the case $\varepsilon = 0.01$. This shows that large ε really makes SLP more conservative e.g. it does not increase its trust region so often and is more reluctant to accept new steps. But it is important to note, that this conservative strategy reduces the amount of iterations needed for this particular experiment.

SLP Tested With the Parabola Test Function

We have tested the SLP algorithm on the Parabola test function and used the same scheme as in section 3.1.3. That is testing for $\varepsilon = 0.01$ and $\varepsilon = 0.25$. The problem has a minimizer in $\mathbf{x}^* = [0\ 0]^T$, and the solution is *not* regular due to only two inner functions being active at the solution.

The first test is done with the following options

$$\text{opts} = [\,\eta,\, \varepsilon,\, k_{max}\,,\delta\,] = [\ 1,\ 0.01,\ 100,\ 10^{-10}\ ]\,, \tag{3.19}$$

Notice the lower δ value. This is because we do not have a strongly unique local minimum in the solution. Therefore the trust region will dictate the precision of which we can find the solution. The results of the test are shown on figure 3.5 (left).

The algorithm used 65 iterations and converged to the solution $\mathbf{x}^*$ with a precision of $\|\mathbf{x}^{(k)} - \mathbf{x}^*\| = 8.8692\text{e-}09$. The algorithm stopped on $\|\mathbf{d}\| \leq \delta$, and 2 functions were active in the solution.

We notice that the trust region η is decreasing in a step wise manner, until the last 10 iterations, where there is a strict decrease of the trust region in every iteration. The reason is that after $F(\mathbf{x})$ has hit the machine accuracy the gain factor ρ settles at a constant level below 0 due to ΔF being negative, so every step proposed by SLP is an uphill step.

The rugged decrease of η is due to the rather large oscillations in the gain factor, as seen on the bottom plot of figure 3.5. Many steps are rejected because $\rho < \varepsilon$. This affects the

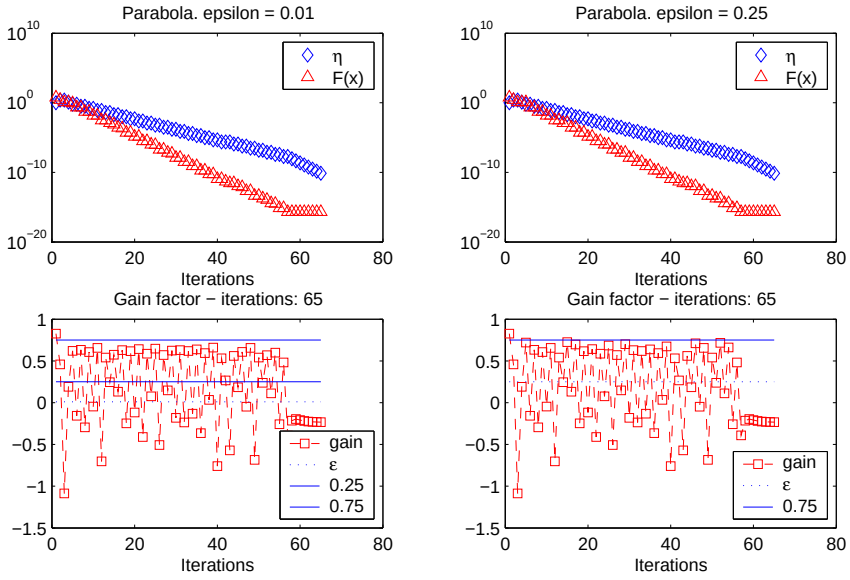

Figure 3.5: Performance of the SLP algorithm when using the Parabola test function. for both $\varepsilon = 0.01$ and $\varepsilon = 0.25$ we see linear convergence.

decrease of $F(\mathbf{x})$, that show no signs of fast convergence. In fact the convergence is linear as according to the theory of stationary points that are not strongly unique local minimizers.

A supplementary test has been made with the following options

$$\text{opts} = [\ \eta,\ \varepsilon,\ k_{max},\ \delta\ ] = [\ 1,\ 0.25,\ 100,\ 10^{-10},\ ]. \tag{3.20}$$

The algorithm used 65 iterations and converged to the solution $\mathbf{x}^*$ with a precision of $\|\mathbf{x}^{(k)} - \mathbf{x}^*\| =$ `8.8692e-09`. The algorithm stopped on $\|\mathbf{d}\| \leq \delta$. Again two functions where active in the solution. The result is shown on figure 3.5 (*right*).

This test shows almost the same results as that of $\varepsilon = 0.01$, even though more steps are rejected because of the high ε. Apparently this has no influence on the convergence rate. From the top plot of figure 3.5 (*right*) we see that $F(\mathbf{x})$ shows a linear convergence towards the minimizer, due to the solution being not regular.

We have seen that the choice of $\varepsilon = 0.25$ has a slightly positive effect on the Rosenbrock problem, and almost no effect on the Parabola problem. Later we will use an addition to the SLP algorithm that uses a corrective step on steps that would otherwise have failed. With such an addition, we are interested in exploiting the flexibility that a low ε gives on the trust region update.

The rate of convergence is for some problems determined by the direction to the minimizer. This is shown in the following.

If we use SLP with $\mathbf{x}_0 = [3\ \ 0]^T$ and $\varepsilon = 0.01$ then the problem is solved in 64 iterations, and we have slow linear convergence. Hence we need second order information to induce faster (quadratic) convergence.

According to Madsen [Mad86, p. 32] for some problems, another direction towards the minimizer will render first order derivatives sufficient to get fast convergence. We tried this on Parabola with $\mathbf{x}_0 = [0 \;\; 30]^T$ and $\varepsilon = 0.01$ and found the minimizer in only 6 iterations.

3.2 The First Order Corrective Step

From the numerical experiments presented in section 3.1, we saw that steps were wasted when $\rho \leq \varepsilon$. In this section we introduce the idea of a corrective step, that tries to modify a step, that otherwise would have failed. The expectation is, that the corrective step can reduce the number of iterations in SLP like algorithms.

We will present a first order corrective step (CS) that first appeared in connection with SLP in [JM92] and [JM94]. The idea is, that if the basic step $\mathbf{h}$ is rejected, then we calculate a corrective step $\mathbf{v}$. A sketch of the idea is shown in figure 3.6. For the sake of the later discussion we use the shorter notation $\mathbf{x}_t = \mathbf{x} + \mathbf{h}$ and $\tilde{\mathbf{x}}_t = \mathbf{x} + \mathbf{h} + \mathbf{v}$.

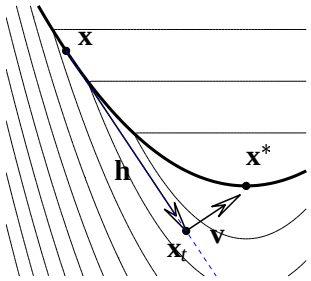

Figure 3.6: The basic step $\mathbf{h}$ and the corrective step $\mathbf{v}$. The idea is that the step $\mathbf{h}+\mathbf{v}$ will move the step towards $f_1(\tilde{\mathbf{x}}_t) = f_2(\tilde{\mathbf{x}}_t)$ (thick line), and in that way reduce the number of iterations.

In the SLP algorithm the basic step $\mathbf{h}_k$ was found by solving the linear subproblem (LP) in (3.2).If $\rho < \varepsilon$ for $\mathbf{x}_k$, the basic step would be discarded, the trust region reduced and a new basic step $\mathbf{h}_{k+1}$ would be tried. Now a corrective step is used to try to save the step $\mathbf{h}_k$ by finding a new point $\tilde{\mathbf{x}}_t$ that hopefully is better than $\mathbf{x}_k$. The important point is that the corrective step cost the same, measured in function evaluations, as the traditional SLP strategy. It could be even cheaper if no new evaluation of the Jacobian was made at $\tilde{\mathbf{x}}_t$.

The corrective step uses the inner functions that are active in the LP problem, where active set in the LP problem is defined as

$$\begin{aligned} \mathcal{A}_{LP} &= \{\; j \mid \ell_j(\mathbf{h}) = \alpha \;\} \\ &= \{\; j \mid \lambda_j > 0 \;\}, \end{aligned} \tag{3.21}$$

where $j = 1, \ldots, m$ and $\ell_j(\mathbf{h})$ is defined in (3.1). We could also use the following multiplier free formulation

$$\mathcal{A}_{LP} = \{\; j \mid |\ell_j(\mathbf{h}) - \alpha| \leq \gamma \;\}, \tag{3.22}$$

where γ is a small value. The difference in definition is due to numerical errors. Here it is important to emphasize that we should use the multipliers whenever possible, because (3.22) would introduce another preset parameter or heuristic to the algorithm.

There is, but one objection, to the strategy of using the multipliers to define the active set. By using the theoretical insight from (2.20), an inner function that is *not* strongly active is

not guaranteed to have a positive multiplier, in fact it could be zero. In other words, if we want to be sure to get *all* the active inner functions we must use (3.22).

The strategy, of using Lagrangian multipliers to define the active set, can sometimes fail when using `linprog ver. 1.23`. This happens when the active functions have contours parallel to the coordinate system. A simple case is described in Section 6.3.

When we have found the active set, the corrective step can be calculated. The reasoning behind the corrective step is illustrated in figure 3.7 (left) where the basic step $\mathbf{h}$ is found at the kink of the linearized inner functions. This kink is, however, situated further away than the kink in the nonlinear problem. At $\mathbf{x}_t$ we use the same gradient information as in $\mathbf{x}$ and we use those functions that were active at the kink (the active functions in the LP problem) to calculate the corrective step. At figure 3.7 (right) the corrective step $\mathbf{v}$ is found at the kink of the linearizations, and hopefully $\tilde{\mathbf{x}}_t$ will be an acceptable point.

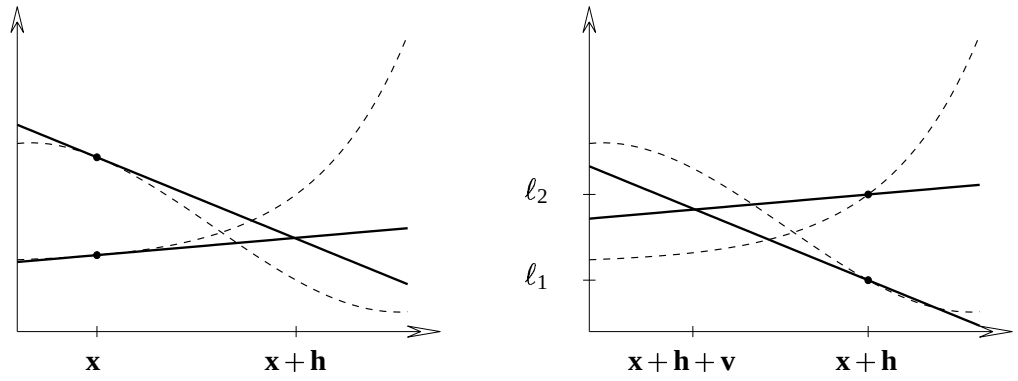

Figure 3.7: Side view of a 2d minimax problem. Left: The gradients at $\mathbf{x}$ indicates a solution at $\mathbf{x}+\mathbf{h}$. Right: The gradients at $\mathbf{x}$ is used to calculate the corrective step, and a solution is indicated at $\mathbf{x}+\mathbf{h}+\mathbf{v}$.

Furthermore the active set $\mathcal{A}_{LP}$ must only consist of functions whose gradients are linearly independent. If this is not the case, then we just delete some equations to make the rest linearly independent.

3.2.1 Finding Linearly Independent Gradients

We have a some vectors that are stored as columns in the matrix $\mathbf{A} \in \mathbb{R}^{n\times m}$. We now want to determine whether these vectors i.e. columns of $\mathbf{A}$ are mutually linearly independent.

The most intuitive ways to do this, is to project the columns of $\mathbf{A}$ onto an orthogonal basis spanned by the columns of $\mathbf{Q} \in \mathbb{R}^{n\times n}$. There exists different methods to find $\mathbf{Q}$ like Gram-Schmidt orthogonalization, Givens transformations and the Housholder transformation, just to mention some. Because $\mathbf{Q}$ is orthogonal $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. The vectors i.e. columns of $\mathbf{A}$ are then projected onto the basis $\mathbf{Q}$, and the projections are then stored in a matrix called $\mathbf{R} \in \mathbb{R}^{n\times m}$.

The method roughly sketched above is called QR factorization, where $\mathbf{A} = \mathbf{QR}$. For more insight and details, the reader is referred to [GvL96, Chapter 5.2] and [Nie96, Chapter 4].

If some columns of $\mathbf{A}$ are linearly dependent, then one of these columns will be given a weight larger than zero on the main diagonal of $\mathbf{R}$, while the rest ideally will appear as zeros. For numerical reasons, however, we should look after small values on the main diagonal as an indication of linear dependence.

We want to find linearly independent gradients in the Jacobian by doing a QR factorization of $\mathbf{A}$, so that $\mathbf{A} = \mathbf{J}(\mathbf{x})^T$, which means that the gradients are stored as columns in $\mathbf{A}$.

To illustrate what we want from the QR factorization, we give an example. We have the following system

$$A = \begin{bmatrix} 1.2 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = [\, 1\, 2\, 3\,]^T, \tag{3.23}$$

where $\mathbf{A}$ is the transposed Jacobian, so that the following first order Taylor expansion can be formulated as

$$\ell(\mathbf{h}) = \mathbf{A}^T\mathbf{h} + \mathbf{b}, \tag{3.24}$$

where $\mathbf{h} \in \mathbb{R}^n$. A QR factorization would then yield the following result

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} 1.2 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.25}$$

As proposed in the previous we look at the main diagonal for elements not equal to zero. In this case it would seem like the system only had $\text{rank}(\mathbf{A}) = 1$, because the last element on the diagonal is zero. This is, however, wrong when obviously $\text{rank}(\mathbf{A}) = 2$.

The problem is that the QR factorization is not *rank revealing*, and the solution is to pivot the system. The permutation is done so that the squared elements in the diagional of $\mathbf{R}$ is decreasing. Matlab has this feature build in its QR factorization, and it delivers a permutation matrix $\mathbf{E} \in \mathbb{R}^{m\times m}$. We now repeat the above example, and do a QR factorization with permutation.

$$\mathbf{Q} = \frac{\sqrt{2}}{2}\begin{bmatrix} -1 & -1 \\ -1 & 1 \end{bmatrix}, \quad \mathbf{R} = \frac{\sqrt{2}}{2}\begin{bmatrix} -2 & -1.2 & -1 \\ 0 & -1.2 & -1 \end{bmatrix} \tag{3.26}$$

where the permutation matrix is

$$\mathbf{E} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{3.27}$$

We see that there are two non-zeros along the diagonal of $\mathbf{R}$ and that this corresponds to the rank of $\mathbf{A}$.

In the examples we used $\mathbf{A}$ and $\mathbf{b}$, these can without loss of generality be replaced by $\mathbf{J}^T$ and $\mathbf{f}$. For a QR factorization of $\mathbf{J}^T$, let i denote the row numbers, where $\text{diag}(\mathbf{R}) \neq 0$, then i is a set containing the indexes that denote the active functions at $\mathbf{x}$. We then find

$$\mathbf{b} = \mathbf{f}_i(\mathbf{x}), \quad \mathbf{A}^T = \mathbf{J}_{i,:}(\mathbf{x}), \; i \in \mathcal{A}_{LP}.$$

We then QR factorize $\mathbf{A}$ and get the basis $\mathbf{Q}$, the projections $\mathbf{R}$ and finally the permutation matrix $\mathbf{E}$. We then apply the permutation so that

$$\tilde{\mathbf{b}} = \mathbf{E}^T\mathbf{b} \quad \text{and} \quad \tilde{\mathbf{A}} = \mathbf{A}\mathbf{E}.$$

Next, we search for non-zero values along the diagonal of $\mathbf{R}$. For numerical reasons we should search for values higher than a certain threshold. The index to the elements of $mathrmdiag(\mathbf{R})$ that is non-zero is then stored in j. Then

$$\overline{\mathbf{J}}^T = \tilde{\mathbf{A}}_{:,j} \quad \text{and} \quad \overline{\mathbf{f}} = \tilde{\mathbf{b}}_i \,, \tag{3.28}$$

and we see that $\overline{\mathbf{J}}$ and $\overline{\mathbf{f}}$ correspond to reduced versions of $\mathbf{J}$ and $\mathbf{f}$, in that sense that all linear dependent columns of $\mathbf{J}$ and all linear dependent rows of $\mathbf{f}$ have been removed. It is those reduced versions we will use in the following calculation of the corrective step.

3.2.2 Calculation of the Corrective Step

We can now calculate the corrective step based on the reduced system $\overline{\mathbf{J}}$ and $\overline{\mathbf{f}}$. We now formulate a problem, whose solution is the corrective step $\mathbf{v}$.

$$\min_{\mathbf{v},\beta} \frac{1}{2}\mathbf{v}^T\mathbf{v} \quad \text{s.t.} \quad \overline{\mathbf{f}}(\mathbf{x}+\mathbf{h}) + \overline{\mathbf{J}}(\mathbf{x}+\mathbf{h})\mathbf{v} = \beta\mathbf{e} \tag{3.29}$$

where $\overline{\mathbf{f}} \in \mathbb{R}^t$ are those components of $\mathbf{f}$ that are active according to definition in (3.21) or (3.22) and linearly independent. β is a scalar, $\mathbf{e} \in \mathbb{R}^t$ is a vector of ones and $\mathbf{v} \in \mathbb{R}^n$. The equation says that the corrective step $\mathbf{v}$ should be as short as possible and the active linearized functions should be equal at the new iterate. This last demand is natural because such a condition is expected to hold at the solution $\mathbf{x}^*$. The corrective step $\mathbf{v}$ is minimized by using the ℓ_2 norm, to keep it as close to the basic step $\mathbf{h}$ as possible. By using the ℓ_2 norm, the solution to (3.29), can be found by solving a simple system of linear equations.

First we reformulate (3.29)

$$\min_{\hat{\mathbf{v}}} \frac{1}{2}(\hat{\mathbf{I}}\hat{\mathbf{v}})^T(\hat{\mathbf{I}}\hat{\mathbf{v}}) \quad \text{s.t.} \quad \overline{\mathbf{f}} + \left[\overline{\mathbf{J}} \; -\mathbf{e}\right]\hat{\mathbf{v}} = \mathbf{0}, \tag{3.30}$$

where $\overline{\mathbf{f}}$ and $\overline{\mathbf{J}}$ are short for $\overline{\mathbf{f}}(\mathbf{x}+\mathbf{h})$ and $\overline{\mathbf{J}}(\mathbf{x}+\mathbf{h})$. Furthermore we have

$$\hat{\mathbf{v}} = \begin{bmatrix} \mathbf{v} \\ \beta \end{bmatrix} \quad , \quad \hat{\mathbf{I}} = \begin{bmatrix} \mathbf{I} & 0 \\ 0 & 0 \end{bmatrix} \quad , \quad \mathbf{I} \in \mathbb{R}^{n\times n}. \tag{3.31}$$

By using the theory of Lagrangian multipliers

$$\mathcal{L}(\hat{\mathbf{v}},\lambda) = \frac{1}{2}(\hat{\mathbf{I}}\hat{\mathbf{v}})^T(\hat{\mathbf{I}}\hat{\mathbf{v}}) + \left(\overline{\mathbf{f}} + \left[\overline{\mathbf{J}} \; -\mathbf{e}\right]\hat{\mathbf{v}}\right)^T\lambda \tag{3.32}$$

and by using the first order *Kuhn-Tucker* conditions for optimality we have that the gradient of the Lagrangian should be zero

$$\mathcal{L}'(\hat{\mathbf{v}},\lambda) = \hat{\mathbf{I}}\hat{\mathbf{v}} + \left[\overline{\mathbf{J}} \; -\mathbf{e}\right]^T\lambda = 0 \,. \tag{3.33}$$

Further, because of the equality constraint: $(\overline{\mathbf{f}} + \left[\overline{\mathbf{J}} \; -\mathbf{e}\right]\hat{\mathbf{v}})^T\lambda = 0$, is seen to be satisfied when

$$\left[\overline{\mathbf{J}} \; -\mathbf{e}\right]\hat{\mathbf{v}} = -\overline{\mathbf{f}} \,. \tag{3.34}$$

To find a solution that satisfy both Kuhn-Tucker conditions, we use the following system of equations

$$\begin{bmatrix} \hat{\mathbf{I}} & \hat{\mathbf{A}} \\ \hat{\mathbf{A}}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{v}} \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ -\overline{\mathbf{f}} \end{bmatrix} \quad , \quad \hat{\mathbf{A}} = \begin{bmatrix} \overline{\mathbf{J}}^T \\ -\mathbf{e}^T \end{bmatrix} \tag{3.35}$$

where $\hat{\mathbf{A}} \in \mathbb{R}^{(n+1)\times t}$. By solving (3.35) we get the corrective step $\mathbf{v}$. For a good introduction and description of the Kuhn-Tucker conditions, the reader is referred to [MNT01, Chapter 2].

In (3.29) we used the Jacobian $\overline{\mathbf{J}}$ evaluated at $\mathbf{x}_t$, however it was suggested in [JM94], that we instead could use $\overline{\mathbf{J}}$ evaluated at $\mathbf{x}$. As Figure 3.8 clearly indicate, this will give a more crude corrective step.

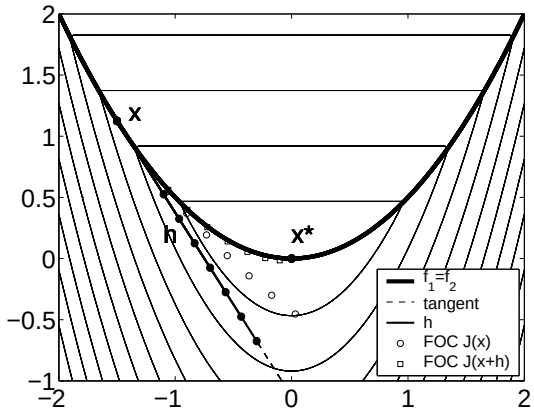

Figure 3.8: Various steps $\mathbf{h}$ of different length. The squares denote the corrective step $\mathbf{v}$ based on $\overline{\mathbf{J}}(\mathbf{x}+\mathbf{h})$. The circles denote $\mathbf{v}$ based on $\overline{\mathbf{J}}(\mathbf{x})$.

The figure shows that $\mathbf{v}$ based on $\overline{\mathbf{J}}(\mathbf{x})$ is not as good as $\mathbf{v}$ based on $\overline{\mathbf{J}}(\mathbf{x}_t)$. The latter gives a result much closer to the intersection between the two inner functions of the parabola test function. This goes in line with [JM94] stating that it would be advantageous to use (3.29) when the problem is highly non-linear and otherwise calculate $\mathbf{v}$ based on $\mathbf{J}(\mathbf{x})$ and save the gradient evaluation at $\mathbf{x}_t$.

It is interesting to notice that Figure 3.8 shows that the the step $\mathbf{v}$ based on $\mathbf{J}(\mathbf{x})$ is perpendicular to $\mathbf{h}$. We will investigate that in more detail in the following. The explanation assumes that the corrective step is based on $\mathbf{J}(\mathbf{x})$.

The length of the basic step $\mathbf{h}$ affects the corrective step $\mathbf{v}$ as seen in figure 3.8. This is because the length of $\mathbf{v}$ is proportional to the difference in the active inner functions $\overline{\mathbf{f}}$ of F. This is illustrated in figure 3.7. If the distance between the active inner functions are large, then the length of the corrective step will also be large and vice versa.

From the figure it is clear, that the difference in the function values of the active inner functions is proportional to the distance between $\mathbf{x}_t$ and $\tilde{\mathbf{x}}_t$. So the further apart the values of the active inner functions is, the further the corrective step becomes, and vice versa.

It turns out that the corrective step is only perpendicular to the basic step in certain cases. So the situation shown in figure 3.8 is a special case. The special case arises when the active set in the nonlinear problem is the same as in the linear problem. That is $\mathcal{A} = \mathcal{A}_{LP}$. We could also say that the basic step $\mathbf{h}$ has to be a tangent to the nonlinear kink where the active functions meet. An illustration of this is given in figure 3.9.

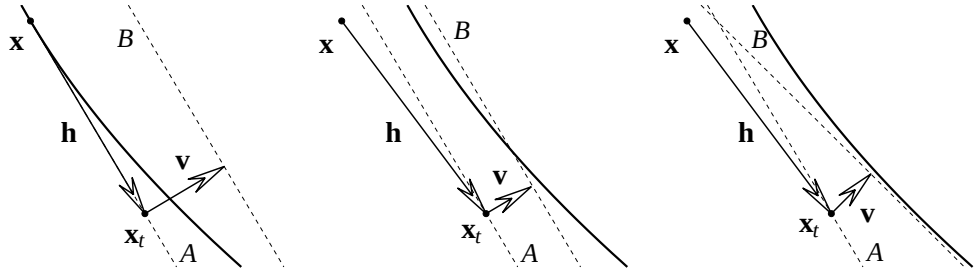

Figure 3.9: The dashed lines illustrates the kink in: A, the LP landscape at $\mathbf{x}$ and B, the linearized landscape at $\mathbf{x}_t$. Left: Only when the basic step is a tangent to the nonlinear kink, we have that $\mathbf{h}$ is perpendicular to $\mathbf{v}$. Middle: Shows that $\mathbf{h}$ is not always perpendicular to $\mathbf{v}$. Right: If $\mathbf{J}(\mathbf{x}_t)$ is used to calculate $\mathbf{v}$ then the two kinks are not always parallel.

The figure shows two dashed lines, where A illustrates the kink in the LP landscape at $\mathbf{x}$ and B illustrates the kink in the linearized landscape at $\mathbf{x}_t$. It is seen that the dashed line B moves because of the change in the active inner functions at $\mathbf{x}_t$. This affects $\mathbf{v}$, because it will always go to that place in the linearized landscape where the active functions at $\mathbf{x}_t$ becomes equal. Therefore the corrective step always goes from A to B.

When using $J(\mathbf{x})$ to calculate $\mathbf{v}$, the dashed lines will be parallel, and hence $\mathbf{v}$ will be orthogonal to A and B because it is the shortest distance between them. In general, however, it is always the case that $\mathbf{v}$ will be orthogonal to the dashed line B as illustrated in figure 3.9 right.

Figure 3.9 left, illustrates the case where $\mathbf{x}$ is situated at A, and where $J(\mathbf{x})$ is used to calculate $\mathbf{v}$. In this case $\mathbf{h}$ will be perpendicular to $\mathbf{v}$.

The same figure middle, shows a situation where $\mathbf{x}$ is not situated at A, hence $\mathbf{h}$ and $\mathbf{v}$ are not orthogonal to each other.

Figure 3.9 right, illustrates a situation where the Jacobian evaluated at $\mathbf{x}_t$ is used. In this case, the dashed lines are not parallel to each other. Still, however, $\mathbf{v}$ is perpendicular to B, because that is the shortest distance from $\mathbf{x}_t$ to B.

The definition of the corrective step in (3.29) says that the corrective step should go to a place in the linearized landscape at $\mathbf{x}_t$ based on either $\mathbf{J}(\mathbf{x})$ or $\mathbf{J}(\mathbf{x}_t)$, where the active functions are equal. Also it says that we should keep the corrective step as short as possible.

We notice that if the active set $\mathcal{A}_{LP}$ is of size $t = 1$, then the corrective step $\mathbf{v}$ will be equal to the null vector.

$$\text{length}(t) = 1\,, \quad \text{then } \mathbf{v} = \mathbf{0}\,. \tag{3.36}$$

This is because the calculation of the corrective step has an equality constraint that says that all active functions should be equal. In the case where there is only one active function, every $\mathbf{v} \in \mathbb{R}^n$ would satisfy the equality constraint. The cost function, however, demands that the length of $\mathbf{v}$ should be as short as possible, and when $\mathbf{v} \in \mathbb{R}^n$ then the null vector would be the corrective step with the shortest length. Hence $\mathbf{v} = \mathbf{0}$.

As stated in (3.29) we should only use the functions that are active in $F(\mathbf{x})$ to calculate the corrective step. The pseudo code for the corrective step is presented in the following.

Function 3.2.1 Corrective Step

```
      v := corr_step(f, J, i)                                   {1°}
begin
  Reduce i by e.g. QR factorization                             {2°}
  if length(i) ≤ 1,
    v := 0;
  else
    find [v̂ λ]^T by solving (3.35)
  return v;
end
```

{1°} **J** is either evaluated at **x** or $\mathbf{x}_t$. **i** is the index to the active functions of the LP problem in (3.2)

{2°} The active set is reduced so that the gradients of the active inner functions evaluated at **x** or $\mathbf{x}_t$ are linearly independent.

3.3 SLP With a Corrective Step

In section 3.1 we saw that minimax problems could be solved by using SLP, that only uses first order derivatives, which gives slow convergence for problems that are not regular in the solution. To induce higher convergence rates, the classic remedy is to use second order derivatives. Such methods use Sequential Quadratic Programming (SQP).

For some problems second order derivatives are not always accessible. The naive solution is to approximate the Hessian by finite difference (which is very expensive) or to do an approximation that for each iteration gets closer to the *true* Hessian. One of the preferred methods that does this, is the BFGS update. Unfortunately for problems that are large and sparse, this will lead to a dense Hessian (quasi-Newton SQP). For large and sparse problems we want to avoid a dense Hessian because it will have $n \times n$ elements and hence for big problems use a lot of memory and the iterations would become more expensive. Methods have also been developed that preserve the sparsity pattern of the original problem in the Hessian [GT82].

In the following we discuss the CSLP algorithm that uses a corrective step in conjuncture with the SLP method. CSLP is Hessian free and the theory gives us reason to believe that a good performance could be obtained. The CSLP algorithm was first proposed in 1992 in a technical note [JM92] and finally published by Jonasson and Madsen in BIT in 1994 [JM94].

3.3.1 Implementation of the CSLP Algorithm

The CSLP algorithm is built upon the framework of SLP presented in algorithm 3.1.1, with an addition that tries a corrective step each time a step otherwise would have failed.

This scheme is expected to reduce the number of iterations in comparison with SLP for certain types of problems. Especially for highly non-linear problems like, e.g, the Rosenbrock function.

The CSLP algorithm uses (like SLP) the gain factor ρ from (3.12) to determine whether or not a the iterate $\mathbf{x}_t = \mathbf{x} + \mathbf{d}$ is accepted or discarded. If the latter is the case, we want to try to correct the step by using function 3.2.1. So if $\rho < \varepsilon$ a corrective step is tried.

When the corrective step $\mathbf{v}$ have been found a check is performed to see if $\|\mathbf{v}\| \leq 0.9\|\mathbf{h}\|$. This check is needed to ensure that the algorithm does not return to where it came from, i.e., the basic step added with the corrective step gives $\mathbf{h} + \mathbf{v} = \mathbf{0}$, which would lead to $\mathbf{x} = \mathbf{x_t}$. The algorithm could then oscillate between those two points until the maximum number of iterations is reached. If the check was not performed a situation like the one shown in figure 3.10 could happen.

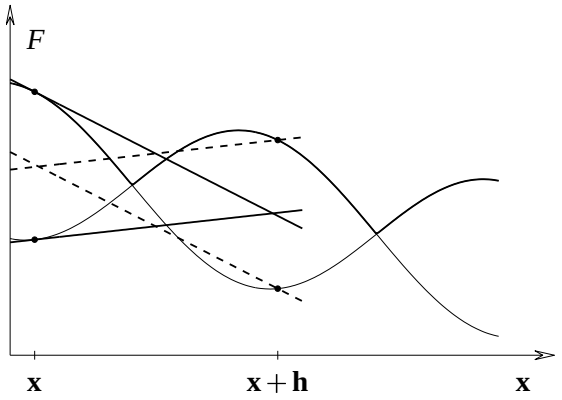

Figure 3.10: Side view of a 2d minimax problem. If not the check $\|\mathbf{v}\| \leq 0.9\|\mathbf{h}\|$ was performed a situation could arise, where the algorithm would oscillate between $\mathbf{x}$ and $\mathbf{x} + \mathbf{h}$. The kink in the solid tangents indicate $\mathbf{h}$ and the kink in the dashed tangents indicate $\mathbf{v}$.

If the check succeeds, then $\mathbf{d} = \mathbf{h} + \mathbf{v}$. To confine the step $\mathbf{d}$ to the trust region we check that $\|\mathbf{d}\| \leq \eta$. If not, we reduce the step so that $\mathbf{d} = \frac{\eta}{\|\mathbf{d}\|}\mathbf{d}$. The reduction of $\mathbf{d}$ is necessary in order to prove convergence.

If the corrective step was taken, we have to recalculate $\Delta F(\mathbf{x}; \mathbf{d})$ to get a new measure of the actual reduction in the nonlinear function $F(\mathbf{x})$. This gives rise to a new gain factor ρ that we treat in the same way as in the SLP algorithm.

The CSLP algorithm is described in pseudo code in algorithm 3.3.1, with comments.

3.3.2 Numerical Results

In this section we test the CSLP algorithms performence on the two test cases from section 3.1.3. This is followed by a test of SLP and CSLP on a selected set of test functions. Those results are shown in table 3.3. Finally, matlab's own minimax solver `fminimax` is tested on the same set of test functions. `fminimax` is a quasi-Newton SQP algorithm that uses soft line search, instead of a trust region.

Algorithm 3.3.1 CSLP

```
begin
  k := 0;  x := x_0;  found := false;  ρ_old := 2ε;                    {1°}
  f := f(x); J := J(x);
  repeat
    calculate x̂ by solving (3.2) or (3.3) by using the setup in (3.6).
    α := x̂(n+1);    h := x̂(n);
    x_t := x + h;   f_t := f(x_t);
    ρ = ΔF(x;h)/ΔL(x;h)
    if ρ ≤ ε                                                           {2°}
      ν := corr_step(f_t, J, h);                                       {3°}
      if ‖ν‖ ≤ 0.9‖h‖
        d := h + ν;
        if ‖d‖_∞ > η
          d := (η/‖d‖_∞)d;
        x_t := x + d;   f_t := f(x_t);
        ρ := ΔF(x;d)/ΔL(x;h);
    if ρ > ε
      x := x_t;   f := f_t;   J := J(x_t);
    Use the trust region update in (3.13)
    ρ_old = ρ;   k = k + 1;
    if stop or k > k_max                                               {4°}
      found := true;
until found
```

{1°} The variables ε, η are supplied by the user. We recommend to use $\varepsilon = 10^{-2}$ or smaller. The trust region $\eta = 1$ is suggested in [JM94], but in reality the ideal η depends on the problem being solved.

{2°} The step $\mathbf{x}+\mathbf{h}$ is rejected, and a corrective step should be taken.

{3°} The corrective step is calculated by using function 3.2.1. We can use either **J** based on $\mathbf{J}(\mathbf{x})$ or $\mathbf{J}(\mathbf{x}+\mathbf{h})$, where the latter is suggested in [JM94] if the problem is highly non-linear. This can also be seen from figure 3.8.

{4°} The algorithm should stop, when a certain stopping criterion indicates that a solution has been reached. We recommend to use (3.15) and (3.16).

CSLP Tested on Rosenbrock's Function

Again we test for two extreme cases of ε, as we did in section 3.1.3 for the SLP algorithm, to see what effect ε has on the convergence rate. We used the following options in the test.

$$(\eta, \varepsilon, k_{max}, \delta) = (1, 0.01, 100, 10^{-10}), \tag{3.37}$$

and the Jacobian for the corrective step was based upon $\mathbf{x}+\mathbf{h}$. The result is shown on figure 3.11 in the left column of plots. The solution was found in 8 iterations with $\mathbf{x} = [1,1]^T$. The algorithm stopped on $\|\mathbf{d}\| \le \delta$ with a precision of $\|\mathbf{x}-\mathbf{x}^*\|_\infty = 0$. We see that $F(\mathbf{x})$ shows

quadratic convergence at the end of the iterations when we are close to $\mathbf{x}^*$. This is because the problem is regular e.g. at least $n+1$ functions are active in the solution and the gradients satisfies the Harr condition. Its interesting to notice that the level of the trust region radius η is higher than that of SLP in section 3.1.3. This is a good sign, because this gives us a quick convergence towards a stationary point in the global part of the iterations.

The gain factor ρ is never below ε which indicates that the corrective step manages to save all the steps that would otherwise have failed.

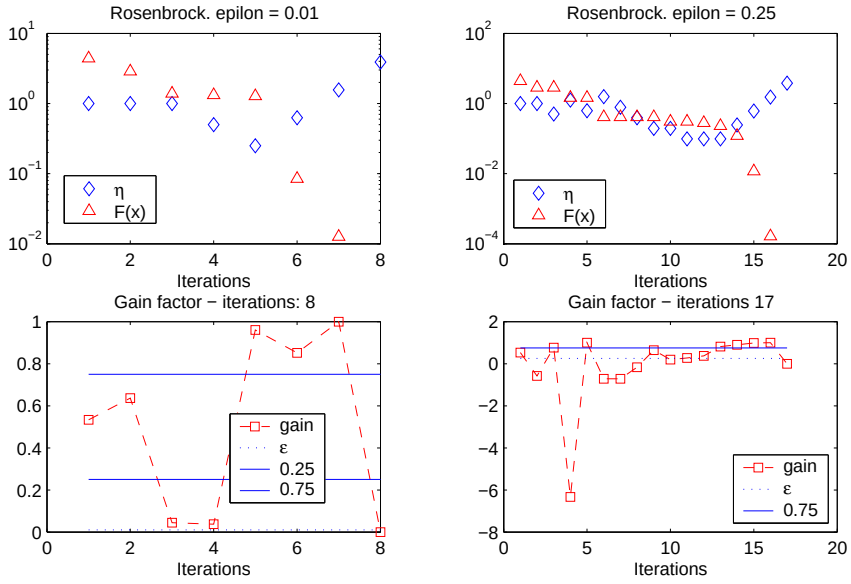

Figure 3.11: Performance of the CSLP algorithm when using the Rosenbrock function with $w = 10$. (Bottom row) The acceptance area of a new step, lies above the dotted line denoting the value of ε.

The same problem was tested with $\varepsilon = 0.25$ and the result is somewhat similar to that of $\varepsilon = 0.01$. The result is shown in figure 3.11 right column of plots. The solution $\mathbf{x} = [1,1]^T$ was found in 17 iterations and the algorithm stopped on a small step $\|\mathbf{d}\| \leq \delta$, with a precision of $\|\mathbf{x} - \mathbf{x}^*\|_\infty = 0$.

Compared with the SLP test, the corrective step has not managed to reduce the number of iterations when the very conservative $\varepsilon = 0.25$ is used. Anyway there is no need to be conservative, when the step is saved by the corrective step.

We see that the corrective step has reduced the number of iterations significantly (see table 3.3) for Rosenbrock like problems, so we can say that the CSLP is a real enhancement of the SLP algorithm.

CSLP Tested on the Parabola Test Function

The same test that was done with SLP in section 3.1.3, is done here with CSLP. The Parabola test function is of special interest because it's not regular at the solution e.g. only two function are active at $\mathbf{x}^* \in \mathbb{R}^2$. Most commonly second order information are needed to get

faster than linear convergence in such a case, and it is therefore interesting to see what effect the corrective step has on such a problem. The test was done with the following options

$$(\eta, \varepsilon, k_{max}, \delta) = (1, 0.01, 100, 10^{-10}). \tag{3.38}$$

The solution was found in 60 iterations with a precision of $\|\mathbf{x} - \mathbf{x}^*\|$ = `5.68e-09`, and stopped on a small step. The result is shown in figure 3.12 left column of plots.

We see a very interesting and fast decrease in $F(\mathbf{x})$ in the first 10-15 iterations. This is because the corrective step very quickly gets $\mathbf{x}$ into the neighbourhood of the stationary point.

When $\mathbf{x}$ is close to the stationary point, the corrective step no longer helps, and thats why we get the linear convergence in the remaining iterations.

Compared to the SLP test, we see that the gain factor is more stable in the start of the iterations (0-15) and at the end, when the trust region is reduced and no new steps are taken.

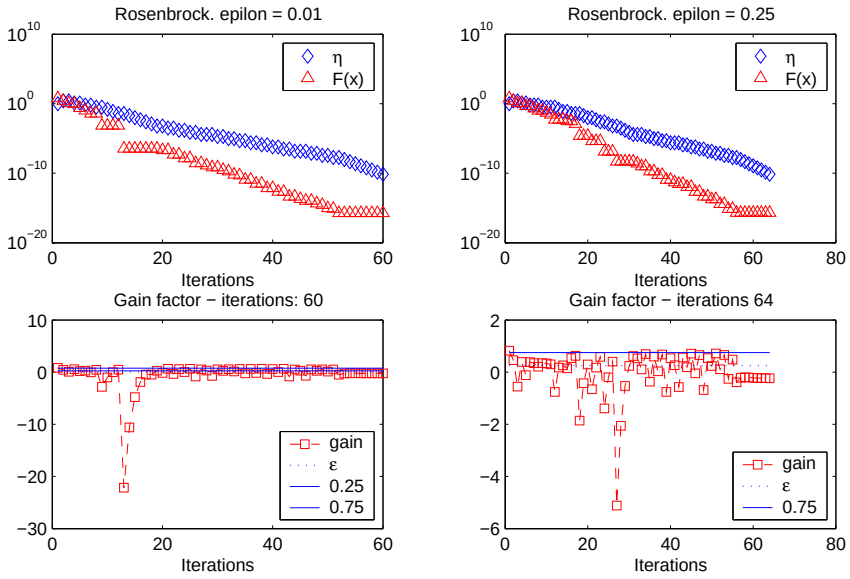

Figure 3.12: Performance of the CSLP algorithm when using the Parabola test function. (Bottom row) The acceptance area of a new step, lies above the stippled line denoting the value of ε.

The same problem was tested with $\varepsilon = 0.25$ and the solution was found in 64 iterations with a precision of $\|\mathbf{x} - \mathbf{x}^*\|$ = `8.87e-09`. The algorithm terminated on a small step. The behavior of the variables in CSLP is shown in figure 3.12 right column.

The course of iterations is almost similar to that of $\varepsilon = 0.01$. The initial reduction of $F(\mathbf{x})$ is somewhat slower, even though the same behavior is present. We see a fast initial reduction of $F(\mathbf{x})$ coursed by the corrective step that brings $\mathbf{x}$ close to the stationary point. When in the vicinity of the stationary point, the corrective step no longer helps, and we get linear convergence. In order to improve the convergence rate, we need second order information.

For this problem CSLP differs from SLP in an important way. From figure 3.13 we see

that the corrective step gets us close to the vicinity of a stationary point, in *fewer* iterations than SLP for the Parabola test function. This yield that CSLP is a very promising method in the global part of the iterations before we reach the vicinity of a stationary point. Another important fact is that CSLP does not perform worse than SLP, as seen in section 3.3.3.

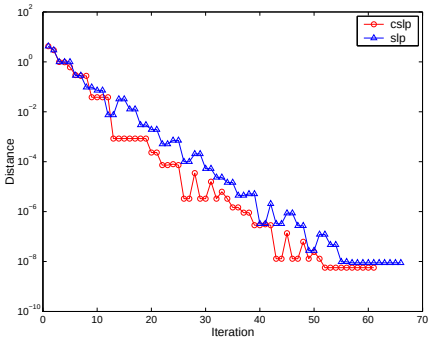

Figure 3.13: Results from SLP and CSLP tested on the Parabola problem. The plot shows the Euclidean distance from $\mathbf{x}^{(k)}$ to the minimizer $\mathbf{x}^*$ as a function of iterations. We see that CSLP reach the vicinity of the minimizer faster than SLP.

3.3.3 Comparative Tests Between SLP and CSLP

In this section we presents numerical results for the SLP and CSLP algorithms presented in the previous. We have used an initial trust region radius $\eta = 1$ and unless otherwise noted $\varepsilon = 0.01$. Table 3.1 shows the test functions that have been used.

Test functions

Name	param	m	n	t^*
Parabola		2	2	2
Rosenbrock1	$w = 10$	2	2	4
Rosenbrock2	$w = 100$	2	2	4
BrownDen		20	4	3
Bard1	y^A	15	3	4
Bard2	y^B	15	3	4
Ztran2f		11	2	2
Enzyme		11	4	22
El Attar		51	6	7
Hettich		5	4	4

Table 3.1: Test functions used in the analysis. See Appendix C for definitions.

The test results for the SLP and CSLP with a corrective step based on $\mathbf{J}(\mathbf{x})$ and $\mathbf{J}(\mathbf{x}+\mathbf{h})$ are presented in Table 3.3.

When comparing the number of iterations, we see that CSLP performs just as well as SLP or better, which also is observed in [JM94].

For functions like Parabola, BrownDen, Ztran2f and Hettich, the number of iterations are either equal or less for CSLP than that of SLP, but when using CSLP, the number of function evaluations grows when compared to SLP. This is because several corrective steps were attempted. For the Hettich function this has lead to a reduction in iterations, but unfortunately also in an increase of function evaluations.

Algorithm fminimax
minimax SQP, Quasi-Newton, line search

Name	10^{-2}	10^{-5}	10^{-8}
Parabola	7(15)	9(19)	8(17)
Rosenbrock1	8(18)	8(18)	8(18)
Rosenbrock2	19(44)	19(44)	19(44)
BrownDen	7(15)	9(19)	9(19)
Bard1	5(11)	6(13)	6(13)
Bard2	7(15)	8(17)	8(17)
Ztran2f	10(21)	14(30)	14(30)
Enzyme	8(17)	-	-
El Attar	16(33)	18(37)	18(37)
Hettich	4(9)	3(7)	3(7)

Table 3.2: Matlab's minimax solver `fminimax` tested upon the test functions. The numbers in the parentheses is the number of function evaluations.

The test functions Bard1, Bard2 and El Attar all share the commen property that CSLP does not use corrective steps, hence for theese functions SLP and CSLP show the same performance. El Attar uses only one corrective step wich only alter the results for CSLP with one iteration.

CSLP has shown itself quite successful on test problems like Rosenbrock1, Rosenbrock2 and Enzyme. Especially the Enzyme problem reveals the strength of CSLP. When the corrective step is based on $\mathbf{J}(\mathbf{x}+\mathbf{h})$ the number of iterations are reduced by almost 75% compared to SLP. The same goes for the number of function evaluations that are reduced by almost 45%.

For Rosenbrock's function we see a huge difference in performance between SLP and CSLP, especially for Rosenbrock2. This get even better when the corrective step is based upon $\mathbf{J}(\mathbf{x}+\mathbf{h})$, which is in accordance with expectations and illustrated in figure 3.8.

The effect of using second order derivatives is seen in table 3.2. Matlabs minimax solver `fminimax` is an SQP algorithm, that uses an approximated Hessian (quasi-Newton) with soft line search. The results are generally very acceptable, and `fminimax` shows good performance for test problems like Parabola, BrownDen, Ztran2f, Enzyme and Hettich, when compared to SLP and CSLP.

For a function like Rosenbrock, `fminimax` uses almost as few iterations as CSLP, but due to its line search it uses a lot more function evaluations.

For the Enzyme problem the number of iterations and function evaluations is very low, but the precision of the solution is not that good. The poor result motivated a further investigation, and by using `optimset`, further information was extracted.

For the Enzyme problem, `fminimax` terminates because the directional derivative is smaller than `2*TolFun`, where `TolFun=1e-6`. The remedy is obviously to lower the value of `TolFun`, in order to get more iterations and a better solution. Rather surprisingly, lowering `TolFun` to `1e-7`, triggers a loop that continues until the maximum number of function

Test of SLP

Name	10^{-2}	10^{-5}	10^{-8}	corrective steps		
Parabola	11(12)	21(22)	31(32)	-	-	-
Rosenbrock1	16(17)	16(17)	16(17)	-	-	-
Rosenbrock2	40(41)	41(42)	41(42)	-	-	-
BrownDen	19(20)	32(33)	42(43)	-	-	-
Bard1	3(4)	4(5)	5(6)	-	-	-
Bard2	3(4)	4(5)	5(6)	-	-	-
Ztran2f	6(7)	21(22)	30(31)	-	-	-
Enzyme	164(165)	168(169)	169(170)	-	-	-
El Attar	7(8)	9(10)	10(11)	-	-	-
Hettich	8(9)	19(20)	30(31)	-	-	-

Test of CSLP - $\mathbf{J(x)}$

Name	10^{-2}	10^{-5}	10^{-8}	corrective steps		
Parabola	8(13)	15(27)	21(39)	4[0]	11[0]	17[0]
Rosenbrock1	10(17)	11(18)	11(18)	7[1]	7[1]	7[1]
Rosenbrock2	17(29)	18(30)	18(30)	11[0]	11[0]	11[0]
BrownDen	15(20)	29(41)	42(57)	4[0]	11[0]	14[0]
Bard1	3(4)	4(5)	5(6)	0[0]	0[0]	0[0]
Bard2	3(4)	4(5)	5(6)	0[0]	0[0]	0[0]
Ztran2f	6(7)	21(30)	30(43)	0[0]	8[0]	12[0]
Enzyme	58(85)	64(94)	65(95)	32[6]	35[6]	35[6]
El Attar	7(8)	9(10)	10(11)	1[1]	1[1]	1[1]
Hettich	7(11)	18(33)	28(53)	5[2]	13[2]	26[2]

Test of CSLP - $\mathbf{J(x+h)}$

Name	10^{-2}	10^{-5}	10^{-8}	corrective steps		
Parabola	8(15)	12(23)	21(41)	6[0]	10[0]	19[0]
Rosenbrock1	7(13)	8(14)	8(14)	5[0]	5[0]	5[0]
Rosenbrock2	9(14)	11(16)	11(16)	4[0]	4[0]	4[0]
BrownDen	15(20)	29(41)	36(52)	4[0]	11[0]	15[0]
Bard1	3(4)	4(5)	5(6)	0[0]	0[0]	0[0]
Bard2	3(4)	4(5)	5(6)	0[0]	0[0]	0[0]
Ztran2f	6(7)	21(30)	30(43)	0[0]	8[0]	12[0]
Enzyme	26(48)	42(75)	43(76)	22[1]	33[1]	33[1]
El Attar	6(8)	8(10)	9(11)	1[0]	1[0]	1[0]
Hettich	5(7)	12(21)	21(39)	3[2]	10[2]	19[2]

Table 3.3: Unless otherwise noted, $\eta_0 = 1$ and $\varepsilon = 0.01$. Column 2–4 number of iterations, ( ) the number of function evaluations. Column 5–7 attempted corrective steps. [ ] failed corrective steps.

evaluations has been reached. Raising the maximum number of function evaluation from 500 to e.g. 1000 , do not help, the precision of the solution stay the same.

In general, however, the table shows, that if we have the possiblity of using second order information, it generally pays to use it. Especially when the problem is not regular at the solution.

3.3.4 Finishing Remarks

For regular solutions where second order information is not necessary to ensure quadratic convergence, the performance of CSLP is very promising. For problems that do not have regular solutions, like the parabola test function, we see a performance of CSLP that is as good as SLP.

The important difference between CSLP and SLP in the non regular case, is that CSLP seems to be able to reach the vicinity of the stationary point faster than SLP. This make CSLP a good algorithm in the global part of the iterations.

The work of Madsen [Mad86], proposes a minimax algorithm that is a two stage method called the *combined method*. For the global part of the iterations a method like SLP (method 1) is used to get into the vicinity of the minimizer. When in the vicinity, a switch is made to a method that uses second order information (method 2), to ensure fast final convergence. It could be interesting to try the combined method of Madsen, with method 1 replaced by CSLP.

As stated in the start of this section its not trivial to use second order information for problems where its not available. In such cases the approximated Hessian (quasi-Newton) tend to be dense. This can pose a problem for large and sparse systems.

Chapter 4

Constrained Minimax

We now turn our attention to the solution of minimax problems with constraints. First we introduce the theory of constrained minimax with examples, followed by a presentation of how to solve constrained minimax problems by using a *penalty function*.

We start by describing the constrained optimization problem

$$\begin{array}{lrcl} \min_{\mathbf{x}} & F(\mathbf{x}) & = & \max\{\mathbf{f}_j(\mathbf{x})\} \\ \text{s.t.} & C(\mathbf{x}) & = & \max\{\mathbf{c}_i(\mathbf{x})\} \leq 0 \ , \end{array} \tag{4.1}$$

where $j = 1, \ldots, m$ and $i = 1, \ldots, p$, and the function of constraints $C(\mathbf{x})$ is a convex function, that is piecewise differentiable. The constraint $C(\mathbf{x}) < 0$ defines a feasible domain $\mathcal{D}$ that is a subset of $\mathbb{R}^n$. Further we assume that inner functions $c_i(\mathbf{x})$ are differentiable in the domain $\mathcal{D}$.

4.1 Stationary Points

We could define a stationary point by using the first order Kuhn-Tucker condition for optimality. That is

$$\mathbf{0} = F'(\mathbf{x}) + \sum_{i=1}^{q} \lambda_i c_i(\mathbf{x}) \quad , \quad \lambda_i \geq 0 \ . \tag{4.2}$$

If $C(\mathbf{x}) < 0$, this generalization is the same as the unconstrained case as shown in (2.9). This is natural when realizing that for $C(\mathbf{x}) < 0$ the constraints have no effect. The interesting case is when $C(\mathbf{x}) = 0$, i.e., we are at the border of the feasible region $\mathcal{D}$.

When noting that $C(\mathbf{x})$ is a minimax function, then we can define the generalized gradient of the constraints $\partial C(\mathbf{x})$ in a similar way as we did with $\partial F(\mathbf{x})$ in (2.6), by using the first order Kuhn-Tucker conditions which yields

$$\partial C(\mathbf{x}) = \{ \sum_{i \in \mathcal{A}_c} \lambda_i c_i'(\mathbf{x}) \mid \sum_{i \in \mathcal{A}_c} \lambda_i = 1, \lambda_i \geq 0 \} \ , \tag{4.3}$$

where

$$\mathcal{A}_c = \{ \ i \mid c_i(\mathbf{x}) = C(\mathbf{x}) \ \} \ . \tag{4.4}$$

The stationary point could, however, be defined slightly differently by using the *Fritz-John* stationary point condition [Man65, p. 94]. This condition holds when $F(\mathbf{x})$ in the nonlinear programming problem (4.1) is a smooth function. Then it holds for a stationary point that $\mathbf{0} \in F'(\mathbf{x})$.

$$\mathbf{0} = \lambda_0 F'(\mathbf{x}) + \lambda \mathbf{c}_i'(\mathbf{x}) \ , \quad \lambda = \sum_{i=1}^{q} \lambda_i \ , \quad \lambda_0 + \lambda = 1 \ , \tag{4.5}$$

where the multipliers $\lambda_i \geq 0$ for $i = 0, \ldots, q$. If constraint i is not violated, then $c_i(\mathbf{x}) < 0$ which leads to $\lambda_i = 0$. For $\lambda_0 > 0$ and $\lambda > 0$, the stationary point condition for the Fritz-John point is

$$\mathbf{0} \in \text{conv}\{\ F'(\mathbf{x}), \partial C(\mathbf{x})\ \} \ . \tag{4.6}$$

So, we say that the null vector should belong to the convex hull spanned by $F'(\mathbf{x})$ and $\partial C(\mathbf{x})$. But this only hold when $F(\mathbf{x})$ is smooth, and $C(\mathbf{x}) = 0$. There are, however, other cases to consider, and for that Merrild [Mer72] defined the map

$$M(\mathbf{x}) = \begin{cases} \partial F(\mathbf{x}) & \text{if } C(\mathbf{x}) < 0 \\ \text{conv}\{\partial F(\mathbf{x}), \partial C(\mathbf{x})\} & \text{if } C(\mathbf{x}) = 0 \\ \partial C(\mathbf{x}) & \text{if } C(\mathbf{x}) > 0 \end{cases} \tag{4.7}$$

As proposed in [Mad86] we will use the Fritz-John stationary point condition in the following, because for some cases the stationary point condition based on Kuhn-Tucker in (4.2) will not suffice. This is illustrated through an example, but first we shall see that there is in fact no huge difference between the two stationary point conditions.

First, the Fritz-John stationary point condition says that

$$\mathbf{0} \in \{\lambda \mathbf{f}' + (1-\lambda)\mathbf{c}' \mid \mathbf{f}' \in \partial F(\mathbf{x}), \mathbf{c}' \in \partial C(\mathbf{x}), 0 \leq \lambda \leq 1\} \ , \tag{4.8}$$

Second, the stationary point condition derived from Kuhn-Tucker in (4.2) says that

$$\mathbf{0} \in \{\ \mathbf{f}' + \mu \mathbf{c}' \mid \mathbf{f}' \in \partial F(\mathbf{x}),\ \mathbf{c}' \in \partial C(\mathbf{x}),\ \mu \geq 0\ \} \ . \tag{4.9}$$

where $\mu = (1-\lambda)/\lambda$. We see that the two conditions are equivalent if $0 < \lambda < 1$. That is, if $\mathbf{0}$ belongs to the convex hull in (4.8), then $\mathbf{0}$ will also belong to the convex hull in (4.9) as seen on figure 4.1.

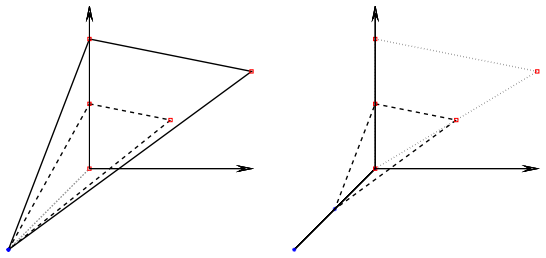

Figure 4.1: The convex hull for three different values of λ. Stationary point condition due to (left) Kuhn-Tucker, (right) Fritz-John. $\lambda = 0$ (dotted line) , $\lambda = 0.5$ (dashed line) and $\lambda = 1$ (solid line).

Before we give the example, we must define what is meant by a stationary point in a constrained minimax framework.

Definition 4.1 $\mathbf{x} \in \mathcal{D}$ *is a stationary point of the constrained minimax problem in (4.1) if*

$$\mathbf{0} \in M(\mathbf{x})$$

We now illustrate why the Fritz-John condition is preferred over Kuhn-Tucker in the following example. We have the following problem

$$\begin{array}{lcl} \min_x F(\mathbf{x}) & = & x_1 \\ \text{subject to} & & \\ c_1(\mathbf{x}) \equiv -x_2 + x_1^2 & \leq & 0 \\ c_2(\mathbf{x}) \equiv x_2 + x_1^2 & \leq & 0 \end{array} \tag{4.10}$$

An illustration of the problem is given in figure 4.2 (left). From the constraints it is seen that the feasible domain $\mathcal{D}$ is the point $\mathbf{x} = [\ 0,\ 0\ ]$, so the solution must be that point regardless of F. If wee used the Kuhn-Tucker stationary point condition, then the convex hull (dashed line) would not have $\mathbf{0}$ as part of its set, and hence the point would not be stationary due to the definition, which is obviously incorrect.

When the Fritz-John stationary point condition is used, the convex hull can be drawn as indicated on the figure by the solid lines, for multiple values of λ. Here we see that $\mathbf{0}$ can be a part of the convex hull if $\lambda = 0$. So the Fritz John condition can be used to define the stationary point successfully.

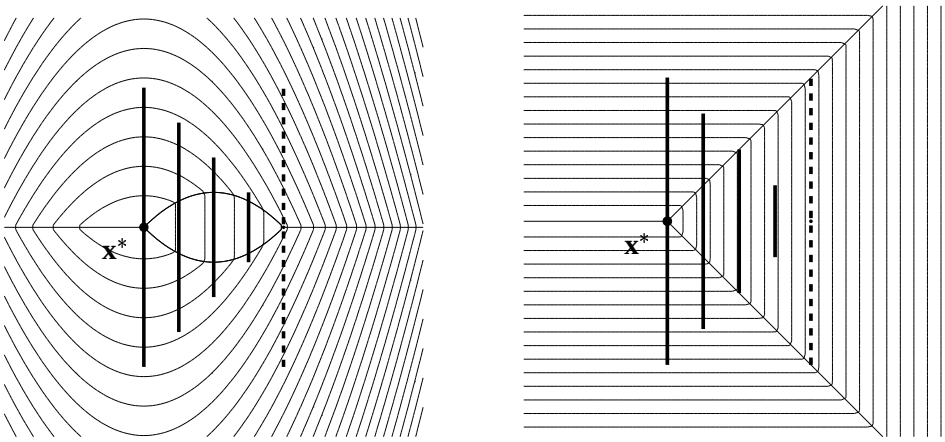

Figure 4.2: The dashed line shows the convex hull due to the Kuhn-Tucker stationary point condition, for $\lambda = 1$. The solid line indicate Fritz-John, for $\lambda = 0, 0.25, \ldots, 1$.

The Fritz-John conditions is, however, not without its drawbacks. If we linearize the constraints in (4.10), then we have the situation shown in figure 4.2 (right). Here we see that the line between $c_1(\mathbf{x})$ and $c_2(\mathbf{x})$ can be a stationary point regardless of F, even when F is increasing along the line.

The last example shows that it is not possible to generalize proposition 2.1 to the constrained case, because if $\lambda = 0$ then it is not certain that $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$ for every direction at $\mathbf{x}$.

As defined in [Mad86] we introduce the *feasible direction.*

Definition 4.2 *For $\mathbf{d} \in \mathbb{R}^n$ and $\mathbf{d} \neq 0$, a feasible direction from $\mathbf{x} \in \mathcal{D}$ exits if $\exists \varepsilon > 0$ so that $\mathbf{x} + t\mathbf{d} \in \mathcal{D}$ for $0 \leq t \leq \varepsilon$.*

Proposition 4.1 *If* $C(\mathbf{x}) = 0$ *and* $\mathbf{d}$ *is a feasible direction, then* $C'_{\mathbf{d}}(\mathbf{x}) \leq 0$.

Proof: [Mad86] pp. 53.

This means that the directional derivative for the constraints $C'_{\mathbf{d}}(\mathbf{x})$ must be negative or zero when whe are at the border of $\mathcal{D}$ and go in a feasible direction. This must hold since $C(\mathbf{x})$ is a convex function.

One way we can avoid the situation described in the last example is to assume that $C(\mathbf{x})$ is negative for some $\mathbf{x}$. In that case the feasible domain $\mathcal{D}$ would not be a point, and F would have an influence on the stationary point condition in such a case because $\lambda > 0$. This leads to the following proposition that holds for stationary points.

Proposition 4.2 *Let* $\mathbf{x} \in \mathcal{D}$. *If* $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$ *for all feasible directions* $\mathbf{d}$ *from* $\mathbf{x}$ *then* $\mathbf{0} \in M(\mathbf{x})$. *On the other hand, the following holds:*

$$\mathbf{0} \in M(\mathbf{x}) \text{ and } \exists \mathbf{y} \in \mathbb{R}^n : C(\mathbf{y}) < 0$$

then $F'_{\mathbf{d}}(\mathbf{x}) \geq 0$ *for all feasible directions* $\mathbf{d}$.

Proof: [Mad86] pp. 54

We see that the first part of the proposition is very similar to the unconstrained case. The second part of the proposition deals with the case where the unconstrained problem at $\mathbf{x}$ would have a negative directional derivative, so in the constrained case where $C(\mathbf{x}) = 0$ we are stopped by constraints at $\mathbf{x}$. An illustration of a simple case is given in figure 4.3.

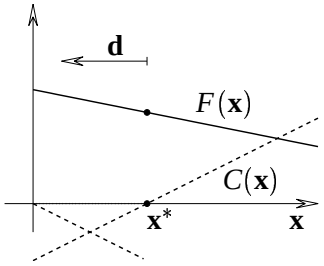

Figure 4.3: The point $\mathbf{x}^*$ indicates a solution to the constrained problem, and the feasible direction is indicated by $\mathbf{d}$.

We see that when $C(\mathbf{x}) = 0$ and $\lambda > 0$ because $\exists \mathbf{y} \in \mathbb{R}^n : C(\mathbf{y}) < 0$ then the convex hull of the Fritz-John stationary point condition can be written as

$$\mathbf{0} = \lambda \mathbf{f} + (1-\lambda)\mathbf{c} \;\Leftrightarrow\; \mathbf{0} = \mathbf{f} + \mu \mathbf{c}\,, \quad \mu = (1-\lambda)/\lambda\,, \tag{4.11}$$

where $\mu > 0$, $\mathbf{f} \in \partial F(\mathbf{x})$ and $\mathbf{c} \in \partial C(\mathbf{x})$. We see that in this case the Fritz-John and Kuhn-Tucker stationary conditions are equivalent. Next we use the feasible direction $\mathbf{d}$.

$$\mathbf{0} = \mathbf{f}^T \mathbf{d} + \mu \mathbf{c}^T \mathbf{d}\,. \tag{4.12}$$

And from proposition 4.1 we see that if $\mathbf{d}$ is a feasible direction, then it must hold that $C'_{\mathbf{d}}(\mathbf{x}) \leq 0$ and therefore due to the definition of the directional derivative in (2.13) and $\mu > 0$, we have that

$$C'_{\mathbf{d}}(\mathbf{x}) \leq 0 \;\Rightarrow\; \mu \mathbf{c}^T \mathbf{d} \leq 0\,, \tag{4.13}$$

which is obvious as illustrated in figure 4.3. This leads to

$$\mathbf{f}^T \mathbf{d} \geq 0 \;\Rightarrow\; F'_{\mathbf{d}}(\mathbf{x}) \geq 0\,. \tag{4.14}$$

4.2 Strongly Unique Local Minima

A strongly unique local minimum in the constrained case, is a minimum, that is not a plane or a line etc. but is a point and can be defined uniquely by using first order information.

As in the unconstrained case, algorithms will have quadratic final convergence, when going to a strongly unique local minimum. The following proposition defines the condition that has to be satisfied in order to have a strongly unique local minimum.

Proposition 4.3 *For $\mathbf{x} \in \mathcal{D}$ we have a strongly unique local minimum if*

$$\mathbf{0} \in \text{int}\{M(\mathbf{x})\} \ .$$

Proof: [Mad86, p. 57].

When $\mathbf{x}$ is interior to the map $M(\mathbf{x})$ we know from the strict separation theorem 2.1 that for all directions $\mathbf{d}$

$$\mathbf{v}^T\mathbf{d} \geq 0 \ , \quad \mathbf{v} \in M(\mathbf{x}) \tag{4.15}$$

If $C(x) < 0$ we just have the unconstrained problem and then we know that $F'_{\mathbf{d}}(\mathbf{x}) > 0$ for all directions $\mathbf{d}$.

We can, however, not generalize this to the directional gradients for $C(\mathbf{x}) = 0$, as in the unconstrained case, because $C(\mathbf{x})$ does not influence the landscape of $F(\mathbf{x})$ in any way – we shall later see that such an influence is possible through the use of a *penalty function*.

For the constrained case we can not directly make a statement that says that $F'_{\mathbf{d}}(\mathbf{x}) > 0$ for all directions $\mathbf{d}$, when $\mathbf{x}$ is a strongly unique local minimum. It is, however, possible to say something about $F'_{\mathbf{d}}(\mathbf{x})$ if $\mathbf{d}$ is a *feasible* direction. We will briefly give an explanation inspired by a proof in [Mad86, p. 56].

We assume that $\mathbf{0} \in M(\mathbf{x})$, and that there exists a vector $\mathbf{z} \in \mathbb{R}^n$. Then we introduce a kind of radius measure $\varepsilon > 0$ so that $\{\|\mathbf{z}\| < \varepsilon\} \subseteq M(\mathbf{x})$. This is illustrated in figure 4.4 with the ℓ_2 norm.

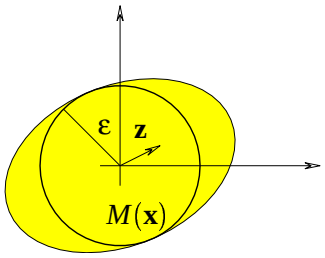

Figure 4.4: $\mathbf{z}$ is defined inside the circle, which is a subset of $M(\mathbf{x})$, so that $\{\|\mathbf{z}\| < \varepsilon\} \subseteq M(\mathbf{x})$.

Let $\mathbf{d} \in \mathbb{R}^n$ be a *feasible direction*, i.e., $C'_{\mathbf{d}}(\mathbf{x}) \leq 0$. Then we introduce a vector that lives on the edge of the circle (in the ℓ_2 norm case) so that $\mathbf{v} = \varepsilon\mathbf{d}/\|\mathbf{d}\|$ and $\|\mathbf{v}\| = \varepsilon$. Then by using the stationary point condition (Fritz-John) we can write the vector $\mathbf{v}$ as

$$\mathbf{v} = \{\lambda\mathbf{f} + (1-\lambda)\mathbf{c} \mid \mathbf{f} \in \partial F(\mathbf{x}), \mathbf{c} \in \partial C(\mathbf{x})\} \ , \tag{4.16}$$

We can do this since the Fritz-John condition can create any vector in $M(\mathbf{x})$ and $\mathbf{v}$ belongs to a subset of $M(\mathbf{x})$. By taking the inner product of $\mathbf{v}$ and the feasible direction $\mathbf{d}$ we see that $\mathbf{v}^T\mathbf{d} = \varepsilon\|\mathbf{d}\|$, which leads to

$$\mathbf{v}^T\mathbf{d} = \lambda\mathbf{f}^T\mathbf{d} + (1-\lambda)\mathbf{c}^T\mathbf{d} = \varepsilon\|\mathbf{d}\| \tag{4.17}$$

Because we go in a feasible direction $C'_{\mathbf{d}}(\mathbf{x}) \leq 0$ due to proposition 4.1 we get the following inequality

$$F'_{\mathbf{d}}(\mathbf{x}) \geq \lambda\mathbf{f}^T\mathbf{d} \geq \varepsilon\|\mathbf{d}\| \ . \tag{4.18}$$

So even in the constrained case a strongly unique local minimum $F'_{\mathbf{d}}(\mathbf{x})$ is strictly positive for all feasible directions.

4.3 The Exact Penalty Function

In constrained minimax optimization, as in other types of optimization, the constraints $\mathbf{c}_i(\mathbf{x})$ does not have an influence on the minimax landscape $F(\mathbf{x})$. Because of this, the theory of constrained minimax is somewhat different from the unconstrained theory. We can, however, use the unconstrained theory in constrained minimax if we use an *exact penalty function*.

An exact penalty function, is a function that has the same minimizer $\mathbf{x}^*$ as the constrained problem. Therefore by using an exact penalty function we can use all the tools and theory from unconstrained minimax on a minimax problem with constraints. We now introduce an exact penalty function for minimax that has the following form for $\sigma > 0$

$$\min_x P(\mathbf{x},\sigma) \equiv \max\{\ \mathbf{f}_j(\mathbf{x})\ ,\ \mathbf{f}_j(\mathbf{x}) + \sigma\mathbf{c}_i(\mathbf{x})\ \}\ , \tag{4.19}$$

where $i = 1,\ldots,m,\ j = 1,\ldots,q$. We see that if $\sigma > 0$, and $\mathbf{x} \notin \mathcal{D}$ then $\{\ \exists i \mid \mathbf{c}_i(\mathbf{x}) > 0\}$. This means that

$$\mathbf{f}_j + \sigma\mathbf{c}_i(\mathbf{x}) > \mathbf{f}_j(\mathbf{x}) \ \Rightarrow\ P(\mathbf{x},\sigma) = \mathbf{f}_j(\mathbf{x}) + \sigma\mathbf{c}_i(\mathbf{x}) \tag{4.20}$$

Of course the situation is the same vice versa, when $\mathbf{x}$ is feasible. Then $\mathbf{c}_i(\mathbf{x}) \leq 0$ for all i, and in this situation we have

$$\mathbf{f}_j(\mathbf{x}) \geq \mathbf{f}_j(\mathbf{x}) + \sigma\mathbf{c}_j(\mathbf{x}) \Rightarrow P(\mathbf{x},\sigma) = \mathbf{f}_j(\mathbf{x})\ . \tag{4.21}$$

Notice that in this case σ is irrelevant. In order to simplify the following notation we introduce the inner functions of $P(\mathbf{x},\alpha)$

$$\begin{array}{lcll} p_t(\mathbf{x},\sigma) & = & \mathbf{f}_j(\mathbf{x}) & ,\ t = 1,\ldots m \\ p_t(\mathbf{x},\sigma) & = & \mathbf{f}_j(\mathbf{x}) + \sigma\mathbf{c}_i(\mathbf{x}) & ,\ t = m+1,\ldots,z \end{array} \tag{4.22}$$

where z is defined in the following discussion in (4.34). The general setup of $p_t(\mathbf{x},\sigma)$ that

will be used here is

$$\begin{array}{lcl} p_1(\mathbf{x},\sigma) & = & f_1(\mathbf{x}) \\ \vdots & \vdots & \vdots \\ p_m(\mathbf{x},\sigma) & = & f_m(\mathbf{x}) \\ \vdots & \vdots & \vdots \\ p_{m+1}(\mathbf{x},\sigma) & = & f_1(\mathbf{x}) + \sigma c_1(\mathbf{x}) \\ \vdots & \vdots & \vdots \\ p_{m+m}(\mathbf{x},\sigma) & = & f_m(\mathbf{x}) + \sigma c_1(\mathbf{x}) \\ \vdots & \vdots & \vdots \\ p_{z-m}(\mathbf{x},\sigma) & = & f_1(\mathbf{x}) + \sigma c_p(\mathbf{x}) \\ \vdots & \vdots & \vdots \\ p_z(\mathbf{x}) & = & f_m(\mathbf{x}) + \sigma c_p(\mathbf{x}) \end{array} \tag{4.23}$$

Further we have that $P(\mathbf{x},\sigma) \equiv \max\{p_t(\mathbf{x},\sigma)\}$ and the gradients are denoted with $\mathbf{p}'_t(\mathbf{x},\sigma)$, finally the generalized gradient is denoted $\partial P(\mathbf{x},\sigma)$.

To resume the discussion from previous, a $\sigma > 0$ is not enough to ensure that $P(\mathbf{x},\sigma)$ solves the constrained problem in (4.1), in fact σ has to be *large enough*. We illustrate this by an example

We have the following equations

$$\begin{array}{lcl} f_1(x) & = & \frac{1}{2}x^2 \\ f_2(x) & = & \frac{1}{2}(x-1)^2 \\ c_1(x) & = & \frac{1}{3}x - \frac{1}{10}\ . \end{array} \tag{4.24}$$

And we solve the problem

$$\begin{array}{l} \min_x F(x) \\ \text{s.t.} \\ C(x) = c_1(x) \leq 0\ , \end{array} \tag{4.25}$$

by using the penalty function $P(x,\sigma)$ in (4.19). So the feasible area $\mathcal{D}$ is defined by $C(x) \leq 0$, and the optimal feasible solution to the constrained minimax problem is $\mathbf{x}^* = 0.3$. However, we see that the unconstrained minimax function $F(x) = \max\{f_1(x), f_2(x)\}$ has an unconstrained minimum at $x_u^* = 0.5$. The problem is shown in figure 4.5

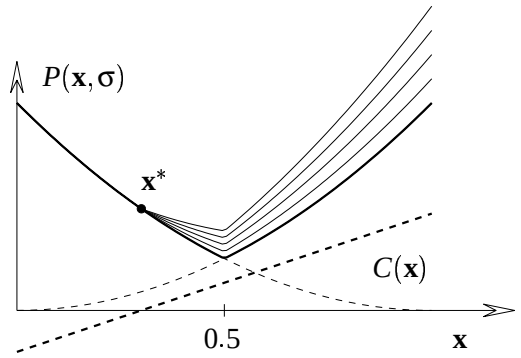

Figure 4.5: The penalty function $P(x,\sigma)$ plotted for $\sigma = 0, 0.25, \ldots, 1$.

The figure clearly illustrate that $\sigma = 1$ is not large enough, because the unconstrained minimizer of $P(x,\sigma)$ is at $x = 0.5$. Indeed if we let σ grow, we will eventually get an unconstrained minimizer of $P(\mathbf{x},\sigma)$ at $x = 0.3$ which is the solution to (4.25).

The example raise a basic question. How can we detect if σ is large enough? When the constrained solution is known, the size of σ can be calculated. We give an example based on the previous example.

At $x^* = 0.3$ we see that there must be two active functions

$$\begin{array}{rcl} p_2(x) & = & \frac{1}{2}(x-1)^2 \\ p_4(x) & = & \frac{1}{2}(x-1)^2 + \sigma(\frac{1}{3}x - \frac{1}{10}) \ . \end{array} \tag{4.26}$$

We see that only $p_4(x)$ is influenced by σ, and by using that $F_d'(x) \geq 0$ at a stationary point we have that

$$\mathbf{p}_4'(x) \geq 0 \ \Rightarrow \ \sigma \geq 3(1-x) \ . \tag{4.27}$$

This yield $\sigma \geq 2.1$ for $x = x^*$. As a theoretical insight, we should choose $\sigma > 2.1$, because then a *posible* strongly unique local minimum at x^*, would give the desired quadratic convergence. If $\sigma = 2.1$ then $F_d'(x) \geq 0$ and this will then not be a strongly unique local minimum.

With the aim of making an algorithm for constrained minimax, we can not evaluate the size of σ as sketched above, for that simple reason that we do not know the solution x^*. The simplest way to check if σ should be increased is to check if any of the constraints have been violated, that is

```
if C(x) > 0
  increase σ
end
```
(4.28)

If the constraints are violated then we just increase $\sigma > 0$ by following some specific scheme, e.g. multiply σ with 10 when $C(\mathbf{x}) > 0$. Such a scheme is also implemented in the fortran program MINCIN for linearly constrained minimax optimization, that is part of the package for Robust Subroutines for Non-linear Optimization [MNS02].

4.4 Setting up the Linear Subproblem

As described in details in chapter 3, we need to sequentially solve a linear subproblem in order to solve the non-linear minimax problem in (2.7) and (4.1). When using an exact penalty function (4.19) we have to set up the linear subproblem in a special way, that is descriped in the following.

The exact penalty function can be written as

$$\min_x P(\mathbf{x},\sigma) \equiv \max\{ f_j(\mathbf{x}) \ , f_j(\mathbf{x}) + \sigma c_i(\mathbf{x})\} = \max\{p_t(\mathbf{x},\sigma)\} \ , \tag{4.29}$$

where $p_t(\mathbf{x},\sigma)$ is defined in (4.22). We see that in order to write the exact penalty function into a linear program, we first need the m functions $f_j(\mathbf{x})$ and then all the combinations of i and j in $f_j(\mathbf{x}) + \sigma c_i(\mathbf{x})$. When $i = 1,\ldots,m$ and $j = 1,\ldots,q$ then the exact penalty function

gives rise to *at least* mq extra inner function not counting the first m inner functions $f_j(\mathbf{x})$. We give an example of this from the problem in (4.25) shown in figure 4.5.

We have that the number of inner functions is $m = 2$, and the number of constraints is $q = 1$. This gives the following linear system

$$\begin{array}{lll} \min_{h,\alpha} & G(\mathbf{h},\alpha) & \equiv \quad \alpha \\ \text{s.t.} & p_t(\mathbf{x},\sigma) + \mathbf{p}_t'(\mathbf{x},\sigma)^T\mathbf{h} & \leq \quad \alpha\,, \end{array} \tag{4.30}$$

where $t = 1,\ldots,4$.

The situation covered in the above example is for inequality constraints of the form $\leq$. We can however also handle equality constraints quite easily by just adding additional constraints to the LP problem. We can write an equality constraint $c_i(x) = \alpha$ as

$$\alpha \leq c_i(\mathbf{x}) \leq \alpha \;\Rightarrow \begin{cases} c_i(\mathbf{x}) \leq \alpha \\ -c_i(\mathbf{x}) \leq \alpha \end{cases} \tag{4.31}$$

So each equality constraint gives rise to two inequality constraints in the LP problem. We order the constraints so that the first r constraints corresponds to equality constraints and the rest $q-r$ corresponds to inequlity constraints. Then

$$\begin{array}{ll} c_i(\mathbf{x}) = 0 & \text{for } i = 1,\ldots,r \\ c_i(\mathbf{x}) \leq 0 & \text{for } i = r+1,\ldots,q\,, \end{array} \tag{4.32}$$

and

$$p_t(\mathbf{x},\alpha) = \begin{cases} f_j(\mathbf{x}) & \text{for } t = 1,\ldots,m \\ c_i(\mathbf{x}) = 0 & \text{for } t = m+1,\ldots,m(1+2r) \\ c_i(\mathbf{x}) \leq 0 & \text{for } t = m(1+2r)+1,\ldots,m(1+r+q)\,. \end{cases} \tag{4.33}$$

The total amount of constraints in the LP problem for m inner functions, p constraints where the r of them is equality constraints is

$$z = m + 2mr + m(q-r)\,. \tag{4.34}$$

4.5 An Algorithm for Constrained Minimax

We can now outline an algorithm that solves a constrained minimax problem by using an exact penalty function.

First we decide an initial penalty factor. This could e.g. be $\sigma = 1$. Then we form the linear programming problem as described above. The easiest way to do this is inside the test function itself.

Due to the nature of the exact penalty function $P(\mathbf{x},\sigma)$, we can solve it as an unconstrained minimax problem. Hence we can use the tools discussed in chapter 3 to solve $P(\mathbf{x},\sigma)$. Of course we kan not guarantee that $\sigma = 1$ is big enough so that the solution of $P(\mathbf{x},\sigma)$ corresponds to the constrained minimizer of $F(\mathbf{x})$. Hence wee need an update scheme like the one proposed in (4.28) to update σ. A stopping criterion for the algorithm will be discussed later. First we give a more formal outline of the algorithm.

Algorithm 4.4.1 CMINMAX

```
begin
  k := 0;  σ > 0;   x := x_0;   found := false;                    {1°}
  repeat
    x = slp(fun, fpar, x, opts);                                   {2°}
    if C(x) > 0
      Increase σ;
    else
      found := true;                                               {3°}
    k = k + 1;
  until found or k > k_max
end
```

{1°} The user selects the initial value of σ. If σ is large enough and $\mathbf{x}_0$ is feasible then the algorithm will behave as an *interior point* method. Otherwise it will behave as an *exterior point* method.

{2°} Here we use an unconstrained minimax solver. We can use the SLP or the CSLP algorithms from chapter 3, or Matlab's own minimax solver `fminimax` The function fun should be an exact penalty function, where σ is passed to fun by using fpar.

{3°} If $C(x) < 0$ then we have found an unconstrained minimizer inside the feasible region $\mathcal{D}$ and we should stop. Else if $C(x) = 0$ then a constrained minimizer has been found, and again we should stop.

We now give two examples that uses CMINMAX to find the constrained minimax solution. Both examples uses the Rosenbrock function subject to inequality and equality constraints and with the classic starting point at $\mathbf{x} = [-1.2, 1]^T$. We have $m = 4$ inner functions and $q = 1$ constraints.

First we look at the system where $\mathbf{x} \in \mathbb{R}^2$

$$\begin{array}{lrcl} \min_x & F(\mathbf{x}) & \equiv & \text{Rosenbrock} \\ s.t. & & & \\ & c_1(\mathbf{x}) & = & \mathbf{x}^T\mathbf{x} - 0.2 \leq 0 \end{array} \tag{4.35}$$

We find the minimizer of this problem by using the CMINMAX algorithm, with $\sigma_0 = 0.05$. The constrained minimizer is found at $\mathbf{x}^* = [0.4289, 0.1268]^T$, for $\sigma = 1$. Figure 4.6 illustrate the behavior of the algorithm for each σ.

At $\sigma = 0.05$ the algorithm finds a solution outside the feasible area indicated by the dashed circle on the figure. The algoritm then increase σ to 0.5, and we get closer to the feasible area but still we are infeasible. Finaly $\sigma = 5$ is large enough, and we find the constrained minimizer.

The following table is a printout of the values of the inner functions of $P(\mathbf{x}^*, \sigma)$ for each value of σ.

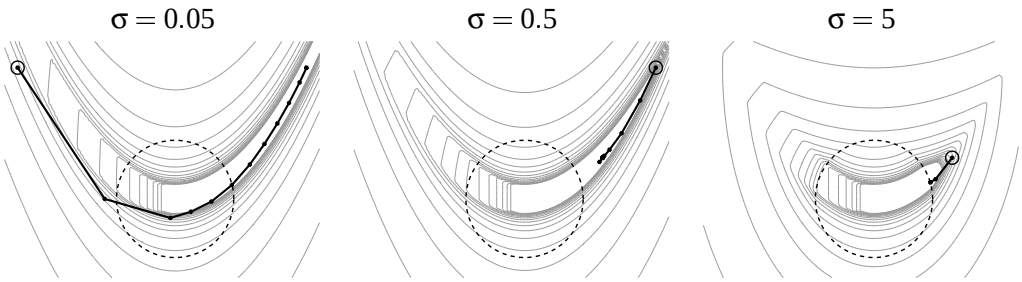

Figure 4.6: Iterations of the CMINMAX algorithm, with an infeasible starting point. The dashed circle contains the feasible area.

		$\sigma = 0.05$	$\sigma = 0.5$	$\sigma = 5$
f_1	p_1	0.0000	-0.4048	-0.5711
f_2	p_2	0.0000	0.4048	0.5711
f_3	p_3	0.0000	0.4048	0.5711
f_4	p_4	-0.0000	-0.4048	-0.5711
	p_5	0.0900	-0.2785	-0.5711
	p_6	0.0900	0.5312	0.5711
	p_7	0.0900	0.5312	0.5711
	p_8	0.0900	-0.2785	-0.5711

From the above function evaluations we notice that $\max\{f_i\} = \max\{p_t\}$ for $\sigma = 5$, this will become important in the later discussion regarding stopping criterions for CMINMAX.

Next we present another example of constrained minimax optimization, where we also use the Rosenbrock function, but with slightly different constraints. We have the following problem

$$\begin{array}{lrcl} \min_x & F(\mathbf{x}) & \equiv & \text{Rosenbrock} \\ s.t. & & & \\ & c_1(\mathbf{x}) & = & \mathbf{x}^T\mathbf{x} - 0.2 = 0 \end{array} \tag{4.36}$$

For this problem with equality constraints the constrained solution is found at $\mathbf{x}^* = [0.4289, 0.1268]^T$ for $\sigma = 5$, this is the same minimizer as in the previous example. The behavior for CMINMAX with $\sigma_0 = 0.05$ can be seen in figure 4.7.

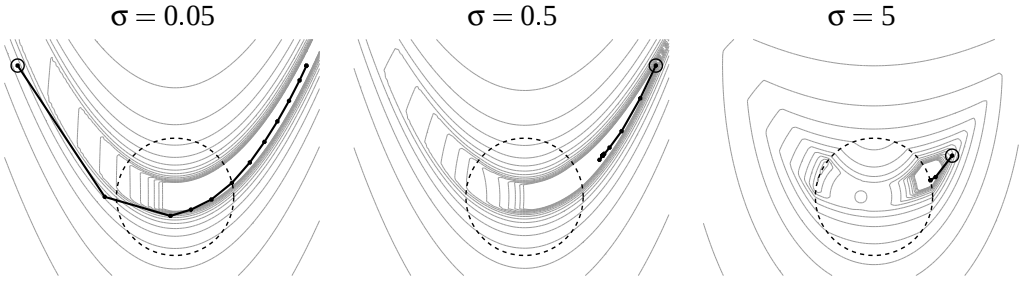

Figure 4.7: Iterations of the CMINMAX algorithm, with an infeasible starting point. The dashed circle indicate the feasible area.

As shown on the figure for $\sigma = 5$ we have two minima. If we had chosen the initial penalty

factor $\sigma_0 = 5$ then we would have found *the other minimum* at $\mathbf{x}_2^* = [-0.3599, 0.2655]^T$.

Again we bring a table of the inner function values of $P(\mathbf{x}^*, \sigma)$, for each value of σ. Again for $\sigma = 5$ we have that $\max\{f_j\} = \max\{p_t\}$.

		$\sigma = 0.05$	$\sigma = 0.5$	$\sigma = 5$
f_1	p_1	0.0000	-0.4048	-0.5711
f_2	p_2	0.0000	0.4048	0.5711
f_3	p_3	0.0000	0.4048	0.5711
f_4	p_4	0.0000	-0.4048	-0.5711
	p_5	0.0900	-0.2785	-0.5711
	p_6	0.0900	0.5312	0.5711
	p_7	0.0900	0.5312	0.5711
	p_8	0.0900	-0.2785	-0.5711
	p_9	-0.0900	-0.5312	-0.5711
	p_{10}	-0.0900	0.2785	0.5711
	p_{11}	-0.0900	0.2785	0.5711
	p_{12}	-0.0900	-0.5312	-0.5711

From the tables of the inner functions in the two examples we saw that $\max\{f_j\} = \max\{p_t\}$ was always satisfied when the constrained minimizer was found, i.e. when σ was large enough.

When the feasible domain $\mathcal{D} \subseteq \mathbb{R}^n$ is more than a point, i.e. $\exists \mathbf{y} \in \mathbb{R}^n$ for which $C(\mathbf{y}) < 0$, then we can use the Fritz-John stationary point condition that says

$$0 \in \{\lambda \mathbf{f} + (1-\lambda)\mathbf{c} \mid \mathbf{f} \in \partial F, \mathbf{c} \in \partial C \}$$

where $\lambda > 0$. This indicate that at least one of the inner functions of the unconstrained minimax problem $F(\mathbf{x})$ has to be active at the constrained solution corresponding to the stationary point for $P(\mathbf{x}, \sigma)$ when σ is large enough. This is also indicated by proposition 4.2.

This knowledge can be used as a stopping criterion for CMINMAX. We check to see whether one of the inner function of $F(\mathbf{x})$ is active.

The inner functions of $F(\mathbf{x})$ correspond to the first $m = 4$ rows in the tables of the two previous examples. We see that

$$\{f_j(\mathbf{x}) \mid j = 1, \ldots, m\} = \{p_t(\mathbf{x}, \sigma) \mid t = 1, \ldots, m\} \,. \tag{4.37}$$

The solution to the exact penalty function in the k'th iteration of CMINMAX is denoted by $\mathbf{x}_k^*$. We can then define a stopping criterion

$$\begin{aligned} &\|P(\mathbf{x}_k^*, \sigma) - F(\mathbf{x}_k^*)\| \leq \varepsilon \;\Leftrightarrow \\ &\|P(\mathbf{x}_k^*, \sigma) - \max\{f_j(\mathbf{x}_k^*)\}\| \leq \varepsilon \;\Leftrightarrow \\ &\|P(\mathbf{x}_k^*, \sigma) - \max\{p_t(\mathbf{x}_k^*, \sigma)\}\| \leq \varepsilon \,, \quad t = 1, \ldots, m \,, \end{aligned} \tag{4.38}$$

where term $0 < \varepsilon \ll 1$ handles the numerical issues regarding the stopping criterion. If the above criterion is not satisfied then it indicates that

$$\max\{f_j(\mathbf{x}_k^*)\} < \max\{p_t(\mathbf{x}_k^*, \sigma)\} \,. \tag{4.39}$$

The only way this can happen is if $\mathbf{x}_k^* \notin \mathcal{D}$. The stoping criterion (4.38) will also work if there is an unconstrained minimum inside the feasible domain $\mathbf{x}_k^* \in \mathcal{D}$, because the convexity of $P(\mathbf{x}, \sigma)$ ensures that

$$\max\{ f_j(\mathbf{x})\} \geq \max\{p_t(\mathbf{x}, \sigma)\} \ .$$

The major advantage with this stopping criterion is that we do not have to supply CMINMAX with the value of $C(\mathbf{x})$.

The disadvantage by using the above stopping criterion (4.38) is that it introduces a new preset parameter ε to the CMINMAX algorithm. However, we can avoid this preset parameter by seeing that (4.38) should interpreted as a check for whether some inner functions of $P(\mathbf{x}, \sigma)$ is active. From the theory in chapter 2 we know that if $p_t(\mathbf{x}, \sigma)$ is active then the corresponding multiplier will be positive. This gives rise to the following preset parameter free, but equivalent stopping criterion

$$\{\max\{\lambda_t\} > 0 \mid t = 1, \ldots, m \} \ \Rightarrow \ \text{stop} \tag{4.40}$$

The demand that $C(\mathbf{x}_k^*) \leq 0$ is not needed when we use (4.38) or (4.40) because if $\mathbf{x} \notin \mathcal{D}$ then $\max\{ f_j(\mathbf{x})\} < \max\{p_t(\mathbf{x}, \sigma)\}$ for $\sigma > 0$ as shown in (4.20) and (4.21). Hence $p_t(\mathbf{x}, \sigma)$ for $t = 1, \ldots, m$ can not be active, and there will be no corresponding positive multiplier.

As an addition to this presentation, we give a short argumentation for why the following stopping criterion does *not* work

$$\|\mathbf{x}_{k-1}^* - \mathbf{x}_k^*\| \leq \varepsilon \ \Rightarrow \ \text{stop} \tag{4.41}$$

As we saw in the start of this section on figure 4.5, the solution $\mathbf{x}_k^*$ is not guaranteed to move when σ is increased. Therefore $\|\mathbf{x}_{k-1}^* - \mathbf{x}_k^*\| = 0$ could happen even if we have not found the constrained minimizer.

4.6 Estimating the Penalty Factor

In the previous description of the CMINMAX algorithm, an increase of the penalty factor σ was proposed if the current iterate $\mathbf{x}_k^*$ was not feasible. It was also suggested that in this case σ should be increased by a factor of ten. One could argue that the multiplication factor also could have been choosen as two, three or maybe even twelve. In other words, we have no idea of what the right size of the multiplication factor should be.

In this section we will take a closer look at the multiplication factor, and use the theory of constrained minimax to give us a value of σ_k^* that guarantees a shift of stationary point for $\mathbf{x}_k^* \notin \mathcal{D}$. In this way an increase of σ can be found that is connected to the problem being solved.

The motivation behind the choise of multiplication factor is that we want to find a σ that is large enough, so that the unconstrained minimizer $\mathbf{x}^*$ of

$$P(\mathbf{x}^*, \sigma) = \max\{ f_j(\mathbf{x}^*) + \sigma c_i(\mathbf{x}^*)\} \ ,$$

corresponds to the minimizer $\mathbf{x}$ of a minimax problem with constraints i.e.

$$\min_{\mathbf{x}} F(\mathbf{x}) \text{ s.t. } C(\mathbf{x}) \leq 0 \ .$$

However one could also look at this from a different perspective, as a question of finding the value of $\sigma = \sigma_k^*$ that will force a shift from a stationary point $\mathbf{x}_k^* \notin \mathcal{D}$ to a new stationary point $\mathbf{x}_{k+1}^*$. By iteratively increasing σ^*, shift of stationary points will be obtained, that eventually leads to an unconstrained minimizer of $P(\mathbf{x}^*)$ that solves the corresponding constrained problem.

We denote a shift of stationary point by $\rightsquigarrow$, and now a more precise formulation of the problem can be given. Find the smallest penalty factor σ_k that triggers a shift of stationary points $\mathbf{x}_k^* \rightsquigarrow \mathbf{x}_{k+1}^*$, where $\mathbf{x}_k^* \notin \mathcal{D}$.

First we look at the condition for a stationary point when a penalty function is being used. If we have a stationary point $\mathbf{x}$ so that $\mathbf{0} \in \partial P(\mathbf{x}, \sigma)$ and the constraints are violated $C(\mathbf{x}) > 0$ then

$$\mathbf{0} = \sum_{t \in \mathcal{A}} \lambda_t \mathbf{p}_t'(\mathbf{x}, \sigma) \ , \ \mathbf{p}_t'(\mathbf{x}, \sigma) \in \partial P(\mathbf{x}, \sigma) \ .$$

In (4.22) we saw that t corresponded to some combinations of $f_j(\mathbf{x})$ and $c_i(\mathbf{x})$. By using this correspondence we get

$$\mathbf{0} = \sum_{t \in \mathcal{A}} \lambda_t (\mathbf{f}_j'(\mathbf{x}) + \sigma \mathbf{c}_i'(\mathbf{x})) \ , \quad \mathbf{f}_j'(\mathbf{x}) \in \partial F(\mathbf{x}), \ \mathbf{c}_i'(\mathbf{x}) \in \partial C(\mathbf{x}) \ .$$

That $t \in \mathcal{A}$ means that $p_t(\mathbf{x}, \alpha) = f_j(\mathbf{x}) + \sigma c_i(\mathbf{x})$ is active. It is seen that $f_j(\mathbf{x})$ and $c_i(\mathbf{x})$ is part of this expression, and hence we call them *pseudo* active, i.e., $j \in \mathcal{P}_f$ and $i \in \mathcal{P}_c$. Then we can write

$$-\sum_{j \in \mathcal{P}_f} \lambda_j \mathbf{f}_j'(\mathbf{x}) = \sigma \sum_{i \in \mathcal{P}_c} \lambda_i \mathbf{c}_i'(\mathbf{x}) \ , \tag{4.42}$$

which leads to

$$\sigma = \frac{-\| \sum_{j \in \mathcal{P}_f} \lambda_j \mathbf{f}_j'(\mathbf{x}) \|^2}{\langle \sum_{j \in \mathcal{P}_f} \lambda_j \mathbf{f}_j'(\mathbf{x}), \ \sum_{i \in \mathcal{P}_c} \lambda_i \mathbf{c}_i'(\mathbf{x}) \rangle} \ . \tag{4.43}$$

We now have an expression that calculates σ when we know the multipliers and the gradients of the inner functions and constraints at $\mathbf{x}$.

We give a brief example based on the two examples in the previous section for $\sigma = 0.5$ where we find σ by using the above formula. In the two examples the solution was the same in the second iteration of CMINMAX where $\mathbf{x}_2^* = [0.5952 \ \ 0.3137]^T$.

The active inner functions at $\mathbf{x}_2^*$ was $p_t(\mathbf{x}_2^*, 0.5)$ with $t = 6, 7$ for the example with inequality constraints.

$$\begin{aligned} p_6(\mathbf{x}_2^*, 0.5) &= f_2(\mathbf{x}_2^*) + 0.5 c_1(\mathbf{x}_2^*) \\ p_7(\mathbf{x}_2^*, 0.5) &= f_3(\mathbf{x}_2^*) + 0.5 c_1(\mathbf{x}_2^*) \end{aligned}$$

Here are the numerical results

$$\lambda_{6:7} = \begin{bmatrix} 0.9686 \\ 0.0314 \end{bmatrix} \ , \quad \mathbf{f}_{2:3}'(\mathbf{x}_2^*) = \begin{bmatrix} -1.0000 & 0.0000 \\ 11.9034 & -10.0000 \end{bmatrix}$$

and $\mathbf{c}_1'(\mathbf{x}_2^*) = [1.1903 \;\; 0.6275]$. Note that due to the structure of $p_t(\mathbf{x}, \alpha)$ we use

$$\mathbf{c}_1'(\mathbf{x}_2^*) = \begin{bmatrix} 1.1903 & 0.6275 \\ 1.1903 & 0.6275 \end{bmatrix} ,$$

in the following calculations. Next we calculate σ

$$\mathbf{v} = \sum_{j \in \mathcal{P}_f} \lambda_j \mathbf{f}_j'(\mathbf{x}) = \mathbf{f}_{2:3}'(\mathbf{x}_2^*)^T \lambda_{6:7} = [-0.5952 \;\; -0.3137]^T$$

and

$$\mathbf{w} = \sum_{i \in \mathcal{P}_c} \lambda_i \mathbf{c}_i'(\mathbf{x}) = \mathbf{c}_1'(\mathbf{x}_2^*)^T \lambda_{6:7} = [1.1903 \;\; 0.6275]^T ,$$

which yield

$$\sigma = -\frac{\|\mathbf{v}\|^2}{\langle \mathbf{v}, \mathbf{w} \rangle} = 0.5000 .$$

We have now shown that we can calculate σ at $\mathbf{x}$, *if* we know the multipliers λ. We can also calculate the multipliers for a given value of σ but not by using (4.43). The value of σ changes the "landscape" of $P(\mathbf{x}, \sigma)$ which again also changes the multipliers λ.

The relation between λ and σ should become apparent, when looking upon λ as gradients [Fle00, 14.1.16]. For a problem with linear constraints, we can write

$$\begin{array}{lllll} \min_{\mathbf{x},\alpha} & G(\mathbf{x}, \alpha) & = & \alpha \\ \text{s.t.} & g(\mathbf{x}, \alpha) & \equiv & \mathbf{c}(\mathbf{x}) + \varepsilon \leq \alpha \end{array} \tag{4.44}$$

Notice the perturbation ε of the constraint, which is visualized in figure 4.8 for $\varepsilon_1 = 0$ and $\varepsilon_2 > 0$.

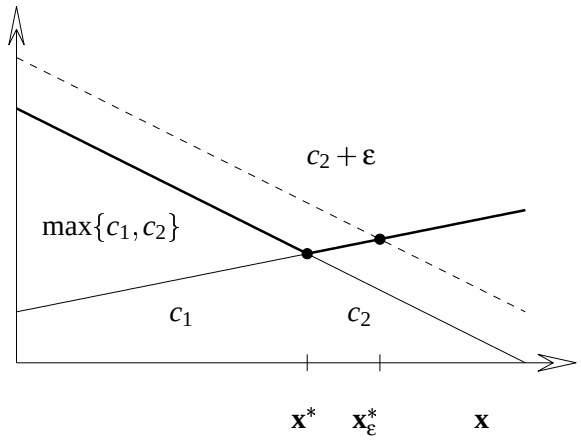

Figure 4.8: Constraint c_2 has been perturbed by ε.

The above system can be solved by the Lagrange function

$$\mathcal{L}(\mathbf{x}, \alpha, \lambda) = \alpha + \sum_i \lambda_i (c_i(\mathbf{x}) + \varepsilon_i - \alpha) , \tag{4.45}$$

and if we differentiate this with respect to ε_i we get

$$\frac{d\mathcal{L}}{d\varepsilon_i} = \lambda_i . \tag{4.46}$$

The above is only valid if the active set in the perturbed and unperturbed system is the same. This shows that the multipliers in fact can be looked upon as gradients. From [HL95, section 4.7] we see that the multipliers are the same as *shadow prices* in operations research. The above figure also illustrate why λ_i can not be negative for a stationary point. If e.g. $c_1(\mathbf{x})$ had a negative gradient, then $\lambda_2 < 0$.

The above shows that an increase of σ will lead to a change of λ_i and (4.43) indicated that a change of λ_i indicate a change of σ. This means that λ and σ is related to each other.

As we have seen in the previous, if the penalty factor is not high enough, then the CMINMAX algorithm will find an infeasible solution $\mathbf{x}_k^*$, and hence increase σ_k. It would be interesting to find that value $\sigma_k = \sigma_k^*$ where the penalty factor will be high enough to trigger a shift of stationary point, so that $\mathbf{x}_k^* \rightsquigarrow \mathbf{x}_{k+1}^*$.

To simplify this problem somewhat, it would be beneficial to ignore the multipliers all together, because of their relation to σ. To do this we use that

$$\mathbf{0} \in \partial P(\mathbf{x},\sigma) = \{ \sum_{t\in\mathcal{A}} \lambda_t \mathbf{p}_t'(\mathbf{x},\sigma) \mid \sum_{t\in\mathcal{A}} \lambda_t = 1, \lambda_t \geq 0 \} , \tag{4.47}$$

is equivalent with

$$\mathbf{0} \in \partial P(\mathbf{x},\sigma) = \mathtt{conv}\{\mathbf{p}_t'(\mathbf{x},\sigma) \mid t : p_t(\mathbf{x},\sigma) = P(\mathbf{x},\sigma)\} \tag{4.48}$$

A shift of stationary point will then occur for some σ_k^*, when $\mathbf{0} \notin \partial P(\mathbf{x}_k^*, \sigma_k^*)$, because then $\mathbf{x}_k^*$ will no longer be a stationary point, and then the CMINMAX algorithm will find a new stationary point at $\mathbf{x}_{k+1}^*$.

The sollution to the problem of finding σ_k^* is described in the following. However, we have two assumptions.

- At $\mathbf{x}_k^*$ only one constraint is pseudo active.
- $\mathbf{x}_k^* \notin \mathcal{D}$ which indicate $P(\mathbf{x}_k^*,\sigma) = p_t(\mathbf{x}_k^*,\alpha) = f_j(\mathbf{x}_k^*) + \sigma c_i(\mathbf{x}_k^*)$.

If we are at an infeasible stationary point and increment the penalty factor $\Delta\sigma$, then

$$\begin{aligned} P(\mathbf{x}_\mathbf{k}^*, \sigma+\Delta\sigma) &= f_j(\mathbf{x}_k^*) + (\sigma+\Delta\sigma)c_i(\mathbf{x}_k^*) \\ &= f_j(\mathbf{x}_k^*) + \sigma c_i(\mathbf{x}_k^*) + \Delta\sigma c_i(\mathbf{x}_k^*) \\ &= p_t(\mathbf{x}_k^*,\sigma) + \Delta\sigma c_i(\mathbf{x}_k^*) , \end{aligned} \tag{4.49}$$

which give rise to the stationary point condition

$$\mathbf{0} \in \partial P(\mathbf{x}_k^*, \sigma+\Delta\sigma) = \mathtt{conv}\{\mathbf{p}_t'(\mathbf{x}_k^*,\alpha) + \Delta\sigma \mathbf{c}_i'(\mathbf{x}_k^*)\} , \tag{4.50}$$

where $\{t \mid p_t(\mathbf{x}_k^*,\sigma) = P(\mathbf{x}_\mathbf{k}^*,\sigma)\}$ In this case we can illustrate the convex hull like the one shown in figure 4.9 left.

We are interested in finding that $\Delta\sigma$ that translates the convex hull $\partial P(\mathbf{x}_k^*,\sigma)$ so that $\mathbf{0}$ is at the edge of the hull, i.e. it holds that $\mathbf{0} \in \partial P(\mathbf{x}_k^*, \sigma+\Delta\sigma)$.

Figure 4.9 right, shows γ defined as the length from $\mathbf{0}$ to the intersection between the convex hull $\partial P(\mathbf{x}_k^*,\sigma)$ and $\mathbf{c}_i'(\mathbf{x}_k^*)$. This is a somewhat ambiguous definition because we always get two points $\mathbf{a}_i$ of intersection with the hull. We chose that point $\mathbf{a}_i$ is the vector $\mathbf{v} = \mathbf{a}_i$ that

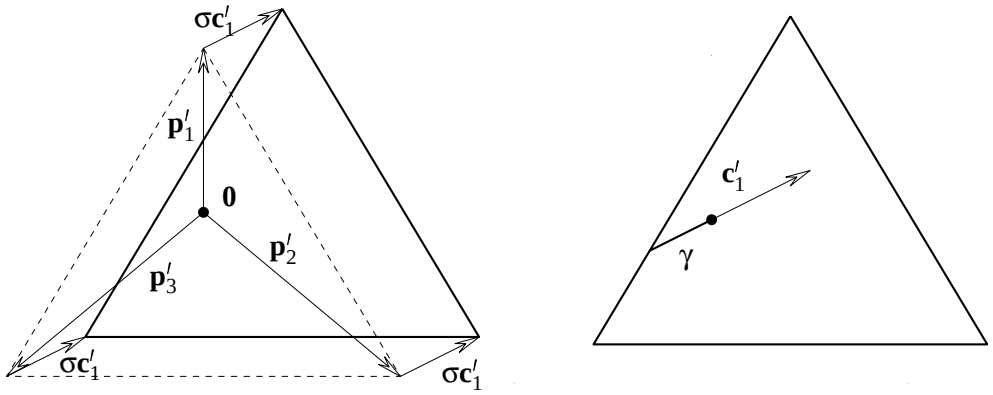

Figure 4.9: $\mathbf{p}'_t(\mathbf{x},\alpha)$ and $\mathbf{c}_i(\mathbf{x})$ has been substituted by $\mathbf{p}'_t$ and $\mathbf{c}'_i$ on the figure. left: The convex hull of $\partial P(\mathbf{x},0)$ is indicated by the dashed triangle. The solid triangle indicates the convex hull of $\partial P(\mathbf{x},\sigma)$. right: The length from $\mathbf{0}$ to the border of the hull in the opposite direction of $\mathbf{c}'_1$ is indicated by γ the length of the solid line.

has $0 > \mathbf{v}^T\mathbf{c}'_i(\mathbf{x}^*_k)$. We can now find the translation of the convex hull so that $\mathbf{0}$ is at the edge of the hull.

$$\Delta\sigma\mathbf{c}'_i(\mathbf{x}^*_k) = \gamma\frac{\mathbf{c}'_i(\mathbf{x}^*_k)}{\|\mathbf{c}'_i(\mathbf{x}^*_k)\|} \quad \Rightarrow \quad \Delta\sigma = \frac{\gamma}{\|\mathbf{c}'_i(\mathbf{x}^*_k)\|} = -\frac{\|\mathbf{v}\|^2}{\mathbf{v}^T\mathbf{c}'_i(\mathbf{x}^*_k)} \; . \tag{4.51}$$

We now have that

$$\mathbf{0} \in \text{edge}\Big\{\partial P(\mathbf{x}^*_k, \sigma_k + \Delta\sigma)\Big\} \; , \tag{4.52}$$

where σ_k indicate the value of σ at the k'th iteration of e.g. CMINMAX. The value $\sigma_k + \Delta\sigma$ does not, however, trigger a shift to a new stationary point $\mathbf{x}^*_k \rightsquigarrow \mathbf{x}^*_{k+1}$, because $\mathbf{x}^*_k$ is still stationary.

In order to find the trigger value σ^*_k, we add a small value $0 < \varepsilon \ll 1$ so that

$$\sigma^*_k = \sigma_k + \Delta\sigma + \varepsilon \; . \tag{4.53}$$

Then it will hold that

$$\mathbf{0} \notin \partial P(\mathbf{x}^*_k, \sigma^*_k) \; , \tag{4.54}$$

hence $\mathbf{x}^*_k$ is no longer stationary and e.g. CMINMAX will find a new stationary point at $\mathbf{x}^*_{k+1}$.

We illustrate the theory by a simple constrained minimax example with linear inner func-

tions and linear constraints. Consider the following problem

$$\begin{array}{lrcl} \min_{\mathbf{x}} & F(\mathbf{x}) & = & \max\{f_j(\mathbf{x})\}\,, \quad j=1,\ldots,4 \\ & f_1(\mathbf{x}) & = & -x_1-x_2 \\ & f_2(\mathbf{x}) & = & -x_1+x_2 \\ & f_3(\mathbf{x}) & = & x_1-4 \\ & f_4(\mathbf{x}) & = & -3x_1 \\ \text{s.t.} & & & \\ & C(\mathbf{x}) & = & \max\{c_i(\mathbf{x})\} \leq 0\,, \quad i=1,\ldots,p \\ & c_1(\mathbf{x}) & = & x_1+\frac{1}{2}x_2-1 \\ & c_2(\mathbf{x}) & = & x_1-\frac{1}{2}x_2+\frac{4}{10} \\ & c_3(\mathbf{x}) & = & -x_1-1 \end{array} \tag{4.55}$$

See figure 4.10 for an illustration of the problem in the area $x_1 \in [-1.5, 3]$ and $x_2 \in [-2, 2]$.

The figure indicates the feasible area $\mathcal{D}$ as the region inside the dashed lines. The generalized gradient $\partial P(x)$ is indicated by the solid lines.

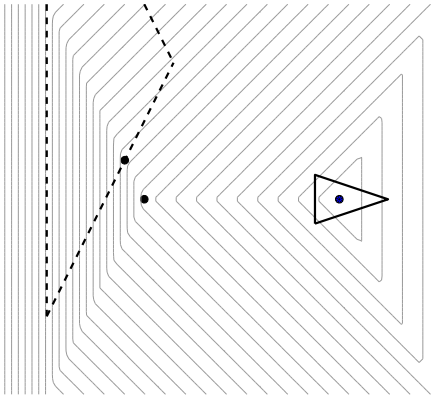

Figure 4.10: Illustration of the problem in (4.55). Constraints indicated by dashed lines. Generalized gradient indicated by solid line. $\sigma = 0$.

For certain values of σ, the problem contains an infinite number of stationary points. For all other values of σ it is only possible to get three different strongly unique local minima, as indicated by the dots on the figure.

The problem has two critical values of σ that will trigger a change of stationary point.

$$\sigma_1^* = 1+\varepsilon,\ \sigma_2^* = 1.5+\varepsilon\,. \tag{4.56}$$

At $\sigma = 1$ and $\sigma = 1.5$ the problem has an infinity of stationary points as illustrated in figure 4.11.

When $\sigma = 1 \pm \varepsilon$ where $\varepsilon = 0.0001$ there is a shift of stationary point as shown on figure 4.12.

In the interval $\sigma \in [1\ \ 1.5]$ the generalized gradient $\partial P(\mathbf{x}_2^*, \sigma)$ based on the multipliers performs a rotation like movement as shown in figure 4.13. Note that $\Delta\sigma$ is calculated on the generalized gradient defined only by the gradients $\mathbf{p}_t(\mathbf{x}_2^*, \sigma)$, and thus the multipliers are not taken into account.

When $\sigma = 1.5 \pm \varepsilon$ where $\varepsilon = 0.0001$, then there is a new shift of stationary point from $\mathbf{x}_2^*$ to $\mathbf{x}_3^*$ as shown on figure 4.14. When $\mathbf{x}_3^*$ is reached the algorithm should stop, because some $\lambda_t > 0$ for $t = 1,\ldots,m$ indicate that $\mathbf{x}_3^* \in \mathcal{D}$ i.e. $p_t(\mathbf{x}_3^*, 1.5001) = f_j(\mathbf{x}_3^*)$.

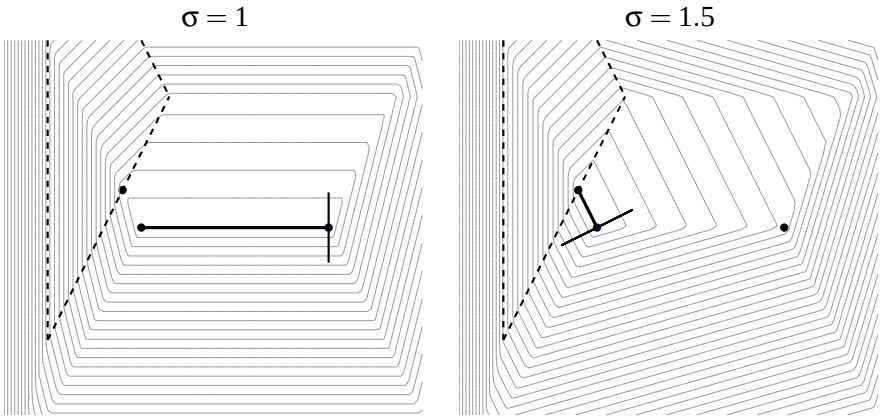

Figure 4.11: For $\sigma = 1$ and $\sigma = 1.5$ there is an infinity of solutions as illustrated by the solid lines. The line perpendicular to the solid line indicate the generalized gradient.

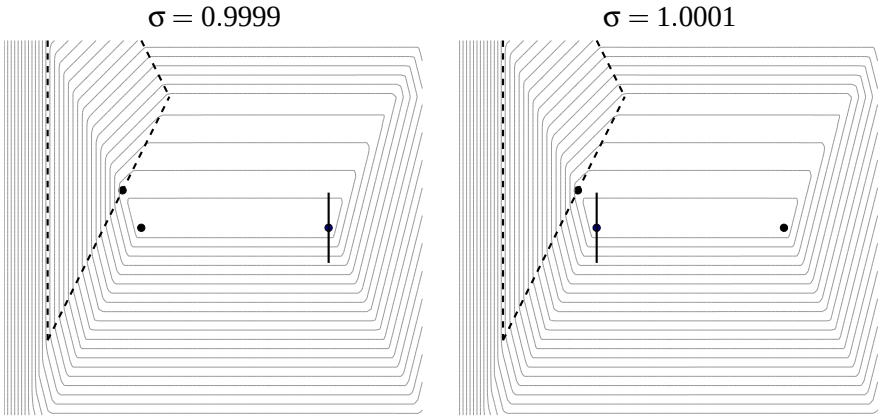

Figure 4.12: For $\sigma = 1 \pm 0.0001$ there is a shift of stationary point $\mathbf{x}_1^* \rightsquigarrow \mathbf{x}_2^*$.

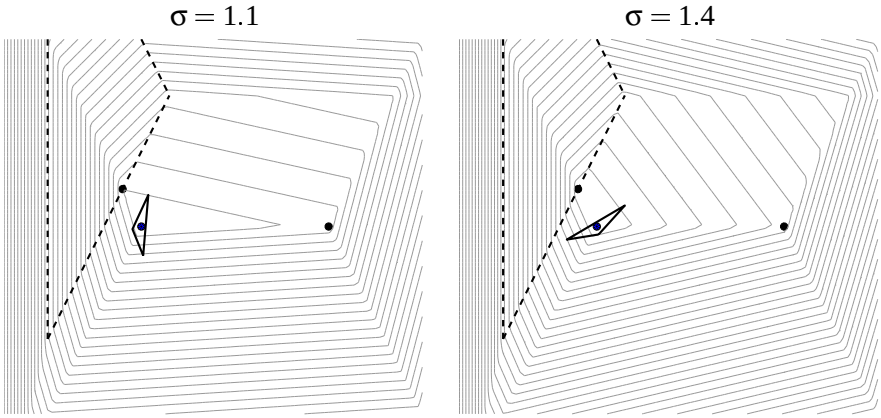

Figure 4.13: For $\sigma \in [1 \;\; 1.5]$, $\partial P(\mathbf{x}_2^*, \sigma)$ based on the multipliers performs a rotation like movement.

The intermediate and the final solution found for this example was

$$\mathbf{x}_1^* = [2 \;\; 0]^T \;, \quad \mathbf{x}_2^* = [0 \;\; 0]^T \;, \quad \mathbf{x}_3^* = [-0.2 \;\; 0.4]^T \;.$$

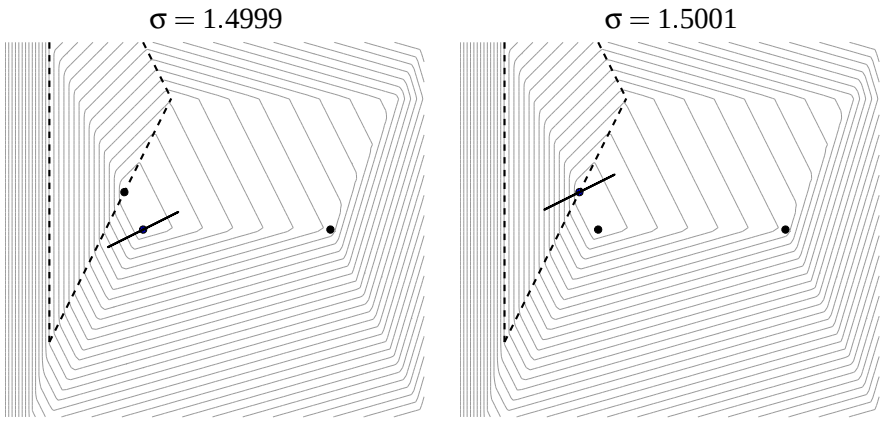

Figure 4.14: For $\sigma = 1.5 \pm 0.0001$ a shift occur from the stationary point $\mathbf{x}_2^*$ to $\mathbf{x}_3^*$.

For e.g. $\sigma = 0.1234$ and $\mathbf{x} = \mathbf{x}_1^*$ the convex hull is defined by the vectors

$$\mathbf{p}_9' = \begin{bmatrix} -0.8766 \\ -1.0617 \end{bmatrix}, \mathbf{p}_{10}' = \begin{bmatrix} -0.8766 \\ 0.9383 \end{bmatrix}, \mathbf{p}_{11}' = \begin{bmatrix} 1.1234 \\ -0.0617 \end{bmatrix}.$$

The pseudo active constraint gradient is $\mathbf{c}_2' = [1.0 \ \ -0.5]^T$. The intersection with the convex hull is found at $\mathbf{v} = [-0.8766 \ \ 0.4383]^T$.

$$\Delta\sigma = -\frac{\|[-0.8766 \ \ 0.4383]\|^2}{\langle [-0.8766 \ \ 0.4383], [1.0 \ \ -0.5] \rangle} = 0.8766 ,$$

which yield $\sigma + \Delta\sigma = 1.0000$. Then for $\sigma_1^* = 1.0000 + \varepsilon$, we get a shift of stationary point $\mathbf{x}_1^* \rightsquigarrow \mathbf{x}_2^*$.

Until now, we have not taken the multipliers into account, because they influence the penalty factor, and vice versa. It turns out, however, that if the edge of the convex hull is known, where there is a point of intersection $\mathbf{v}$ with $\mathbf{c}$, so that $\mathbf{v}^T\mathbf{c} < 0$, then we can solve the problems by using the multipliers quite easily.

According to [HL95, Chapter 6] every linear programming problem is associated with another programming problem, called the *dual* problem. Let us say that the LP problem in (3.4) is the primal problem.

$$(P) \qquad \min_{\hat{\mathbf{x}}} \ \mathbf{g}'(\mathbf{x}, \alpha)^T \hat{\mathbf{x}} \quad \text{s.t.} \quad \mathbf{A}\hat{\mathbf{x}} \leq \mathbf{b} ,$$

then, by using [HL95, Table 6.14], we can find the corresponding dual problem

$$(D) \qquad \max_{\mathbf{y}} \ \mathbf{b}^T\mathbf{y} \quad \text{s.t.} \quad \mathbf{A}^T\mathbf{y} = \mathbf{g}'(\mathbf{x}, \alpha) .$$

Surprisingly it turns out, that there is a connection between the distance γ to the edge of the generalized gradient $\partial P(\mathbf{x}, \sigma)$ and the dual problem. Both can be used to find the penalty factor. In other words, the dual solution to the primal problem, can be visualized as a distance from origo to the edge of a convex hull.

We give an example of this, that is based on the previous examples. At $\mathbf{x} = [2,0]^T$, we had three active functions $\mathbf{f}_j(\mathbf{x})$, $j = 1,\ldots,3$.

$$\begin{bmatrix} \mathbf{f}_1'(\mathbf{x}) & \mathbf{f}_2'(\mathbf{x}) & \mathbf{f}_3'(\mathbf{x}) \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} .$$

And the solution is $\lambda_1 = 0.25$, $\lambda_2 = 0.25$ and $\lambda_3 = 0.5$. From the previous we know that the intersection point on the convex hull with $\mathbf{c}_2'(\mathbf{x})$ is at the line segment

$$\{\mathbf{u} \mid \mathbf{u} = \lambda_1 \mathbf{f}_1'(\mathbf{x}) + \lambda_2 \mathbf{f}_2'(\mathbf{x}), \sum \lambda_i = 1, \lambda_i \geq 0, i = 1,2\}$$

So we can write the following system of equations

$$\begin{bmatrix} \mathbf{f}_1'(\mathbf{x}) & \mathbf{f}_2'(\mathbf{x}) & \mathbf{c}_2'(\mathbf{x}) \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \sigma \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} ,$$

where we, compared with the previous system, have replaced a column in $\mathbf{A}^T$ and a row in $\mathbf{y}$. We get the following result $\lambda_1 = 0.25$, $\lambda_2 = 0.75$ and $\sigma = 1$. We see that $\sigma^* = \sigma + \varepsilon$ would trigger a shift of stationary point in the primal problem.

At $\mathbf{x} = [0,0]^T$ the penalty factor can be found by solving this system

$$\begin{bmatrix} \mathbf{f}_2'(\mathbf{x}) & \mathbf{f}_4'(\mathbf{x}) & \mathbf{c}_2'(\mathbf{x}) \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \sigma \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} ,$$

where the solution is $\lambda_2 = 0.75$, $\lambda_4 = 0.25$ and $\sigma = 1.5$. Again $\sigma^* = \sigma + \varepsilon$ will trigger a shift of stationary point in the primal problem.

The above shows, that the estimation of the penalty factor σ can be obtained by solving a dual problem. We now have a theory for finding the exact value of σ that triggers a shift of stationary point, which is interesting in itself. In practice, this can be exploited by CMINMAX, in the part of the algorithm where we update σ.

A new update heuristic can now be made, where we use, e.g.,

$$\sigma_k = \xi \sigma_k^* , \tag{4.57}$$

where $\xi > 1$ is a multiplication factor. The above heuristic guarantees to trigger a shift of stationary point.

For non-linear problems, it is expected that using σ_k^* as the penalty factor, will trigger a shift of stationary point, but the iterate $\mathbf{x}_k$ found, could be close to $\mathbf{x}_{k-1}$ and hence the CMINMAX algorithm will use many iterations to converge to the solution. This is why we in (4.57) propose to use a σ where σ_k^* is multiplied with ξ. The above heuristic is, however, still connected to the problem being solved through σ_k^*.

Chapter 5

Trust Region Strategies

In this section we will take a closer look at the various update strategies for trust regions. Further we will also discuss the very important choice of parameters for some of the trust region strategies.

Practical tests performed on the SLP and CSLP algorithms suggest that that the choice of both the initial trust region radius and its update, have a significant influence on the performence. In fact this holds not only for SLP and CSLP, but is a common trademark for all trust-region methods [CGT00, p. 781].

In the following we will look closer at three update strategies, and finally we look at good initial values for the trust region radius. The discussion will have its main focus of the ℓ_∞ norm.

All basic trust region update strategies implement a scheme that on the basis of some measure of performance, decides whether to increase, decrease or keep the trust region radius. The measure of performance is usually the *gain* factor that was defined in (3.12). The gain factor is a measure of how well our model of the problem corresponds to the real problem, and on basis of this we can write a general update scheme for trust regions.

$$\eta_{new} \in \begin{cases} [\eta, \infty) & \text{if } \rho \geq \xi_2 \\ [\gamma_2\eta, \eta] & \text{if } \rho \in [\xi_1, \xi_2) \\ [\gamma_1\eta, \gamma_2\eta] & \text{if } \rho < \xi_1 \end{cases} \tag{5.1}$$

where $0 < \gamma_1 < \gamma_2 \leq 1$. In the above framework, we have that $\rho \geq \xi_1$ indicate a successful iteration

$$\mathcal{S} = \{\rho \geq \xi_1\}\,, \tag{5.2}$$

and $\rho \geq \xi_2$ indicate a very successful iteration

$$\mathcal{V} = \{\rho \geq \xi_2\}\,, \tag{5.3}$$

We see that it always hold that $\mathcal{V} \subseteq \mathcal{S}$. An iteration that is neither in $\mathcal{V}$ or $\mathcal{S}$ is said to be unsuccessful, and will trigger a reduction of the trust region radius. If the iteration is in $\mathcal{S}$ we have a successful iteration and the trust region radius is left unchanged if $\gamma_2 = 1$ or reduced. Finally if the iteration is very successful the trust region radius is increased.

The above description gives rise to the following classical update strategy, with $\gamma_1 = \gamma_2 = 0.5$, $\xi_1 = 0.25$ and $\xi_2 = 0.75$. If the iteration is very successful the trust region radius is increase by 2.5.

$$\begin{aligned} &\texttt{if}(\rho > 0.75) \\ &\quad \eta = 2.5 * \eta; \\ &\texttt{elseif } \rho < 0.25 \\ &\quad \eta = 0.5\eta; \end{aligned} \tag{5.4}$$

The above factors were used in the SLP and CSLP algorithms and were proposed in [JM94] on the basis of *practical* experience. An illustration of the update strategy is shown in figure 5.1. The figure clearly shows jumps in η_{new}/η when it passes over 0.25 and 0.75, hence the expression discontinuous trust region update.

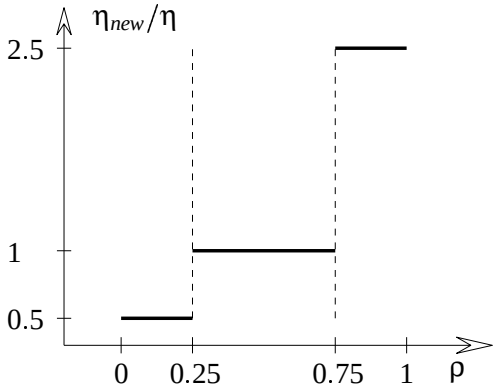

Figure 5.1: The classical trust region update in (5.4) gives rise to jumps in η_{new}/η across 0.25 and 0.75.

The factors are most commonly determined from practical experience, because the determination of these values have no obvious theoretical answer. In [CGT00, chapter 17] they give this comment on the determination of factor values: "As far as we are aware, the only approach to this question is experimental".

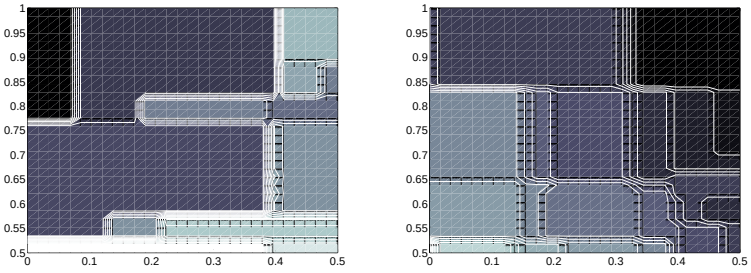

Figure 5.2: The number of iterations as function of ξ_1 and ξ_2, low values are indicated by the darker areas. (left) Rosenbrock, (right) ztran2f.

To clarify that the optimal choice of ξ_1 and ξ_2 is very problem dependent, we give an example for the Rosenbrock and ztran2f test functions. The number of iterations is calculated as a function of ξ_1 and ξ_2 and $\mathbf{x}_0 = [-1.2, 1]^T$ was used for Rosenbrock and $\mathbf{x}_0 = [120, 80]^T$ was used for ztran2f. The result is shown in figure 5.2.

It is seen from the figure that the optimal choice is $\xi_1 \in [0, 0.1]$ and $\xi_2 \in [0.8, 1]$ for the Rosenbrock function. For the ztran2f problem we get the different result, that $\xi_1 \in [0.3, 0.5]$

and $\xi_2 \in [0.85, 1]$. Further, the above result is also dependent on the starting point and choice of γ_1 and γ_2. This illustrate how problem dependent the optimal choice of parameters is, and hence we resort to *experience*.

An experiment has been done where $\gamma_1 = \gamma_2 = 0.5$ and the multiplication factor for a very successful step was 2.5. The aim with the experiment was to see whether or not $\xi_1 = 0.25$ and $\xi_2 = 0.75$ was optimal when looking at the whole test set. The result is shown in figure 5.3.

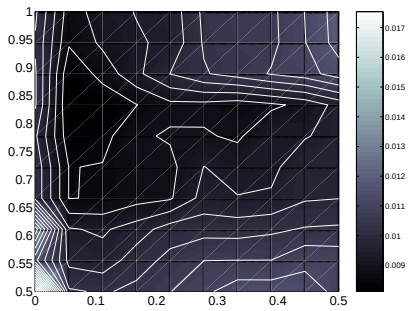

Figure 5.3: The optimal choice of ξ_1 and ξ_2 with $\gamma_1 = \gamma_2 = 0.5$ and the multiplication factor 2.5 for a very successful step fixed. Calculated on basis of a set consisting of twelve test problems.

The figure shows that $\xi_1 = 0.25$ and $\xi_2 = 0.75$ is not a bad choice. However, the figure seems to indicate that the choice is not the most optimal either. It is important to realize that the plot was produced only from twelve test problems, and that [JM94] claimed that the above parameters was optimal based on the cumulated experience of the writers. Therefore we can not conclude, that the above choice of parameters is bad. The figure was made by taking the mean value of the normalized iterations. Further the figure shows a somewhat flat landscape, which can be interpreted as the test problems having different optimal values of ξ_1 and ξ_2.

5.1 The Continuous Update Strategy

We now move on to looking at different update strategies for the trust region radius. We have already in the previous seen an example of a discontinuous update strategy. We will now look at an continuous update strategy, that was proposed for ℓ_2 norm problems in connection with the famous Marquardt algorithm.

An update strategy that gives a continuous update of the damping parameter μ in Marquardt's algorithm has been proposed in [Nie99a]. The conclusion is that the continuous update has a positive influence on the convergence rate. The positive result is the motivation for also trying this update strategy in a minimax framework.

Without going in to details, a Marquardt step can be defined as

$$(\mathbf{f}''(\mathbf{x}) + \mu\mathbf{I})\mathbf{h}_m = -\mathbf{f}'(\mathbf{x}) \tag{5.5}$$

We see that μ is both a damping parameter and trust region parameter. If the Marquardt step fails μ is increased with the effect that $\mathbf{f}''(\mathbf{x}) + \mu\mathbf{I}$ becomes increasingly diagonal dominant. This has the effect that the steps are turned towards the steepest descent direction.

When μ is increased by the Marquardt update, the length of $\mathbf{h}_m$ is reduced as shown in [MNT99, 3.20] that

$$\mathbf{h}_M = \min_{\|\mathbf{h}\| \le \|\mathbf{h}_M\|} \{L(\mathbf{h})\} ,$$

where $L(\mathbf{h})$ is a linear model. Further [FJNT99, p. 57] shows that for large μ we have $\mathbf{h}_m \simeq \frac{-1}{\mu}\mathbf{f}'(\mathbf{x})$. Therefore μ is also a trust region parameter.

Lets look at the update formulation for Marquardts method as suggested in [Nie99a] and described in [FJNT99].

$$\begin{array}{l} \texttt{if } \rho > 0 \texttt{ then} \\ \quad \mathbf{x} := \mathbf{x} + \mathbf{h} \\ \quad \mu := \mu * \max\{1/\gamma, 1-(\beta-1)(2\rho-1)^p\};\ \nu := \beta \\ \texttt{else } \mu := \mu * \nu:\ \nu := 2 * \nu \end{array} \tag{5.6}$$

where p has to be an odd integer, and β and γ is positive constants.

The Marquardt update, however, has to be changed so that it resembles the discontinuous update strategy that we use in the SLP and CSLP algorithms.

$$\begin{array}{l} \texttt{if} \rho > 0 \texttt{ then} \\ \quad \eta = \eta * \min\left\{ \max\{\frac{1}{\gamma}, 1+(\beta-1)(2\rho-1)^p\}, \beta \right\};\ \nu = 2; \\ \texttt{else } \eta = \eta/\nu;\ \nu = 2 * \nu; \end{array} \tag{5.7}$$

We see that this change, does not affect the essence in the continuous update. That is, we still use a p degree polynomial to fit the three levels of η_{new}/η shown in figure 5.1. An illustration of the continuous update strategy is shown in figure 5.4.

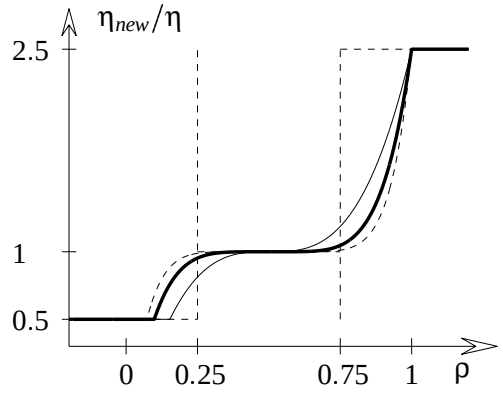

Figure 5.4: The continuous trust region update (solid line) eliminates the jumps in η_{new}/η across 0.25 and 0.75. Values used, $\gamma = 2$, $\beta = 2.5$ and $p = 5$. The thin line shows $p = 3$, and the dashed line shows p_7.

The interpretation of the variables in (5.7) is straight forward. $1/\gamma$ controls the value of the lowest level value of η_{new}/η, and β controls the highest value. The intermediate values are defined by a p degree polynomial , where p has to be an odd number. The parameter p controls how well the polynomial fits the discontinuous interval $[0.25, 0.75]$, where $\eta_{new}/\eta = 1$ in the discontinuous update strategy.

A visual comparison between the discontinuous and the continuous update strategy is seen in figure 5.5. The left and right plots show the results for the Enzyme and Rosenbrock

test functions. The continuous update strategy show superior results iteration wise, on the Enzyme problem, but shows a somewhat poor result when compared to the discontinuous update for the Rosenbrock problem. The results are given in table 5.1.

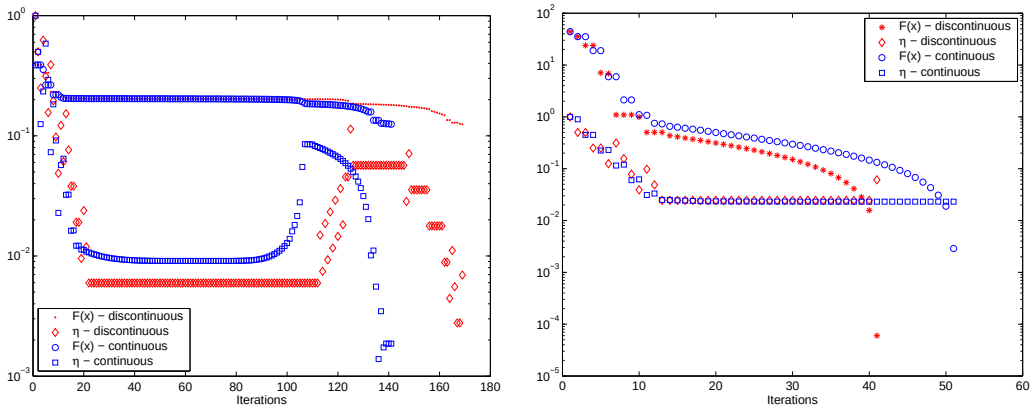

Figure 5.5: The continuous vs. the discontinuous update, tested on two problems. Left: The Enzyme problem. Right: The Rosenbrock problem.

From the figure we see that the continuous update give a much smoother update of the trust region radius η for both problems. The jumps that is characteristic for the discontinuous update is almost eliminated by the continuous strategy. However, the performance of the continuous update does not always lead to fewer iterations.

Iterations	(5.4)	(5.7)	(3.13)	(5.10)
Parabola	31	33	31	23
Rosenbrock1	16	20	16	24
Rosenbrock2	41	51	41	186
BrownDen	42	47	42	38
Ztran2f	30	30	30	20
Enzyme	169	141	169	242
El Attar	10	8	10	10
Hettich	30	28	30	19

Table 5.1: Number of iterations, when using different trust region update strategies, indicated by there. The update strategies are indicated by there equation number.

The continuous vs. the discontinuous update have been tried on the whole test set, and there was found no significant difference in iterations when using either. This seems to suggest that they are equally good, still, however, one could make the argument that the discontinuous update gives a more intuitively interpretable update.

5.2 The Influence of the Steplength

In the latter general formulation (5.1) of the trust region we saw that the step length $\mathbf{s}_k$ had no influence on the radius of the trust region. This is not reasonable, when realizing that

the gain factor ρ_k is calculated upon a sample $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$ and is the only parameter that determines an increase or decrease of the radius.

One could make the argument that an update strategy that includes $\mathbf{s}_k$ will be more correct on a theoretical level. If the step does not go to the edge of the trust region then ρ_k is not a measure of trust for the whole region, but just for a subregion that is defined by $\eta_{sub} = \|\mathbf{s}_k\|$. Figure 5.6 tries to illustrate why it is reasonable to include $\mathbf{s}_k$ in an update strategy. The subregion is illustrated by the dashed border.

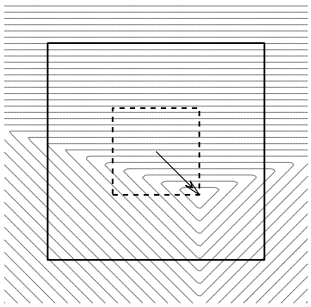

Figure 5.6: When the step $\mathbf{s}_k$ does not go to the border of the trust region, then ρ_k is only a measure of trust for a subregion, illustrated by the dashed lines.

All manipulation of the trust region should then be performed by using the subregion. Here is a more formal definition [CGT00, p. 782]

$$\eta_{new} \in \begin{cases} \max\{\alpha_1\|\mathbf{s}_k\|_\infty, \eta\} & \text{if } \rho \geq \xi_2 \\ \eta & \text{if } \rho \in [\xi_1, \xi_2) \\ \alpha_2\|\mathbf{s}_k\|_\infty & \text{if } \rho < \xi_1 \end{cases}, \tag{5.8}$$

where $0 < \alpha_2 < 1 < \alpha_1$. We saw in the previous numerical experiment regarding SLP and CSLP, that when the algorithms approached a strongly unique local minimum, then the trust region radius grew. An illustration of this effect is shown in figure 3.4. It is exactly effects of this kind that (5.8) aims to eliminate, because there is no theoretical motivation to increase the trust region when $\|\mathbf{s}_k\| \to 0$.

The factors α_1 and α_2 should as above, be determined from practical experience. In [CGT00, (17.1.2)] they propose to use the values

$$\alpha_1 = 2.5 \text{ and } \alpha_2 = 0.25 \tag{5.9}$$

They do not say on which type of functions this experience was gained, so it could be any kind of problems. A test was done to see if the above factors is suitable for our minimax test problems. We have examined the average amount of normalized iterations as a function of α_1 and α_2 in an area of ± 0.2 around the point suggested in (5.9). The result is shown in figure 5.7.

The result show that the values of $\alpha_1 = 2.5$ and $\alpha_2 = 0.25$ is not a bad choice, however, the figure seems to suggest that a small decrease in iterations could be made by choosing other values. However, this reduction does not seem to be significant.

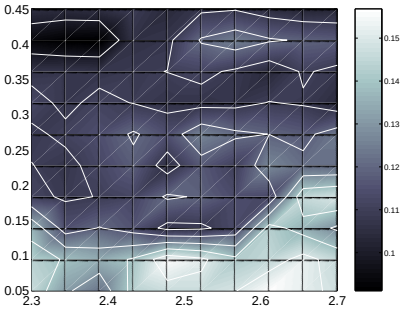

Figure 5.7: The average of the normalized iterations as a function of α_1 and α_2. Twelve test problems was used in the evaluation.

Using the above factors yields the following update strategy

$$\eta_{new} \in \begin{cases} \max\{2.5\|\mathbf{s}_k\|_\infty, \eta\} & \text{if } \rho \geq 0.75 \\ \eta & \text{if } \rho \in [0.25, 0.75) \\ 0.25\|\mathbf{s}_k\|_\infty & \text{if } \rho < 0.25 \end{cases}, \tag{5.10}$$

The strategy has also been used on the test problems and the results are given in table 5.1. We see that the performance is better when looking at the number of iterations compared against the other update strategies. The only exceptions are the Rosenbrock and the Enzyme test problems, where the performance is significantly more poor than for the other update strategies.

Finally we give an example of the step update strategy in (5.8). We can expect that such a strategy will be more dynamic and therefore be able to reduce its trust region more quickly than the discontinuous update, because every update involves the step length $\|\mathbf{s}_k\|$. Figure 5.8 shows the step update strategy vs. the discontinuous strategy on the ztran2f problem.

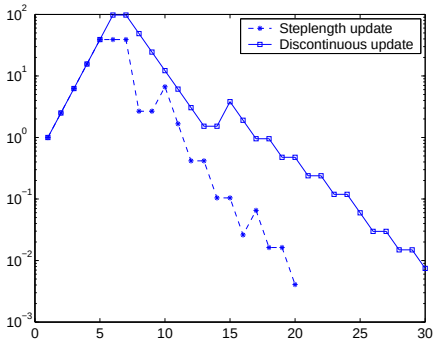

Figure 5.8: The discontinuous update strategy vs. the step update strategy in (5.8) tested on the ztran2f problem.

The figure shows that the step update is able to reduce the trust region radius faster than the discontinuous update. This gives nice results for the ztran2f problem, but for a problem like Rosenbrock this property of the step update gives rise to a lot more iterations. See table 5.1.

5.3 The Scaling Problem

The scaling problem arise when the variables of a problem is not of the same order of magnitude, e.g. a problem of designing an electrical circuit, where one variable is measured

in *Farad* and the other in *Ohm*. The value in *Farad* would be of magnitude 10^{-12} and *Ohm* would be of magnitude 10^6. This is a huge difference, that not alone is an indication of bad scaling, but would also introduce severe numerical problems.

One such problem is loss of information [CGT00, p. 162]. We have that $\mathbf{c}_k$ is measured in *Farad* and $\mathbf{r}_k$ is measured in *Ohm*. Now we wish to calculate the norm of a step $h_k = r_k + c_k$

$$\|\mathbf{h}_k\|^2 = \|\mathbf{r}_k + \mathbf{c}_k\|^2 = \|\mathbf{r}_k\|^2 + \|\mathbf{c}_k\|^2$$

We see that $\|\mathbf{r}_k\| \approx 10^{12}$ and $\|\mathbf{c}_k\| \approx 10^{-24}$, and if the computer is limited to 15 digit precision then $\|\mathbf{h}_k\|^2 = \|\mathbf{r}_k\|^2$. In this way the information from $\mathbf{c}_k$ is lost!

In the implementation of the SLP and CSLP algorithms there is an underlying assumption that the problems are well scaled. This is not an unreasonable assumption, because it is expected that qualified personal is able to do such rescaling of their problems fairly easy. However we will in the following give a short outline on how such a rescaling could be made internally in an algorithm.

All numerical calculations inside an algorithm should be done in a scaled space, so a *scaled* variable $\mathbf{w}$ is needed

$$\mathbf{S}_k \mathbf{w} = \mathbf{h} \ . \tag{5.11}$$

The scaling matrix $\mathbf{S}_k \in \mathbb{R}^{n \times n}$ is a diagonal matrix when only a scaling parallel to the axis are needed, however it could also be something different from than a diagonal matrix. In this case the scaling would also be a rotation of the scale space. In the non diagonal case, $\mathbf{S}_k$ could be determined from a eigenvalue decomposition of the Hessian $\mathbf{H}$.

From the previous example, a user could supply the algorithm with the scaling matrix $\mathbf{S}$, as a pre process step.

$$\mathbf{S} = \begin{bmatrix} 10^6 & 0 \\ 0 & 10^{-12} \end{bmatrix} \tag{5.12}$$

Here the subscript k has been omitted, because the scaling matrix is only given in the pre process stage and never changed again. A scaling matrix could, however, depend on the current iterate if the problem is non linear and have *local* bad scaling.

We have a model $M_k(\mathbf{x}_k + \mathbf{h})$ of the problem we want to solve. Because of bad scaling we re-scale the problem by transforming the problem into a scaled space denoted by superscript s

$$M_k^s(\mathbf{x}_k + \mathbf{w}) \approx f(\mathbf{x}_k + \mathbf{S}_k \mathbf{w}) \equiv f^s(\mathbf{w}) \ . \tag{5.13}$$

We can then write a linear model of the problem

$$M_k^s(\mathbf{x}_k) = f(\mathbf{x}_k), \quad \mathbf{g}_k^s = \nabla_w M_k^s(\mathbf{x}_k) = \nabla_w f^s(0) = \nabla_x f(\mathbf{x}_k)\mathbf{S}_k \ , \tag{5.14}$$

where

$$\begin{aligned} M_k^s(\mathbf{x}_k + \mathbf{w}) &= M_k^s + \mathbf{g}_k^s \mathbf{w} \\ &= f(\mathbf{x}_k) + \nabla_x f(\mathbf{x}_k)\mathbf{S}_k \mathbf{w} \\ &= f(\mathbf{x}_k) + \nabla_x f(\mathbf{x}_k)\mathbf{h} \\ &= M_k(\mathbf{x}_k + \mathbf{h}) \end{aligned} \tag{5.15}$$

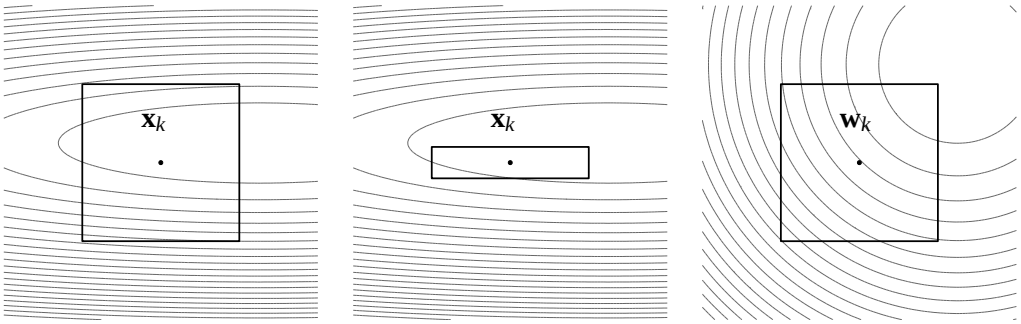

Figure 5.9: Left: The unscaled space and the unscaled trust region. Middle: The unscaled space and the scaled trust region. Right: The scaled space and the scaled trust region.

The trust region is affected by the transformation to the scaled space and back to the original space. In the scaled space the trust region subproblem is defined as

$$\min_{\|\mathbf{w}\|_\infty \leq \eta} M_k^s(\mathbf{x}_k + \mathbf{w}) \ , \tag{5.16}$$

and in the original space defined as

$$\min_{\|\mathbf{S}^{-1}\mathbf{h}\|_\infty \leq \eta} M_k(\mathbf{x}_k + \mathbf{h}) \ . \tag{5.17}$$

We see that the trust region in original space is no longer a square but a rectangle, however, it is still a square in the scaled space. This is shown in figure 5.9. If $\mathbf{S}_k$ is a symmetric positive definite matrix with values off the diagonal, then we will get a scaling *and* rotation of the scaled space.

Chapter 6

Linprog

Matlabs optimization toolbox offers a wide range of tools to solve a variety of problems. The tools span form general to large scale optimization of nonlinear functions, and further it also contains tools for quadratic and linear programming.

At first, the source code in the toolbox can seem overwhelming, but is in fact both well structured and well commented. Still, however, the algorithms may at first sight seem more logical complex, than they ought to be. For instance, many algorithms have branching (`if`, `switch` etc.) that depends on the calling algorithm. A straight forward explanation is, that in this way code redundancy i.e. the amount of repeated code, is reduced to a minimum. This is good seen from a maintenance perspective.

The toolbox offers many exciting possibilities and have many basic parts that one can use to build new algorithms. Just to list a few of the possibilities this toolbox offers.

- Linear and quadratic programming.
- Nonlinear optimization e.g. 2-norm and minimax, with constraints
- A flexible API that ensures that all kinds of parameters can be passed directly to the test problems.

A good overview and documentation of the toolbox is given in [Mat00] that covers the basic theory, supplemented with small tutorials. This chapter will focus on the program `linprog ver.1.22` that is the linear solver in Matlab's optimization toolbox. The version of the optimization toolbox investigated here is
`Version 2.1.1 (R12.1) 18-May-2001`

Currently a newer version 2.2 of the toolbox is available at MathWorks homepage[1].

This chapter is build like this: bla bla.

6.1 How to use linprog

Linprog is a program that solves a linear programming problem. A linear programming problem seeks to minimize or maximize a linear cost function, that is bounded by linear

[1] http://www.mathworks.com/products/optimization/

constraints. This means that the feasible region of an LP problem is convex, and since the cost function is linear, the solution of the LP problem must exist at a extreme point or a vertex of the feasible region.

The constraints in linear programming can be either equality or inequality constraints, and hence `linprog` can solve the following type of problem.

$$\min_{\mathbf{x}} F(\mathbf{x}) , \quad \mathbf{x} \in \mathbb{R}^n$$

subject to

$$\begin{aligned} c_i(\mathbf{x}) &= 0 , \quad i = 1, \ldots, m_e \\ c_i(\mathbf{x}) &\leq 0 , \quad i = m_e, \ldots, m \\ \mathbf{a} &\leq \mathbf{x} \leq \mathbf{b} , \end{aligned}$$

where **a** and **b** are lower and upper bounds. The above problem is non-linear so in order to get a linear programming problem, all the constraints should be linearized by a first order Taylor approximation, so that we can formulate the above problem as

$$\min_x \mathbf{f}^T\mathbf{x} \text{ subject to } \begin{aligned} \mathbf{A}\mathbf{x} &\leq \mathbf{b} \\ \mathbf{A}_{eq}\mathbf{x} &= \mathbf{b}_{eq} \\ lb &\leq \mathbf{x} \leq ub \end{aligned}$$

where $\mathbf{f} = \nabla F(\mathbf{x})$ is a vector, and the gradients of the constraints $\mathbf{A}_{eq}$ and $\mathbf{A}$ are matrices. The above problem can be solved by `linprog`, by using the following syntax

```
x = linprog(f,A,b,Aeq,beq,lb,ub,x0,options)
[x,fval,exitflag,output,lambda] = linprog(...)
```

6.2 Why Hot Start Takes at Least *n* Iterations

Simplex algorithms have a feature that the Interior Point method does not have. It can take a starting point supplied by the user, i.e. a so-called *warm start*. The use of warm start, has the great advantage that if the starting point is close to the solution then the Simplex algorithm will not have to traverse so many vertices in order to find the solution, and hence terminate quickly.

A warm start is also a good idea for nonlinear problems, where a linear subproblem is solved in each iteration. When we are near the solution to the nonlinear problem, we can expect that the active set remains the same, and hence there is a good change, that the solution to the LP problem in the previous iteration could provide the Simplex algorithm with a good warm start for the next iteration.

Tests of `linprog` for medium scaled problems, has shown that even if we supply it with a very good starting point, it will use at least *n* iterations. In this section we investigate why this is so. In the latter we assume that the problems are not degenerate.

For problems that are medium scale, `linprog` uses an active set method, implemented as `qpsub` in the optimization toolbox, to solve an optimization problem on the form

$$\min_x \mathbf{f}^T\mathbf{x} , \quad \mathbf{x} \in \mathbb{R}^n$$

subject to

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$
$$\mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq}$$
$$lb \leq \mathbf{x} \leq ub$$

When solving medium scaled problems, `linprog` offers the possibility of using a supplied $\mathbf{x}$ as a way to warm start the algorithm, i.e. giving it a good start point that may be close to the solution $\mathbf{x}^*$. In this way, one can reduce the number of iterations that `linprog` uses to find the solution.

The title of this section includes the words *hot start*, and by this we mean, that not only is the supplied $\mathbf{x}$ close to the solution, it is in fact *the* solution. One could then expect `linprog` to terminate after only one iteration, however this is *not* what happens.

The Fourier series problem, illustrated in the introduction of this report, revealed an interesting aspect of the hot start problem. Even if `linprog` was used with hot start, it would still use n iterations to solve the problem, this is illustrated in Figure 6.1.

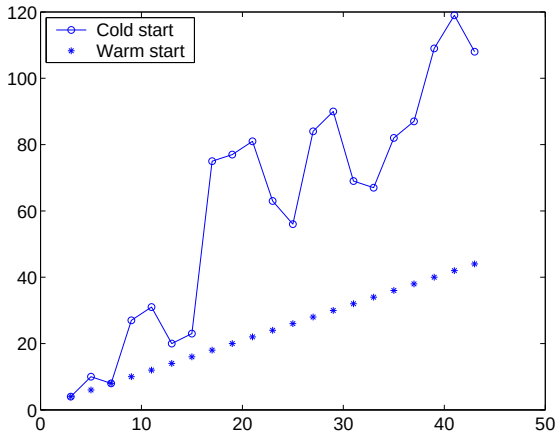

Figure 6.1: The Fourier series problem solved with linprog using a guess as cold start, and the solution as warm start. The abscissae show the dimension of the problem, and the ordinate show the number of iterations. It is seen that linprog uses at least n iterations when supplied with a hot start.

As mentioned, `linprog` uses an active set method for medium scale problems, but it is not `linprog` that solves the problem. Instead `linprog` calls `qpsub` to do the actual calculations, hence the rest of this investigation will have its main focus on `qpsub`.

In the following we investigate what happens in `qpsub` when we supply `linprog` with a hot start.

In `linprog` at line 171 the call to `qpsub` is made. As part of the initialization of `qbsub` the variable `simplex_iter` is set to zero. This will become important later on, because `simplex_iter` equal to one is the only way to get out of the main while loop by using the return call at line 627.

```
318  while iterations < maxiter
⋮
592  if ACTCNT == numberOfVariables - 1, simplex_iter = 1; end
⋮
744  end
```

We can not use `maxiter` to break the while loop, because it is set to `inf`. If the caller of `qpsub` had been different, e.g. a large scale algorithm, then `maxiter` would have been set to a finite value.

In the initilization phase, the system $\mathbf{Ax} \leq \mathbf{b}$ is scaled and the active set `ACTCNT` is initialized to the empty set.

The constraint violations `cstr`$= \mathbf{Ax} - \mathbf{b}$ is found as part of finding an initial feasible solution at line 232–266. If we assume that a feasible solution do exist, then `cstr` must contain elements that are less or equal to zero, due to the definition of the constraints. Further, it must be the case, that when `qpsub` is supplied with a hot start, then at least n elements in `cstr` is zero.

The `cstr` vector is recalculated at line 549 in the main loop each time a new `X` is found. As we shall see later, a hot start has the affect that `X` does not move, which is natural because `X` *is* the solution. The vector `cstr` is used for calculating the distance `dist`, that is calculated at each iteration in the main loop at line 341

```
dist = abs(cstr(indf)./GSD(indf));
```

where `GSD` is the gradient with respect to the search direction. The vector `indf` contains all the indexes in `GSD` that are larger than some small value. In this way, a singular working set is avoided. If this was not done, it would be possible to have constraints parallel to the search direction. See the comments in the code at line 330–334.

The `dist` vector is then used to find the minimum step `STEPMIN`, and because of the hot start, this distance is zero. The algorithm now select the nearest constraint

```
[STEPMIN,ind2] =  min(dist);
ind2 = find(dist == STEPMIN);
```

So `ind2` holds the indexes to the nearest constraints. One may notice that the way `ind2` is found is not ideal when thinking of numerical errors. It is, however, not crucial to find all distances to the constraints that are equal to `STEPMIN`. This is seen at line 346 where the constraint with the smallest index is selected, and the rest is discarded.

```
ind=indf(min(ind2));
```

This is a normal procedure also known as Bland's rule for anti-cycling. This ensures that we leave a degenerate vertex after a finite number of iterations. In this way the Simplex algorithm avoids being stuck in an infinite loop, see [Nie99b, Example 4.4].

A step in the steepest decent direction is made at line 365, `X = X+STEPMIN*SD;`. Because of the hot start `STEPMIM` is zero, which have the result that the new `X` is equal to old `X`.

In each iteration, at line 587–595, the variable `ACTCNT` that hold the number of active constraints is increased with one. Because of Bland's rule, only one constraint becomes active in each iterations. So at the $n-1$ iteration, `ACTCNT == n-1`, where n in `qpsub` is stored in `numberOfVariables`.

If the following condition11 is satisfied, then the `simplex_iter` flag is set to one.

```
if ACTCNT == numberOfVariables - 1, simplex_iter = 1; end
```

then the `simplex_iter` flag is set to one. When using hot start this will happen at the $n-1$ iteration.

When `simplex_iter == 1`, the algorithm enters the simplex iterations at line 595–629, where the solution is found by traversing the edges of the problem. Because of the hot start, there is no edges to traverse, that reduces the cost function and hence all the all the Lagrangian multipliers are non negative, which indicate that the solution has been found, see line 640–641. The algorithm will then terminate after doing $n-1$ active set iterations and at least 1 simplex iteration.

The execution then return to `linprog` where the output information from `qpsub` is given the right format.

Figure 6.2 shows the results from a runtime experiment done on linprog, by using the linear Fourier problem described in appendix A.

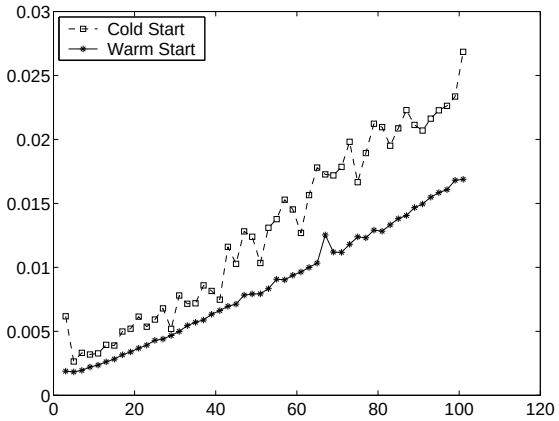

Figure 6.2: Timing results from linprog when solving the linear Fourier problem. The abscissae shows the number of variables in the problem, and the ordinate shows the average time per iteration.

The results clearly show, that when using hot start, the average time per iteration is lower than that for cold start. When using a hot start, `qpsub` does $n-1$ active set iterations and at least one simplex iteration. Hence the results indicate that the active set iterations are cheaper than the simplex iterations.

6.3 Issues regarding linprog.

The strategy, of using Lagranian multipliers to define the active set needed for the corrective step, can fail when using `linprog ver. 1.23`. This happens when the active functions have contours parallel to the coordinate system. This can give rise to an infinite number of solutions, which is illustrated in the following.

We show three examples from `linprog`, where the trust region radius is varied with $\eta = \{0.2, 0.3, 0.5\}$, and in each case the solution is analysed. Figure 6.3 shows the three situations.

The figure shows the LP minimax landscape for Rosenbrock's function evaluated in $\mathbf{x} = [-1.2, 1]^T$, where $f_1'(\mathbf{x}) = [-20x_1\ 10]^T$ and $f_2'(\mathbf{x}) = [-1, 0]^T$.

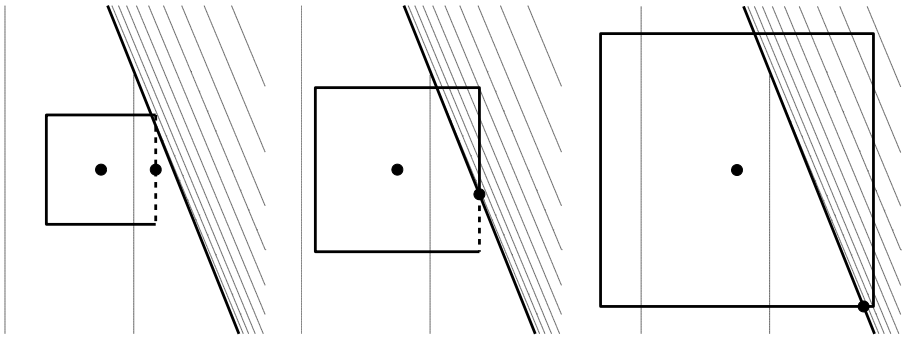

Figure 6.3: Plot of the minimax LP landscape of the Rosenbrock function in $\mathbf{x} = [-1.2, 1]^T$. Left: $\eta = 0.2$. Middle: $\eta = 0.3$. Right: $\eta = 0.5$. The dashed lines indicate an infinity of solutions.

For $\eta = 0.2$ the solution $\mathbf{h} = [0.2, 0.0]^T$ is not unique. Both $\ell_1(\mathbf{x})$ and the upper bound on x_1 is active according to the λ multipliers from `linprog`. The dashed line in Figure 6.3 indicates an infinity of solutions.

For $\eta = 0.3$ the sollution $\mathbf{h} = [0.30, -0.09]^T$ is not unique either. Both $\ell_1(\mathbf{x})$ and the upper bound of x_1 are active. The dashed line in Figure 6.3 middle, says that there is an infinity of solutions.

For $\eta = 0.5$ the sollution $\mathbf{h} = [0.464, -0.500]^T$ is unique and both functions $\ell_1(\mathbf{x})$, $\ell_2(\mathbf{x})$ and the lower bound of x_2 are active, according to the `linprog` multipliers.

The example shows that `linprog ver. 1.23` not always deliver the multipliers one would expect. For instance, it could be expected for a situation as the one illustrated in Figure 6.3 middle, that both the multipliers for $\ell_1(\mathbf{x})$ and $\ell_2(\mathbf{x})$ are larger than zero. In this way the multipliers will indicate that both functions are active at $\mathbf{x}$. The reason for this prior expectation is found in the definition of an active function

$$\mathcal{A} = \{ j \mid \ell_j(\mathbf{x}) = \alpha \} \ ,$$

where both $\ell_1(\mathbf{x})$ and $\ell_2(\mathbf{x})$ would be active. It is then strange that the multipliers do not indicate the same – or is it?

Turning to the theory of generalized gradients, we see that this is not so strange at all, in fact the above behavior is in good alignment with the theory. But first we look at how `qpsub` deals with bounds on the variables.

From line 74–82 in `qpsub` we see that the bounds are added to the problem as linear constraints. This means that we extend the problem we want to solve, and in turn our solution will be confined to the bounded area.

The added linear constraints from the bounds, have to be taken into consideration, when evaluating a stationary point. It is then seen that in all three cases the stationary point condition is fulfilled, because the null vector is interior to the convex hull of the generalized gradient, as illustrated in Figure 6.4.

The figure shows that for the first case $\eta = 0.2$ we have that the null vector is part of the convex hull. So the solution found is of course a stationary point, but not a strongly unique

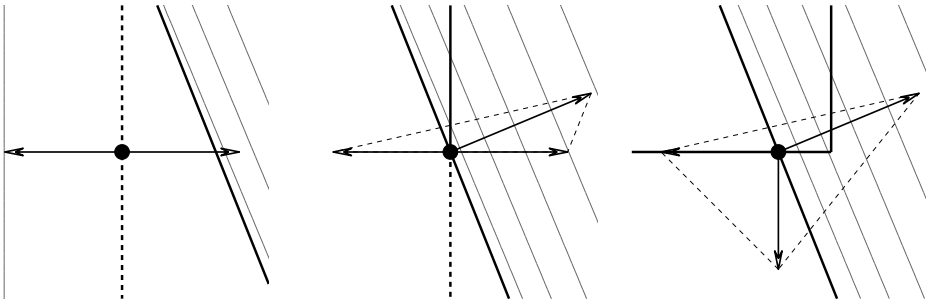

Figure 6.4: Left: $\eta = 0.2$, the null vector is part of the convex hull. Middle: $\eta = 0.3$, The gradient from $\ell_2(\mathbf{x})$ shown as the arrow that is not horizontal, is only weakly active i.e. its multiplier is zero. Right: $\eta = 0.5$. This is a strongly unique local minimum, and all multipliers are greater than zero.

local minimum, because the null vector is not interior to the hull. This also explains why we have an infinity of solutions along the dashed line in Figure 6.3 left.

The point found in the second case $\eta = 0.3$ is also a stationary point. As seen in Figure 6.4 middle, the gradient corresponding to $\ell_2(\mathbf{x})$, shown as the arrow that is not horizontal, is only weakly active. This means that if we removed it, the point found would still be stationary. This is why the multiplier for $\ell_2(\mathbf{x})$ is zero in this case. Basically the situation is then the same as for the previous case, and we again have an infinity of stationary points along the dashed line shown in Figure 6.3 middle.

In the last case $\eta = 0.5$ the solution found is not just a stationary point but a strongly unique local minimum. As seen in Figure 6.4 right, the null vector is interior to the convex hull of the generalized gradient, and hence the multipliers for $\ell_1(\mathbf{x})$, $\ell_2(\mathbf{x})$ and the lower bound on x_2 are positive.

This behavior described above, was first noticed during the work with the corrective step. For a problem like Rosenbrock, the case corresponding to the one shown in Figure 6.3 middle, has a damaging affect if the multipliers are used to indicate the active functions. In this case it would seem like $\ell_2(\mathbf{x})$ was not active, and the corrective step would not be calculated correctly. Even worse, in this case there would only be one active function, and hence the corrective step would be the null vector. This seriously hamper the performance of the CSLP algorithm. It was on this basis that it was decided to use

$$\mathcal{A} = \{ j \mid |\ell_i(\mathbf{x}) - \alpha| \leq \varepsilon \} ,$$

to indicate whether or not, a function is active.

6.4 Large scale version of linprog

For medium scale problems `linprog` uses an active set method to find $n-1$ active constraints at the edge of the feasible area, and then use Simplex iterations to find the solution. It is, however, not feasible to use such a method for large problems. Hence, the optimization toolbox offers the possibility of using an interior point method. By using

```
opts = optimset('LargeScale','on') ,
```

and passing `opts` to an optimization algorithm, that algorithm will be set to use an interior point method. The interior point method is implemented in `lipsol` (large scale interior point solver) in the optimization toolbox.

The active set method (AS) as implemented in the optimization tool box by `qpsub`, is basically a Simplex algorithm, and hence we will in the following regard it as a Simplex-like algorithm.

The Simplex method is in the worst case an exponential time algorithm, whereas the Interior Point (IP) method is a polynomial time algorithm [HL95, chapter 4]. We are, however, often interested in the *average* case performance. Here both the Simplex, and the (IP) method has a polynomial runtime.

The Simplex method finds a solution that lies on a vertex of a convex set that defines the feasible area. It decreases the cost function by traversing the edges of the hull going from one vertex to the next. For a thorough introduction see [Nie99b, chapter 4], and [HL95, chapter 4,5,6].

The IP method follows a central path *trough* the feasible domain, and can therefore in some cases outperform the Simplex method that is confined to the edge of the feasible area. The IP method uses a logarithmic barrier function, to confine the iterations to the interior of the feasible domain, and use a Newton step to find the next iterate. The IP method is also described in [Nie99b, chapter 3] and somewhat sketchy in [HL95, chapter 7].

When comparing the Simplex and the IP method, one must look at the amount of work per iteration, and the number of iterations. The IP method is more complex than the Simplex method, and it requires more computer time per iteration. In turn, it does not take as many iterations as the Simplex method. In fact, large problems does not require many more iterations than small problems, as shown in Table 6.1.

m	n	Iter. (AS)	Iter. (IP)	sec. (AS)	sec. (IP)
130	34	67	15	0.6372	1.2350
⋮	⋮				
370	94	271	13	7.1073	18.2001
410	104	359	11	9.2024	20.6553
450	114	465	16	14.3509	38.4359
490	124	515	16	17.1492	49.6527

Table 6.1: The Fourier problem solved as ℓ_∞, where m is the number of constraints and n is the number of variables.

The table clearly shows that the number of iterations for the IP method is almost constant, regardless of the size of the problem. However, we also see that the runtime results indicate that the IP iterations is more computational costly than the AS iterations. This is not surprising, when looking at the theory behind these two methods.

Without going into any details, the main work of a simplex iteration involves solving two systems

$$\mathbf{B}^T\mathbf{g} = \mathbf{c}_{\mathcal{B}} \quad \text{and} \quad \mathbf{Bh} = \mathbf{C}_{:,s}$$

where $\mathbf{B} \in \mathbb{R}^{m \times m}$. For the purpose of our discussion it is not important to know the exact details, instead we refer to [Nie99b, Chapter 4]. The inner workings of the Simplex method requires, that we move from one vertex to another until the solution is found. This is done by moving one column in each iteration in and out of the basis $\mathbf{B}$. In this way the basis $\mathbf{B}^{k-1}$ differs from $\mathbf{B}^k$ by only one column. This can be exploited quite efficiently by updating the factorization of $\mathbf{B}^{k-1}$, and in `qpsub` this is done by the two Matlab functions `qrinsert` and `qrdelete`.

The Interior Point method has more work to do in each iteration. Again, it is not important for the discussion to go into any specific details, instead we refer to [Nie99b, Chapter 3]. The IP method uses a logarithmic barrier function to create a landscape of the feasible region. The minimizer of this landscape is a non linear KKT system, that is approximated by a linearization. The minimizer of the linearized KKT system is found by using a Newton step. The Newton step can be reduced to the following expression [Nie99b, (3.22)]

$$(\mathbf{A}\mathbf{D}^2\mathbf{A}^T)\mathbf{h}_y = \mathbf{f}_y + \mathbf{A}\mathbf{D}^2\mathbf{f}_x$$

where $(\mathbf{A}\mathbf{D}^2\mathbf{A}^T) \in \mathbb{R}^{m \times m}$, and we solve for $\mathbf{h}_y$. The above system can be solved by, e.g., using a factorization, but unlike the Simplex method, we can not update the factorization. Hence the computational work in each iteration is greater than that of the Simplex method, which is confirmed by the results in Table 6.1.

We have to note, that the timing results in the table should be looked upon, just as an indication of the amount of work the methods do, in order to solve the problem. Since the tests were made on a non dedicated system, the timings are not precise. Still, however, they provide a rough indication of the amount of work done.

In the following, we present some results of test performed on `linprog` using medium and large scale settings. All problems were solved as Chebyshev ℓ_∞ problems

$$\min_{\hat{x}} \mathbf{c}^T\hat{\mathbf{x}} \text{ subject to } \mathbf{A}\hat{\mathbf{x}} \leq \mathbf{b} ,$$

where $\mathbf{A} \in \mathbb{R}^{2m \times n+1}$, $\mathbf{b} \in \mathbb{R}^{2m}$ and $\hat{\mathbf{x}} \in \mathbb{R}^{n+1}$.

We have performed the tests by using two very different test problems. The first problem is the Fourier problem described in Appendix A, and is very dense, due to the nature of the problem.

The second test problem, is the problem of finding the solution to the Laplace equations numerically, by using a rectangular grid. The problem is described in detail in Appendix B, and is sparse.

We have set up six test cases, described in Table 6.2.

When `linprog` is set to large scale mode, it uses an algorithm called `lipsol` that is stored in Matlab's private directory. Similarly, when we use medium scale mode, `linprog` will use `qpsub`, an algorithm that is also stored in Matlab's private directory. Therefore the following tests of the AS and IP methods are in reality tests of `lipsol` and `qpsub`.

First we investigate how well the AS and IP methods handles sparse vs. dense problems. The results are shown in Figure 6.5 and 6.6.

	Setting	Problem	n	m
1:	Medium scale	Fourier	$n = 3,5,\ldots,301$	$m = n+4$
2:	Large scale	Fourier	$n = 3,5,\ldots,301$	$m = n+4$
3:	Medium scale	Fourier	$n = 3,5,\ldots,151$	$m = 2n-1$
4:	Medium scale	Laplace	$n = 3,5,\ldots,301$	$m = n$
5:	Large scale	Laplace	$n = 3,5,\ldots,301$	$m = n$
6:	Large Scale	Fourier	$n = 3,5,\ldots,151$	$m = 2n-1$

Table 6.2: The test cases. Because we are solving the above as an ℓ_∞ problem, we have that $\mathbf{A} \in \mathbb{R}^{2m\times n+1}$.

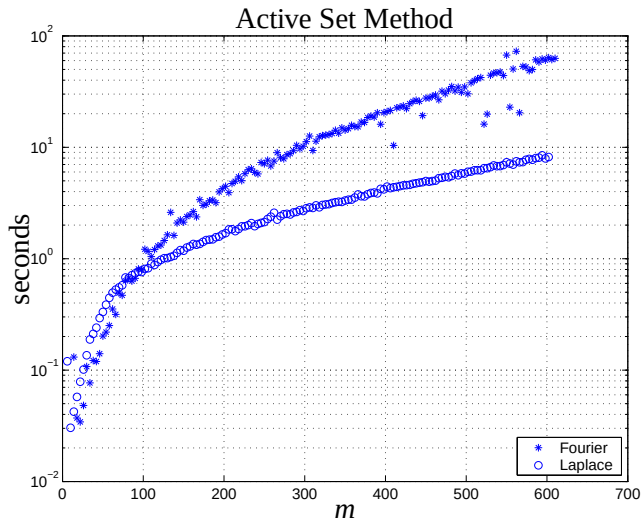

Figure 6.5: Test cases 1 and 4. Runtime results for the Active Set method when solving a dense '$*$' and a sparse '$\circ$' problem. All timings are an average of three observations.

Figure 6.5 shows runtime results for test case 1 and 4, i.e., using the AS method to solve the dense Fourier, and sparse Laplace problem. The results clearly show, that `qpsub` can take advantage of the sparsity in the Laplace problem. For problem sizes where $0 < m < 100$, we see that the AS method uses an equal amount of time to solve the dense and sparse problem. For $m > 100$ the sparse Laplace is solved much faster than the dense Fourier, which indicate that the sparsity is exploited by `qpsub`.

Figure 6.6 shows the result for test cases 2 and 5, i.e., using Matlabs implementation of the IP method `lipsol`, on the dense Fourier, and the sparse Laplace problem. We see that `lipsol` exploits the sparsity of the Laplace problem quite well. For all m, we have that `lipsol` uses more time to solve the dense than the sparse case.

For $m = 500$ something interesting happens. The decrease in time to solve both the dense and sparse problem drops. Especially is the reduction for the sparse problem rather dramatic. The only way to investigate this is to look inside the code for `lipsol`.

At line `643` a check is performed and if $m > 500$, a parameter `nzratio` is set to 0.2, otherwise it is set to 1. If `nzratio < 1` at line `647`, then `lipsol` checks to see if $\mathbf{A}$ contains any dense columns. If it does, then the flag `Dense_cols_exist = 1`. For ℓ_∞ problems, the last

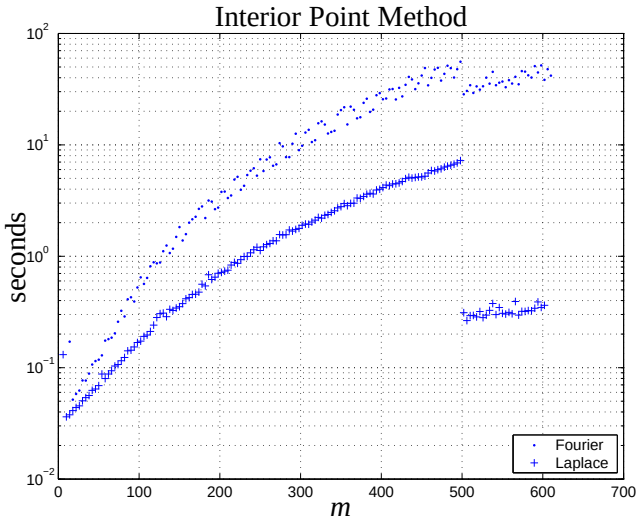

Figure 6.6: Test cases 2 and 5. The Interior Point method tested on a dense '·' and a sparse '+' problem. All timings are an average of three observations.

column of **A** will always be dense, see (3.5).

If `Dense_cols_exist = 1` then conjugated gradient iterations is performed at line `887` and `911`. Because of those conjugated gradient iterations, `lipsol` is able to reduce the computation time, which for the sparse Laplace problem is quite dramatic.

It is also interesting to note, that the reduction in computation time is not that big for the dense Fourier problem when compared to the sparse problem. In the following we compare the dense and the sparse problems. Figure 6.7 shows the timing results for the Fourier problem solved by the AS and IP methods.

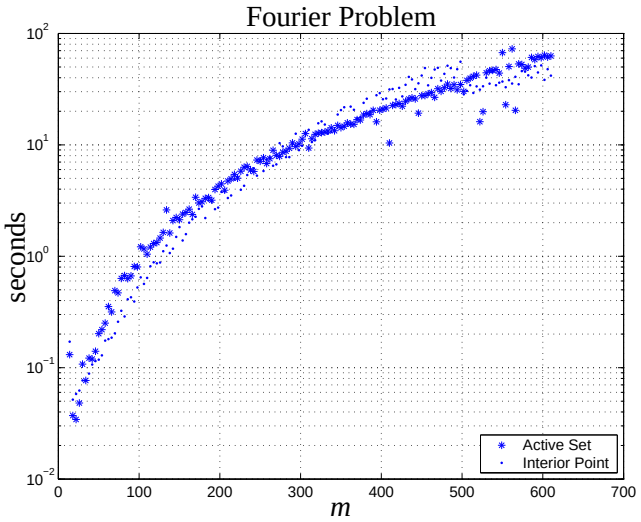

Figure 6.7: Test cases 1 and 2. The dense Fourier problem, solved by the AS method '∗' and by the IP method '·'. All timings are an average of three observations

From the figure we see that the AS and IP methods uses nearly the same time to solve the problem. For $m < 300$ IP is slightly faster than AS, but in turn AS is faster when $300 < m < 500$. For $m > 500$ the IP method uses conjugated gradient iterations, and this gives a slight decrease in computation time. In fact, just enough to make it up to speed with the AS method.

Figure 6.8 shows the timing results for the sparse problem, solved by the AS and IP methods.

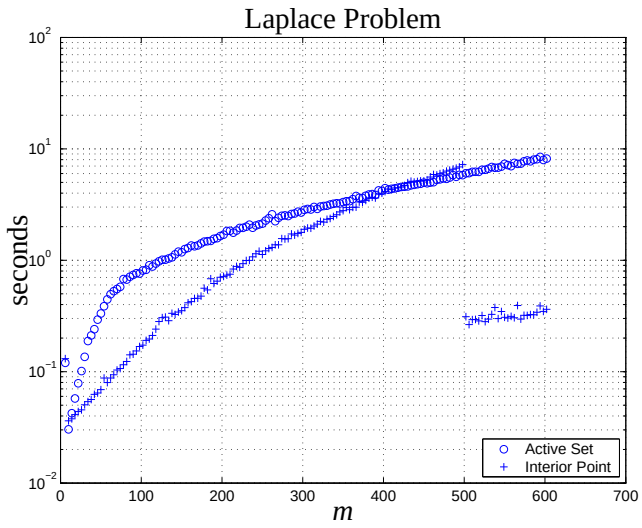

Figure 6.8: Test cases 4 and 5. The sparse Laplace problem, solved by the AS method '∘' and the IP method '+'. All timings are an average of three observations.

The figure clearly shows that the IP method is the fastest method for the sparse problem, however, when $400 < m < 500$ then the AS method seems to catch up with the IP method. When $m > 500$ the IP method begins to use conjugated gradient iterations, and this gives a dramatic decrease in computation time for this kind of sparse problem.

We have also tested the AS and IP methods with the Fourier problem, but with even more constraints than before. This test corresponds to test case 3 and 6 in Table 6.2. The many constraints (large m compared to n) and the dense problem, will have the effect that the IP iterations will take even longer than the AS iterations. As mentioned in the previous discussion, the workload for both methods grows with the number of constraints. Further, the Simplex method can use a smart factorization update, while this can not be exploited in the IP method. Therefore it is interesting to see if the IP method (test case 6) will do as well as the AS method (test case 3). Figure 6.9 shows the results from test case 3 and 6.

The figure shows that AS method is slightly faster than the IP method, but still their computational time results are comparable. This seems to suggest that even though the IP iterations becomes more and more expensive as m grows, it is able to compete with the Simplex method for large sized problems.

In [HL95, Chapter 4] there is a discussion about the pro and cons when comparing the interior point method and the Simplex method. They say that for large problems with thousands of functional constraints the IP method are likely to be faster than the simplex method.

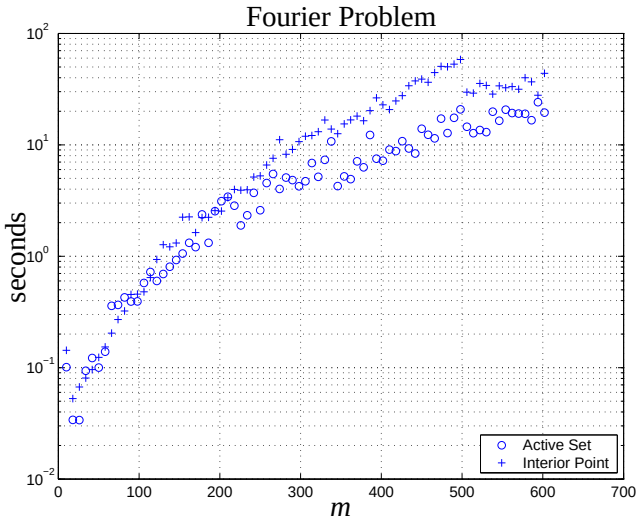

Figure 6.9: Test cases 3 and 6. The sparse Fourier problem, solved by the AS method '∘' and the IP method '+'. All timings are an average of three observations.

We have seen how the conjugated gradient (CG) iterations could speed up the IP method for the sparse Laplace system. From the comments in `lipsol` it is seen that preconditioned conjugated gradients is used during the CG iterations. When looking on the structure of **A** for the Laplace problem we notice that **A** has a block Toeplitz structure and perhaps this kind of structure is well suited for precondition. This could maybe explain the dramatic reduction in computational time we get when solving the Laplace problem. However, a thorough investigation of this, falls outside the scope of this thesis. Instead the reader is referred to [GvL96, Section 4.7 and 10.3] for more information about Toeplitz and preconditioned conjugated gradients.

6.4.1 Test of SLP and CSLP

In this section the SLP and CSLP algorithms are tested on a set of test functions that are given in Table 3.1 on page 38. The tests are performed with `linprog` set to large scale mode, and the results are presented in Table 6.3 on page 88.

The table shows results, that are comparable with the result given in Table 3.3, where `linprog` is used with medium scale setting. In other words, the performance of SLP and CSLP are not changed significantly when using either medium scale or large scale mode.

Test of SLP

Name	10^{-2}	10^{-5}	10^{-8}	corrective steps		
Parabola	11(12)	21(22)	31(32)	-	-	-
Rosenbrock1	18(19)	18(19)	18(19)	-	-	-
Rosenbrock2	19(20)	19(20)	19(20)	-	-	-
BrownDen	19(20)	32(33)	42(43)	-	-	-
Bard1	3(4)	4(5)	5(6)	-	-	-
Bard2	3(4)	4(5)	5(6)	-	-	-
Ztran2f	6(7)	21(22)	30(31)	-	-	-
Enzyme	179(180)	184(185)	185(186)	-	-	-
El Attar	7(8)	9(10)	10(11)	-	-	-
Hettich	8(9)	19(20)	26(27)	-	-	-

Test of CSLP - $\mathbf{J}(\mathbf{x})$

Name	10^{-2}	10^{-5}	10^{-8}	corrective steps		
Parabola	9(13)	15(25)	24(43)	4[1]	10[1]	19[1]
Rosenbrock1	12(14)	12(14)	12(14)	5[4]	5[4]	5[4]
Rosenbrock2	17(28)	18(29)	18(29)	11[1]	11[1]	11[1]
BrownDen	15(20)	29(41)	42(57)	4[0]	11[0]	14[0]
Bard1	3(4)	4(5)	5(6)	0[0]	0[0]	0[0]
Bard2	3(4)	4(5)	5(6)	0[0]	0[0]	0[0]
Ztran2f	6(7)	21(30)	30(43)	0[0]	8[0]	12[0]
Enzyme	58(85)	61(89)	61(89)	32[6]	33[6]	33[6]
El Attar	7(8)	9(10)	10(11)	1[1]	1[1]	1[1]
Hettich	7(11)	18(33)	29(55)	5[2]	16[2]	27[2]

Test of CSLP - $\mathbf{J}(\mathbf{x}+\mathbf{h})$

Name	10^{-2}	10^{-5}	10^{-8}	corrective steps		
Parabola	7(13)	13(25)	20(39)	5[0]	11[0]	18[0]
Rosenbrock1	11(14)	13(16)	13(16)	6[4]	6[4]	6[4]
Rosenbrock2	9(13)	11(15)	11(15)	4[1]	4[1]	4[1]
BrownDen	15(20)	29(41)	36(52)	4[0]	11[0]	15[0]
Bard1	3(4)	4(5)	5(6)	0[0]	0[0]	0[0]
Bard2	3(4)	4(5)	5(6)	0[0]	0[0]	0[0]
Ztran2f	6(7)	21(30)	30(43)	0[0]	8[0]	12[0]
Enzyme	26(48)	34(62)	35(63)	22[1]	28[1]	28[1]
El Attar	6(8)	8(10)	9(11)	1[0]	1[0]	1[0]
Hettich	5(7)	13(23)	21(39)	3[2]	11[2]	19[2]

Table 6.3: Results for SLP and CSLP with `linprog` set to large scale mode. Unless otherwise noted, $\eta_0 = 1$ and $\varepsilon = 0.01$. Column 2–4: number of iterations. ( ): The number of function evaluations. Column 5–7: Attempted corrective steps. [ ]: Failed corrective steps.

Chapter 7

Conclusion

In this chapter we give a summary of the performed work, and comments on the insight gained through this project.

The project started out with an implementation of the CSLP algorithm, described in Chapter 3.3 and originally presented in [JM94]. The work of analyzing the CSLP algorithm was a motivation for learning more about the theory behind minimax optimization.

In the process of implementing CSLP, a more basic algorithm SLP was created. This is described in Chapter 3 and corresponds to *method 1* in [Mad86]. SLP does not use a corrective step like CSLP, and is therefore more easy to analyze. The insights gained from this analysis were used in the process of implementing CSLP.

The corrective step was described and analyzed in detail in Chapter 3.2. In order to find the linearly independent gradients needed for the corrective step, we used a QR factorization. The work showed that we do not need **Q** which is good, especially when the problem is large and sparse, **Q** will be huge and dense. Further the QR factorization should be rank revealing, and in Matlab, the sparse version of `qr` does not do column permutations, which is needed for our purpose. Hence we use the economy version of `qr` for full matrices.

Both, SLP and CSLP uses only first order information, and studies of the performance in Chapter 3.3 showed that if a given problem was not regular in the solution, then only linear convergence could be obtained in the final stages of convergence. From a theoretical standpoint, this comes as no surprise.

Studies of the CSLP algorithm showed, however, that it uses fewer iterations to get into the vicinity of a local minimum. Further it was shown that CSLP was cheap, in the sense that it uses almost the same number of function evaluations as SLP.

CSLP showed in particular a good performance on problems that were highly non-linear. Both SLP and CSLP are well suited for large scale optimization, as tests have shown.

Large parts of the work presented in thesis is of theoretical character. The theory of unconstrained and constrained minimax was presented in Chapter 2 and 4. The study of the theory was in itself very interesting, and new insights and contributions were given, especially in the area regarding the penalty factor σ for exact penalty functions. It was revealed, that the estimation of σ was a problem of finding the distance to the edge of a convex hull, also

known as the generalized gradient. Further it was shown, that this in fact corresponds to solving a dual linear problem.

The work with trust regions, showed that the continuous trust region update presented in [Nie99a], tweaked in the right way, has just as good a performance as the discontinuous update strategy, used in most trust region algorithms.

The algorithms presented in this thesis are implemented in a small toolbox, and the source code can be found in Appendix D.

Finally, tests presented in Chapter 6 showed that `linprog` is well suited for large and sparse problems.

7.1 Future Work

In this work We have covered many areas of minimax optimization, but there is still room for further studies. In the following we give some suggestions for future investigations.

In the process of describing and implementing the SLP and CSLP algorithms, we did not use the multipliers to find the active set. Instead we used

$$\mathcal{A}_{LP} = \{\ j \mid |\ell_j(\mathbf{h}) - \alpha| \leq \gamma\ \}\ ,$$

where all the inner functions ℓ_j close to α are regarded as active. The drawback of this approach is that it requires the preset parameter γ. An area of future research would be to find a good heuristic to determine γ on the basis of the problem being solved.

The work with trust regions showed, that there are many ways to update the trust region radius. All of them, are heuristics that have proven their worth in practice. However, none of these depend on the problem being solved. It would be useful to find an update heuristic that was somehow connected to the underlying problem, e.g., by using neural networks.

Last but not least, the theoretical work done with the penalty factor, would be well suited for further research. One such research area, could be to find extensions that could be used to identify an initial basic feasible solution in the Simplex algorithm.

Appendix A

The Fourier Series Expansion Example

A periodic function $f(t)$ with a period of $T = 2\pi/\omega$ can be expanded by the Fourier series, which is given by

$$f(t) = \frac{1}{2}a_0 + \sum_{p=1}^{\infty} a_p \cos p\omega t + \sum_{p=1}^{\infty} b_p \sin p\omega t \tag{A.1}$$

where the coefficients can be found by

$$\begin{aligned} a_p &= \frac{1}{T}\int_d^{d+T} f(t)\cos p\omega t \,\mathrm{d}t \\ b_p &= \frac{1}{T}\int_d^{d+T} f(t)\sin p\omega t \,\mathrm{d}t \end{aligned} \tag{A.2}$$

In e.g. electrical engineering the periodic function $f(t)$ can also be looked upon as a design criterion i.e. we have a specification of design, and we want to fit it with a Fourier expansion.

Finding the Fourier coefficients by using (A.2) corresponds to the ℓ_2 norm solution i.e. the least squares solution. We can, however, also find these coefficients by using the *normal equations*, and solve it by simple linear algebra.

If we arrange the coefficients in a vector $x \in \mathbb{R}^n$ and set up a matrix $A \in \mathbb{R}^{m\times n}$, where m is the number of samples, and setup a vector $b \in \mathbb{R}^m$ that consists of function evaluations of $f(t)$ at the sample points. Then we can find the solution to the system

$$Ax = b \ ,$$

by using the normal equations

$$x = (A^T A)^{-1} A^T b \ ,$$

which find the ℓ_2 norm solution. A more numerical stable way to solve the normal equations is to use QR factorization.

$$x = R^{-1}_{1:n,:} Q^T_{:,1:n} b \ .$$

In Matlab this can be done efficiently for over determined systems, by using `x = A\b;`.

We can also find the ℓ_1 and ℓ_∞ solutions if we solve a linear programming problem with an LP solver e.g. Matlab's `linprog`. In the following we define a residual $r(x) = y - Fx$.

A.1 The ℓ_1 norm fit

To find the ℓ_1 solution we set up the following primal system

$$(P)\ \min_{\hat{x}} c^T\hat{x}\ \text{ subject to }\ A\hat{x} \le b$$

where

$$\hat{x} = \begin{bmatrix} x \\ z \end{bmatrix},\ c = \begin{bmatrix} 0 \\ e \end{bmatrix},\ A = \begin{bmatrix} F & -I \\ -F & -I \end{bmatrix},\ b = \begin{bmatrix} y \\ -y \end{bmatrix}$$

where $x \in \mathbb{R}^n$, $z, e, y \in \mathbb{R}^m$ and e is a column vector of ones and y is a vector with m samples of $f(t)$. Further, $F \in \mathbb{R}^{n\times m}$ and the unit matrix $I \in \mathbb{R}^{m\times m}$.

We see that a residual can be expressed by the above equations so that

$$\begin{array}{rcrcl} Fx-y & \le & Iz & \Leftrightarrow & r_i(x) \ge -z_i \\ Fx-y & \ge & -Iz & & r_i(x) \le z_i \end{array}$$

where $i = 1,\ldots,m$. By using the above, we see that minimizing the cost function

$$\min_{\hat{x}} c^T\hat{x} \Leftrightarrow \min_{\hat{x}} [0\ \ e] \begin{bmatrix} x \\ z \end{bmatrix} \Rightarrow \min_{z} \sum_{i=1}^{m} z_i \Rightarrow \min_{x} \sum_{i=1}^{m} r_i(x)\ ,$$

is exactly the same as minimizing the sum of the residuals. This is also the definition of the ℓ_1 norm.

The primal system is large, and we can get the exact same solution by solving a smaller dual problem, as shown in [MNP94]. The dual problem can be expressed as

$$(D)\ \max_{u} y^T u\ \text{ subject to }\ F^T u = 0\ \text{ and }\ -e \le u \le e\ ,$$

Because of the symmetry between the primal and dual problem, we have that the Lagrangian multipliers of the dual problem correspond to x.

A.2 The ℓ_∞ norm fit

The solution to the problem of minimizing the largest absolute residual, can also be found by solving a linear problem. We solve the following system

$$(P)\ \min_{\hat{x}} c^T\hat{x}\ \text{ subject to }\ A\hat{x} \le b$$

where

$$\hat{x} = \begin{bmatrix} x \\ \alpha \end{bmatrix},\ c = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\ A = \begin{bmatrix} F & -e \\ -F & -e \end{bmatrix},\ b = \begin{bmatrix} y \\ -y \end{bmatrix},$$

with the same dimensions as described in the former section. By using the above we see that

$$\begin{array}{rcrcl} Fx-y & \le & \alpha e & \Leftrightarrow & -r_i(x) \le \alpha \\ -Fx+y & \le & \alpha e & & r_i(x) \le \alpha \end{array}\ .$$

So when minimizing the cost function, we are in fact minimizing the maximum absolute residual

$$\min_{x,\alpha} \alpha \quad \alpha = \max_{x} \{r(x), -r(x)\} .$$

The above system for ℓ_∞ is identical with (3.7), it is just two different ways to get to the same result. If we define a residual function $f(x) = y - Fx$ and minimize it with respect to the ℓ_∞ norm, then

$$\begin{array}{rcl} f(x) + J(x)\Delta x & \leq & \alpha \\ -f(x) - J(x)\Delta x & \leq & \alpha \end{array} \Leftrightarrow \begin{bmatrix} J(x) & -e \\ -J(x) & -e \end{bmatrix} \begin{bmatrix} \Delta x \\ \alpha \end{bmatrix} \leq \begin{bmatrix} -f(x) \\ f(x) \end{bmatrix}$$

Appendix B

The Sparse Laplace Problem

The Laplace equation can be used to solve steady heat conduction, aerodynamics and electrostatic problems, just to mention a few. Formally we seek a solution to the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{B.1}$$

which turns out to be somewhat difficult to solve analytically. However, there is an easy numerical solution by using a mesh. Information about the boundary condition of the solution region is given, so all the mesh points has to be solved simultaneously. This is done by solving a linear system equations, by using a finite difference approximation to (B.1).

$$\frac{u(i+1,j) - 2u(i,j) + u(i-1,j)}{\Delta x^2} + \frac{u(i,j+1) - 2u(i,j) + u(i,j-1)}{\Delta y^2} = 0\,. \tag{B.2}$$

If the grid is equidistant in the x and y direction, then we can reformulate the above to this, somewhat simpler expression

$$4u(i,j) = u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1)\,. \tag{B.3}$$

We see that the above gives rise to the famous five point computational molecule. The idea is now to use a grid of the solution area that contains $m \times n$ points and to solve $Au = b$.

A simple grid of size $m = 2$ (columns) and $n = 2$ (rows) is illustrated on figure B.1 and by using (B.3) we get the following equations on matrix form

$$\begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{B.4}$$

When implementing this problem in Matlab, one must remember to use sparse matrices, because the sparsity can be exploited to conserve memory, and to solve the Laplace system *much* faster than by using dense matrices. One can test this, by using `u = A\b` for both dense and sparse matrices.

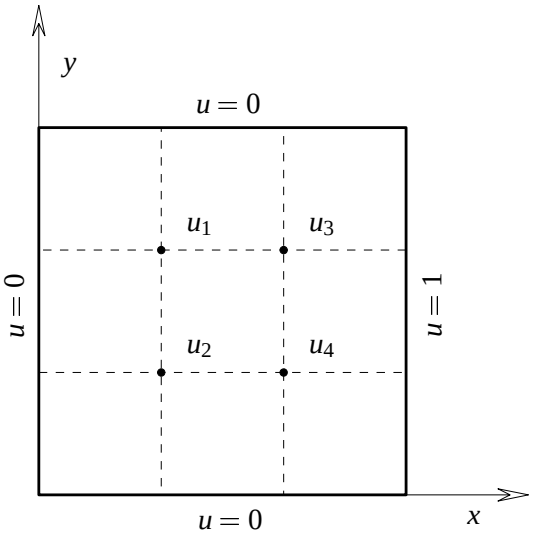

Figure B.1: The grid used for solving the Laplace problem. The grid dimension is $m = 2$, $n = 2$.

For large grids we use the *Kronecker product* $\otimes$ to generate A, so that

$$A = I \otimes B + L \otimes C\,,$$

and

$$B = \begin{bmatrix} 4 & -1 & \cdots & 0 \\ -1 & 4 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -1 \\ 0 & \cdots & -1 & 4 \end{bmatrix}, \quad C = \begin{bmatrix} -1 & 0 & \cdots & 0 \\ 0 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -1 \end{bmatrix},$$

where $B, C \in \mathbb{R}^{m\times m}$. Further, we have that $I \in \mathbb{R}^{n\times n}$ is the unit matrix and

$$L = \begin{bmatrix} 0 & 1 & \cdots & 0 \\ 1 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \cdots & 1 & 0 \end{bmatrix}, \; L \in \mathbb{R}^{n\times n}\,.$$

The resulting system will have $A \in \mathbb{R}^{mn\times mn}$ and $u, b \in \mathbb{R}^{mn}$. In Matlab, we write `A = kron(I,B) + kron(L,C);`.

For the purpose of making a sparse test problem for `linprog`, we change the least squares problem $Au = b$ into a Chebyshev problem ℓ_∞, by giving `linprog` the following problem

$$F = \begin{bmatrix} A & -e \\ -A & -e \end{bmatrix}, \; y = \begin{bmatrix} b \\ -b \end{bmatrix},$$

where $e \in \mathbb{R}^{mn}$ is a vector of ones, and solving $\min_{\hat{u}} c^T \hat{u}$ subject to $F\hat{u} \leq y$.

A visualization of the solution for a grid of size $m = 20$ and $n = 20$ with the same boundary conditions as in figure B.1 is shown in figure B.2.

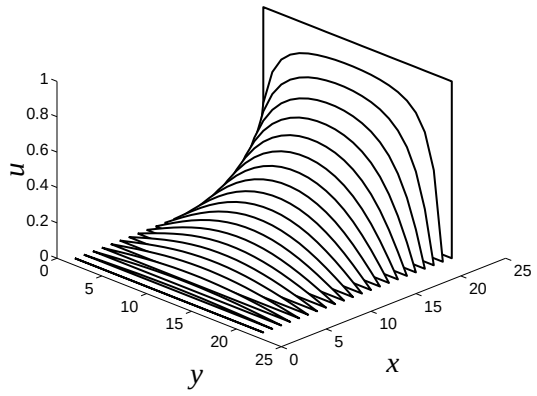

Figure B.2: The solution to the Laplace problem. The grid dimension is $m = 20$, $n = 20$ and the same boundary conditions is used as those shown in figure B.1.

Appendix C

Test functions

The test functions used in this repport, is defined in the following. For convenience we repeat the definition of the different types of problems here.

Minimax problems is defined as

$$\min_x F(\mathbf{x}) \quad , \quad F(\mathbf{x}) \equiv \max\{f_j(\mathbf{x})\} \ , \tag{C.1}$$

where $j = 1,\ldots,m$. The Chebyshev norm $\|\cdot\|_\infty$ can be formulated as a minimax problem

$$\min_x F(\mathbf{x})\ F(\mathbf{x}) \equiv \max\{\max_j(f_j(\mathbf{x})), \max_j(-f_j(\mathbf{x}))\}. \tag{C.2}$$

The test functions are presented in the list below. We use the following format.

a) Type of problem and dimension.
b) Function definition.
c) Start point.
d) Solution and auxiliary data.

Bard [Bar70].

a) Type (C.2). Dimension: $\mathbf{y}'$: $n = 3$, $m = 5$, $t^* = 3$. $\mathbf{y}''$: $n = 3$, $m = 4$, $t^* = 4$.

b)
$$f_j(\mathbf{x}) = y_j - x_1 + \frac{u_j}{v_j x_2 + w_j x_3} \ , \ j = 1,\ldots,15$$
where $u_j = j$, $v_j = 16 - i$ and $w_j = \min\{u_j, v_j\}$, $y_j = y'_j$ or $y_j = y''_j$.

c) $\mathbf{x}_0 = [\ 1\ \ 1\ \ 1\ ]^T$.

d) For $y_j = y'_j$: $F(\mathbf{x}^*) \simeq 0.050816326531$. For $y_j = y''_j$: $F(\mathbf{x}^*) \simeq 0.0040700234725$. Data:

i	y'_j	y''_j	i	y'_j	y''_j	i	y'_j	y''_j
1	0.14	0.16	6	0.32	0.37	11	0.73	0.83
2	0.18	0.21	7	0.35	0.40	12	0.96	1.10
3	0.22	0.26	8	0.39	0.43	13	1.34	1.54
4	0.25	0.30	9	0.37	0.53	14	2.10	2.43
5	0.29	0.34	10	0.58	0.66	15	4.39	5.10

Brown-Den Brown and Dennis [BD71].

a) Type: (C.2). Dimension: $n = 4$, $m = 20$, $t^* = 3$.

b)
$$f_j(\mathbf{x}) = (x_1 + t_j x_2 - e^{t_j})^2 + (x_3 + x_4 \sin t_j - \cos t_j)^2$$
where $t_i = i/5$, $i = 1, \ldots, 20$.

c) $\mathbf{x}_0 = [\ 25\ \ 5\ \ -5\ \ -1\ ]^T$.

d) $F(\mathbf{x}^*) = 115.70643952$.

Rosenbrock [Ros60].

a) Type (C.2). Dimension: $n = 2$, $m = 2$, $t^* = 3$.

b)
$$f_1(\mathbf{x}) = w(x_2 - x_1^2)\ ,\ f_2(\mathbf{x}) = 1 - x_1$$
where $w = 10$ or $w = 100$.

c) $\mathbf{x}_0 = [\ -1.2\ \ 1\ ]^T$.

d) $F(\mathbf{x}^*) = 0$.

Parabola

a) Type: (C.1). Dimension: $n = 2$, $m = 2$, $t^* = 2$.

b)
$$f_1(\mathbf{x}) = x_1^2 - x_2\ ,\ f_2(\mathbf{x}) = x_2$$

c) $\mathbf{x}_0 = [\ -3\ \ 3\ ]^T$.

d) $F(\mathbf{x}) = 0$.

Enzyme [KO68]

a) Type: (C.2). Dimension: $n = 4$, $m = 11$.

b)

$$f_j(\mathbf{x}) = v_j - \frac{x_1(y_j^2 + x_2 y_j)}{y_j^2 + x_3 y_j + x_4} \quad , \quad j = 1, \ldots, 11.$$

c) $\mathbf{x}_0 = [\ 0.5\ \ 0.5\ \ 0.5\ \ 0.5\ ]^T$.

d) Data:

j	v_j	y_j	j	v_j	y_j
1	0.1957	4.0000	7	0.0456	0.1250
2	0.1947	2.0000	8	0.0342	0.1000
3	0.1735	1.0000	9	0.0323	0.0833
4	0.1600	0.5000	10	0.0235	0.0714
5	0.0844	0.2500	11	0.0246	0.0625
6	0.0627	0.1670	-	-	-

El-Attar [EAVD79]

a) Type: (C.2). Dimension: $n = 6$, $m = 51$.

b)

$$f_j(\mathbf{x}) = x_1 e^{-x_2 t_j} \cos(x_3 t_j + x_4) + x_5 e^{-x_6 t_j} - y_j$$

$$y_j(\mathbf{x}) = \frac{e^{-t_j}}{2} - e^{-2t_j} + \frac{e^{-3t_j}}{2} + 3e^{-3t_j/2}\frac{\sin(7t_j)}{2} + e^{-5t_j/2}\sin(5t_j)\ ,$$

where $j = 1, \ldots, 51$ and $t_j = (j-1)/10$.

c) $\mathbf{x}_0 = [\ 2\ \ 2\ \ 7\ \ 0\ \ -2\ \ 1\ ]^T$.

d) $F(\mathbf{x}) = 0.034904$.

Hettich [Wat79]

a) Type: (C.2). Dimension: $n = 4$, $m = 5$.

b)

$$f_j(\mathbf{x}) = \sqrt{t_j} + ((x_1 t_j + x_2)t_j + x_3)^2 - x_4$$

where $j = 1, \ldots, 5$ and $t_j = 0.25 + (j-1)0.75/4$.

c) $\mathbf{x}_0 = [\ 0\ \ -0.5\ \ 1\ \ 1.5\ ]^T$.

d) $F(\mathbf{x}^*) = 0.002459$.

Appendix D

Source code

D.1 Source code for SLP in Matlab

```
function [x,info, perf] = slp4(fun, fpar, x0, opts)
%
% Usage: [x, info, perf] = slp2(fun, fpar, x0, opts)
%
% Sequential Linear Programing solver with continous
% trust region update [2]. To use default setting,
% omit opts from the argument list.
%
% INPUT:
%   fun : a string with a function name.
%         All function should follow this API
%         [f,J] = fun(x, fpar);
%   fpar: parameters to the function, (may be empty).
%   x0 : start point.
%   opts      : options.
%
% The options are set by using SETPARAMETERS.
% call SLP with no arguments to see default values.
%   rho   : Set trust region radius rho.
%   epsilon   : Threshold for accepting new step.
%   maxIter   : Maximum number of iterations.
%   minStep   : If ||step||_2 < minStep then stop.
%   trustRegion : choose trust-region update.
%               1: Classical update with damping [1]
%               2: Continous update of trust region. See [2].
%               3: Trust region update. See [3].
%               4: Enhanced classical update with damping.
%               5: Classical update strategy.
%               6: Udate strategy that uses the steplength.
%   precisionDelta : Stop when a precision delta is reached.
%   precisionFun    : The function value at the solution.
%
% output:
%   X:    a vector containing all points visited.
%   info(1) : Number of iterations.
%   info(2) : Active stopping criterion.
%             1: Maximum number of iterations reached..
%             2: Small step
%             3: dL <= 0.
%             4: Desired precision reached.
%   info(3) : Function value F(x).
%   info(4) : Number of function evaluations.
%   perf: Contains the following fields:
%         F    : Function evaluation.
%         R    : Trust region radius.
%         X    : Steps.
%         fval: Function evaluation at the solution.
%         multipliers : Multipliers at the solution.
%
% In part based upon:
% [1]: K. Jonasson and K. Madsen, Corrected Sequential Linear Programming
%      for Sparse Minimax Optimization. BIT 34 (1994), 372-387.
% [2]: H.B. Nielsen. Damping parameter in Marquardt's method.
%      Technical Repport IMM-REP-1999-05. DTU, Kgs. Lyngby.
% [3]: K. Madsen. Minimax Solution of Non-linear Equations without
%      calculating derivatives.
%
% Release version 1.0  28-10-02.
```

```
% Mark Wrobel. proj76@imm.dtu.dk

if nargin < 1,
  fprintf('                    rho : [ Default = 0.1*max(x0) ]\n');
  fprintf('                epsilon : [ Default = 0.01 ]\n');
  fprintf('                maxIter : [ Default = 100 ]\n');
  fprintf('                minStep : [ Default = 1e-10 ]\n');
  fprintf('            trustRegion : [ Default = 1 ]\n');
  fprintf('         precisionDelta : [ Default = 0 ]\n');
  fprintf('           precisionFun : [ Default = 0 ]\n');
  fprintf('        derivativeCheck : [ Default = "off" ]\n');
  fprintf('             largescale : [ Default = "off" ]\n');
  return
end

if ~exist('opts')
  opts = setparameters('empty');
end
if isempty(opts)
  opts = setparameters('empty');
end

rho = setChoise(opts.rho, 0.1*max(x0));
epsilon = setChoise(opts.epsilon, 0.01);
max_iter = setChoise(opts.maxIter, 100);
min_step = setChoise(opts.minStep, 1e-10);
trustregion = setChoise(opts.trustRegion, 1);
derivativeCheck = setChoise(opts.derivativeCheck, [], 1);
precision_delta = setChoise(opts.precisionDelta, 0);
precision_f = setChoise(opts.precisionFun, 0);
largescale = setChoise(opts.largescale, [], 1, 1);

if derivativeCheck,
    % See p. 3 in "Checking Gradients" H.B. Nielsen IMM 21.9.200
    % Download at www.imm.dtu.dk/~hbn
    [maxJ, err, index] = checkjacobi(fun, fpar, x0, eps^(1/3)*norm(x0));
    if abs(err(3)) > 0.9*abs(err(1)),
        error('Derivative check failed');
    end
end

% Initialization.
F = []; R = [];
stop = 0; iterations = 0; fcount = 0; nu = 2;
flag = 1;  %<-Take the role as gain_old. (gain_old = 2*epsilon, see [1])
x = x0(:); %<- ensures a column vector
X = x;     %<-X is a matrix of x's

fcount = fcount + 1;
[f, J] = feval(fun, x, fpar);
[m, n] = size(J);
activeset = zeros(2*m,1);

%--------------MAIN LOOP-----------
while ~stop
  LB = [repmat(-rho,n,1); -inf]; UB = -LB;
  % defining constraints for linprog (2.4)
  A = [J -ones(m,1)];
  b = -f;

  options = optimset('Display','off','LargeScale', largescale);
  [s, fevl, exitfl, outpt, lambda] = ...
      linprog([repmat(0,1,n) 1], A, b, [], [], LB, UB, [], options);
  if exitfl < 0,
    error(sprintf('%s: linprog failed', algorithm));
  end

  alpha = s(end); d = s(1:n);
  xt = x + d;
  [ft, Jt] = feval(fun, xt, fpar); fcount = fcount + 1;

  FX = max(f);
  dF = FX - max(ft);
  dL = FX - alpha;

  % storing information.
  F = [F FX]; R = [R rho]; X = [X x];

  % do not calculate gain = dF/dL due to numeric unstability.
  if dF > epsilon*dL & (dL > 0),
    x = xt; f = ft; J = Jt; flag = 1;
  else, flag = 0; end


  switch trustregion
   case 1 %<- Classical update with damping [1]
    if (dF > 0.75*dL) & flag,
      rho = 2.5*rho;
```

```
    elseif dF < 0.25*dL
      rho = rho/2;
    end
   case 2 %<- Continous update of trust region. See [2].
    gamma = 2; beta = 2.5;
    if (dF > 0) & (dL > 0)
      if norm(d,inf) > 0.9*rho,
        rho = rho * min(max(1/gamma, 1 + (beta - 1)*(2*dF/dL - 1)^5),beta);
        nu = 2;
      end
    else
      rho = rho/nu; nu = nu*2;
    end
   case 3 %<- Trust region update. See [3].
    rho1 = 0.01; rho2 = 0.1; sigma1 = 0.5; sigma2 = 2;
    if dF <= rho2*dL,
      rho = sigma1*max(norm(d,inf), 1/sigma2*rho);
    else
      rho = min(sigma2*max(norm(d,inf), 1/sigma2*rho), 2);
    end
   case 4 %<- Enhanced classical update with damping.
    if (dF > 0.75*dL) & flag,
      rho = 2.5 * min(rho, norm(d, inf));
    elseif dF < 0.25*dL
      rho = 0.5 * min(rho, norm(d, inf));
    end
   case 5 %<- Classical update strategy.
    if (dF > 0.75*dL),
      rho = 2.5*rho;
    elseif dF < 0.25*dL
      rho = rho/2;
    end
   case 6 %<- Udate that use steplength.
    if (dF > 0.75*dL),
      rho = max(2.5*norm(d, inf), rho);
    elseif dF < 0.25*dL
      rho = 0.25*norm(d, inf);
    end
   otherwise
    error(sprintf('No trust region method %d',trustregion));
  end  %<-end switch

  iterations = iterations + 1;

  if iterations >= max_iter, stop = 1;
  elseif norm(d) < min_step, stop = 2;
  elseif dL < 0, stop = 3;
  end

  if precision_delta,
    % use the precision stop criteria.
    F_x = max(feval(fun,x,fpar));
    if ((F_x - precision_f)/max(1,precision_f) <= precision_delta),
      stop = 4;
    end
  end
end  %<-end while.

% storing performence parameters
perf.F = F; perf.R = R; perf.X = X; perf.fval = f;
perf.multipliers = lambda.ineqlin;
info(1) = iterations; info(2) = stop; info(3) = max(f);
info(4) = fcount;


%----------INNER FUNCTIONS----------

function [value] = setChoise(parameter, default, booleancheck, boolstring)

if nargin > 2 & ~isempty(default),
  error('Can not set a default value for a boolean expression');
end

if nargin < 3
  if isempty(parameter),
    value = default;
  else
    value = parameter;
  end
else
  if isempty(parameter),
    value = 0;
  else
    if strcmp(parameter, 'on'),
      value = 1;
    else
      value = 0;
```

```
    end
  end
end

if nargin == 4,
  if value,
    value = 'on';
  else
    value = 'off';
  end
end
```

D.2 Source code for CSLP in Matlab

```
function [x, info, perf] = cslp4(fun, fpar, x0, opts)
% Usage: [x, info, perf] = cslp4(fun, fpar, x0, opts)
% Minimax solver that uses a corrective step. To use
% default settings, omit opts from the argument list.
% This algorithm is based on [1].
%
% INPUT:
%   fun       : a string with a function name.
%               All functions should follow this API
%               [f,J] = fun(x,fpar);
%   fpar      : parameters to the function, if any.
%   x0        : start point.
%   opts      : options.
%
% The options are set by using SETPARAMETERS.
% call CSLP with no arguments to see default values.
%   rho   : Set trust region radius rho.
%   epsilon   : Threshold for accepting new step.
%   maxIter   : Maximum number of iterations.
%   minStep   : If ||step||_2 < minStep then stop.
%   trustRegion : choose trust-region update.
%               1: Classical update with damping [1]
%               2: Continous update of trust region. See [2].
%               3: Trust region update. See [3].
%               4: Enhanced classical update with damping.
%               5: Classical update strategy.
%               6: Udate strategy that uses the steplength.
%   useTentativeJacobian : If "yes" use the Jacobian at x+h
%                          for the corrective step.
%   precisionDelta : Stop when a precision delta is reached.
%   precisionFun   : The function value at the solution.
%
% output:
%   x       : Resulting x.
%   info(1): number of iterations.
%   info(2): active stoping criterion.
%            1 : Maximum number of iterations reached.
%            2 : Small step.
%            3 : dL <= 0
%          : 4 : Desired precision reached.
%   info(3): Function value F(x).
%   info(4): Number of function evaluations.
%   info(5): Number of active steps.
%   info(6): Number of failed active steps.
%   perf    : Contains the following fields:
%             F    : Function evaluation.
%             R    : Trust region radius.
%             X    : Steps.
%             fval: Function evaluation at the solution.
%             multipliers : Multipliers at the solution.
%
% References
% [1]: K.Jonasson and K.Madsen, Corrected Sequential Linear Programming
%      for Sparse Minimax Optimization. BIT 34 (1994), 372-387.
% [2]: H.B.Nielsen. Damping parameter in Marquardt's method.
%      Technical Repport IMM-REP-1999-05. DTU, Kgs. Lyngby.
% [3]: K. Madsen. Minimax Solution of Non-linear Equations without
%      calculating derivatives.
%
% Mark Wrobel. proj76@imm.dtu.dk

if nargin < 1,
  fprintf('                    rho : [ Default = 0.1*max(x0) ]\n');
  fprintf('                epsilon : [ Default = 0.01 ]\n');
  fprintf('                maxIter : [ Default = 100 ]\n');
  fprintf('                minStep : [ Default = 1e-10 ]\n');
  fprintf('             trustRegion : [ Default = 1 ]\n');
  fprintf('          precisionDelta : [ Default = 0 ]\n');
  fprintf('            precisionFun : [ Default = 0 ]\n');
  fprintf('         derivativeCheck : [ Default = "off" ]\n');
  fprintf('    useTentativeJacobian : [ Default = "off" ]\n');
  fprintf('              largescale : [ Default = "off" ]\n');
  return
end

if ~exist('opts')
  opts = setparameters('empty');
end
if isempty(opts)
  opts = setparameters('empty');
end

rho = setChoise(opts.rho, 0.1*max(x0));
epsilon = setChoise(opts.epsilon, 0.01);
max_iter = setChoise(opts.maxIter, 100);
min_step = setChoise(opts.minStep, 1e-10);
```

```
trustregion = setChoise(opts.trustRegion, 1);
derivativeCheck = setChoise(opts.derivativeCheck, [], 1);
precision_delta = setChoise(opts.precisionDelta, 0);
precision_f = setChoise(opts.precisionFun, 0);
corrstepJX = setChoise(opts.useTentativeJacobian, [], 1);
largescale = setChoise(opts.largescale, [], 1, 1);

if derivativeCheck,
    % See p. 3 in "Checking Gradients" H.B. Nielsen IMM 21.9.200
    % Download at www.imm.dtu.dk/~hbn
    [maxJ, err, index] = checkjacobi(fun, fpar, x0, eps^(1/3)*norm(x0));
    if abs(err(3)) > 0.9*abs(err(1)),
        error('Derivative check failed');
    end
end

F = []; R = [];
fcount = 0;  stop = 0;  iterations = 0;
active_steps = 0; wasted_active = 0;
flag = 1;               % use flag instead of gain_old = 2*epsilon;
x = x0(:);              % ensures a column vector
X = x;                  % X is a matrix of x's


fcount = fcount + 1;
[f, J] = feval(fun, x, fpar);
[m, n] = size(J);

while ~stop
  LB = [repmat(-rho,n,1); -inf]; UB = -LB;
  % defining constraints for linprog (2.4)
  A = [J -ones(m,1)];
  b = -f;

  % solve the LP-problem.
  options = optimset('Display','off','LargeScale',largescale);
  [s, fevl, exitfl, outpt, lambda] = ...
      linprog([repmat(0,1,n) 1], A, b, [], [], LB, UB, [], options);
  if exitfl < 0,
    error('linprog failed');
  end

  alpha = s(end); h = s(1:n); d = h;

  xt = x + d;
  fcount = fcount + 1;
  [ft Jt] = feval(fun, xt, fpar);

  FX = max(f);
  dF = FX - max(ft);
  dL = FX - alpha;


  if (dL >= eps) & (dF <= epsilon*dL)  %<-Take a corrective step
    active_steps = active_steps + 1;
    if corrstepJX,
      v = corrstep(ft, Jt, h, f, J);  %<-Use J(x+h);
    else
      v = corrstep(ft, J, h, f, J);   %<-Use J(x);
    end

    % angle = acos((h'*v)/(norm(h)*norm(v)));
    % if pi/32 < abs(angle-pi),

    if (norm(v) <= 0.9*norm(h)) & (norm(v) > 0),
      d = h + v;
      d_length = norm(d,inf);
      % truncating to fit trust region
      if d_length > rho,
        d = rho*d/d_length;
      end
      xt = x + d;
      [ft Jt] = feval(fun, xt, fpar); fcount = fcount + 1;
      dF = FX - max(ft);
    else
      wasted_active = wasted_active + 1;
    end
  end

  % Storing information.
  F = [F FX]; R = [R rho]; X = [X x];

  if dF > epsilon*dL
    x = xt; f = ft; J = Jt; flag = 1;
  else
    flag = 0;
  end
```

```
  switch trustregion
   case 1 %<- Classical update with damping [1]
    if (dF > 0.75*dL) & flag,
      rho = 2.5*rho;
    elseif dF < 0.25*dL
      rho = rho/2;
    end
   case 2 %<- Continous update of trust region. See [2].
    gamma = 2; beta = 2.5;
    if (dF > 0) & (dL > 0)
      if norm(d,inf) > 0.9*rho,
        rho = rho * min(max(1/gamma, 1 + (beta - 1)*(2*dF/dL - 1)^5),beta);
        nu = 2;
      end
    else
      rho = rho/nu; nu = nu*2;
    end
   case 3 %<- Trust region update. See [3].
    rho1 = 0.01; rho2 = 0.1; sigma1 = 0.5; sigma2 = 2;
    if dF <= rho2*dL,
      rho = sigma1*max(norm(d,inf), 1/sigma2*rho);
    else
      rho = min(sigma2*max(norm(d,inf), 1/sigma2*rho), 2);
    end
   case 4 %<- Enhanced classical update with damping.
    if (dF > 0.75*dL) & flag,
      rho = 2.5 * min(rho, norm(d, inf));
    elseif dF < 0.25*dL
      rho = 0.5 * min(rho, norm(d, inf));
    end
   case 5 %<- Classical update strategy.
    if (dF > 0.75*dL),
      rho = 2.5*rho;
    elseif dF < 0.25*dL
      rho = rho/2;
    end
   case 6 %<- Udate that use steplength.
    if (dF > 0.75*dL),
      rho = max(2.5*norm(d, inf), rho);
    elseif dF < 0.25*dL
      rho = 0.25*norm(d, inf);
    end
   otherwise
    disp(sprintf('No trust region method %d',trustregion));
    break;
  end

  iterations = iterations + 1;

  if iterations >= max_iter,
    stop = 1;
  elseif norm(d) < min_step,
    stop = 2;
  elseif dL < 0,
    stop = 3;
  end

  if precision_delta, %<-use the precision stop criteria.
    F_x = max(feval(fun,x,fpar));
    if ((F_x - precision_f)/max(1,precision_f) <= precision_delta),
      stop = 4;
    end
  end
end %<- end while

% Formating output.
perf.F = F; perf.R = R; perf.X = X; perf.fval = f;
perf.multipliers = lambda.ineqlin;
info(1) = iterations; info(2) = stop; info(3) = max(f);
info(4) = fcount; info(5) = active_steps;
info(6) = wasted_active;


% ========== auxiliary function ==================================

function i = activeset(f)
%
% Usage: i = activeset(f)
% Finds the active set.
% Input:
%   f    : function evaluations (for cslp based on linarizations of f).
% Output:
%   i : The active set.

i = find(abs(f - max(f)) <= 1e-3);

%-------------------------------------------------------------------
```

```
function v = corrstep(f, J, h, f_lin, J_lin)
%
% Usage: v = corrstep(f, J, h, mode)
% Calculates the corrective step. Now with
% pivoting of the Jacobian.
% Input:
%   f    : Function evaluations of active functions.
%   J    : Jacobian of active functions.
%   h    : A step h, proposed by a LP-solver.
%   f_lin: Used to determin active functions in linprog.
%   J_lin: Used to determin active functions in linprog.
%
% Output:
%   v : The corrective step.

%[m,n] = size(J);
i = activeset(f_lin+J_lin*h);

A = J(i, :)'; b = f(i);
[n, m] = size(A);
% need some rank-revealing QR (RRQR) of A to avoid linear
% dependent gradients.
[Q,R,ee] = qr(A,0); E = sparse(m,m); E(ee + (0:m-1).*m) = 1;
%[Q,R,E] = qr(A);
b = E'*b;   %<-pivoting b
A = A*E;    %<-pivoted version of A.
j = find(abs(diag(R)) > 1.01*n*eps);%<- find small values on diagonal(R).
A = A(:,j);
b = b(j);

t = length(j);  %<-number of linear independent gradients.

if (t <= 1)     %<-Null solution
    v = zeros(size(h));
else
    I_h = zeros(n+1); I_h(1:n,1:n)=eye(n);
    A_h = [A; -ones(1,t)];
    v_h = [I_h  A_h; A_h' zeros(t)] \ [zeros(n+1,1); -b];
    v = v_h(1:n);
end

%-------------------------------------------------------------

function [value] = setChoise(parameter, default, booleancheck, boolstring)

if nargin > 2 & ~isempty(default),
  error('Can not set a default value for a boolean expression');
end

if nargin < 3
  if isempty(parameter),
    value = default;
  else
    value = parameter;
  end
else
  if isempty(parameter),
    value = 0;
  else
    if strcmp(parameter, 'on'),
      value = 1;
    else
      value = 0;
    end
  end
end

if nargin == 4,
  if value,
    value = 'on';
  else
    value = 'off';
  end
end
```

D.3 Source code for SETPARAMETERS in Matlab

```
function [opts] = setparameters(varargin)
%
% Usage: [opts] = setparameters(varargin)
%
% Takes variable input on the form
% options = setparameters('param1', value1, 'param2', value2, ...)
%
% Parameters
%------------------------------------------------------------------
% rho - Trust region radius.
% epsilon - if gainfactor > rho then a step is accepted.
% maxIter - Maximum number of iterations.
% minStep - If a step becomes lower than min_step, then the
%           algorithm stops.
% useSolver -
% trustRegion - Set the type of trustregion update.
%               1: Johnasson.
%               2: H.B. Nielsen.
%               3: K. Madsen.
%               4: Modified Johnasson.
%               5: Classical update.
%               6: Update that incorporates the steplength.
% precisionDelta - The precision of the solution. Requires that
%                  also precision_fun is set.
% precisionFun - The value of function in the solution.
% derivativeCheck - Check the gradients.
% useTentativeJacobian - Use the Jacobian evaluated at X tentative.
% empty - returns the empty struct.

if nargin == 0,
    fprintf('                   rho : [ positive scalar ]\n');
    fprintf('               epsilon : [ positive scalar ]\n');
    fprintf('               maxIter : [ positive integer ]\n');
    fprintf('               minStep : [ positive scalar ]\n');
    fprintf('            trustRegion : [ positive integer ]\n');
    fprintf('         precisionDelta : [ positive scalar ]\n');
    fprintf('           precisionFun : [ positive scalar ]\n');
    fprintf('        derivativeCheck : [ on | off ]\n');
    fprintf('   useTentativeJacobian : [ on | off ]\n');
    fprintf('             largescale : [ on | off ]\n');
    return
end

% Create struct.
opts = struct( 'rho', [], ...
               'epsilon', [], ...
               'maxIter', [], ...
               'minStep', [], ...
               'trustRegion', [], ...
               'precisionDelta', [], ...
               'precisionFun', [], ...
               'derivativeCheck', [], ...
               'useTentativeJacobian', [], ...
               'largescale', []);



if ~strcmp(lower(char(varargin(1))),'empty'),
  % Check input, assing values.
  numargs = nargin;
  if mod(numargs,2),
    error('Arguments must come in pairs');
  end

  for i = 1:2:numargs,
    parameter = lower(char(varargin(i)));
    value = cell2mat(varargin(i+1));
    if ~ischar(parameter), error('Parameter is not a string'); end
    try
      switch parameter,
       case 'rho'
        value = checkScalar(parameter, value);
        opts.rho = value;

       case 'epsilon'
        value = checkScalar(parameter, value);
        opts.epsilon = value;

       case 'maxiter'
        value = checkInteger(parameter, value);
        opts.maxIter = value;

       case 'minstep'
        value = checkScalar(parameter, value);
        opts.minStep = value;
```

```
        case 'trustregion'
         value = checkScalar(parameter, value);
         opts.trustRegion = value;

        case 'precisiondelta'
         value = checkScalar(parameter, value);
         opts.precisionDelta = value;

        case 'precisionfun'
         value = checkScalar(parameter, value);
         opts.precisionFun = value;

        case 'derivativecheck'
         value = checkYesNo(parameter, value);
         opts.derivativeCheck = lower(value);

        case 'usetentativejacobian'
         value = checkYesNo(parameter, value);
         opts.useTentativeJacobian = lower(value);

        case 'largescale'
         value = checkYesNo(parameter, value);
         opts.largescale = lower(value);

        case 'empty'
         % return the empty struct.

        otherwise
         error(sprintf('Unrecognized parameter: %s',parameter));
       end %<- end switch
     catch
       error(sprintf('An error accured: %s', lasterr));
     end
   end
end

if ~isempty(opts.precisionDelta) | ~isempty(opts.precisionFun),
   flag = 0;
   if isempty(opts.precisionDelta),
     flag = 1;
   elseif isempty(opts.precisionFun),
     flag = 1;
   end
   if flag,
     error('precisionFun and precisionDelta must both be set');
   end
end

%--------INNER FUNCTIONS---------------

function [value] = checkScalar(parameter, value)
if ~isreal(value) | ischar(value),
   error(sprintf('%s - is not a scalar',parameter ));
end

function [value] = checkInteger(parameter, value)
if ~isreal(value) | ischar(value),
   error(sprintf('%s - is not a scalar',parameter ));
elseif (value - floor(value)) > 0,
   error(sprintf('%s - is not an integer',parameter));
end

function [value] = checkYesNo(parameter, value)
if ~ischar(value),
   error(sprintf('%s - is not a string',parameter ));
elseif ~strcmp('off', lower(value)) & ~strcmp('on', lower(value)),
   error(sprintf('%s - "%s" is not a valid option', parameter, value));
end
```

D.4 Source code for CMINIMAX in Matlab

```
function [x, exitflag, info] = cminimax(fun,x,A,b,Aeq,beq, ...
                                        nonlcon,opts,fpar)
% Constrained minimax solver. Uses an exact penalty function.
% x = cminimax(fun,x0,A,b)
% INPUT:
%   nonlcon : Function that returns the nonlinear constraints.
%

if nargin < 1,
  % display default values.
  cslp4;
  return
end

if ~exist('fpar'), fpar=[]; end
if ~exist('A'), A=[];end
if ~exist('b'), b=[];end
if ~exist('Aeq'), Aeq=[];end
if ~exist('beq'), beq=[];end
if ~exist('nonlcon'), nonlcon=[];end

x = x(:);
sigma = 0.1;

% get unconstrained size of problem.
f = feval(fun,x,fpar); n = length(f);
unconstr_multipliers = zeros(1,n);

while max(unconstr_multipliers) < eps,
    intern = internOpts(fun,fpar,sigma,A,b,Aeq,beq,nonlcon);
    [x, info, perf] = cslp4(@penaltyFunction,intern,x);
    unconstr_multipliers = perf.multipliers(1:n);
    sigma = sigma + 0.1;
end

%----------INNER FUNCTIONS----------
function [f, J] = penaltyFunction(x,intern)

fun = intern.fun;
fpar = intern.fpar;
A = intern.A;
b = intern.b;
Aeq = intern.Aeq;
beq = intern.beq;
sigma = intern.sigma;
nonlcon = intern.nonlcon;

[fun_f, fun_J] = feval(fun, x, fpar);
[m,n] = size(fun_J);

f_container = [];
J_container = [];

% Equality constraints
if ~isempty(A) & ~isempty(b)
  b = A*x - b;
  p = length(b);
  c = repmat(b, 1, m); c = c';
  f_container = [f_container; repmat(fun_f, p, 1) + sigma*c(:)];
  A = repmat(A, 1, m)';
  A = reshape(A, n, m*p)';
  J_container = [J_container; repmat(fun_J, p,1) + sigma*A];
else
  if ~isempty(A) | ~isempty(b)
    error('Both A and b has to be given');
  end
end

% Equality constraints.
if ~isempty(Aeq) & ~isempty(beq)
  Aeq = [Aeq; -Aeq]; beq = [beq;-beq]; %<- Double the constraints.
  beq = Aeq*x - beq;
  p = length(beq);
  c = repmat(beq, 1, m); c = c';
  f_container = [f_container; repmat(fun_f, p, 1) + sigma*c(:)];
  Aeq = repmat(Aeq, 1, m)';
  Aeq = reshape(Aeq, n, m*p)';
  J_container = [J_container; repmat(fun_J, p,1) + sigma*Aeq];
else
  if ~isempty(Aeq) | ~isempty(beq)
    error('Both Aeq and beq has to be given');
  end
end

% Nonlinear constraints.
if isa(nonlcon, 'function_handle') | isa(nonlcon,'char'),
```

```
  [fun_c, fun_A] = feval(nonlcon, x);
  c = repmat(fun_c, 1, m); c = c';
  f_container = [f_container; repmat(fun_f, p, 1) + sigma*c(:)];
  A = repmat(fun_A, 1, m)';
  A = reshape(A, n, m*p)';
  J_container = [J_container; repmat(fun_J, p,1) + K*A];
else
  if ~isempty(nonlcon),
    error('nonlcon is not a function_handle nor a string');
  end
end

% Upper and lower bound.
% Not implemented yet.


% Add the containers with the function.
f = [fun_f; f_container];
J = [fun_J; J_container];

%-------------------------------------------------------------

function [sigma] = getSigma(PF, c)
% [sigma] = getSigma(PF, c)
% Calculate the sigma that triggers a shift to a
% new stationary point.
%
% INPUT:
%    PF : The generalized gradient of F.
%     C : The gradient of C. If more than one
%         constraint is active then just chose
%         one.
% OUTPUT:
%  sigma : The penalty factor.

% Not implemented yet.

%-------------------------------------------------------------

function [intern] = internOpts(fun,fpar,sigma,A,b,Aeq,beq,nonlcon)

% Create struct.
intern = struct( 'fun',[],'fpar',[],'sigma',[],'A',[],'b',[], ...
                 'Aeq',[],'beq',[],'nonlcon', []);

intern.fun = fun;
intern.fpar = fpar;
intern.sigma = sigma;
intern.A = A;
intern.b = b;
intern.Aeq = Aeq;
intern.beq = beq;
intern.nonlcon = nonlcon;
```

Bibliography

[Bar70] Y. Bard. Comparison of gradient methods for solution of nonlinear parameter estimation problems. *SIAM Journal of Numerical Analysis.*, (7):157–186, 1970.

[BD71] K. M. Brown and J. E. Dennis. A new algorithm for nonlinear least squares curve fitting. *Mathematical Software*, pages 391–396, 1971.

[CGT00] Andrew. R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *Trust-Region Methods*. MPS-SIAM Series on Optimization, 2000.

[Cla75] F.H. Clarke. Generalized gradients and applications. *Trans. Am. Math. Society*, 205:247–262, 1975.

[EAVD79] El-Attar, M. Vidyasagar, and S.R.K. Dutta. An algorithm for ℓ_1 approximation. *Siam J. Numer. anal*, 16(1):70–86, 1979.

[FJNT99] Poul Erik Frandsen, Kristian Jonasson, Hans Bruun Nielsen, and Ole Tingleff. *Unconstrained Optimization*. IMM, DTU, first edition, 1999. `http://www.imm.dtu.dk/courses/02611/uncon.pdf`.

[Fle00] R. Fletcher. *Practical Methods of Optimization*. Wiley, 2nd edition edition, 2000.

[GT82] A. Griewank and Ph. L. Toint. Partitioned variable metric updates for large structured optimization problems. *Nummer. Math.*, 39:119–137, 1982.

[GvL96] G. H. Golub and C. F. van Loan. *Matrix Computation*. The John Hopkins University Press, third edition, 1996.

[HL95] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. McGraw Hill, sixth edition, 1995.

[Hub81] Peter J. Huber. *Robust Statistics*. John Wiley, 1981.

[JM92] K. Jonasson and K. Madsen. Corrected sequential linear programming for sparce minimax optimization. Technical Report NI-92-06, Institute for Numerical Analysis, Technical University of Denmark, september 1992.

[JM94] K. Jónasson and K. Madsen. Corrected sequential linear programming for sparse minimax optimization. *BIT*, 34:372–387, 1994.

[KO68] J. S. Kowalik and M. R. Osborne. *Methods for Unconstrained Optimization Problems*. American Elsevier Publishing Co., 1968.

[Mad86] Kaj Madsen. *Minimization of Non-linear Approximation Functions*. Polyteknisk Forlag, first edition, 1986.

[Man65] O.L. Mangasarian. *Nonlinear Programming*. McGraw-Hill Book Company, 1965. Newer title available from SIAM Society for Industrial & Applied Mathematics ISBN 0898713412.

[Mat00] MathWorks. *Optimization Toolbox For Use with Matlab*. The MathWorks, fifth edition, 2000. `http://www.mathworks.com/access/helpdesk/help/pdf_`

doc/optim/optim_tb.pd\%f.

[Mer72] O.H. Merrild. *Applications and extensions of an algorithm that computes fixed points of certain upper semicontinuous point to set mappings*. PhD thesis, University of Michigan, Ann Arbor, USA, 1972.

[MN02] K. Madsen and H.B. Nielsen. Supplementary notes for 02611 optimization and data fitting. http://www.imm.dtu.dk/02611/SN.pdf, 2002.

[MNP94] K. Madsen, H.B. Nielsen, and M.C. Pinar. New characterizations of solutions to overdetermined systems of linear equations. *Operations Research Letters*, 3(16):159–166, 1994.

[MNS02] K. Madsen, H.B. Nielsen, and Jacob Søndergaard. Robust subroutines for non-linear optimization. Technical Report IMM-REP-2002-02, Institute for Numerical Analysis, Technical University of Denmark, 2002. http://www.imm.dtu.dk/~km/F-pak.html.

[MNT99] Kaj Madsen, Hans Bruun Nielsen, and Ole Tingleff. *Methods for Non-linear Least Squares Problems*. IMM, DTU, first edition, 1999. http://www.imm.dtu.dk/courses/02611/NLS.pdf.

[MNT01] K. Madsen, H.B. Nielsen, and O. Tingleff. *Optimization with constraints*. IMM, DTU, 2001. http://www.imm.dtu.dk/courses/02611/H40.pdf.

[Nie96] Hans Bruun Nielsen. *Nummerisk Lineær Algebra*. IMM, DTU, 1996.

[Nie99a] H. B. Nielsen. Damping parameter in marquardt's method. Technical Report IMM-REP-1999-05, IMM, DTU, 1999. http://www.imm.dtu.dk/~hbn/publ/TR9905.ps.z.

[Nie99b] H.B. Nielsen. *Algorithms for linear optimization*. IMM Department of Informatics and Mathematical Modelling, DTU, 2nd edition, 1999. http://www.imm.dtu.dk/~hbn/publ/ALO.ps.

[Ros60] H.H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *Comput. J*, 3:175–184, 1960.

[Wat79] Watson. The minimax solution of an overdetermined system of non-linear equations. *J. Inst. Math. Appl.*, 23:167–180, 1979.