On Curvature and Separability in Unconstrained Optimisation

Lennart Frimannslund

Thesis for the degree of Philosophiae Doctor (Ph.D.)
Department of Informatics
University of Bergen
Norway

March 2006
Acknowledgments

I would like to thank my principal advisor over the past years, professor Trond Steihaug, without whose blend of patience, impatience, competence and professionalism this thesis would never have been completed. Thank you! Thanks also to my second advisor, professor Hans Julius Skaug, for being patient and treating me more like a peer than a student.

On the personal side, thanks to my girlfriend, family and friends, for believing in me and supporting me, not least at the times when I didn’t deserve it.

Notation

In this thesis we use superscripts to denote the iteration a variable corresponds to. We use subscripts to denote variable names and indices. If a variable has more than one item of information which is to appear in the same position, we use parentheses. Some examples are given below:

\[ x^k \]  The variable \( x \), at iteration \( k \).
\[ x^* \]  The optimal value of \( x \).
\[ (C_Q)_{ij} \]  Element \( ij \) of the matrix \( C_Q \).
\[ (y^k)^T \]  The transpose of the variable \( y \) at iteration \( k \).
\[ e_1 \]  The first unit coordinate vector.
\[ (q_r)_i \]  Element \( i \) of the vector \( q_r \).

In addition, we sometimes use the notation \( x^{ij} \) in conjunction with a matrix element \( (C_Q)_{ij} \), which means \( x^{ij} \) is a vector, which has a connection with element \( ij \) of the matrix \( C_Q \).
Contents

1 Introduction 4

2 Generating Set Search 9

3 Using Curvature Information 17
   3.1 Using Curvature Information in GSS methods . . . . . . . . . . . 17
   3.2 Paper I — A GSS Method Using Curvature Information . . . . 20

4 Using Separability Information 25
   4.1 Separability and Sparsity . . . . . . . . . . . . . . . . . . . . . . 25
   4.2 Exploiting Separability in Direct Search Methods . . . . . . . . 27
   4.3 Paper II & IV — GSS Methods Exploiting Curvature and Sepa-
       rability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

5 Limited Memory Methods 32
   5.1 Quasi-Newton Methods . . . . . . . . . . . . . . . . . . . . . . 32
   5.2 Derivative-Based Limited Memory Methods . . . . . . . . . . . . 35
   5.3 Hessian Sparsity in Derivative-Based Methods . . . . . . . . . . 37
   5.4 Paper V — A Class of Methods Combining TN and L-BFGS . . 39

6 Application of GSS to Maximum Likelihood Estimation 41
   6.1 Maximum Likelihood Estimation . . . . . . . . . . . . . . . . . . 41
   6.2 Paper III — Estimating Mortality of Norwegian Spring Spawning
       Herring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

7 Concluding Remarks 45
   7.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
   7.2 Possible Further Research . . . . . . . . . . . . . . . . . . . . . 45
Chapter 1

Introduction

Optimisation is a diverse field, where the only common denominator of all its incarnations can be said to find the best choice in a situation where one has several possibilities to choose from. For instance, one might be adjusting the ratio of components to make a chemical process run as smoothly as possible, or the properties of physical object such as its shape, mass distribution and stiffness, one might be trying to make a production line run more efficiently, increase the revenue of a factory, or try to produce the largest number of clothes from a given amount of fabric or any other similar situation. To give an example, consider the problem of a farmer looking for a site to build a farm. The farm must have irrigation, with water supplied from a nearby reservoir, electricity from a local power station, and road access to a processing plant, where the products of the farm will be delivered. There is a cost associated with building a water pipeline, installing cables for electricity and paving a road to the processing plant. The farmer wants to build the farm in the site which corresponds to the lowest cost, so the optimisation problem can be formulated as

$$\min_{\text{Farm position}} \text{Pipeline cost} + \text{cable cost} + \text{road cost}. \quad (1.1)$$

As one learns from calculus, extreme values are found where the gradient vanishes. Apart from cases involving very simple objective functions one needs iterative solution methods. Many methods are implicitly based on Taylor’s theorem by constructing polynomial model functions. Newton’s method assumes a quadratic model function, and steps to the minimum of that function. A third-order tensor method assumes a cubic model function, and similarly steps to the minimum of that function. Since Taylor’s theorem tells us that the higher the degree of the polynomial, the better we can approximate an arbitrary function, one might expect that methods of this kind would become more and more powerful as their order increases. However, trying to build the highest-order method possible would present new problems along the way, for instance that:

1. The desired derivatives may be expensive, or difficult to compute.
2. The objective function might not be differentiable, or yield unhelpful derivatives.

3. The desired derivatives may be expensive to store.

The fact that derivatives may be expensive to compute is addressed by quasi-newton methods like the BFGS method (see e.g. [50] and the references therein), which make use of gradient information from past iterations to produce an approximation to the Hessian. Compared to Hessian matrices, gradients can be computed relatively cheaply using Automatic Differentiation (AD) techniques, see e.g. [36]. However, if gradients are not directly available, for example if the function is written in several programming languages, precluding the use of AD, or is for some other reason only available as a “black box”-procedure, one might have to turn to other alternatives.

If the objective function is the result of a simulation on a computer, then the resulting function might well not be smooth, although the same input will always give the same output except in the case of random effects. For example, determining the time of day it is fastest or slowest for a cargo train to get from one city to another, given ordinary traffic that uses the same network of rails, is a problem which can be cast as a continuous optimisation problem, since the input variable (start time) and output variable (travel time) are continuous. Since two trains cannot occupy the same piece of rail at the same time, there could be situations where the cargo train would have to wait for oncoming trains, or wait until a track is cleared for some other reason. This could result in the objective function resembling a staircase, like in Figure 1.1, which would make it a non-differentiable function in some parts of its domain and the gradient zero wherever it is defined, which would certainly be a challenging function for a gradient-based method. A different example — simulations might be used in weather forecasting. One could if one were to calibrate a model for weather forecasting based on historical weather data, encounter an objective function which would probably involve solving a differential equation. Differential equations are solved using iterative methods, such as for example Runge-Kutta methods. These methods inevitably involve some sort of discretisation, which can result in that even if there exists an underlying smooth function in the mathematical sense, the numerical function the optimisation method works with appears as non-smooth. This can also happen in the context of flow optimisation, e.g. [8]. To generalise this kind of functions, suppose we are dealing with a function that is very expensive to compute accurately, but which can be computed approximately at a lower cost. Such a function could, apart from differential equations for instance contain integrals. High-dimensional integrals can be expensive to compute, especially if they cover a broad region. One might solve the integral using a coarse discretisation, which would make the integral – and in turn the function – relatively cheap to compute, although one would then only obtain an approximation to the underlying function value. A third example, the objective function itself could contain an “inner” optimisation problem, so that evaluation of the objective function would mean solving an optimisation problem in the process. The inner problem could be solved to varying degrees of accuracy,
which would again influence computation time and accuracy of the final numerical objective function value. In chapter 6 we will encounter a function of this type, where the inner optimisation is part of an approximation to an integral.

If the function values are expensive to obtain accurately, then it is not hard to imagine that the same is the case for the derivatives. If the function when computed to limited accuracy is not smooth, it is not necessarily clear what an inaccurately computed derivative represents or if it contains useful information about the underlying smooth objective function at all. In an example in flow optimisation [9], a discretisation in the computed objective function causes the corresponding computed gradient to be inconsistent with the underlying mathematical function, and causes gradient-based optimisation methods to terminate at non-stationary points.

In all of the cases mentioned one can turn to derivative-free optimisation methods. These methods in their most basic form use only function values. This is a strength, since it makes these methods robust in the sense that they in practice can perform very well on functions which are not differentiable, even though for many of these methods there is often no theoretical guarantee for convergence on such functions. The fact that these methods only use function values is also a source of potential weakness, since these methods in their simplest form typically converge very slowly, unless the objective function by chance is particularly well suited to the method at hand.

When we talk about expensive functions, the case might also be that the objective function is so expensive to evaluate that one may only perform a limited number of evaluations, since evaluating the function more times is impractical, or even impossible. If this is the case one often works with a surrogate model function instead, which is in part based on the values of the actual objective

Figure 1.1: Cost function where the derivative is not always defined.
function. In surrogate modeling derivative-free methods which we will be concerned with can play a part (see e.g. [28, 6, 7]), although we will not investigate this kind of modeling directly.

That derivatives are expensive to store is an obstacle that limited memory methods attempt to deal with. The gradient of the objective function has \( n \) elements, which allows for quite large \( n \) on today’s computers, although concepts like small and large will of course always be relative. A Hessian matrix has \( n(n+1)/2 \) unique elements, and the quadratic growth of the storage requirements as \( n \) grows puts a limit on the size of variables it is practical to work with. Even more so in case of the third derivatives, which have \( n(n+1)(n+2)/6 \) unique elements. In this case one often tries to exploit sparsity if it is present, which can reduce the need for storage. Even if the derivatives are not sparse, if the function is separable then one it might still be possible to cut back on storage requirements, for example in the case of a Hessian matrix where all the diagonal elements are always equal to each other, and all the off-diagonal elements are equal to each other as well. In this case, even if the Hessian were of infinite size, it would only contain two unique elements and could be stored using only two floating point numbers. If one cannot cut storage requirements through separability, one may use a limited number of lower-order derivative values to approximate higher-order derivatives. Two important methods which do this are the limited memory BFGS (L-BFGS) method [54, 48] and the discrete Newton method [27, 56]. Both these methods use gradients to approximate Hessian information. The L-BFGS method is a variant of the BFGS method. Whereas the storage requirement for BFGS is \( O(n^2) \), the L-BFGS method uses only \( O(mn) \) space, where \( m \) is a variable which can be controlled by the user. The discrete truncated Newton uses the property of most iterative methods for solving linear equations that they do not need an explicit representation of the coefficient matrix, in this case the Hessian. Instead, an iterative method usually only needs the ability to compute the product of the Hessian with an arbitrary vector. Such a product one can approximate using two values of the gradient. Newton methods which use iterative equation solvers usually do not solve the Newton step equations to full precision at each iteration, and hence are called (discrete) truncated Newton methods.

What we wish to investigate in this thesis is how we can speed up methods that deal with the three points listed. To see how this can be done, let us put ourself in the place of an optimisation method for a moment. If one as a human being were to determine what is the best choice in a situation where one has several or infinite options to choose from, one would probably do some research before one starts experimenting, as well as memorising one’s experience during the testing phase. Optimisation methods are little different from humans in this respect, they can make use of information about the problem provided to them before they start evaluating the function. We will make use of the separability properties of the objective function, which are sometimes available before the optimisation process starts. Similarly, optimisation methods can remember what has happened in earlier iterations, and use for instance previous function values and gradients to make an informed decision about where to search for the
next solution. We intend to use curvature information to this end. Thus, we will try to improve existing methods and we will keep in mind that objective functions are not always smooth, even if the underlying mathematical representation might be. Specifically, we will deal with generating set search methods, which deal with potentially non-smooth functions and functions where the derivatives are difficult to obtain, and hence address points 1 and 2, and limited memory methods which address point 3. If the function is not smooth, we will use average curvature information to try to capture the curvature properties of the underlying function. For example, in the case of the function in Figure 1.1, this would mean capturing the curvature properties of a quadratic function. As for limited-memory methods, although the L-BFGS method approximates curvature over time, previous research has shown that its initial curvature approximation is very important when it comes to ensuring quick convergence. We will try to enhance this method by trying to make use of second derivative information, that is, given that we can compute the product of the Hessian with an arbitrary vector, we will try to make L-BFGS exploit this information in its initial curvature approximation.

The thesis is organised as follows. In chapter 2 we introduce unconstrained optimisation, briefly review derivative free methods, and introduce generating set search methods, as well as reiterating the types of functions we will be concerned with. In chapter 3 we show how a generating set search method can be made to reach the optimal solution faster, by making it detect and exploit average curvature information. In chapter 4 we introduce the concept of separability, and show how it is not necessarily related to differentiability. We extend the improved generating set search method of chapter 3 to take separability into account. In chapter 5 we introduce quasi-Newton methods and limited memory methods, particularly L-BFGS and the (discrete) truncated Newton method. We present a new class of hybrid methods which encompasses L-BFGS and truncated Newton. The new class can be viewed either as a class of L-BFGS methods enhanced by curvature information, or as a class of truncated Newton methods enhanced by using previously acquired gradient values. Chapter 6 contains an application of the generating set search method of chapter 3 to a concrete problem from statistics, and chapter 7 offers some concluding remarks and proposes some possible avenues of further research. The five papers themselves appear at the end of this thesis.
Chapter 2

Generating Set Search

In this thesis we will address problems of the form

$$\min_x f(x),$$  \hspace{1cm} (2.1)

where

$$f : \mathbb{R}^n \rightarrow \mathbb{R}.$$  

The function may or may not be differentiable, and may or may not be continuous. We assume that the function is not stochastic, that is, the same input always gives the same output. We will try to obtain local minima of the objective function $f$ from an arbitrary starting point, we will not try to predictably obtain global minima.

We can define a concrete version the farm problem (1.1) to obtain a problem of the form (2.1). Let the processing plant be at the origin, let the water reservoir be at position $x_w = (100, 0)$ and the power plant be at position $x_{pow} = (150, 50)$, where the units along the axes are kilometers. Let the cost of of cable to the power plant be $7000$ per kilometer, the cost of a pipeline from the water reservoir be $8000$ per kilometer, and the cost to build a road to the processing plant be $9000$ per kilometer. Let the position of the farm be $x_f$. Again, where is the cheapest site to build the farm? We can model this as

$$\min_{x_f} 9000 \cdot \|x_f\|_2 + 8000 \cdot \|x_f - x_w\|_2 + 7000 \cdot \|x_f - x_{pow}\|_2.$$  \hspace{1cm} (2.2)

This is a smooth optimisation problem, with well-defined derivatives except at the origin, $x_w$ and $x_{pow}$. In real life on the other hand, the costs involved can depend on things like the nature of the terrain, cost of subcontracting, available manpower, overtime payment — simply put any number of problems one can think of. In other words, although one would like the cost function for for example the water pipeline to look like the lowermost curve in Figure (2.1), it might well look like the staircase-curve in the middle, or even the uppermost noisy curve, in complicated cases. In all three curves of the figure we have an
underlying, simple function, but the two uppermost curves present potential problems for gradient-based methods.

If the function is challenging for gradient-based methods, there are many derivative-free method classes to choose from. One may for instance sample the function at particular points, and construct an interpolating model function. Then, one could minimise the interpolating function, and evaluate the objective function at the optimum of the model function, construct a new model function, and repeat the process. Polynomials, either linear or quadratic are often the interpolating functions of choice in this context. Using this method one might get an interpolating function which is a poor approximation to the objective function, especially far from the current best point. An extension to the above strategy is to consider the model function to be valid only within a specific radius of the current best point, and to search for the minimum of the model function within this region only. This is called a trust region approach. Some of these methods, including linear model functions are reviewed in [16, 59, 61]. Methods that use quadratic approximations include those in [18, 60, 62, 14], a recent method with both a linear and a quadratic variant is given in [51]. A recent extension towards separable function is given in [15].

A different approach entirely is that of simplex methods. One of the most widely cited and popular simplex methods is the method of Nelder and Mead [52]. The Nelder-Mead method in $n$ dimensions keeps a simplex of $n+1$ points, and changes one of these points per iteration. We will not give a full description of the method, but highlight one of its features, which is of interest to us. In Figure 2.2 we see what is called an outside contraction, in $\mathbb{R}^2$. The three points of the triangle with dashed sides are the points corresponding to, say, iteration $k$, and the three points of the triangle with solid sides correspond to iteration...
Figure 2.2: An outside contraction in the Nelder-Mead method.

$k + 1$. In other words, the uppermost point is substituted with the lowermost point in the figure. Similarly, the algorithm can also perform what is called a reflection and expansion step, which in the figure corresponds to the solid triangle belonging to iteration $k$ and the dashed triangle to iteration $k + 1$. In any event, what we wish to highlight is that the shape of the simplex is not constant, since the angles between the sides of the simplex can vary. This can cause the shape of the simplex to be for instance very long and narrow, which in turn can cause the method to stagnate. Torczon [68] demonstrated that the direction of progress of the Nelder-Mead method can become nearly orthogonal to the negative gradient for even the simplest functions. Torczon introduced a new method, the multi-directional search method, which just like the Nelder-Mead method keeps a simplex of $n + 1$ points in $n$ dimensions, but which does not allow the angles between its sides to change. The numerical results in [68] show the multi-directional search method to perform better than the Nelder-Mead method.

That the function may cause the points the Nelder-Mead method keeps to lie close to each other is a phenomenon which can also affect trust region interpolation methods in the form we have described them above. The reason for this is that we have not imposed any restrictions on the geometry of the points we include in our model, so for a given function, the equations that define the interpolating function might be singular, or near-singular, as pointed out in [17]. Interpolation methods can be made to enforce a certain geometry, which eliminates this particular problem.

As far as derivative-free methods are concerned, we wish to develop methods that enforce a certain geometry on its points and search directions, but do not wish to do this within the trust-region framework, that is, we do not want an explicit model function. It is then natural to turn to methods like the multi-directional search algorithm.

This method is in the class of what is called generating set search (GSS)
CHAPTER 2. GENERATING SET SEARCH

methods. Until the review article [42] coined this name (GSS) there seems to have been some overlapping use of terminology in this field. In the review article [73], both the Nelder-Mead method and the multi-directional search algorithm are called “direct search methods”, Torczon et al use the name “pattern search”, in e.g. [70, 71, 30, 47] to signify (a subset of) GSS methods, whereas Conn and Toint in [18] use the name “pattern search” for both the Nelder-Mead method and the multi-directional method. Exactly how to define these methods has not been an easy task either, leading to headings like “I know it when I see it: Toward a definition of direct search methods” [72] and indirect definitions like that a direct search method does only uses function values and does not “in its heart” develop an approximate gradient [73].

A framework encompassing all the GSS algorithms we will be concerned with was given in [42], and we will stick with the name and definitions of that review. GSS methods are in their simplest form very intuitive methods. As stated in the preface of [25], one such method was used when one first started to optimise functions on computers. Two issues however, that for many years made direct search methods an unattractive option for many users were as pointed out in for instance [67, 42], the lack of convergence theory, and the fact that the methods often converge slowly in practice. The multi-directional search algorithm was presented along with a proof of convergence [68, 69], and this convergence theory was generalised in [70], which also presented a unifying framework for a particular class of GSS algorithms. This class is called generalised pattern search (GPS) algorithms in [4], where the convergence theory of [44, 45, 70] was unified. GPS with derivative information was studied in [1], and GPS with inexact derivative information in [23].

Convergence theory for GSS methods we will call moving grid methods was presented in [21] and for what we will call sufficient decrease methods in [49]. Extensions toward constrained optimisation have been discussed in [29, 44, 45, 46], and an algorithm designed specifically for noisy problems appeared in [2].

The methods for which [70, 4] provides convergence theory are methods which restrict all the iterates to lie on a rational lattice, which essentially is a regular grid in $n$ dimensions. Mathematically, this can be stated as that given an initial iterate $x^0$, search directions $d_i, i = 1, \ldots, r$ and initial corresponding step lengths $\delta_i$, then any iterate on $x^k$ on a given lattice defined by $d$ and $\delta$, satisfies

$$x^k = x^0 + \sum_{i=1}^{r} \zeta_i \delta_i d_i,$$

(2.3)

where the coefficients $\zeta_i$ are all nonnegative integers. The lattice can be successively refined according to rules imposed by the convergence theory, for simplicity one can imagine the refinement as restricted to halving the step lengths $\delta$. Since the resulting lattice is rational, the convergence result becomes that

$$\lim_{k \to \infty} \inf \|\nabla f(x^k)\| = 0.$$

This result cannot always be strengthened, as discussed in [3].
CHAPTER 2. GENERATING SET SEARCH

Moving grid methods can be viewed as an extension to rational lattice methods. Moving grid methods allow the grid of points to be reconfigured once a grid minimum is found. A grid minimum is a point on a fixed grid which corresponds to a lower objective function value than all of its neighbouring points. This reconfiguration can be for instance a rotation, scaling, a shear transformation or a combination of some or all of these transformations. (For a definition of shear, see e.g. [32] section 5.2.)

Sufficient decrease methods do not impose restrictions on grids, as long as a new point is only accepted if it produces sufficient decrease, that is,

\[ f(x_{\text{new}}) \leq f(x_{\text{old}}) - \rho(\delta), \]

where

\[ \delta = \|x_{\text{new}} - x_{\text{old}}\|, \]

and

\[ \rho(\delta) = a(\delta), \text{ as } t \downarrow 0. \]

All of these results where summarised in [42], which also credits [74] and [43] for the convergence results. Central to all GSS methods is the presence of a generating set, or positive basis as it is also called. A generating set is a set of vectors \( v_i, i = 1, \ldots, r \) such that for each \( x \) in \( \mathbb{R}^n \), we have

\[ x = \sum_{i=1}^{r} c_i v_i, c_i \geq 0, \quad i = 1, \ldots, r. \]

This requires [26] that \( r \geq n + 1 \). A GSS method uses these directions as its search directions, or search basis. We will, for the rest of this thesis only consider methods where \( r = 2n \), where the search directions are the positive and negative of the columns of an orthogonal matrix \( Q \), unless otherwise stated. We will occasionally state results for methods which search along the positive and negative of the columns of an \( m \times n \) matrix \( P \).

A very simple example of a GSS method which we will call compass search, can be written as in Figure 2.3. From the code one can gather what is meant by an iterative method. The method takes an approximation to, or guess of the optimal solution \( x^k \), and produces a better approximation \( x^{k+1} \). Then the process repeats, and can be repeated until no better function value is found, or one is satisfied with the current approximation. There are several aspects of this method which can be decided upon by the user, for instance which directions \( G \) should contain, in which order the method should search along the directions, as well as when step lengths are increased and decreased. It is common to halve the step lengths if no search progress is made, either by halving all the step lengths if no progress is made along any direction, or treating the directions individually and halving the step lengths corresponding to directions with no progress. One can also increase the step lengths, for instance by doubling them, if for instance the method has stepped along a certain direction or certain directions many times in a row. There are a few technical requirements for the method to be
Given $f$, $k = 0$, $x^k$, search directions $\mathcal{G}$, step lengths $\delta_i$.

While $\max_i \delta_i > \text{threshold}$

Set $x_+ \leftarrow x^k$,

For each direction $d_i$ in $\mathcal{G}$, with associated step length $\delta_i$

If $f(x_+ + \delta_i d_i) < f(x_+)$

Set $x_+ \leftarrow x_+ + \delta_i d_i$.

end.

Set $x^{k+1} \leftarrow x_+$.

Update step lengths $\delta_i$

Set $k \leftarrow k + 1$.

end.

Figure 2.3: Pseudocode for Compass Search.

Figure 2.4: Progress of Compass Search.

convergent, most importantly that the set of search directions multiplied by all possible step lengths spans the entire space of interest.

An illustration of how the method searches in $\mathbb{R}^2$ when $\mathcal{G}$ consists of the positive and negative of the unit vectors is given in Figure 2.4. In the figure each point is marked by a node. The directions in $\mathcal{G}$ are searched in the order east, north, south, west. In addition, step lengths are unchanged as long as there is progress along one of the four directions. The search starts at the black node, and the points are evaluated in the order they are numbered. First, the method searches to the right/east, finds a better function value, and steps to the grey node. Then, it searches upward/north, but does not find a better value, which is marked by a white node. Then it searches to the south/down, but does not find a better value. Then it searches to the left/west, and tests the original black point again, but does not step since the starting point does not give a lower function value. In the form given in Figure 2.3 the method would
perform this evaluation twice, hence it is marked \( 0/4 \). Then the cycle repeats, the method tests in the order east, north, south, west. Note that the points marked \( 3/8 \) and \( 7/12 \) are also visited twice.

We can apply compass search to the farm problem (2.2). Let us take the point \( x_f = (50, 50) \) as the initial guess. We use the positive and negative of the unit coordinate vectors as the set \( \mathcal{G} \), the same step length \( \delta \) for all directions in \( \mathcal{G} \), and halve \( \delta \) if we cannot find reduction along any direction. Initially \( \delta = 10 \). The result of the first 25 iterations are given in Table 2.1. As one can see in the table,

<table>
<thead>
<tr>
<th>Iteration</th>
<th>( f )</th>
<th>( \delta )</th>
<th>(( x_f ))_1</th>
<th>(( x_f ))_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.90208e+06</td>
<td>1.00000e+01</td>
<td>5.00000e+01</td>
<td>5.00000e+01</td>
</tr>
<tr>
<td>1</td>
<td>1.85853e+06</td>
<td>1.00000e+01</td>
<td>4.00000e+01</td>
<td>5.00000e+01</td>
</tr>
<tr>
<td>2</td>
<td>1.83126e+06</td>
<td>1.00000e+01</td>
<td>3.00000e+01</td>
<td>5.00000e+01</td>
</tr>
<tr>
<td>3</td>
<td>1.82114e+06</td>
<td>5.00000e+00</td>
<td>2.00000e+01</td>
<td>4.00000e+01</td>
</tr>
<tr>
<td>4</td>
<td>1.82114e+06</td>
<td>2.50000e+00</td>
<td>2.00000e+01</td>
<td>4.00000e+01</td>
</tr>
<tr>
<td>5</td>
<td>1.82093e+06</td>
<td>2.50000e+00</td>
<td>2.25000e+01</td>
<td>4.00000e+01</td>
</tr>
<tr>
<td>6</td>
<td>1.82080e+06</td>
<td>2.50000e+00</td>
<td>2.50000e+01</td>
<td>4.25000e+01</td>
</tr>
<tr>
<td>7</td>
<td>1.82080e+06</td>
<td>1.25000e+00</td>
<td>2.50000e+01</td>
<td>4.25000e+01</td>
</tr>
<tr>
<td>8</td>
<td>1.82074e+06</td>
<td>1.25000e+00</td>
<td>2.12500e+01</td>
<td>4.12500e+01</td>
</tr>
<tr>
<td>9</td>
<td>1.82074e+06</td>
<td>6.25000e-01</td>
<td>2.12500e+01</td>
<td>4.12500e+01</td>
</tr>
<tr>
<td>10</td>
<td>1.82071e+06</td>
<td>6.25000e-01</td>
<td>2.18750e+01</td>
<td>4.12500e+01</td>
</tr>
<tr>
<td>11</td>
<td>1.82071e+06</td>
<td>3.12500e-01</td>
<td>2.18750e+01</td>
<td>4.12500e+01</td>
</tr>
<tr>
<td>12</td>
<td>1.82071e+06</td>
<td>3.12500e-01</td>
<td>2.18750e+01</td>
<td>4.15625e+01</td>
</tr>
<tr>
<td>13</td>
<td>1.82071e+06</td>
<td>1.56250e-01</td>
<td>2.18750e+01</td>
<td>4.15625e+01</td>
</tr>
<tr>
<td>14</td>
<td>1.82071e+06</td>
<td>1.56250e-01</td>
<td>2.18750e+01</td>
<td>4.14062e+01</td>
</tr>
<tr>
<td>15</td>
<td>1.82071e+06</td>
<td>7.81250e-02</td>
<td>2.18750e+01</td>
<td>4.14062e+01</td>
</tr>
<tr>
<td>16</td>
<td>1.82071e+06</td>
<td>7.81250e-02</td>
<td>2.17969e+01</td>
<td>4.14844e+01</td>
</tr>
<tr>
<td>17</td>
<td>1.82071e+06</td>
<td>7.81250e-02</td>
<td>2.17969e+01</td>
<td>4.14062e+01</td>
</tr>
<tr>
<td>18</td>
<td>1.82071e+06</td>
<td>3.90625e-02</td>
<td>2.17969e+01</td>
<td>4.14062e+01</td>
</tr>
<tr>
<td>19</td>
<td>1.82071e+06</td>
<td>3.90625e-02</td>
<td>2.17969e+01</td>
<td>4.14453e+01</td>
</tr>
<tr>
<td>20</td>
<td>1.82071e+06</td>
<td>1.95312e-02</td>
<td>2.17969e+01</td>
<td>4.14453e+01</td>
</tr>
<tr>
<td>21</td>
<td>1.82071e+06</td>
<td>1.95312e-02</td>
<td>2.18164e+01</td>
<td>4.14258e+01</td>
</tr>
<tr>
<td>22</td>
<td>1.82071e+06</td>
<td>9.76562e-03</td>
<td>2.18164e+01</td>
<td>4.14258e+01</td>
</tr>
<tr>
<td>23</td>
<td>1.82071e+06</td>
<td>9.76562e-03</td>
<td>2.18164e+01</td>
<td>4.14355e+01</td>
</tr>
<tr>
<td>24</td>
<td>1.82071e+06</td>
<td>4.88281e-03</td>
<td>2.18164e+01</td>
<td>4.14355e+01</td>
</tr>
<tr>
<td>25</td>
<td>1.82071e+06</td>
<td>4.88281e-03</td>
<td>2.18164e+01</td>
<td>4.14355e+01</td>
</tr>
</tbody>
</table>

Table 2.1: First 25 compass search iterations on problem (2.2).

the sequence of iterates tends towards the point \( x_f = (21.81 \ldots, 41.43 \ldots) \), and \( \delta \) declines as the method closes in on this point. The corresponding placement of the farm is visualised in Figure 2.5. Informally, one can say that GSS methods converge linearly. If we call the white nodes in Figure 2.4 \( x_{\text{unsuccessful}}^l \) where \( l \) is an index which is updated as the white nodes are discovered, then technically,
Figure 2.5: Positions of the processing plant, water reservoir and electricity plant and optimal placement of farm.

and under reasonable assumptions on the objective function and the method, the sequence

$$\|x^f_{\text{unsuccessful}} - x_*\|$$

converges $r$-linearly to zero. (See e.g. [55], chapter 3.3 for more on convergence rates.)
Chapter 3

Using Curvature Information

3.1 Using Curvature Information in GSS methods

Compass search as presented in the previous chapter is in a sense memory-less, as it makes no use of previously sampled function values. As we will show, these values can be very useful. The search progress of compass search in \( \mathbb{R}^2 \) we have already illustrated to some extent in Figure 2.4, but in Figure 3.1 one can see one of its major drawbacks, what we will call zig-zagging. Since compass search (and all other methods we consider) require that the function value at a new point is lower than at the current point, it is often forced to take very small steps. The reason for this is that the angle between the negative of the gradient and the available search directions can be relatively large (about 45° in the figure), and the effect of this is that the method may have to perform a large number of function evaluations to cover the distance from the initial solution to the optimal solution. In the worst case, for instance a gradient of all ones, the smallest angle in \( \mathbb{R}^2 \) between any of the positive and negative unit vectors is 45°. This number depends on the dimension \( n \). Zig-zagging can be illustrated with a simple example. Let

\[
Q = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}, \quad A = \begin{bmatrix} 10^{-4} \\ 100 \end{bmatrix},
\]

and

\[
f(x) = x^T Q^T A Q x.
\]

As one can see the optimal solution is \( x_\star = 0 \), and \( f(x_\star) = 0 \) as well. Let us apply compass search to the problem. If we take \( x^0 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}^T \) as our initial approximation, we have

\[
\|x^0 - x_\star\| = 2.8284, \text{ and } f(x^0) = 8 \cdot 10^{-4}.
\]
Let us take 0.1 to be the initial step lengths, and search along the positive and negative of the coordinate axes. Let us halve the step length \( \delta_i \) if the search fails along both the direction \( d_i \) and \(-d_i\). If the search succeeds along a direction, we immediately test if we can step again along the same direction. If this succeeds, we accept the second step and double the step length. This is how the subroutine \textit{exploratory moves} in paper I searches. Over the first 14 iterations, all that happens is that the method halves the step lengths, before starting to progress when the step lengths are of the order \( 10^{-6} \). After 30 iterations, we have,

\[
\|x^{30} - x_*\| = 2.8280, \text{ and } f(x^{30}) = 7.9976 \cdot 10^{-4},
\]

which is very little progress, considering the number of iterations that have been performed. If we instead of searching along the coordinate directions had searched along the positive and negative of the column vectors of \( Q \), then we would have obtained

\[
\|x^{30} - x_*\| = 3.0510 \cdot 10^{-6}, \text{ and } f(x^{30}) = 9.3088 \cdot 10^{-16},
\]

which is very close to the optimal solution.

As the example shows, the choice of search basis can to a large extent influence how much the method is able to progress per iteration. It would therefore be desirable to have a method which can automatically choose its search directions. This idea was behind two early direct search methods, the method of Hooke and Jeeves [40], which implicitly searches along the direction of average progress in addition to searching along the coordinate directions, and Rosenbrock’s method [64], which is basically a variant of compass search, combined with rotation of all its search directions. Specifically, Rosenbrock’s method aligns the principal search direction to the direction of average progress. Figure 3.2 illustrates what this means. In the left of the figure we see an example of search progress. One can see that the general direction of progress is down and
Figure 3.2: Illustration of the logic behind Rosenbrock’s method. On the left, search before rotation of search directions, on the right, search after directions are rotated. The step lengths and time of rotation in this figure differ from Rosenbrock’s implementation.

to the right. Therefore, a suitable choice of search directions, hopefully allowing for longer steps would be the directions in the right of the figure, one search direction being the direction of average progress, and the rest orthogonal to it. At suitable intervals Rosenbrock’s method creates an $n \times n$ matrix where the first column contains the change in the variable since the last basis rotation. The second column contains the change in the variable along all the search directions except the first search direction, the third column the change along all the search directions except the first and the second search direction, an so on. The new $n$ search directions are obtained by applying the Gram-Schmidt process to the columns of this matrix. This way the method adapts its search directions to the function, while maintaining the geometry of mutually orthogonal search directions.

**From one to $n$ Function-based Directions** Out of the Rosenbrock method’s $n$ unique (apart from sign) orthogonal search directions, only one is based on information obtained from the function itself, namely the direction of average progress. The remaining $n - 1$ directions are based on a decomposition of the information stored in the first direction. We would like a method where all its $n$ unique directions are based on information from the function, while still preserving a healthy geometry, that is, we want the directions to be orthogonal.

Often, optimisation methods are designed to work well on quadratic functions. We will adopt this methodology, and implicitly use a quadratic model function. The question becomes what can be considered good orthogonal directions for a quadratic function.

We would like directions which allow us to take long steps along as many of the $n$ unique directions as possible, which would then reduce zig-zagging, since zig-zagging is caused by the method only being able to take short steps along its search directions. We know from the theory behind the conjugate gradient method [39] for solving linear equations with symmetric positive definite coefficient matrices that conjugate directions with respect to the coefficient matrix
are directions which do not lead to zig-zagging on quadratic functions, since if one minimises the corresponding quadratic function along such a direction, the direction need not be considered again. Recall that two directions \( p \) and \( q \) are by definition conjugate with respect to the matrix \( A \) if and only if

\[ p^T A q = 0. \]

Since minimising a quadratic function, say,

\[ f(x) = c - b^T x + \frac{1}{2} x^T A x, \]

with a symmetric positive definite Hessian \( A \) is the same as solving the linear equation system

\[ Ax = b, \]

then conjugate directions should make good search directions, as studied in [10]. Indeed, derivative-free methods that search along conjugate directions have been proposed [20, 58]. However, conjugate directions are not necessarily orthogonal, so if the underlying implicit quadratic model is inaccurate, we could in theory end up with a worse zig-zagging problem than what we sought to prevent. Thus, ideally we would like conjugate, orthogonal search vectors based on curvature information from the function. Fortunately, such directions are easily identifiable, namely the eigenvectors of the Hessian of the underlying model function. We make use of these directions in the method of Paper I.

3.2 Paper I — A GSS Method Using Curvature Information

In paper I we present a moving grid GSS method which makes use of curvature information. This is done by observing that, as can be seen in Figure 2.4, when performing compass search along orthogonal directions, the points which the algorithm evaluates will lie in constellations which can be used by the formula

\[ q_i^T \nabla^2 f(\bar{x}) q_j = \frac{f(x + h q_i + k q_j) - f(x + h q_i) - f(x + k q_j) + f(x)}{hk}, \hspace{1cm} (3.1) \]

where the equation holds for two times continuously differentiable functions. To see this for \( n > 2 \), consider Figure 3.3. The situation depicted in the figure is that after two search directions have been considered. Independent of whether or not the search along either direction is successful, the constellation of points given by the three nodes as drawn in the figure, or rotated or mirrored, will always be found. If, for example both steps are successful, then the constellation occurs when stepping from node 1 to 2 to 3, which is what is shown in the figure. As before the search starts at the black point, and grey nodes signify points which are accepted. If only the first step is successful, then the constellation occurs if the method for instance again starts at point 1, steps to point 2, and
CHAPTER 3. USING CURVATURE INFORMATION

considers point 3 but does not step. (The colours in the figure are from now on inconsistent with previous usage.) Similarly, with failure along first direction but not along the second direction (start at point 2, try point 1 but fail, try point 3 and step), as well as failure along both directions (start at point 2, try point 1, fail, try point 3, fail), the constellation occurs. If one, given such a constellation computes the function value at point 4 explicitly, one gets four points in a rectangle, which is exactly the set of points needed by formula (3.1). The algorithm shuffles the order of the search directions in order to obtain curvature information along all pairs of directions. An outline of the algorithm is given below:

- Perform compass search along the directions in the set $G$, initially consisting of the columns of $I$ and $-I$.
- Use formula (3.1) to compute a matrix $C_Q$, as the search progresses. If the function is two times continuously differentiable $C_Q$ will have, as entry $(i, j)$,
$$\left(C_Q\right)_{ij} = q_i^T \nabla^2 f(x^{ij}) q_j,$$
for various $x^{ij}$.
- Once $C_Q$ is complete, calculate the Hessian approximation
$$\nabla^2 f(x) \approx C = QCQ^T.$$
- Compute the eigenvectors of $C$ and replace $G$ with the positive and negative of these eigenvectors.

A Small Example We illustrate a slightly simplified version of the algorithm on a small example. There are a few technical requirements on the algorithm imposed by the convergence theory which are addressed in the paper. The example is illustrated in figure 3.4. Let:
$$f(x) = f(x_1, x_2) = 5x_1^2 + 2x_1x_2 + 10x_2^2.$$
Let the initial variable be
\[ x^0 = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \]
and let the search basis be
\[ Q = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}. \]
Let the initial step lengths be
\[ \delta = \begin{bmatrix} 0.1 \\ 3 \end{bmatrix}. \]
We order the directions
\[ \begin{bmatrix} q_1 \\ q_2 \\ -q_1 \\ -q_2 \end{bmatrix}. \]
Now, the fact that \( q_1 \) and \( q_2 \) immediately follow each other as the first two search directions enables us to compute \((C_Q)_{12}\). (The algorithm of paper I actually makes use of search directions in groups of three, but we do not touch upon this behaviour in this example.) At our initial point \( x = x^0 \) we have
\[ f(x) = 49. \]
We search along \( q_1 \) and find
\[ f(x + \delta_1 q_1) = 47.0851, \]
a function value reduction. We then immediately test if we can double the step length along this direction and test
\[ f(x + 2\delta_1 q_1) = 45.3002, \]
an additional reduction. So now we set
\[ x \leftarrow x + 2\delta_1 q_1, \]
and compute, by (3.1) (with \( q_i = q_j \) and \( \delta_i = \delta_j \))
\[ (C_Q)_{11} \leftarrow \frac{45.3002 - 2 \cdot 47.0851 + 49}{0.12} = 13. \]
Finally we set
\[ \delta_1 \leftarrow 2\delta_1 = 0.2. \]
Now we are ready to search along \( q_2 \). We test
\[ f(x + \delta_2 q_2) = 237.5941. \]
The search along $q_2$ does not provide us with a lower function value, but we leave $\delta_2$ unchanged for the time being. Since we were able to step along $q_1$ but not along $q_2$, we are in the situation depicted in Figure 3.3, with the black point in Figure 3.4 corresponding to point 1 in Figure 3.3, and points 2, 3 and 4 being the same. We need to compute the function value at the point marked by a cross, namely

$$f(x - \delta_1 q_1 + \delta_2 q_2) = 244.2939.$$ 

Now we can compute, by (3.1)

$$(C_Q)_{12} = \frac{237.5941 - 244.2939 - 45.3002 + 49}{0.2 \cdot 3} = -5.$$ 

Note that we have used the updated $\delta_1$ in the denominator. The next direction scheduled for search is $-q_1$. Now, since

$$x - \delta_1 q_1 = x^0,$$

the search direction $-q_1$ is skipped. The next direction for search is $-q_2$. We get

$$f(x - \delta_2 q_2) = 6.0063.$$ 

Immediately, again, we test

$$f(x - 2\delta_2 q_2) = 119.7123.$$ 

Stepping twice was not successful, so we set

$$x \leftarrow x - \delta_2 q_2.$$ 

Now we can compute

$$(C_Q)_{22} = \frac{119.7123 - 2 \cdot 6.0063 + 45.3002}{3^2} = 17.$$ 

All elements in $C_Q$ are now determined and we can solve for our new search basis. We compute

$$C \leftarrow Q \begin{bmatrix} 13 & -5 \\ -5 & 17 \end{bmatrix} Q^T = \begin{bmatrix} 10 & 2 \\ 2 & 20 \end{bmatrix},$$

the exact Hessian, since $f$ is quadratic. All that remains is to eigenvalue-factorise $C$ and use its eigenvectors as the new search basis.

**Numerical Results** Our numerical experiments show that this new method performs much better than compass search in terms of the number of function evaluations required to reach the optimal solution, except when the coordinate directions themselves are near-conjugate directions. This is the case for both smooth and noisy problems.
Figure 3.4: Visualisation of the algorithm’s progress in the example. (The axes are unequally scaled.) The black point is the starting point. Grey points are taken, the white points are not. The cross marks the extra point evaluated for the computation of \((C_Q)_{12}\). The numbers indicate the order in which the points are evaluated.
Chapter 4

Using Separability Information

4.1 Separability and Sparsity

First derivative and curvature information are good examples of information one can gather about the function as the search progresses. It is also sometimes possible to make statements about the function before it is evaluated, by looking at its representation. The property of partial separability falls into this category. Given the sets

\[ \chi_1, \chi_2, \ldots, \chi_n, \quad \chi_i \subseteq \{1, 2, \ldots, n\}, \text{ for all } i. \]

Consider the function

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}, \quad f = \sum_{i=1}^{n} f_i, \quad f_i : \mathbb{R}^{\chi_i} \rightarrow \mathbb{R}, \quad (4.1) \]

where each function depends only on a few of the components of \( x \), specified by the \( \chi \)-sets. That is, \( f_i \) depends on the components of \( x \) corresponding to the indices in the set \( \chi_i \). An example in \( \mathbb{R}^3 \) is

\[ f(x) = \sin(x_1) + \cos(x_2x_3), \quad (4.2) \]

which can be written

\[ f = f_1 + f_2, \]

with

\[ \chi_1 = \{1\}, \quad \chi_2 = \{2, 3\}. \]

Thus, \( f \) is the sum of \( \nu \) element functions, each of which ideally depends on only a few variables, and thus has an invariant subspace. Such a function is called a partially separable function. Partial separability was introduced by Griewank and Toint, see e.g. [37], and [38] for the incorporation of partial separability into quasi-Newton methods.
The Covariation Graph  Separability is usually associated with differentiability. We present a new definition which is applicable to non-differentiable functions as well. Define, for any function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ the covariation graph,

$$G(V, E), |V| = n,$$  \hspace{1cm} (4.3)

that is, a graph with $n$ nodes. Let $e_i$ be the $i$th unit coordinate vector. If, for some $x$, we have

$$f(x + e_i + e_j) - f(x + e_i) - f(x + e_j) + f(x) \neq 0,$$

then let there be an edge between from node $i$ to node $j$ in the graph. If there is an edge from node $i$ to $j$ there must also be an edge from node $j$ to node $i$.

In other words, the graph $G$ can be viewed as undirected. In the case of the function (4.2), we get the graph in Figure 4.1.

**Theorem 1** Let a continuous function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, and its covariation graph be given. If the graph is not complete then the function is separable.

**Proof.** If the graph is not complete then there exist $i$ and $j$ such that

$$f(x + e_i + e_j) - f(x + e_i) - f(x + e_j) + f(x) = 0,$$

for all $x$. Assume without loss of generality that $n = 2$, and that $i = 1$ and $j = 2$. Then we have, for all $h$ and $k$,

$$f(x_1 + h, x_2 + k) - f(x_1 + h, x_2) - f(x_1, x_2 + k) + f(x_1, x_2) = 0.$$

Now, let $x_1$ and $x_2$ be identical to zero, and let $h$ and $k$ be the independent variables. Then we get

$$f(h, k) - f(h, 0) - f(0, k) + f(0, 0) = 0,$$

which can be written

$$f(h, k) = f(h, 0) + f(0, k) - f(0, 0).$$

Now we can define, for instance

$$f_1(h) = f(h, 0) + f(0, 0),$$

and

$$f_2(k) = f(0, k),$$
and we have that $f$ can be written on the form (4.1), and thus is separable. □

From the covariation graph we can make a graph adjacency matrix, which is symmetric since the graph $G$ is undirected. The matrix for (4.2) becomes

$$
\begin{bmatrix}
\times & \times & \times \\
\times & \times & \\
\times & & \\
\end{bmatrix}.
$$

(4.4)

If the function is two times differentiable, then the structure of the adjacency matrix arising from the covariation graph is the structure of the Hessian. In addition, the result of Theorem 1 can then be stated in terms of partial derivatives, that is, that if any of the cross-derivatives $\frac{\partial f}{\partial x_i \partial x_j}$ is identical to zero, then the function is partially separable [37].

In paper IV we extend our definition of separability so that it is not dependent on the vectors $e_1, \ldots, e_n$, instead depending on $n$ orthogonal or $n$ general linearly independent vectors.

### 4.2 Exploiting Separability in Direct Search Methods

It is possible to exploit information about separability in direct search methods. A method which does this is the method of Price and Toint, which assumes that each individual element function is available, and takes advantage of this.

**The method of Price and Toint** In [63], a direct search method which makes use of partial separability of the objective function is presented. Specifically, it does this by noting that for separable functions where the individual element functions are known, evaluating the function at points of the form

$$
f \pm he_i, \ i = 1, \ldots, n,
$$

is relatively inexpensive, since one may not need to evaluate the entire function, only the element functions which are affected by the change in the variable $x$. In addition, one sometimes obtains function values at specific points at no extra cost. For instance, if the function is totally separable, that is, for instance

$$
f(x) = \sum_{i=1}^{n} f_i(x_i),
$$

then evaluating the entire function at two points $x$ and $y$ gives the values at the points

$$
z, \text{ such that } z_i = x_i \text{ or } z_i = y_i, \ i = 1, \ldots, n,
$$

provided that the element functions are known explicitly. This approach enables the algorithm to solve problems with a large number of variables (as many as 5000 in the paper) at little cost.
In papers II and IV, we employ separability without assuming that the individual element functions are available. We extend the algorithm of paper I to take advantage of knowledge about the structure of the adjacency matrix of the covariation graph, so that curvature information can be gathered more efficiently.

**4.3 Paper II & IV — GSS Methods Exploiting Curvature and Separability**

**Computing Curvature Information Element by Element** In paper II we present an extension to the moving grid method of paper I, by imposing a structure on the matrix $C$ used in the first paper.

In paper IV we extend the algorithm of further and answer theoretical questions which were left open in paper II. We impose a sufficient decrease condition on the exploratory moves making the method of paper IV a (provably convergent) sufficient decrease method.

Recall that the algorithm of paper I computed curvature in a rotated coordinate system and assembled it in a matrix $C_Q$, such that

$$\nabla^2 f \approx C = QC_Q Q^T.$$  

Equation (4.5) can be written in a different form, using Kronecker products. Kronecker products can be defined by

$$AXB = C \iff (B^T \otimes A)\text{vec}(X) = \text{vec}(C),$$  

Where $\text{vec} : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^{n^2}$ stacks the entries of the matrix $X$, column by column in a vector. The matrix $(B^T \otimes A)$ is $n^2 \times n^2$, with entries

$$(B^T \otimes A) = \begin{bmatrix} B_{11}A & \cdots & B_{n1}A \\ \vdots & \ddots & \vdots \\ B_{1n}A & \cdots & B_{nn}A \end{bmatrix}. \quad (4.7)$$

If we apply this to the last part of the equation (4.5) we can write

$$C = QC_Q Q^T \iff (Q^T \otimes Q^T)\text{vec}(C) = \text{vec}(C_Q). \quad (4.8)$$

We impose the sparsity structure of the adjacency matrix of the covariation graph of $f$ on $C$. Now if $Q = I$ the $n^2 \times n^2$ equation system

$$(Q^T \otimes Q^T)\text{vec}(C) = \text{vec}(C_Q) \quad (4.9)$$

can be reduced in size based on knowledge of which elements of $C$ are zero, as well as the fact that $C$ is symmetric, so that we only need to compute the upper or lower triangular elements of $C_Q$.

We can apply the same idea even if $Q \neq I$, as long as we are a little careful. It turns out not to be obvious which elements of $C_Q$ are to be computed in
this case. The resulting coefficient matrix might be singular if elements are not properly chosen. In paper II we employ a simple heuristic for choosing these elements, in paper IV we extend this heuristic to a method which we prove always returns a nonsingular coefficient matrix. In addition, the solution itself will depend on which elements are chosen if the right-hand side $\text{vec}(C_Q)$ contains truncation error. To clarify, consider the following two examples. Let the Hessian be diagonal and constant, say,

$$\nabla^2 f = \begin{bmatrix} 1 & 2 \\ & 3 \end{bmatrix}.$$ 

Let

$$Q = \begin{bmatrix} \sqrt{3} & \sqrt{2} & \sqrt{2} \\ \sqrt{2} & \sqrt{2} & \sqrt{2} \\ \sqrt{3} & \sqrt{2} & \sqrt{2} \end{bmatrix}.$$ 

$(Q^T \otimes Q^T)$ is a $9 \times 9$ matrix, but since we require all off-diagonal elements of $C$ to be zero, we can cut all but three of its columns, giving us the an overdetermined system which can be written

$$(Q^T \otimes Q^T)P_c \begin{bmatrix} C_{11} \\ C_{22} \\ C_{33} \end{bmatrix} = \text{vec}(C_Q).$$

(4.10)

Define

$$\text{vec}(C) = \begin{bmatrix} C_{11} \\ C_{22} \\ C_{33} \end{bmatrix},$$

Then $P_c$ is the $(9 \times 3)$ matrix such that

$$\text{vec}(C) = P_c \text{vec}(C).$$

To turn (4.10) into an equation system with a square coefficient matrix we can either solve for the least squares solution, or we can cut rows from the coefficient matrix, which corresponds to computing only selected elements in $C_Q$. We would like to compute no more elements of $C_Q$ than strictly needed when the adjacency matrix of the covariation graph is sparse, so we choose the latter option. Now, if we choose to evaluate the three off-diagonal unique elements in $C_Q$, namely

$$C_Q : \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \end{bmatrix},$$

then the resulting coefficient matrix $A$, which can be written

$$A = P_c (Q^T \otimes Q^T)P_c,$$
will be singular. This can be seen if one tries to compute
\[ A \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = P_r \text{vec}(Q^T IQ) = 0, \]
since \( P_r \) cuts all the elements corresponding to the diagonal of \( C_Q \). If one instead tries to compute for instance the elements
\[ C_Q : \begin{bmatrix} \times & \times \\ \times & \times \end{bmatrix}, \]
then \( A \) will be nonsingular. Furthermore if the function is two times continuously differentiable, and the elements of \( C_Q \) contain truncation errors or equivalently are correspond to derivatives computed at different \( x \), that is
\[
(C_Q)_{ij} = q_i^T \nabla^2 f(x^{ij}) q_j, \\
(C_Q)_{rs} = q_r^T \nabla^2 f(x^{rs}) q_s,
\]
where
\[ x^{ij} \neq x^{rs}, \]
then the solution \( C \) will also depend on which elements of \( C_Q \) we compute. This is also true if the function is not differentiable. To see this, assume that \( \nabla^2 f \) exists and is diagonal but not constant, and that
\[ Q = \begin{bmatrix} q_1 & q_2 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \]
Let \( e \) be a constant, and let
\[
(C_Q)_{11} = q_1^T \begin{bmatrix} 1 - e & 0 \\ 0 & 1 - e \end{bmatrix} q_1, \quad (C_Q)_{21} = q_2^T \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} q_1, \\
(C_Q)_{12} = (C_Q)_{21}, \quad (C_Q)_{22} = q_2^T \begin{bmatrix} 1 + e & 0 \\ 0 & 1 + e \end{bmatrix} q_2.
\]
Then, if we compute \( C \) from \( (C_Q)_{11} \) and \( (C_Q)_{12} \) we get,
\[ C = \begin{bmatrix} 1 - e & 0 \\ 0 & 1 - e \end{bmatrix}, \]
whereas if we compute \( C \) from \( (C_Q)_{12} \) and \( (C_Q)_{22} \) we get
\[ C = \begin{bmatrix} 1 + e & 0 \\ 0 & 1 + e \end{bmatrix}, \]
and least squares solution from all elements of \( C_Q \) yields \( C = I \). Either way, the difference between the actual Hessian and \( C \), roughly speaking, grows with
the size of \( h \) and \( k \) used to compute \( C_Q \) elements in formula (3.1), and depends on the condition number of the matrix \( A \). This is elaborated upon in the two papers.

Numerical results show that the algorithms of the two papers both outperform the method of paper I, especially as \( n \) grows, and the adjacency matrix of the covariation graph has many identical zeros.
Chapter 5

Limited Memory Methods

5.1 Quasi-Newton Methods

Quasi-Newton methods are powerful gradient-based methods which use gradient information from past iterations to approximate the Hessian. For a comprehensive treatment of the subject, see e.g. [50] and the references therein. The original idea of using acquired gradient information to approximate the Hessian was introduced in 1959 by Davidon [24]. (Available as [25], since the original paper was never published until the first issue of SIAM Journal on Optimization in 1991.) The work of Davidon was continued by Fletcher and Powell [31], resulting in what is called the DFP method. The DFP method maintains a positive definite approximation to either the Hessian, its inverse, or a factorisation of either of the two. At iteration $k$, let us call the approximation to the Hessian $B^k$, and the method sets the point $x^{k+1}$ by the formula

$$x^{k+1} = x^k - \alpha^k (B^k)^{-1} \nabla f(x^k),$$

where $\alpha^k$ is usually taken to satisfy e.g. the strong Wolfe conditions, which are

$$f(x^k + \alpha^k p^k) \leq f(x^k) + c_1 \alpha^k \nabla f(x^k)^T p^k$$

and

$$|\nabla(f + \alpha^k p^k)^T p^k| \leq c_2 |\nabla f(x^k)p^k|,$$

where

$$0 < c_1 < c_2 < 1,$$

and in practice typically $c_1 = 10^{-4}$ and $c_2 = 0.9$ for the methods we will be concerned with (see e.g. [55], section 3.1). The matrix $B^k$ is then updated by the formula

$$B^{k+1} = (I - \gamma^k y^k (s^k)^T) B^k (I - \gamma^k s^k (y^k)^T) + \gamma^k y^k (y^k)^T,$$

where

$$s^k = x^{k+1} - x^k, y^k = \nabla f(x^{k+1}) - \nabla f(x^k), \gamma^k = \frac{1}{(y^k)^T s^k}.$$
One arrives at the formula (5.3) by solving the problem
\[ B^{k+1} = \arg\min_B \|B - B^k\|, \]  
where \( B \) is required to be symmetric, and in addition satisfy the secant equation
\[ B^{k+1}s^k = y^k. \]  
The choice of norm in (5.4) plays a role in the sense that different norms result in different update formulas, but this is beyond the scope of this discussion. The most important properties as far as we are concerned is that the method generates descent directions. For this to be the case \( B^k \) needs to be positive definite. The reason for this is that only if \( B^k \) is positive definite, the angle between the negative of the gradient and the search direction \( -B^k \nabla f(x^k) \) is provably in the range \((-90^\circ, 90^\circ)\). The cosine of this angle, say, \( \eta \) is given as
\[ \cos(\eta) = \frac{\nabla f(x^k)^T B^k \nabla f(x^k)}{\|\nabla f(x^k)\| \|B^k \nabla f(x^k)\|}. \]  
Since the matrix \( B^k \) is positive definite, and hence that \( x^T B^k x > 0, \forall x \in \mathbb{R}^n \),
the numerator of the fraction in (5.6) is always strictly positive, and hence the direction the angle between \( -\nabla f(x^k) \) and \( -B^k \nabla f(x^k) \) is in the range \((-90^\circ, 90^\circ)\) as required. Furthermore, as long as \( B^k \) is positive definite and \( \gamma^k > 0 \), \( B^{k+1} \) is also positive definite.

The most popular and efficient quasi-Newton method in use today is the BFGS method [5]. One can derive this method by instead of solving (5.4), minimising with respect to the inverse of \( B \), which we call \( H \), and solve
\[ H^{k+1} = \arg\min_H \|H - H^k\|, \]  
where \( H \) is required to be symmetric, as well as to satisfy the inverse of the secant equation (5.5), namely
\[ H^{k+1}y^k = s^k. \]  
The formulas for \( x^{k+1} \) and \( H^{k+1} \) in the BFGS method are
\[ x^{k+1} = x^k - \alpha^k H^k \nabla f(x^k), \]
and
\[ H^{k+1} = (I - \rho^k s^k (y^k)^T) H^k (I - \rho^k y^k (s^k)^T) + \rho^k s^k (s^k)^T, \]  
where as before
\[ s^k = x^{k+1} - x^k, \]
\[ y^k = \nabla f(x^{k+1}) - \nabla f(x^k), \]
Given $f$, $k = 0$, $x^0$, $H^0$, $\nabla f(x^0)$,

While $\|\nabla f(x^k)\| > \text{tolerance},$

Set $x^{k+1} \leftarrow x^k - \alpha^k H^k \nabla f(x^k)$, for some $\alpha^k$ satisfying the (strong) Wolfe conditions.

Compute $\nabla f(x^{k+1})$.

Compute $H^{k+1}$ by (5.9).

Set $k \leftarrow k + 1$.

end.

Figure 5.1: Pseudocode for the BFGS algorithm.

and in addition

$$\rho^k = \frac{1}{(y^k)^T s^k}.$$  

$H^{k+1}$ is positive definite as long as $H^k$ is positive definite, and $\rho^k > 0$. It can be shown that if $\alpha^k$ is required to conform to the Wolfe conditions, then $\rho^k$ is positive. The Wolfe conditions are a slightly less restrictive version of the strong Wolfe conditions, or vice versa. Keeping an approximation to the inverse of the Hessian rather than the Hessian itself gives a cheaper algorithm in terms of operation count, since one does not have to solve an equation system at every iteration. As mentioned, it is also possible to store either $B$ or $H$ in factorised form, for instance the Cholesky factorisation of $B$, or a conjugate factorisation of $H$, as is done in [19]. As is tested and discussed in [11] among others, there is little difference whether one chooses a factorised or non-factorised implementation as long as gradients are available to machine precision. The authors claim in [11] that when gradients are only available to limited precision (e.g. are approximated with finite difference formulas), then factorised implementations perform better, in the sense that they converge more often.

Pseudo code for the BFGS method is given in Figure 5.1. We now turn our attention to how the initial approximation to the Hessian $B^0$ or its inverse $H^0$ should be chosen. The most obvious choice, if one knows little about the function is $H^0 = I$. It has been suggested to let $H^0$ be a diagonal matrix whose entries are the inverses of the diagonal entries of the Hessian. These are, as noted in [65] expensive to compute, and if the resulting matrix is not positive definite, it cannot be used unaltered, since it might not generate descent directions. It is suggested in [65] to compute $x^1$ using $H^0 = I$, and scale $H^0$ before $H^1$ is computed, by the formula

$$\hat{H}^0 = \frac{(y^1)^T s^1}{(y^1)^T y^1},$$  

and then forming $H^1$ from $\hat{H}^0$ instead of from $H^0$. The numerical results of [65] show that except for small $n$, computing $H^1$ from $\hat{H}^0$ gives a much more
effective algorithm in terms of iterations than computing $H^1$ from $H^0 = I$. If we apply BFGS (updating $H^1$ from $H^0$) to the farm problem (2.2), we get the results in Table 5.1. BFGS does not converge as rapidly as Newton’s method, but is much faster than for GSS, or e.g. steepest descent. BFGS converges superlinearly, that is,

$$\lim_{k \to \infty} \frac{\|x^{k+1} - x_\star\|}{\|x^k - x_\star\|} = 0.$$  

Since quasi-Newton algorithms perform only one gradient evaluation per iteration disregarding gradient computations in the line search, the same as steepest descent, they are a good example of how one can use previously obtained information effectively.

### 5.2 Derivative-Based Limited Memory Methods

When $n$ is large, the memory required to store a full Hessian or approximations to it, namely $O(n^2)$ may be more than is practical on a given computer system. They aim of limited-memory methods is be able to solve the step equation using only $O(n)$ memory. Two such methods are limited memory BFGS [48, 54, 12] and discrete Newton [56, 27].

Limited memory BFGS is a variant of BFGS which instead of maintaining an approximation $H^k$ to the inverse of the Hessian based on the $k$ vector pairs $y^0, \ldots, y^k$ and $s^0, \ldots, s^k$, 

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$f$</th>
<th>$|\nabla f(x_f)|$</th>
<th>$(x_f)_1$</th>
<th>$(x_f)_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.90208e+06</td>
<td>5.07036e+03</td>
<td>5.00000e+01</td>
<td>5.00000e+01</td>
</tr>
<tr>
<td>1</td>
<td>1.83968e+06</td>
<td>3.40752e+03</td>
<td>1.07749e+01</td>
<td>4.44757e+01</td>
</tr>
<tr>
<td>2</td>
<td>1.82294e+06</td>
<td>9.94100e+02</td>
<td>2.60807e+01</td>
<td>4.35143e+01</td>
</tr>
<tr>
<td>3</td>
<td>1.82083e+06</td>
<td>1.77030e+02</td>
<td>2.25124e+01</td>
<td>4.27033e+01</td>
</tr>
<tr>
<td>4</td>
<td>1.82072e+06</td>
<td>6.05756e+01</td>
<td>2.18219e+01</td>
<td>4.19566e+01</td>
</tr>
<tr>
<td>5</td>
<td>1.82071e+06</td>
<td>2.05783e+01</td>
<td>2.17742e+01</td>
<td>4.15489e+01</td>
</tr>
<tr>
<td>6</td>
<td>1.82071e+06</td>
<td>1.28433e+00</td>
<td>2.18062e+01</td>
<td>4.14305e+01</td>
</tr>
<tr>
<td>7</td>
<td>1.82071e+06</td>
<td>8.37229e-02</td>
<td>2.18109e+01</td>
<td>4.14313e+01</td>
</tr>
<tr>
<td>8</td>
<td>1.82071e+06</td>
<td>5.27479e-04</td>
<td>2.18112e+01</td>
<td>4.14316e+01</td>
</tr>
<tr>
<td>9</td>
<td>1.82071e+06</td>
<td>7.85562e-06</td>
<td>2.18112e+01</td>
<td>4.14316e+01</td>
</tr>
<tr>
<td>10</td>
<td>1.82071e+06</td>
<td>6.87501e-06</td>
<td>2.18112e+01</td>
<td>4.14316e+01</td>
</tr>
<tr>
<td>11</td>
<td>1.82071e+06</td>
<td>2.73346e-11</td>
<td>2.18112e+01</td>
<td>4.14316e+01</td>
</tr>
<tr>
<td>12</td>
<td>1.82071e+06</td>
<td>1.25056e-12</td>
<td>2.18112e+01</td>
<td>4.14316e+01</td>
</tr>
<tr>
<td>13</td>
<td>1.82071e+06</td>
<td>1.50071e-12</td>
<td>2.18112e+01</td>
<td>4.14316e+01</td>
</tr>
<tr>
<td>14</td>
<td>1.82071e+06</td>
<td>9.11269e-13</td>
<td>2.18112e+01</td>
<td>4.14316e+01</td>
</tr>
</tbody>
</table>

Table 5.1: 14 BFGS iterations on problem (2.2).
keeps only the $m$ most recent pairs, where $m$ can be chosen by the user. This requires little storage if $m$ is not too large (usually less than about 30), since the product of $H^k$ with an arbitrary vector $v$ can be computed without constructing $H^k$ itself, using the procedure in Figure 5.2. Between the two for-loops we have

Given $p$, $H^k_0$, $s^{k-m}$, $s^{k-1}$, $y^{k-m}$, $y^{k-1}$,

$q \leftarrow v$,

For $i = k - 1$ to $k - m$,

$\alpha_i \leftarrow \rho^i (s^i)^T q$,

$q \leftarrow q - \alpha^i y^i$,

end.

$r = H^k_0 q$,

For $i = k - m$ to $k - 1$,

$\beta \leftarrow \rho^i (y^i)^T r$,

$r \leftarrow r + s^i (\alpha_i - \beta)$,

end.

Figure 5.2: Two-loop procedure to compute $r = H^k v$. $\alpha$ is a vector of length $m$ used for storage.

The assignment

$r = H^k_0 q$.

The matrix $H^k_0$ plays the same role as the matrix $H^0$ in the first line of the regular BFGS method in Figure 5.1, in the sense that it serves as the initial approximation to the inverse of the Hessian, which the gradient $(s^k, y^k)$ pairs subsequently modify. In the regular BFGS method $H^0$ is only used as the approximation of the inverse of the Hessian at the starting point $x^0$. In L-BFGS, since we only keep the $m$ most recent difference pairs, $H^k_0$ at iteration $k$ serves as an approximation to the inverse of the Hessian at $x^j$, $j = \max \{1, k - m\}$, that is, at iteration $k$, $k > m$ the method assumes that

$$H^k_0 \approx \left[ \nabla^2 f(x^{k-m}) \right]^{-1}.$$  

This leads to the question of how to choose $H^k_0$. Liu and Nocedal [48] discuss several choices for $H^k_0$, and find that the dynamic scaling of the identity matrix

$$H^k_0 = \frac{(y^{k-1})^T s^{k-1}}{(y^{k-1})^T y^{k-1}} I,$$  

(5.11)
leads to a more effective algorithm than for instance $H_0^k = I$, $k = 1, \ldots$ or

$$
H_0^k = \frac{(y^k)^T s^k}{(y^k)^T y^k} I, \quad k = 1, \ldots,
$$

as is done in the regular BFGS method. Different rules for $H_0^k$ are also discussed in e.g. [34], where the best choice tested for $H_0^k$ is a diagonal matrix based on a full Hessian approximation $B$.

Discrete Newton methods are variants of Newton’s method that use an iterative method for solving the step equation, that is

$$
\nabla^2 f(x^k) p^k = -\nabla f(x^k).
$$

Since iterative solvers such as the conjugate gradient method need only the product of the coefficient matrix with an arbitrary vector, not the matrix itself, the approximation

$$
\nabla^2 f(x)v \approx \frac{\nabla f(x + \epsilon v) - \nabla f(x)}{\epsilon},
$$

(5.13)

can be used to this end. In addition, although not related to memory, one does not have to solve the step equation to full accuracy at each iteration. A useful rule is to solve (5.12), at iteration $k$, to accuracy

$$
\frac{||\nabla^2 f(x^k)p^k + \nabla f(x^k)||}{||\nabla f(x^k)||} \leq \min \left\{ \frac{1}{k}, \frac{\|\nabla f(x^k)\|}{\epsilon} \right\},
$$

(5.14)

where $p^k$ is an approximate solution to (5.12) [27]. This is called a discrete truncated Newton method, or a truncated Newton method if the product $\nabla^2 f v$ is computed from Hessian information rather than from (5.13).

### 5.3 Hessian Sparsity in Derivative-Based Methods

Sparsity can be used effectively to evaluate Hessian matrices at little cost (see e.g. [22, 13, 53]). For instance, assume that the Hessian of the objective function is tridiagonal, and that it is known only as the product of the Hessian with an arbitrary vector, $\nabla^2 f \cdot v$, either by a finite difference formula or through automatic differentiation. Then, an $n \times n$ Hessian can be directly determined using only three evaluations. To see how this can be the case, consider the tridiagonal matrix structure

$$
\nabla^2 f = \begin{bmatrix}
  h_{11} & h_{12} & & & \\
  h_{21} & h_{22} & h_{23} & & \\
  & \ddots & \ddots & \ddots & \\
  & & h_{n-1,n-2} & h_{n-1,n-1} & h_{n-1,n} \\
  & & & h_{n,n-1} & h_{nn}
\end{bmatrix}.
$$

(5.15)
Assume without loss of generality that \( n \) is divisible by 3. If we now choose to compute the three products

\[
\nabla^2 f \cdot s_1, \; \nabla^2 f \cdot s_2, \; \nabla^2 f \cdot s_3,
\]

where

\[

\begin{align*}
  s_1 &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ \vdots \\ 1 \\ 0 \\ 1 \end{bmatrix}, \\
  s_2 &= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ \vdots \\ 0 \\ 1 \\ 0 \end{bmatrix}, \\
  s_3 &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ \vdots \\ 0 \\ 1 \end{bmatrix},
\end{align*}
\]

then the resulting vectors will be

\[

\nabla^2 f \cdot s_1 = \begin{bmatrix} h_{11} \\ h_{21} \\ h_{34} \\ h_{44} \\ h_{54} \\ h_{67} \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}, \\
\nabla^2 f \cdot s_2 = \begin{bmatrix} h_{12} \\ h_{22} \\ h_{32} \\ h_{45} \\ h_{55} \\ \vdots \\ \vdots \end{bmatrix}, \\
\nabla^2 f \cdot s_3 = \begin{bmatrix} h_{23} \\ h_{33} \\ h_{43} \\ h_{56} \\ h_{66} \\ \vdots \end{bmatrix},
\]

and we can read the values of the sought Hessian elements directly. The vectors \( s_1, s_2, s_3 \) are often called seed vectors. Since the Hessian is symmetric and only has \( 2n - 1 \) unknowns in this case and each Hessian vector product produces \( n \) equations we can determine a tridiagonal Hessian from only two such products, but we then have to solve a system of equations to find the explicit values of the elements. Assuming, again without loss of generality, that \( n \) is even, then a pair of seed vectors which accomplishes this is

\[

\begin{align*}
  s_1 &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}, \\
  s_2 &= \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.
\end{align*}
\]

Note that many choices of seed vectors would help us determine the Hessian, as long as the resulting equation system

\[

\nabla^2 f \cdot \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = W,
\]

is solved.
has a unique solution. Thus, if one knows in advance that the Hessian of the objective function is sparse, one can compute the entire Hessian cheaply, and hence optimise the function with a Newton-type method with less operations than if one does not make use of information about sparsity.

5.4 Paper V — A Class of Methods Combining TN and L-BFGS

In this paper we present a new class of hybrid limited memory methods. The class can be described based on the L-BFGS method. Instead of performing the assignment

\[ r = H_k^0 q, \tag{5.20} \]

between the two for-loops in the procedure to compute \( H_k^v \), we solve the equation system

\[ \nabla^2 f(x^k) r = q, \tag{5.21} \]

inexactly, using an iterative equation solver, and subject to some forcing sequence. Apart from this assignment and that we will allow \( m \) to vary, the methods in the class are equal to L-BFGS. This way, if \( m = 0 \) throughout, the method reduces to a truncated Newton algorithm, and if \( m \) is constant and the iterative equation solver applied to (5.21) returns (5.20), then the method becomes L-BFGS.

We test a member of the class with \( m = 3 \) and a simple forcing sequence for (5.21). We simulate the cost for different situations, whether or not derivatives are available through AD, and whether or not we are able to exploit the techniques of section 5.3. We find our preliminary numerical results very promising, and for the concrete method tested we find that it preforms well compared with L-BFGS with \( m = 3 \) and truncated Newton with forcing sequence (5.14), and that it generally performs well if one of its parents methods does. A nice example is given in Figure 5.3, where the method starts as effectively as L-BFGS, and when L-BFGS stagnates it continues progressing towards the optimal solution at about the rate of truncated Newton. We conclude that the class of methods is a promising one, and that more research should be conducted to identify forcing sequences and values of \( m \) which can give rapid convergence with little memory.
Figure 5.3: Plot of a hybrid method, TN and L-BFGS on the Penalty I function from the CUTEr collection [35]. See paper V for a detailed description of the plot.
Chapter 6

Application of GSS to Maximum Likelihood Estimation

6.1 Maximum Likelihood Estimation

Consider the situation where we want to determine some quantity found in nature. This may be, for instance the height of men aged 25. We assume, correctly or incorrectly, that this quantity has a normal distribution, with mean \( \mu \) and variance \( \sigma^2 \). The probability density in the normal distribution is given as

\[
\frac{1}{\sqrt{2\pi}\sigma} \exp\left( -\frac{1}{2} \frac{(x - \mu)^2}{\sigma^2} \right). \tag{6.1}
\]

We conduct multiple observations with different results. Let us say we observe ten men, with observations as in Table 6.1. The table consists of values drawn from a pseudo-random normal distribution with \( \mu_{\text{real}} = 180 \) and \( \sigma_{\text{real}} = 10 \). What are the most likely values of the quantities \( \mu \) and \( \sigma \), given the observations? This is what maximum likelihood estimation addresses. Associated with the distribution function (6.1) and the recorded observations is a log-likelihood function, in this case given as

\[
l(\mu, \sigma) = -10 \log(\sqrt{2\pi}\sigma) - \frac{\sum_{i=1}^{10} (x_i - \mu)^2}{2\sigma^2}, \tag{6.2}
\]

We can now use optimisation methods to maximise this function, to obtain the estimates

\[
\mu = 182.31 \text{ and } \sigma = 8.38.
\]

It should be noted that the log-likelihood function (6.2) can be maximised analytically, but for the more complicated model we shall discuss below analytical results are not available. Even though the data set we generated comes from
Table 6.1: Observed heights in centimeters of ten men.

<table>
<thead>
<tr>
<th>Observed heights</th>
</tr>
</thead>
<tbody>
<tr>
<td>178.13</td>
</tr>
<tr>
<td>187.25</td>
</tr>
<tr>
<td>174.11</td>
</tr>
<tr>
<td>201.83</td>
</tr>
<tr>
<td>178.63</td>
</tr>
<tr>
<td>181.13</td>
</tr>
<tr>
<td>190.66</td>
</tr>
<tr>
<td>180.59</td>
</tr>
<tr>
<td>179.04</td>
</tr>
<tr>
<td>171.67</td>
</tr>
</tbody>
</table>

a normal distribution (or a distribution very close to one) we do not recover the true values for $\mu$ and $\sigma$. Maximum likelihood estimates are unbiased in the limit, which means that if we had an infinite number of samples from the true distribution, we would recover the values $\mu_{\text{real}}$ and $\sigma_{\text{real}}$. Since we have a quite small data set, we experience bias. See for instance [57] for more on likelihood estimation.

6.2 Paper III — Estimating Mortality of Norwegian Spring Spawning Herring

In this paper we try to determine the amount of data required to make statements about variations in the mortality of Norwegian spring spawning herring. By data sets we mean acoustical observations of the herring stock like those presented in [41]. This we do via a model which can be applied to other species as well. A cohort is the group of herring born a particular year. Let $N_{jt}$ be the size of cohort $j$ the year $t$, with $t = 0$ being the year it is born in our model. Let $C_{jt}$ be the number of herring from cohort $j$ caught in year $t$. (The age of herring can be determined by looking at their shells, which is similar to determining the age of a tree by looking at its growth rings.) In addition, a portion of each cohort dies of causes not related to catch, so that we can express the size of a cohort recursively by the expression

$$N_{jt} = (N_{jt-1} - C_{jt-1})e^{-(M + \epsilon t_{t-1})}.$$  \hspace{1cm} (6.3)

The portion of a cohort which does not survive to the next year due to other causes than catch is controlled by $M + \epsilon t_{t-1}$, which we call the mortality parameter for year $t - 1$. We assume that the individual $\epsilon_t$, $t = 0, \ldots, n - 1$, are normally distributed, with mean 0 and variance $\tau^2$. Our goal is to estimate $\tau$. 
CHAPTER 6. APPLICATION OF GSS TO MAXIMUM LIKELIHOOD ESTIMATION

To obtain estimates of $\tau$ we construct artificial data sets, the same way we did for the example of heights, and test for which sizes of the data set the maximum likelihood estimate exists, and its level of accuracy. The likelihood function we are interested in can be written as $l(\theta, \epsilon)$, where $\theta$ and $\epsilon$ both are vectors of variables. $\epsilon$ is the vector containing $\epsilon_t$, $t = 0, \ldots, n - 1$ from (6.3), and $\theta$ is a vector containing $\tau$ as well as some additional variables not of primary interest to us, such as $M$ and $N_{j,0}$, $j = 1, \ldots, A$, from (6.3), $A$ being the number of cohorts under consideration. We are interested in the marginal likelihood function with respect to the components of $\theta$, so we are interested in maximising the function

$$l(\theta) = \log \left[ \int \exp \left( l(\theta, \epsilon) \right) d\epsilon \right].$$  \hfill (6.4)

Since $\epsilon$ in our experiments consists of 20 variables, solving the integral (6.4) is very expensive, so instead we maximise an approximation. The approximation to the integral (6.4) we use is called a Laplace approximation [66], and has the form

$$l^*(\theta) = -\frac{1}{2} \log \det \left( -\nabla^2 l(\theta, \bar{\epsilon}) \right) + l(\theta, \bar{\epsilon}),$$  \hfill (6.5)

where we by

$$\nabla^2 l(\theta, \bar{\epsilon}),$$

mean the Hessian of $l$ with respect to the elements of $\epsilon$, given $\theta$ and evaluated at $\bar{\epsilon}$, det is the determinant and

$$\bar{\epsilon} = \arg \max_{\epsilon} l(\theta, \epsilon).$$  \hfill (6.6)

That is, $\bar{\epsilon}$ is the value of $\epsilon$ that maximises $l$ for a given value of $\theta$. This is a two-level optimisation problem, since evaluating the function $l^*$ requires the optimisation of the function $l$. $l^*$ is therefore expensive to evaluate, and its derivatives are difficult to obtain. It is possible to compute $\bar{\epsilon}$ inexactly to reduce the cost of evaluating (6.5). This introduces numerical noise, and optimising $l^*$ is therefore a problem well suited to GSS methods. In Table 6.2 we show the results of performing 25 iterations on (6.5) with our GSS method of paper I, with the slight modification that it enforces sufficient decrease rather than simple decrease as in the paper. 25 iterations are usually sufficient to obtain an estimate of $\tau$ with a reasonable degree of precision. As a starting value we use the “true” values from which our data sets are generated. Let $\nabla l$ denote the gradient of $l(\theta, \epsilon)$ with respect to $\epsilon$. In the heading of Table 6.2 the convergence criterion used for obtaining $\bar{\epsilon}$ is listed. As one can see in the table, the level of accuracy used for obtaining $\bar{\epsilon}$ can be very loose and still only affect the estimated value of $\tau$ to little extent. In addition, a loose tolerance translates to less computation time. In the implementation behind the results using GSS, Newton’s method was used for maximising $l(\theta, \epsilon)$. It is likely that the relative gains in time can be made bigger by for instance employing truncated Newton instead.

In paper III we were able to work with a proprietary commercially available package for nonlinear statistical models, AD Model builder [33] (ADMB).
CHAPTER 6. APPLICATION OF GSS TO MAXIMUM LIKELIHOOD ESTIMATION

<table>
<thead>
<tr>
<th>Problem</th>
<th>GSS, $|\nabla l| \leq 10^{-6}$</th>
<th>GSS, $|\nabla l| \leq 1$</th>
<th>ADMB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\tau$</td>
<td>time</td>
<td>$\tau$</td>
</tr>
<tr>
<td>1</td>
<td>1.50266e-01</td>
<td>18.0s</td>
<td>1.50137e-01</td>
</tr>
<tr>
<td>2</td>
<td>1.39902e-01</td>
<td>17.8s</td>
<td>1.39901e-01</td>
</tr>
<tr>
<td>3</td>
<td>2.31647e-01</td>
<td>14.4s</td>
<td>2.31647e-01</td>
</tr>
<tr>
<td>4</td>
<td>2.41657e-01</td>
<td>17.5s</td>
<td>2.41657e-01</td>
</tr>
<tr>
<td>5</td>
<td>2.09133e-01</td>
<td>17.1s</td>
<td>2.09133e-01</td>
</tr>
<tr>
<td>6</td>
<td>1.83468e-01</td>
<td>17.8s</td>
<td>1.83466e-01</td>
</tr>
<tr>
<td>7</td>
<td>1.80755e-01</td>
<td>21.1s</td>
<td>1.80755e-01</td>
</tr>
<tr>
<td>8</td>
<td>1.93294e-01</td>
<td>21.8s</td>
<td>1.93294e-01</td>
</tr>
</tbody>
</table>

Table 6.2: Estimates of $\tau$ on eight randomly generated data sets using our GSS method with two tolerances for the inner optimisation (6.6), and the time used to obtain the estimate. The estimate in the last column is from AD Model Builder, and should be considered the most accurate for each problem.

ADMB is able, through a combination of AD and hand-coded derivatives to obtain the gradient of the Laplace approximation (6.5), and hence optimise $l^*$ effectively with a quasi-Newton method. Nevertheless, we contend that the problem (6.5) is a useful and realistic example of the applicability of our method to a numerically noisy problem.

**Statistical Findings**  As for the results pertaining to the likelihood estimation itself, we find that it is difficult to obtain an accurate estimate of $\tau$ if the true value of $\tau$ is low, say, 0.05. If the true value is somewhat larger, e.g. 0.2, we have a better chance of getting a good estimate.
Chapter 7

Concluding Remarks

7.1 Summary

In this thesis we have showed how we can take a very basic and arguably slow GSS algorithm, compass search, and speed it up using average curvature information accumulated over a region. We have shown that the necessary curvature information can be obtained effectively for partially separable functions, in turn leading to a more effective method. In addition, separability is not intrinsically linked to differentiability, which makes it a useful concept for functions which are not differentiable as well.

Numerical testing has indicated that updating the search basis of GSS algorithms using the eigenvectors of a matrix based on curvature information reduces zig-zagging, sometimes significantly.

We have applied one of our methods to a problem from maximum likelihood estimation, showing that the method is applicable to a difficult class of numerically noisy problems.

We have defined a new class of limited memory methods for smooth unconstrained optimisation, and shown that a particular instance of the class performs well compared to its parent methods.

7.2 Possible Further Research

There are several issues we have not been able to address, which could be studied in the future. An obvious avenue of further research is developing effective forcing sequences and schemes for varying $m$ for the class of methods presented in paper V. For GSS methods, some possibilities are outlined below.

Enhancing the Exploratory Moves  Our GSS algorithms base themselves on a particular method for their exploratory moves, listed in Paper I. This
subroutine searches in $2n$ directions at each iteration. In theory this may lead to points being evaluated more than once, and if there is a clear trend as to where in $n$-space the method progresses, as many as half of these evaluations if not more may easily be unsuccessful. It should be possible to incorporate a heuristic which tries to minimise the number of unsuccessful evaluations, while still collecting curvature information.

**Expanding the Role of the Quadratic Model Function** We have so far only used an implicit quadratic model function, since the curvature matrix $C$ we work with can be used as a basis for a family of quadratic functions. We have not tried to exploit a model function further, for instance by using interpolation to predict suitable values for step lengths, or where in $n$-space to search next. This could possibly speed our methods.

**Treating Element Functions individually** In our work on separable functions, we have yet to study the case where the individual element functions are available to us. If this were the case, we could build up (average) curvature information about each of the element functions, and then assemble this information in $C$ prior to the eigenvalue-factorisation.

**Computing Curvature Information by Approximate Gradients** So far, our work on GSS methods has revolved around computing curvature information element by element. It is also possible to apply techniques like those of section 5.3, that is, computing (approximate) Hessian-vector products and exploiting sparsity to obtain the full Hessian at little cost. Hessian-vector products can be obtained with a formula such as the one used by the discrete truncated Newton method. Gradients can be approximated by the formula

$$
(Q^T \nabla f(x))_i \approx \frac{f(x + \epsilon q_i) - f(x)}{\epsilon}.
$$

The constellation of points needed to compute the entire gradient in the coordinate system defined by the columns of $Q$ can be obtained if a compass search iteration tries many points before accepting a new point. A simple example is given in Figure 7.1. In the figure the algorithm tries three different points before stepping to the fourth point evaluated. The function values of the unsuccessful points can then be used for gradient computation. In this case we have four points around the starting point so that we can compute a central-difference approximation to the derivative. If the function is two times differentiable, then given two such gradient approximations one can either approximate the product of the Hessian with a vector, or create a quasi-Newton-like approximation to the entire Hessian matrix.

This matrix we could then, for instance, eigenvalue-factorise as we do in our current methods, and use the eigenvectors as the new search directions.
Figure 7.1: Possible outcome after an iteration of compass search.


