New York University

Master's Thesis

# A Large-Scale Constrained Optimizer for NonSmooth Functions

*Author:*
Wilmer Henao

*Supervisor:*
Michael L. Overton

*A thesis submitted in fulfilment of the requirements*
*for the degree of Master of Science in Scientific Computing*

*in the*

Courant Institute of Mathematical Sciences
Department of Mathematics

March 2014

# Declaration of Authorship

I, Wilmer Henao, declare that this thesis titled, 'A Large-Scale Constrained Optimizer for NonSmooth Functions' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

NEW YORK UNIVERSITY

# *Abstract*

Faculty Name
Department of Mathematics

Master of Science in Scientific Computing

**A Large-Scale Constrained Optimizer for NonSmooth Functions**

by Wilmer Henao

The Thesis Abstract is written here ...

# *Acknowledgements*

I would like to than my advisor Michael Overton for all the hours of hard work and for all the great recommendations and changes that suggested for the creation of this thesis.

I also would like to thank Allan Kaku and Anders Skaaja for earlier contributions to other versions of the code. . . .

# Contents

# List of Figures

# List of Tables

# Abbreviations

**LAH** **L**ist **A**bbreviations **H**ere

# Physical Constants

Speed of Light $c = 2.997\ 924\ 58 \times 10^8\ \mathrm{ms}^{-\mathrm{S}}$ (exact)

# Symbols

| | | |
|---|---|---|
| $a$ | distance | m |
| $P$ | power | W ($Js^{-1}$) |
| $\omega$ | angular frequency | $rads^{-1}$ |

*For/Dedicated to/To my...*

# Chapter 1

# Introduction

The goal in this thesis is to find a solution of the nonsmooth minimization problem

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & l_i \leq x_i \leq u_i, \\ & i = 1, \ldots, n. \end{aligned} \tag{1.1}$$

where $f \colon \mathbb{R}^n \to \mathbb{R}$, $n$ is a very large but finite number. And $l_i$ and $u_i \in \mathbb{R}$

Larger problems not only mean that the problem will take a longer time to solve compared to a similar problem. But storing and calculating a Hessian matrix is prohibitively expensive. There a few techniques that have already been developed for the case when $n$ is very large and this is what we know as large scale optimization. Also, Several techniques have already been developed to handle this type of problems as long as the function $f$ is smooth.

In this thesis $f(x)$ is a nonsmooth function.

For the particular case when $n$ is a small number, several methods that solve optimization problems of nondifferentiable functions in lower dimensions [2] have been developed. In the case of smooth functions, it is possible to use Newton iteration algorithms and achieve quadratic convergence, the problem with Newton algorithms is that they require second derivatives to be provided[1]. In the 1950's and several years after that, several quasi-newton methods were proposed where the second derivative Hessian matrix is "approximated" step by step [1]. These approximations or "updates" are calculated after every iteration and the way in which this update is found defines a new method

[1] the main issue with the second derivative is that it requires a total of $n \times n$ partial derivatives. Which is impractical for medium and for some small-size problems

depending on the particular needs. This thesis will only be concerned with the $BFGS$. [2] which can achieve super linear convergence, has proven to work in most practical purposes and posseses very nice self correcting features [3]. In other words, it doesn't matter that one update incorrectly estimates the curvature in the objective function, $BFGS$ will always correct itself in just a few steps. This self-correcting property is very desired in the nonsmooth case, since changes in curvature could be large near the optimal point. $BFGS$ is not the right tool for large scale optimization and therefore an $L-BFGS$ adaptation is needed for 1.1

A final assumption in this thesis is that the Hessian matrix is not sparse. In this case, there are other algorithms that may be more suitable [5, 6], some of them have even been implemented in fortran [7].

In this sense, chapter 2 will extend on $BFGS$ and on the basic reasons and steps to move into a large scale $L-BFGS$. During the solution of an unconstrained problem, a template version of $C++$ code was created which enhances the code originally written by Allan Kaku and Anders Skaaja [8]. This software maintains Allan's high-precision libraries. A link to this $BFGS+L-BFGS$ software is provided in the thesis website. An introduction to the different quasi-newton methods is provided, with a simple introduction to the most important of them and the some of the most basic techniques to guarantee a reasonable speed of convergence.

The next chapter is chapter 3 and this chapter introduces the algorithm by Nocedal, and what changes were introduced into the code in order to produce a better and stable version in the nonsmooth case. The website also contains a $FORTRAN$ implementation derived from the original implementation by Nocedal.

Nocedal's original algorithm consists of 2 steps. In the first step most of the dimensions in the problem should be removed, making the problem a lot simpler. And in the second step there is some fine tuning to guarantee better than just linear speed of convergence.

[2] BFGS stands for the last names of its authors Broyden, Fletcher, Goldfarb and Shanno

# Chapter 2

# BFGS and L-BFGS and implementations in the unconstrained case

The *BFGS* method is a line search method. For this reason an introduction to line search methods is important

## 2.1 Line Search Methods:

Line search methods are iterative methods where every step the researcher has to decide a direction to move or "search direction" and also, how much to move in that direction or "step length". In general all line search methods are characterized by the equation

$$x_{k+1} = x_k + \alpha_k d_k \tag{2.1}$$

where $\alpha_k$ is the step length and $d_k$ is the search direction.

The main difference between line search methods will be in the selection of $\alpha_k$ and $d_k$ and depending on this selection it will be possible to achieve a convergence that will typically be somewhere between linear and quadratic.

In general it is desirable that the search direction is a descent direction in every step. So one usually also checks that the property $d_k^T \nabla f_k = |d_k||\nabla f_k| \cos\theta < 0$ is satisfied.

In general the search direction will have the form

$$d_k = -B_k^{-1} \nabla f_k \tag{2.2}$$

And the choice of matrix $B_k$ will define what type of method we are using. This means that together with the search direction condition $d_k^T \nabla f_k < 0$ we need that:

$$-\nabla f_k B_k^{-1} \nabla f_k < 0 \tag{2.3}$$

And this would require that $B_k$ be positive definite.

Other conditions that are of the utmost importance in this thesis are the conditions on step length. These conditions will be reviewed in a separate section.

Next a presentation of some of the most important line search methods

### 2.1.1 Steepest Descent

The most naive choice of search direction is the negative of the gradient $\nabla f_k(x)$, which is equivalent to choosing $B_k = I$. This method is called steepest descent and its main advantage is obviously that it only requires $\nabla f_k$ as an input.

If the condition of the function's Hessian matrix is low, then steepest descent will converge very fast, but if this condition is high, the method will take a lot of iterations. In the worst cases, steepest descent can show a zig-zagging pattern that slows down convergence.

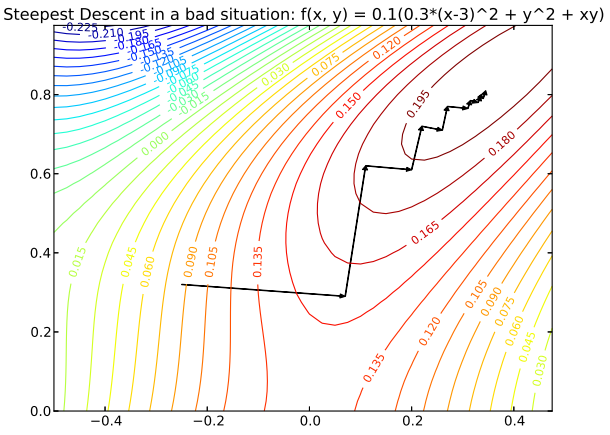



FIGURE 2.1: Steepest Descent example. NOTE: This figure will be replaced by good graphical examples that show disadvantages of steepest descent. Rosenbrock???

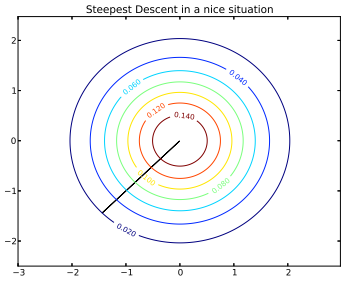



FIGURE 2.2: Steepest Descent example.

### 2.1.2 Newton's Method

Newton's method highly improves upon steepest descent by incorporating information from the Hessian second derivative. It's derivation comes from the second-order Taylor series model of the function around the current point $x_k$.

$$f(x_k + d_k) = f(x_k) + d_k^T \nabla f_k + \frac{d_k^T \nabla^2 f_k d_k}{2} \tag{2.4}$$

And assuming positive definitiness of the Hessian matrix, the optimal value of $d_k$ can be obtained by deriving and equating to zero. Thus the direction is determined by the following formula.

$$d_k^N = -\left(\nabla^2 f(x_k)\right)^{-1} \nabla f(x_k) \tag{2.5}$$

This new search direction stablishes a change in the metric by means of the Hessian matrix $\nabla^2 f$. With this change, the new search direction will point more towards the actual minimum, and not in the same direction of the gradient. [insert graphic here].

Not only that, but the newton direction $d_k^N$ is exactly the length necessary to minimize the model on equation 2.4. Which is great news because under regular assumptions a step size of 1, $\alpha_k = 1$ is all we need to achieve a good convergence and most times we don't need to use an independent algorithm to calculate $\alpha_k$. This change is excellent in terms of convergence. When the method is within a neighborhood of the solution, the convergence is very fast. Indeed, the convergence is quadratic under regular assumptions[4].

Besides that, there are a few big problems with Newton's method. First of all a Hessian has to be provided for each iteration and this makes the method impractical for medium and large size problems. Second, even if the Hessian is provided, the inversion operation

to go from $\nabla^2 f_k$ to $\left(\nabla^2 f_k\right)^{-1}$ requires operations in the order of $O(n^3)$ which is very expensive when n is large.

Given these problems the newton method is ideal for small problems, but it's nearly useless for the large scale problems that are the scope of this thesis.

### 2.1.3 quasinewton methods

Newton's method provides a desireable search direction, but its calculation is very expensive, It is logical that some middle ground solution must be found. In this sense quasinewton methods try to approximate the inverse of the Hessian matrix $\nabla^2 f_k$ saving the algorithm from having to calculate an inverse and to have to provide the hessian matrix everytime.

$$d_k^{QN} = B_k^{-1} \nabla f(x_k) \tag{2.6}$$

In this method the identity matrix[1] is supplied at the first iteration $B_k^{-1} = \mathbb{I}$, which is nothing but steepest descent for the first step. After that, the matrix $B_k^{-1}$ is updated with the $BFGS$ update formula [2]

$$B_{k+1}^{-1} = (I - \rho_k s_k y_k^T) B_k^{-1} (I - \rho_k y_k s_k^T) + (\rho_k s_k s_k^T) \tag{2.7}$$

where:

$$\rho_k = \frac{1}{y_k^T s_k}$$
$$s_k = x_{k+1} - x_k$$
$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$

For the smooth case it can be proven that the algorithm achieves superlinear convergence after it is in a neighborhood of the solution. It also improves on the speed of convergence because it is not necessary to perform an inverse matrix operation at the cost of $O(n^3)$ operations at the end of every step. Finally, as already mentioned, no Hessian matrix has to be provided, so the $BFGS$ method is the right choice for medium to large size problems.

[1] It can be shown that other matrices might be more efficient at a starting point. However, in this thesis, $\mathbb{I}$ will do for simplicity

[2] As I already mentioned. BFGS is not the only update available, but from now on, it is the only that is focused on in this thesis

Unfortunately, the method still requires to store a matrix of size $n^2$ and this could be a problem when dealing with several thousands of variables. In this sense an improvement is suggested

### 2.1.4 Limited Memory Quasinewton methods: L-BFGS

The main idea of limited memory is that instead of building the whole matrix $B_k^{-1}$ from $BFGS$, it is possible to store only the $m$ previous vectors $s_k$ and $y_k$. The way this is done is by recursively applying 2.7 [4]

**Algorithm 1:** L-BFGS construction

**Data**: $y_k$, $s_k$

**Result**: The search direction

$q \leftarrow \nabla f(x_k)$;
**for** $i$ = *k-1, k-2, . . . , k-m* **do**
  $\alpha_i \leftarrow \rho_i s_i^T q$;
  $q \leftarrow q - \alpha_i y_i$;
**end**
$r \leftarrow H_k^0 q$;
**for** $i$ = *k-m, k-m+1, . . . , k-1* **do**
  $\beta \leftarrow \rho_i y_i^T r$;
  $r \leftarrow r + s_i(\alpha_i - \beta)$;
**end**

This method adds an inexpensive number of calculations every step but removes the storage of $B_k^{-1}$ completely from the picture.

In practice, only a handful of points have to be kept in memory. While a very tiny $m$ is bad and hurts the convergence because the approximation to the true hessian is poor. Large values of $m$ slow down the calculations and start to hurt. Typically a value between 5 and 20 is what is used in practice.

In this thesis $L - BFGS$ is the focus of study. First a tool was created to study $L - BFGS$ in the smooth case and then a tool was also created to study the method in the constrained scenario.

### 2.1.5 Convergence of search direction methods

Finally there is the question of convergence and for the case where $f(x)$ is a smooth function, line search convergence is guaranteed for the steepest descent method via

Zoutendijk's theorem [9] in [10]. This theorem quantifies the impact of the step length $\alpha_k$ on convergence. It also, produces a set of criteria for other methods so that they too are globally convergent as long as they don't deviate too far away from steepest descent. Unfortunately there is no such theorem for the nonsmooth case, and we are only sure of convergence from practical results.

While these methods are well studied and they work well for the smooth case, the reality is that most of the convergence of theorems do not apply to the nonsmooth case, although in general, these methods have been used on nonsmooth functions with a good degree of success[11].

### 2.1.6 step length and wolfe conditions

The methods so far suggested give the direction on which to move. All of them will be descent directions given that some conditions are satisfied. But let's remember that according to equation 2.1 a constant $\alpha_k$ has to be determined. How much to move in each direction is a different kind of problem and solving it is just as important as finding the direction $d_k$.

In this sense it's required in line search algorithms to satisfy at least two conditions. The first one is to guarantee that there is some descent in the direction. It is given already that since $d_k$ is a search direction there exists at least some $\epsilon > 0$ s. t. $\forall \alpha < \epsilon$ $f(x_k + \alpha d_k) < f(x_k)$. However, this only implies the existence. If an optimal level of decrease is to be found, better criteria need to be applied. This is where the Armijo and the Wolfe conditions enter into play.

In the rest of this section, $x_k$ and $d_k$ are already being given and it's therefore simpler to focus on the variable $\alpha$. For this reason, the variable $\phi$ is introduced:

$$\phi(\alpha) = f(x_k + \alpha d_k) \tag{2.8}$$

#### 2.1.6.1 Armijo and Strong Wolfe Conditions

The first condition that is required is that the function decreases a little bit. It is also desireable that the more significant the step is, the more significant the decrease in magnitude should be. This is known as the armijo or sufficient decrease condition which is presented as:

$$\phi(\alpha) < \phi(0) + c_1 \phi'(0) \tag{2.9}$$

where $0 < c_1 < 1$. Notice that $\phi'(0) < 0$ because $d_k$ is a descent direction. In practice this constant $c_1$ is chosen to be quite small; in the order of $10^{-4}$. However, this condition alone is not enough because there is always a very tiny $\alpha$ for which the condition is satisfied, so it is desireable that the step length is something of substance. The second condition that we want a smooth function to satisfy is that the step length shoul be long enough so that the gradient almost vanishes. In other words, if the step length is short and the gradient's norm doesn't decrease enough, A longer step should be chosen, The algorithm should do this until the benefits from increasing the step length are minimal. This can be represented in the equation:

$$|\phi'(\alpha)| \leq c_2|\phi'(0)| \tag{2.10}$$

where $0 < c_1 < c_2 < 1$. Typical values of $c_2$ are in the order of 0.9.

Several things can be said about the Stronge Wolfe conditions; first of all, these are not the only conditions out there and another famous set of conditions are the Goldstein conditions, but they apply specially to the pure Newton method [4]. Second, there is always a solution to the strong wolfe conditions, so it is always possible to find an $\alpha$ that satisfies the conditions (as long as the function is smooth). It can also be proved that these strong wolfe conditions guarantee convergence of quasinewton methods via Zoutenijk's theorem [9]

#### 2.1.6.2 The Nonsmooth case and Weak Wolfe Conditions

Of course the main problem with nonsmooth functions is that Zoutendijk's theorem does not apply anymore and therefore there is no guarantee that a step that satisfies both strong Wolfe conditions can be found. A good example of this situation is presented in [11] It is therefore possible to relax the strong Wolfe condition and to not impose an upper bound on the new derivative. Therefore the new weak Wolfe condition that should apply better for the nonsmooth case is:

$$\phi'(\alpha) \geq c_2\phi'(0) \tag{2.11}$$

This weak wolfe condition should be better for the nonsmooth case.

# Chapter 3

# Constraints and L-BFGS-B: Algorithm and implementations

In this chapter the algorithm to handle the constrained case is presented. The original algorithm [12] has an accompanying software written on *FORTRAN* and this thesis builds upon that software by making sufficient changes to make it applied to the nonsmooth case.

The original problem was stated in equation 1.1. In this case an optimizer was created for the case when $n$ is large and $f$ is nonsmooth. The method is based on a gradient projection method similar to the one outlined in [13, 14] which is used to determine an active set corresponding to those variables that are bound at a certain step.

The active set at $x^*$:

$$\mathcal{A}(x^*) = \{i \in \{1 \dots n\} | x_i^* = l_i \vee x_i^* = u_i\} \tag{3.1}$$

It seems like working on this active set is efficient in large problems and according to previous research its properties are theoretically good [4]. The most important reason in theory is that the gradient projection step is able to find most of the active set variables in a single stroke. Usually, while performing quadratic optimizations, the active set changes by one variable at a time during the line search step [1]. So, if 1 million constraints are active at the nondenerage solution, at least 1 million iterations will be needed. Gradient projection gets rid of that problem, diminishing the number of iterations, and at the same reducing the number of variables for the next step.

[1] the line search is cut short immediately after the first bound is hit, so only one active constraint will change at every step... that is, unless the line search hits several constraints at the same time by coincidence, which is very unlikely

[Try to introduce a graphic of gradient projection here]

Just like its unconstrained counterparty $L-BFGS$, $L-BFGS-B$ does not require a Hessian matrix to be provided and instead the Hessian is approximated. Following some simple computational techniques that will all be explained, the number of computations and the storage required, are all kept in the order of $n$.

## 3.1 Overview of the Algorithm

At the beginning of the $k^{th}$ step, the function $f(x)$ is approximated by a second-order Taylor expansion around $x_k$. This approximation will be refered to as "the model".

$$m_k(x) = f(x_k) + \nabla f(x_k)^T(x - x_k) + \frac{(x - x_k)^T B_k(x - x_k)}{2} \tag{3.2}$$

In this model it is important to point out that the gradient $\nabla f$ is provided, however $B_k$ will be a $L-BFGS-B$ approximation to the Hessian $\nabla^2 f$ and will not be calculated explicitly. The algorithm assumes that the function is similar to the quadratic model at least in a neighborhood of $k$, therefore, the first step will be to try and minimize the model 3.2 subject to the constraints $l_i < x_{k,i} < u_i$ via the gradient projection method.

On each iteration of the gradient projection algorithm there will be two stages, a cauchy point computation and a subspace minimization. In the first stage the algorithm starts on the current point $x_k$ searching on the direction of $-\nabla f(x_k)$. Now, whenever this search direction encounters one of the constraints, the search direction turns on the boundaries in order to remain feasible. The path is nothing but the feasible piecewise projection of the steepest descent search direction on the contraint "box" determined by the values $\overrightarrow{l}$ and $\overrightarrow{u}$. At the end of this stage, the value of $x$ that minimizes $m_k(x)$ on this piecewise gradient projection path is known as the "Cauchy point" $x^c$.

For stage two, define the working set as the active set defined on the cauchy point $\mathcal{A}(x^c)$. In order to do this, solve the quadratic subproblem in which all values of the working set $\mathcal{A}(x^c)$ are fixed at the values corresponding to $x^c$

### 3.1.1 Generalized Cauchy Point Calculation

The generalized cauchy point that minimizes 3.2 and lies along the gradient projection path can be calculated by a simple algorithm. First of all, any point on the space can be projected on the feasible region by the following formula:

$$\mathcal{P}(x,\vec{l},\vec{u}) = \begin{cases} l_i, & \text{if } x_i < l_i \\ x_i, & \text{if } x_i \in [l_i, u_i] \\ u_i, & \text{if } x_i > u_i \end{cases} \tag{3.3}$$

The piecewise linear path that starts on $x_k$ and goes on the projected direction of the negative gradient at $x_k$ can be parameterized by $t$ as:

$$x(t) = \mathcal{P}(x_k - t\nabla f(x_k), \vec{l}, \vec{u}) \tag{3.4}$$

With this definition the $k^{th}$ Generalized Cauchy point is the first local minimizer of the function $m_k(x(t))$ where $t > 0$. The minimizer is obtained by examining each of the different segments one by one in order. In order to do this, it is important to find out the elements at which the breakpoints occur. These elements are characteristized disorderly by:

$$\hat{t_i} = \begin{cases} \frac{x_{i,k}-u_i}{\nabla f(x_k)_i}, & \text{if } \nabla f(x_k)_i < 0 \\ \frac{x_{i,k}-l_i}{\nabla f(x_k)_i}, & \text{if } \nabla f(x_k)_i > 0 \\ \infty, & \text{otherwise} \end{cases} \tag{3.5}$$

where $x_{i,k}$ represents the $i^{th}$ coordinate of $x_k$ and $\nabla f(x_k)_i$ stands for the $i^{th}$ coordinate of the gradient. It is therefore very important to find a piecewise representation of the components of $x(t)$ which is piecewise defined as

$$x_i(t) = \begin{cases} x_{i,k} - t\nabla f(x_k)_i, & \text{if } t \leq \hat{t_i} \\ x_{i,k} - \hat{t_i}\nabla f(x_k)_i, & \text{otherwise} \end{cases} \tag{3.6}$$

Now, in order to iterate in order through the different $\hat{t_i}$ values. It is necessary to sort the values with a heapsort algorithm [2] which can do the job in $O(n \log(n))$. Eliminating repeated and zero values of $\hat{t_i}$, this ordering defines the intervals $[0, t_1]$, $[t_1, t_2]$, ... [3].

When this algorithm is run, the intervals are analysed one by one starting with $[0, t_1]$. If the optimal is found in this interval, the algorithm stops. If it has not been found the algorithm continues to the next interval and so on. Assuming that the algorithm has

[2] several algorithms can be chosen in this situation

[3] Notice that the $\hat{hat}$ represents the unordered breakpoints, whereas the lack of represents the ordered breakpoints

not found a solution until $t_{j-1}$, and the point $x(t_{j-1})$ has been found and the direction of the projected gradient is given by

$$d_i^{j-1} = \begin{cases} -\nabla f(x_k)_i, & \text{if } t_{j-1} < \hat{t}_i \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

the model 3.2 on the next line segment will be

$$\begin{aligned} m_k(x) = f(x_k) + \nabla f(x_k)^T (x_{j-1} - x_k + \Delta t d^{j-1}) + \\ \frac{(x_{j-1} - x_k + \Delta t d^{j-1})^T B_k (x_{j-1} - x_k + \Delta t d^{j-1})}{2} \end{aligned} \tag{3.8}$$

where

$$\Delta t = t - t^{j-1}$$

With this equation it is now possible to obtain a formula for the optimal $\Delta t$. All there is to do is to group the terms in powers of $\Delta t$, derive, equate to zero and obtain the optimal values. Grouping the terms from 3.8

$$\begin{aligned} m_k(x) = f(x_k) + \nabla f(x_k)^T (x_{j-1} - x_k) + \frac{1}{2}(x_{j-1} - x_k)^T B_k (x_{j-1} - x_k) + \\ \left( \nabla f(x_k)^T d^{j-1} + {d^{j-1}}^T B_k (x_{j-1} - x_k) \right) \Delta t + \\ \left( \frac{{d^{j-1}}^T B_k d^{j-1}}{2} \right) \Delta t^2 \end{aligned} \tag{3.9}$$

And defining the corresponding terms to be

$$f_{j-1} := f(x_k) + \nabla f(x_k)^T (x_{j-1} - x_k) + \frac{1}{2}(x_{j-1} - x_k)^T B_k (x_{j-1} - x_k) \tag{3.10}$$

$$f'_{j-1} := \nabla f(x_k)^T d^{j-1} + {d^{j-1}}^T B_k (x_{j-1} - x_k) \tag{3.11}$$

$$f''_{j-1} := {d^{j-1}}^T B_k d^{j-1} \tag{3.12}$$

The optimal value will be $\Delta t^* = \frac{-f'_{j-1}}{f''_{j-1}}$ as long as this $\Delta t^*$ is located within the interval $[0, t_j - t_{j-1}]$. If it is not, there are two other alternatives. Either $f'_{j-1} \geq 0$ in which case the optimal cauchy point is located right at the breakpoint $x_{j-1}$, or $f'_{j-1} < 0$ in

which case the solution is not in the current interval, or finishes at $t_j$ if this was the last interval analysed, or it moves over to the interval $[t_j, t_{j+1}]$, replacing

$$x_j = x_{j-1} + \Delta t^{j-1} d^{j-1}, \text{ where} \Delta t^{j-1} = \hat{t_j} - \hat{t_{j-1}} \tag{3.13}$$

and of course, replacing the search direction $d^j$ by cancelling the corresponding coordinate to 0 zero. That is,

$$d^j = d^{j-1} + \nabla f(x_k)_b e_b \tag{3.14}$$

Where $b$ represents the variable that becomes active at $\hat{t_j}$

A more surprising result [12] not demonstrated here is that it is possible to calculate $f_j'$ and $f_j''$ using the $L-BFGS$ formulae recursively as

$$\begin{aligned} f_j' &= f_{j-1}' + \Delta t^{j-1} f_{j-1}'' + (\nabla f(x_k)_b)^2 + \theta \nabla f(x_k)_b (x_j - x_k)_b - \\ &\quad \nabla f(x_k)_b w_b^T M W^T (x_j - x_k) \\ f_j'' &= f_{j-1}'' + 2\theta \nabla f(x_k)_b^2 - 2\nabla f(x_k) w_b^T M W^T d^{j-1} + \theta (\nabla f(x_k)_b)^2 - \\ &\quad \nabla f(x_k)_b^2 w_b^T M w_b \end{aligned} \tag{3.15}$$

which is very cheap to calculate since it only implies an $O(n)$ calculations, coming from the the matrix products $W^T(x_j - x_k)$ and $W^T d^j$. Looking closely however, these values are updated everytime just one iteration. So it makes sense to store the vectors and update them everytime.

$$p_j := W^T d^j = W^T (d^{j-1} + \nabla f(x_k)_b e_b) = p_{j-1} + \nabla f(x_k)_b w_b \tag{3.16}$$

$$c_j := W^T (x_j - x_k) = W^T (x_{j-1} - x_k + \Delta t^{j-1} d^{j-1}) = c_{j-1} + \Delta t^{j-1} p_{j-1} \tag{3.17}$$

which brings the order of calculation updates down to $O(m^2)$. Since only the first iteration requires a total calculation of the order of $O(n)$, the algorithm is very cheap to calculate

In synthesis the algorithm works like this, there is an initialization phase that works for the first segment and an iteration phase that works for the other segments.

**Algorithm 2:** Generalized Cauchy Point

**Data**: $x$, $\vec{l}$, $\vec{u}$, $\nabla f(x_k)$, and $B_k$

**Result**: The Generalized Cauchy Point

**for** $i \in 1, \ldots, n$ **do**
  Calculate $\vec{t_i}$ according to 3.5;
  Calculate $d_i$ based on $\vec{t_i}$;
**end**
initialization;
$\mathcal{F} := \{i : \vec{t_i} > 0\}$;
$p := W^T d$;
$c := 0$;
$f' := -d^T d$;
$f'' := -\theta f' - p^T M p$;
$\Delta t^* := \frac{-f'}{f''}$;
$t_{old} := 0$;
$t := min\{t_i : i \in \mathcal{F}\}$;
$b := i$ s.t. $t_i = t$ the boundary just hit;
remove b from $\mathcal{F}$;
$\Delta t := t - 0$;
Subsequent;
**while** $\Delta t^* \geq \Delta t$ **do**
  $x_b^{cp} := u_b$ or $l_b$ Depending on what boundary was hit;
  $z_b := x_b^{cp} - x_b$;
  $c := c + \Delta t p$;
  $f' := f' + \Delta t f'' + (\nabla f_b)^2 + \theta \nabla f_b z_b - \nabla f_b w_b^T M c$;
  $f'' := f'' - \theta(\nabla f_b)^2 - 2\nabla f_b w_b^T M p - (\nabla f_b)^2 w_b^T M w_b$;
  $p := p + \nabla f_b w_b$;
  $d_b := 0$;
  $\Delta t^* := -frac f' f''$;
  $t_{old} := t$;
  $t := min\{t_i : i \in \mathcal{F}\}$;
  $b := i$ s.t. $t_i = t$;
  $\Delta t := t - t_{old}$;
**end**
$\Delta t^* := \max\{\Delta t^*, 0\}$;
$t_{old} := t_{old} + \Delta t^*$;
$x_i^{cp} := x_i + t_{old} d_i$ , whenever $t_i \geq t$;
$\forall i \in \mathcal{F}$ with $t_i$ remove i from the set $\mathcal{F}$;
$c := c + \Delta t^* p$;

Overall, the major cost of the algorithm is the heapsort step for $t_i$, which is run in $O(n \log(n)$

### 3.1.2 Subspace Minimization

The final step on this alrorithm is the subspace minimization. Once the gradient direction step has been taken, several dimensions in the problem will have been removed. Here, the algorithm takes advantage of this situation and solves the quadratic model 3.2 restricted to the simple constraints. In order to do this, a new search direction is proposed. And the step length is such that it stays within the bounds stablished in the problem.

The idea at a higher level is to solve the constrained problem, but only on those dimensions that are free (not at bound). In the notation set forth in the previous algorithm. $\mathcal{F}$ represents the set of indices corresponding to the $t$ free variables. $Z_k$ is the matrix formed by the $t$ unit vector columns that span the dimensions of the free variables and $A_k$ is the corresponding matrix that represents the active constraint gradient (also unit vectors). The dimension of $Z_k$ would be $n \times t$ and the dimension of $A_k$ $n \times (n - t)$.

The starting point for this new problem will be the previously found cauchy point $x^c$, and the algorithm only moves in a direction that lives in the space generated by the columns of $Z_k$. In other words, if $\hat{d}$ is the $t$-dimensional search direction,

$$x = x^c + Z_k \hat{d} \tag{3.18}$$

Under this equation and replacing in 3.2, it's obtained

$$m_k(x) = \hat{d}^T \hat{r}^c + \frac{1}{2} \hat{d}^T \hat{B}_k \hat{d} + \gamma \tag{3.19}$$

where $\gamma$ is a constant and $\hat{B}_k = Z_k^T B_k Z_k$, or the Hessian restricted to the "free" dimensions in the problem. $\hat{r}^c = Z_k^T (g_k + B_k(x^c - x_k))$ is the corrected gradient restricted to the same "free" dimensions. Which restates the problem as

$$\begin{aligned} \min_{\hat{d} \in \mathbb{R}^t} \quad & \hat{m}_k(\hat{d}) = \hat{d}^T \hat{r}^c + \frac{1}{2} \hat{d}^T \hat{B}_k \hat{d} + \gamma \\ \text{s.t.} \quad & l_i - x_i^c \leq \hat{d}_i \leq u_i - x_i^c, \\ & i \in \mathcal{F} \end{aligned} \tag{3.20}$$

And the solution is simply $\hat{d}^u = -\hat{B}_k^{-1}\hat{r}^c$[4]. Now the step length should be $\hat{d}^* = \alpha^*\hat{d}^u$ where $\alpha^*$ is chosen so that the new point $\bar{x}_i$ stays within the constraints originally imposed.

[4]Notice that this implies the inversion of $\hat{B}_k$. However there is a numerical trick explained in [4], that makes this operation trivial, since $\hat{B}_k$ is a small-rank correction of a diagonal matrix and its inverse can be computed by the Sherman-Morrison-Woodbury formula

# Chapter 4

# Modifications to the original algorithm

## 4.1 Convex Hull and termination conditions

The most important requirement of a practical algorithm is that it ends in a finite time. For the case of smooth functions, the formal way to do this is to check whether the gradient has dimension zero 0 wherever the constraints are not at bound. In the case of nonsmooth functions however, this is not necessarily true and the function at the minimum, may have a kink. In this kink the gradient may not vanish. Furthermore, if there is a sequence of points that approaches the optimum $x$ from the right, the gradients corresponding to this sequence of points might be completely different from the gradients associated to a sequence gradients associated to a sequence of points that approaches the optimum from the left. In other cases, the optimum might be located right at one of the boundaries, In this case the gradient does not necessarily vanish.

Given this set of conditions, there is the need for a special set of rules to establish the finalization of each optimization.

Since BFGS approximations typically converge to Clarke-Stationary points. The right methodology is to calculate the subgradient. One particular methodology that guarantees an end to the algorithm is suggested in [15]. In order to make sure that the gradient zero $\vec{0}$ is part of the subgradient calculated over a neighbourhood of the optimum. The algorithm keeps a record of the latest gradient vectors in a small neighbourhood of the point that we suspect is the optimum, This collection is called $G_k$ in [15], This collection of gradients spans an associated convex hull of gradients. If this convex hull contains at least one vector of dimension smaller than a tiny number $\tau_k$, the algorithm ends.

Of course the best way to find a vector with such properties is to find the vector with the minimal norm that resides in the convex hull generated by $G_k$

#### 4.1.0.1 minimization of the quadratic program

This solution is guarantees to end up at a local optimum. However, one subalgorithm needs to be solved. This algorithm is a practical primal-dual algorithm. In this case in particular the best solution is to implement a variation of Mehrotra's Predictor-Corrector algorithm applied to quadratic programming. The primal dual method requires the solution of a system in order to calculate the search direction. The most expensive part of this solution is the calculation of the cholesky decomposition. Mehrotra's algorithm uses the same cholesky decomposition to calculate both directions. the predictor, and the corrector.

Currently there is not a theoretical calculation of the complexity of this algorithm but it is very used in practice. The implementation here is exactly the one on [**?** ]. It was implemented in fortran as part of the optimizer.

## 4.2 cubic interpolation replaced with line search

The original software by Nocedal[**?** ], included a cubic line search. The idea of a cubic interpolation line search is to take advantage of the smooth properties of functions and take advantage of the curvature properties. It is debatable to see which one works better whether a simple line search or a cubic interpolation. However, In the case of nonsmooth functions, this line search does not serve our purpose and therefore a more typical line search that implements a bisection was used.

In general, a step length is selected. If this step length does not satisfy the sufficient decrease and curvature conditions, then a step length of half or double the size is selected. the algorithm is guaranteed to converge under a careful selection of the parameters, however, in case this does not happen, the software will stop with a warning.

## 4.3 the functions to be tested

In order to make some tests, a few functions will be evaluated. The most important function to test this non-smooth optimizer is a modified version of rosenbrock's:

$$f(x) = (x_1 - 1)^2 + \sum_{i=2}^{n} |x_i - x_{i-1}^2|^p \tag{4.1}$$

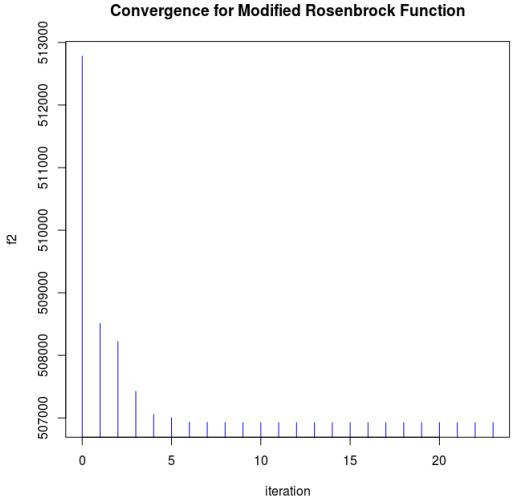
Where the value of $p$ changes the behaviour of the optimizer. This function can be proven to be lipschitz continuous whenever $p > 1$ if restricted to the domain defined by

$$x_i = \begin{cases} [-100, 100] & \text{if } i \in \text{ even numbers} \\ [10, 100] & \text{if } i \in \text{ odd numbers} \end{cases} \tag{4.2}$$

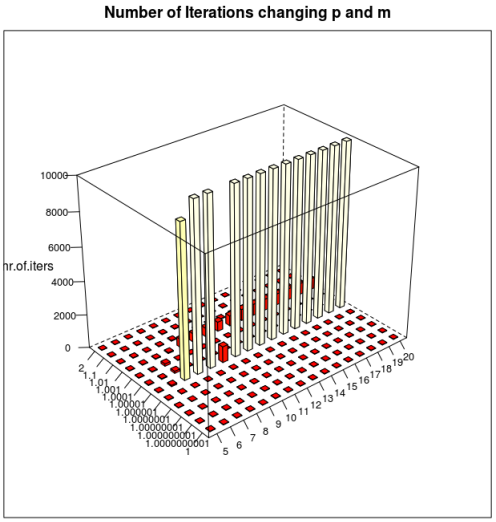
and in fact, whenever the function is restricted to a finite domain. Because whenever $p > 1$ we have that the function $f(z) = |z|^p$ is zero 0 around zero because the derivative $p|z|^{p-1}$ is zero whenever $z$ tends to zero from the right. (from the left also because it is an even function). However the second derivative will not be as nice.

For the case when $p \leq 1$ we have that the second derivative tends to infinity. $\lim_{x \to 0^+} f' = \infty$. Which is already well known given the "heavyside" look of $f(z) = |z|$.

The convergence of the algorithm smoothly descends to the objective




The converge is adversely affected by the selection of $p$ as one would expect. Values of $p$ descending to 1 make the function less "smooth" and have the adverse effect of making the convergence much more difficult. In this exercise it is noticeable how slow the convergence becomes for a few specific values of $p$. In particular for 1.0001




## 4.4 Weak wolfe conditions

Probably the most important change made to the original code was the change in the curvature condition. Originally there are two Wolfe conditions, one of them is the Armijo condition, also known as the sufficient decrease conditions.

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k p_k^T \nabla f(x_k) \tag{4.3}$$

and the other one is the curvature condition, of which the most popular version is the strong wolfe curvature condition:

$$|p_k^T \nabla f(x_k + \alpha_k p_k)| \leq |p_k^T \nabla f(x_k)| \tag{4.4}$$

The strong wolfe is a more natural way to see and achieve convergence, but the problem is that it does not work well for the nonsmooth case. This is because near the minimal points, there maybe abrupt changes in curvature, in these cases there is no other option but to relax the curvature condition as long as the sufficient decrease condition is satisfied. The suggested new decrease condition is this.

$$p_k^T \nabla f(x_k + \alpha_k p_k) \geq p_k^T \nabla f(x_k) \tag{4.5}$$

It is noticeable that with this new condition the algorithm does not crash, as opposed to when the hard wolfe condition is used.

# Appendix A

# Appendix Title Here

Write your Appendix content here.

# Bibliography


[1] J.E. Dennis and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice Hall, SIAM Publications, Englewood Cliffs, NJ, 1983, 1993.

[2] Krzysztof C. Kiwiel. *Methods of Descent for Nondiferentiable Optimization.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.

[3] Jorge Nocedal. Theory of algorithms for unconstrained optimization. *Acta Numerica*, 1:199–242, January 1992.

[4] Stephen J. Wright Jorge Nocedal. *Numerical Optimization.* Springer Series in Operations Research, 2nd edition, 2006.

[5] Roger Fletcher, Andreas Grothey, and Sven Leyffer. Computing sparse hessian and jacobian approximations with optimal hereditary properties. Technical report, Large-Scale Optimization with Applications, Part II: Optimal Design and Control, 1996.

[6] R. Fletcher. An optimal positive definite update for sparse hessian matrices. *SIAM Journal on Optimization*, 5:192–218, 1995.

[7] Prof. Dr. Ph. L. Toint Dr. A. R. Conn, Dr. N. I. M. Gould. *LANCELOT: A fortran package for large-scale nonlinear optimization (Release A).* Springer Series in Computational Mathematics, Berlin Heidelberg, 1992.

[8] Allan Kaku. Implementation of high-precision arithmetic in the bfgs method for nonsmooth optimization. Master's thesis, New York University, Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, NY 10012, 2011.

[9] G. Zoutendijk. *Nonlinear Programming, computational methods*, pages 37–86. In Abadie [10], 1970.

[10] J. Abadie. *Integer and Nonlinear Programming.* North-Holland Publishing company, Amsterdam-London, 1970.

[11] Anders Skaaja. Limited memory bfgs for nonsmooth optimization. Master's thesis, New York University, Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, NY 10012, 2010.

[12] J. Nocedal R.H. Byrd, P. Lu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, 16(5):1190–1208, 1995.

[13] PH.L.Toint A.R. Conn, N.I.M. Gould. Global convergence of a class of trust region algorithms for optimization with simple bounds. *SIAM Journal of Numerical Analysis*, 25:433–460, 1988.

[14] G. Toraldo J.J. More. Algorithms for bound constrained quadratic programming problems. *Numerical Math.*, 55:377–400, 1989.

[15] AdrianS. Lewis and MichaelL. Overton. Nonsmooth optimization via quasi-newton methods. *Mathematical Programming*, 141(1-2):135–163, 2013. ISSN 0025-5610. doi: 10.1007/s10107-012-0514-2. URL http://dx.doi.org/10.1007/s10107-012-0514-2.