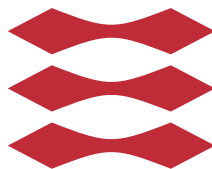


# Implementation and Test of Numerical Optimization Algorithms in C/C++

Christian Wichmann Moesgaard

DTU



Kongens Lyngby 2012  
IMM-B.Sc.-2012-36

Technical University of Denmark  
Informatics and Mathematical Modelling  
Building 321, DK-2800 Kongens Lyngby, Denmark  
Phone +45 45253351, Fax +45 45882673  
[reception@imm.dtu.dk](mailto:reception@imm.dtu.dk)  
[www.imm.dtu.dk](http://www.imm.dtu.dk) IMM-B.Sc.-2012-36

# Summary (English)

---

The goal of this report is to describe the twice continuously differentiable unconstrained optimization problem. Recognizing that it is impossible to check every point, it provides the mathematical foundation for the BFGS algorithm, to solve the problem locally.

Once this is introduced, the report moves on to a practical implementation - the steps required to begin using scientific computing by including BLAS and LAPACK. It then provides the documentation for the implementation in C and an example of how to use it.

Then, it describes how the optimization algorithms can be called from C, C++, Objective-C++, and FORTRAN.

Finally, it does some performance tests, and it is found that libopti is fast compared to scripting based solutions, and as much as 7-8 times faster than the GNU Scientific Library, however it is also not entirely reliable for solving every problem in its class. In fact, no implementation is tested which could solve every problem in its class.



# Summary (Danish)

---

Målet for denne afhandling er at beskrive dobbelt-differentiable, ikke begrænsede optimeringsproblemer. Det indses, at det er umuligt at gennemsnøge ethvert punkt, så der bruges en metode kaldet BFGS til at løse problemet lokalt.

Når dette er gjort, er det videre mål med afhandlingen at give en praktisk implementering. De trin der er nødvendige for at begynde på Scientific Computing ved at inkludere både BLAS og LAPACK. Derefter leveres dokumentation for implementeringen i C og instruktioner på, hvordan det bruges.

Dernæst beskrives hvordan implementeringen kan kaldes fra C, C++, Objective-C++ og FORTRAN.

Til sidst udføres nogle tests. Målet er at vise, at libopti er en hurtig implementering i forhold til scripting-baserede metoder, og at den er hurtig nok til at kunne bruges i stedet for andre, lignende biblioteker til C.

Desværre viser det sig, at algoritmen ikke altid er helt pålidelig, men det er der ingen af dem, der er.



# Preface

---

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

Optimization is an ancient field of mathematics. It all started back in 300 B.C. when Euclid considered the minimal distance between two points. Since then, development was slow until the 17th and 18th century. There Newton's Method was discovered.[oF12]

A pretty common theme among all the optimization algorithms is that, especially for large-dimensional problems, their solution time is very large.[JN00] Because of this, optimization as we know it today did not take off until the middle of the 20th century with the invention of computers usable for scientific computing.[oF12] Since then, many algorithms have appeared to solve various different problems.

The purpose of this project is to create an algorithm designed to solve a specific type of optimization problem using the C language on Linux systems.

C is a rather old programming language with a very powerful, free software compiler[Fou12] and long-standing support. The speed with which C code runs is very high, and so writing the project in C with POSIX subsystems has two main advantages:

- Fast calculation is achievable.
- The project can be built entirely upon free software, making it relatively

easy to port to many hardware architectures and making sure that users of the software do not face restrictions.

Optimization is an important tool in all manner of decision-making processes. It is important in physics, operations research, investments, and many other fields.[\[JN00\]](#)

An optimization problem consists of an objective function, which depends on a potentially large amount of variables, but gives a single output, alternatively with constraints. The goal is to find the minimum or maximum to the objective function within the constraints of the problem.

A problem with no constraints is called an unconstrained optimization problem. This software focuses specifically on this type of problem with an additional property: The objective must be twice continuously differentiable.

In other words, it will solve the problem:[\[JN00\]](#)

$$\min_x f(x)$$

Where  $f(x) \in \mathcal{C}^2 : \mathbb{R}^n \rightarrow \mathbb{R}$ .

Contained within is a description of the algorithm and what it does, and from which sources it was derived. This is followed by an implementation guide and a guide to using the software in C. Several test cases of the software will be demonstrated and the performance will be measured compared to similar software.

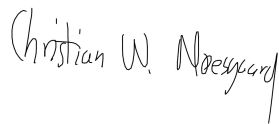
The library will be using the ATLAS Basic Linear Algebra Subroutines[\[ABB+99\]](#) for it's heavy-duty computations for performance benefits.



## Problem definition

- Provide a guide to using BLAS within GNU/Linux Ubuntu systems.
- Provide a usable implementation of an optimization algorithm for twice continuously differentiable unconstrained problems.
- Provide documentation for the function, as well as any other function in use, unless such is already provided elsewhere.
- Provide a guide on how to include and use them in your project.
- Provide a way to use the library functions within other languages.

Lyngby, November 30, 2012-2012

A handwritten signature in black ink that reads "Christian W. Moesgaard". The signature is written in a cursive style with a long, vertical tail on the final letter 'd'.

Christian Wichmann Moesgaard



# Acknowledgements

---

I would like to thank my supervisor professor John Bagterp Jørgensen, DTU IMM, for giving the project ideas and direction to move forward, and for supervising the project.

Many of the algorithms and methods explored in the project have been provided professor Hans Bruun Nielsen at IMM DTU, whom I would like to thank. He wrote the IMM Optibox library for `MATLAB`, upon which several of the included algorithms are based.

Large parts of the algorithms are based upon the work done by `NetLib` with their package `BLAS`, without which most of this work would have taken significantly longer, and the algorithms would have been significantly slower.



# Contents

---

<b>Summary (English)</b>	<b>i</b>
<b>Summary (Danish)</b>	<b>iii</b>
<b>Preface</b>	<b>v</b>
<b>Acknowledgements</b>	<b>ix</b>
<b>1 Theory of Unconstrained Optimization</b>	<b>1</b>
1.1 Optimality Conditions . . . . .	2
1.2 Finding the minimizer . . . . .	4
1.3 Line-search . . . . .	6
1.4 Using a trust region with line search . . . . .	10
1.5 Quasi-Newton BFGS method . . . . .	10
1.6 Finite Difference . . . . .	14
<b>2 Installing BLAS and LAPACK in Ubuntu</b>	<b>15</b>
2.1 Installing Eclipse, BLAS & LAPACK . . . . .	15
2.2 Creating a new project . . . . .	16
2.3 Importing, using linking BLAS & LAPACK . . . . .	17
2.4 First Matrix-multiplications . . . . .	19
2.4.1 Matrix-Vector . . . . .	19
2.4.2 Matrix-Matrix . . . . .	20
2.5 Solving your first LA-equations . . . . .	22
<b>3 Library documentation</b>	<b>25</b>
3.1 Usage example . . . . .	35
<b>4 Mixing FORTRAN and C</b>	<b>37</b>

<b>5</b>	<b>Using the library in ...</b>	<b>41</b>
5.1	FORTRAN . . . . .	41
5.2	Mac OS X and iOS . . . . .	44
<b>6</b>	<b>Performance testing</b>	<b>45</b>
6.1	Himmelblau . . . . .	47
6.2	Rosenbrock . . . . .	49
6.3	Beale . . . . .	51
6.4	Easom . . . . .	53
6.5	Ackley . . . . .	55
6.6	Zakharov . . . . .	58
<b>7</b>	<b>Conclusion</b>	<b>61</b>
<b>A</b>	<b>C library</b>	<b>63</b>
A.1	Library Header . . . . .	63
A.2	BLAS usage . . . . .	67
A.3	Create I matrix . . . . .	67
A.4	Forward difference . . . . .	68
A.5	Quadratic Interpolation . . . . .	69
A.6	Line-search . . . . .	70
A.7	Norm calculation . . . . .	75
A.8	Print performance to file . . . . .	76
A.9	Print performance to folder . . . . .	78
A.10	Unconstrained BFGS minimization . . . . .	80
<b>B</b>	<b>Wrappers to external libraries</b>	<b>89</b>
B.1	Use Apple Accelerate . . . . .	89
B.2	Use ATLAS . . . . .	90
<b>C</b>	<b>Test runs</b>	<b>93</b>
C.1	Apple Test . . . . .	93
C.2	Himmelblau . . . . .	96
C.2.1	MATLAB test . . . . .	96
C.2.2	minlbfgs test . . . . .	96
C.2.3	libopti test . . . . .	98
C.3	Rosenbrock . . . . .	99
C.3.1	MATLAB test . . . . .	99
C.3.2	minlbfgs test . . . . .	99
C.3.3	libopti test . . . . .	101
C.4	Beale . . . . .	102
C.4.1	MATLAB test . . . . .	102
C.4.2	minlbfgs test . . . . .	102
C.4.3	libopti test . . . . .	104

---

C.5	Easom	106
C.5.1	MATLAB test	106
C.5.2	minlbfgs test	106
C.5.3	libopti test	108
C.6	Ackley	109
C.6.1	MATLAB test	109
C.6.2	minlbfgs test	110
C.6.3	libopti test	112
C.7	Zakharov	113
C.7.1	MATLAB test	113
C.7.2	minlbfgs test	114
C.7.3	libopti test	116
C.8	Plot Ackley performance	117
<b>Bibliography</b>		<b>119</b>





## CHAPTER 1

# Theory of Unconstrained Optimization

---

The goal of solving any unconstrained minimization problem is to get the values of  $x$  for which [JN00](#):

$$\min_x f(x)$$

Where  $f(x) \in \mathcal{C}^2 : \mathbb{R}^n \rightarrow \mathbb{R}$ .

The goal of any unconstrained maximization problem is to get the values of  $x$  for which

$$\max_x f(x) \Rightarrow \min_x -f(x)$$

Where  $f(x) \in \mathcal{C}^2 : \mathbb{R}^n \rightarrow \mathbb{R}$ .

This simplifies the entire process as we no longer have to consider the maximization problem.

This value of the vector  $x$  at which it obtains the properties needed to fulfil the objective is denoted  $x^*$ .

A point  $x^*$  is a global minimizer if and only if

$$f(x^*) \leq f(x) \quad \forall x \in \mathbb{R}^n$$

The global minimizer can be difficult to find, because the function  $f(x)$  may not be known analytically and can only be sampled. Since  $f(x)$  spans over at least one real line, there is at least one uncountably infinite amount of points to consider[[Han08](#)]. Thus it is impossible to visit every location.

However, because the function is twice continuously differentiable, any one point, and in particular it's derivatives, have a lot to say about the surrounding area on the function.[[Sch07](#)]

Thus it is far more feasible, for this type of problem, to consider minimizing more locally.

A point  $x^*$  is a local minimizer if there is an open set  $\mathcal{S}$  containing  $x^*$ , such that  $f(x^*) \leq f(x) \quad \forall x \in \mathcal{S}$

## 1.1 Optimality Conditions

An immediate question which arises when asked to find a global or even local minimizer, is one of how to check whether you are in one. It is, as previously discussed, infeasible to check every point. In fact, even the open set in an uncountably infinite set is itself uncountable.[[Kre89](#)]

To get started, Taylor's Theorem must be invoked[[JN00](#)].

Suppose that  $f \in \mathcal{C}^1 : \mathbb{R}^n \rightarrow \mathbb{R}$ , that  $\alpha \in \mathbb{R}$  and that  $p \in \mathbb{R}^n$ . Then:

$$f(x + p) = f(x) + \nabla f(x + \alpha p)p \quad (1.1)$$

for some  $\alpha \in [0, 1]$ .

Furthermore, if  $f$  is twice continuously differentiable:

$$\nabla f(x + p) = \nabla f(x) + \int_0^1 \nabla^2 f(x + \alpha p)p d\alpha \quad (1.2)$$

$$f(x + p) = f(x) + \nabla f(x)p + \frac{1}{2}p \nabla^2 f(x + \alpha p)p \quad (1.3)$$

From this, one can derive the First-Order Optimality Condition[JN00]:

If  $x^*$  is a local minimizer and  $f \in \mathcal{C}^1$  at and near said local minimizer, then  $\nabla f(x^*) = 0$

And the Second-Order Sufficient Optimality Condition[JN00]:

If  $x^*$  is a local minimizer of  $f$  and  $\nabla^2 f$  exists and is continuous in an open neighbourhood of  $x^*$ , then  $\nabla f(x^*) = 0$  and  $\nabla^2 f(x^*)$  is positive definite.

The algorithm which will be written does not test for the Second-order optimality condition. It is quite work-intensive to check, because it involves lots of determinant calculations for use in Sylvester's Criterion[Mey00]. On top of that, other conditions can ensure that this condition is fulfilled most of the time[JN00]. On top of that, the search direction algorithm which will be used assumes that  $\nabla^2 f(x)$  is indeed positive definite for any  $x$  visited. More on this later.

In practice, the imprecision of the data types available on a computer means that the first order optimality condition is actually very rarely achievable, due to cumulative rounding errors in the calculations.

So, rather than finding the exact point, the algorithm will stop once it is "close enough" or have done an unreasonable number of calculations. What "close enough" and "unreasonable" exactly means can be specified by the user.

There are several methods of finding out if the first order optimality condition is almost fulfilled[JN00]:

- $\|\nabla f(x^k)\| \leq \varepsilon_1$
- $\|x^{k+1} - x^k\| \leq \varepsilon_2$
- $f(x^k) - f(x^{k+1}) \leq \varepsilon_3$
- $k > k_{max}$  or  $neval > neval_{max}$  aka. the "give up" condition.

Where *neval* is the number of function evaluations and *k* is the number of steps taken already. Now that we know when we land on a minimizer, it is necessary to find out how to best arrive to the minimizer.

## 1.2 Finding the minimizer

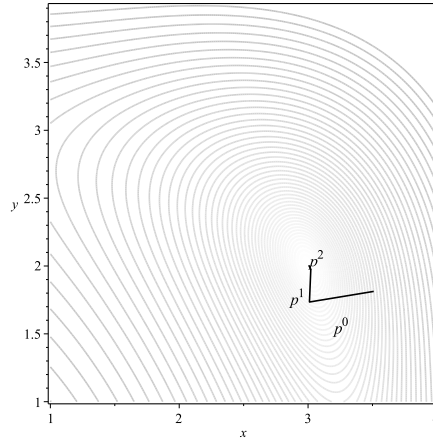
As mentioned earlier, an optimization problem on the real numbers has an infinite number of points to check, but it is only feasible to visit a finite amount of points. Thus, an algorithm that only visits a finite number of points is needed, and it needs to start somewhere.

That "somewhere" is defined by the user and is called  $x^0$ . The goal is to go through a series of steps whereby an  $x^k, k = 1..N$  for each iteration is attained until we reach  $x^*$ , the desired minimizer. This process is performed by finding a step, also known as a descent direction,  $p^k$  by:

$$x^{k+1} = x^k + \alpha^k p^k \tag{1.4}$$

where  $\alpha^k$  is the step-length. More on step-lengths later. For now, assume that  $\alpha^k = 1$ .

Step  $p^k$  is an  $n$ -dimensional vector which is quite well named: It is the direction and magnitude which, if one is to take a step towards  $p^k$  using the current iterate, there is good reason to believe that the function value will decrease.



**Figure 1.1:**  $p^0$  to  $p^2$  on Himmelblau

Perhaps the most important search direction is found using a process called Newton's Method, which results in a Newton direction [JN00]. This direction is derived from the second-order Taylor series approximation:

$$f(x^k + p^k) \approx f + p^k \nabla f(x^k + p^k) + \frac{1}{2} p^k \nabla^2 f(x^k + p^k) p^k := m(p^k) \quad (1.5)$$

Assuming that  $\nabla^2 f^k$  is positive definite, you can simply set  $\nabla m^k(p^k) = 0$  and solve for  $p^k$ . This gives:

$$p^k = -\frac{\nabla f^k}{\nabla^2 f^k} \quad (1.6)$$

Thus each update will result in a step such as this one:

$$x^{k+1} = x^k - \alpha^k \frac{\nabla f^k}{\nabla^2 f^k} \quad (1.7)$$

Thus, so far the algorithm looks like this:

1. Begin with a starting guess  $x^0$  on the at least locally convex function  $f \in \mathcal{C}^2 : \mathbb{R}^n \rightarrow \mathbb{R}$  where  $n$  is the number of input dimensions in the function.

2. Sample the function value  $f(x^0)$ , the gradient  $\nabla f(x^0)$  and the Hessian  $\nabla^2 f(x^0)$
3. While the first order optimality conditions are not yet met:
  - (a) Find  $p^k = -\frac{\nabla f(x^k)}{\nabla^2 f(x^k)}$
  - (b) Use some kind of line-search technique, one is described later, to find a value  $\alpha^k$  which satisfies the line-search conditions.
  - (c) Take the step:  $x^{k+1} = x^k + \alpha^k p^k$
  - (d) Increment  $k$ .
  - (e) Sample the function value  $f(x^k)$ , the gradient  $\nabla f(x^k)$  and the Hessian  $\nabla^2 f(x^k)$

### 1.3 Line-search

Once a descent direction  $p^k$  has been found by using, for example, Newton's method, it should be decided how far to go in this descent direction, henceforth referred to as line-search. This is specifically because when the descent direction was determined we only had local information about the function, and, when taking the full step, it is actually possible that the cost at the new point is larger than the one at the old point[JN00].

Defining a new function, where  $\alpha$  is the search length given at the iteration  $k$ , which can be put in place of  $\alpha^k$  we get:

$$\varphi(\alpha) = f(x^k + \alpha p^k) \quad (1.8)$$

The derivation of how to get the Newton step was explained in the previous chapter, and it assumed that the step-length  $\alpha^k = 1$  was used. This is a good place to start the search[HBN10].

First it is important to look at the slope of  $\varphi(\alpha)$ . Basic vector calculus[HEJK06] immediately reveals the following:

The slope of the function  $\varphi(\alpha)$  can be found as:

$$\frac{d\varphi}{d\alpha} = \frac{\partial f}{\partial x_1} \frac{dx_1}{d\alpha} + \frac{\partial f}{\partial x_2} \frac{dx_2}{d\alpha} + \dots + \frac{\partial f}{\partial x_n} \frac{dx_n}{d\alpha} < 0 \quad (1.9)$$

Which is also known as:

$$\frac{d\varphi}{d\alpha} = p^k \nabla f(x^k + \alpha p^k) < 0 \quad (1.10)$$

As a result of this, if  $\alpha$  is small, we are almost guaranteed to get a satisfying point [HBN10]. However, when taking the actual step, the step will also be small, leading to a slower approach to the minimizer with far more calculations necessary.

The aim of line-search is to find a value of  $\alpha$  such that we get a useful decrease in the value of the original function. In soft line-search we are satisfied with a region in which this is true and are willing to pick any point within this region. The interval in which it does so can be defined by the following conditions, known as the Wolfe-conditions [JN00]:

$$\varphi(\alpha) \leq \varphi(0) + c_1 \frac{d\varphi(0)}{d\alpha} \quad (1.11)$$

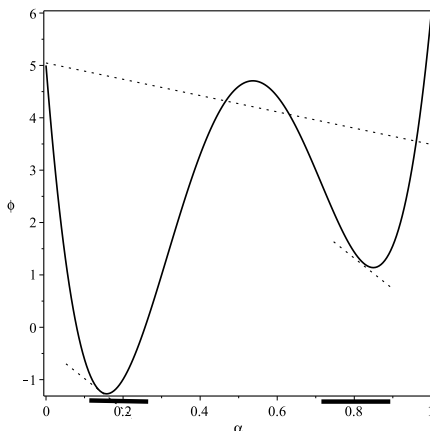
$$\frac{d\varphi(\alpha)}{d\alpha} \geq c_2 \frac{d\varphi(0)}{d\alpha} \quad (1.12)$$

Where  $0 < c_1 < 0.5$  and  $c_1 < c_2 < 1$ . Usually,  $c_1$  is chosen small and  $c_2$  is chosen very large. Typical values could be:  $c_1 = 10^{-4}$ ,  $c_2 = 0.9$ .

The first condition, also known as the Armijo rule, ensures that the function value has decreased enough for the new choice of  $\alpha$  to make the step "worth it".

The second condition, known as the curvature condition, ensures that the slope at the new point has been reduced sufficiently. This hopes to ensure that the next iteration does not start from an area with a very steep slope, which could otherwise cause the iteration sequence to shoot very far off the target.

The following image displays a rough estimate of an example of what acceptable ranges could be a soft line-search algorithm. The thick black marker indicates acceptable ranges.



**Figure 1.2:** Example of line-search acceptable ranges

A possible algorithm to do this would be[HBN10]:

1.  $\alpha := 0, k := 0$ . If  $\frac{d\varphi(0)}{d\alpha} \geq 0$ , stop immediately - 1st order necessary conditions were already met or the step  $p$  was not determined correctly.
2. Define  $\alpha_{max}$  so be a user-specified maximum line-search length, and  $k_{max}$  to be the largest amount of iterations tolerable.
3.  $a := 0, b := \min(1, \alpha_{max})$ .
4. While  $k < k_{max}$ 
  - (a) Increment  $k$ .
  - (b) If  $\varphi(\alpha) < \varphi(0) + c_1 \frac{d\varphi(0)}{d\alpha}$ , set  $a = b$ .
    - i. If  $\frac{d\varphi(b)}{d\alpha} < c_2 \frac{d\varphi(0)}{d\alpha}$ , set  $b = \min(2b, \alpha_{max})$ .
    - ii. Otherwise, terminate the while loop.
  - (c) If the above was not true, check if  $a = 0$  and  $\frac{d\varphi(b)}{d\alpha} < 0$ . If it is, set  $b = b/10$ .
  - (d) In all other cases, exit the while loop.
5. Set  $\alpha = b$  and terminate the algorithm entirely if: ( $a > 0$  and curvature condition is fulfilled) or ( $b \geq \alpha_{max}$  and the curvature condition is not fulfilled).
6. While  $k < k_{max}$



- (a) Increment  $k$ .
  - (b) Refine  $\alpha, a, b$  using a following method.
  - (c) Exit this loop immediately if the Wolfe conditions are satisfied.
7. If, in spite of everything, the new location still has a higher function value, set  $\alpha = 0$ .

A possible refinement method is to create a parabola with similar characteristics and finding the minimum value of it, such as this one:

$$\psi(t) = \varphi(a) + \frac{d\varphi(a)}{d\alpha}(t - a) + c(t - a)^2$$

Where  $c = \frac{\varphi(b) - \varphi(a) - (b-a)\frac{d\varphi(a)}{d\alpha}}{(b-a)^2}$ . A possible way to do this is with the following algorithm[HBN10]:

1.  $D := b - a$ .  $c := \frac{\varphi(b) - \varphi(a) - D\frac{d\varphi(a)}{d\alpha}}{D^2}$ .
2. If  $c > 0$ , set  $\alpha = \frac{a - \frac{d\varphi(a)}{d\alpha}}{2c}$ , because the parabola is a "frown".  
 $\alpha = \min(\max(\alpha, a + 0.1D), b - 0.1D)$  to ensure no divergence.
3. Otherwise, set  $\alpha = (a + b)/2$ .
4. If the armijo-condition is satisfied by this  $\alpha$ , set  $a = \alpha$ , otherwise set  $b = \alpha$ .  
 Return  $a$  and  $b$ .

Another method entirely is to use exact line-search, which is designed to result in a step-length close to a value of  $\alpha$  such that  $\varphi(\alpha)$  is minimized. The algorithm to do this is similar to soft line-search, but the main differences are that, in the first loop,  $a$  is only updated when (but before)  $b$  is. In addition, the condition for stopping the final loop in that algorithm is changed to:

$$\left( \left| \frac{d\varphi(\alpha)}{d\alpha} \right| \leq \tau \left| \frac{d\varphi(0)}{d\alpha} \right| \right) \text{ or } (b - a \leq \varepsilon)$$

The good thing about this approach is that, in theory, it should result in the following relation (on successful runs, it approximately does):

$$\nabla f(x^k + \alpha p^k) \perp p^k \tag{1.13}$$

The bad thing about it is that it takes much longer to compute it, and thus it has fallen out of favour in recent years[HBN10].

## 1.4 Using a trust region with line search

It is also possible to combine some of the methods of the line search and trust region methods. A trust region  $\mathcal{T}$  is defined as such:[HBN10]

$$\mathcal{T} = \{h \mid \|h\| \leq \Delta\}, \quad \Delta > 0 \quad (1.14)$$

The goal here is to adjust  $\Delta$  in such a way that it represents a "trust-worthy" spherical area with the radius  $\Delta$ , in order to reduce the computation time of the line search method.

One way to do this is to first normalize the step if the step is too long. Once this is done, a normal line search, as described above, is performed. The line search gives a step length  $\alpha$  which tells how far it was deemed beneficial to in the direction of the normalized step.

- If less than the full step was taken,  $\alpha < 1$ , reduce the trust region by  $\rho_1$ .
- If the slope was too steep at the final point of the line search, the trust region may be too small. Increase it by a factor of  $\rho_2$ .
- At the end of each iteration, make sure that the trust region is larger than the stopping criteria for too small step length.

Where  $0 < \rho_1 < 1$  and  $\rho_2 > 1$ .

By doing this, the step from each iteration is fitted into a trust region. The better the steps have been in the past, the larger steps we take, and vice versa, which can save computation time for each iteration's line search.

## 1.5 Quasi-Newton BFGS method

A quasi-newton optimization method is a method which uses iterates to arrive at a minimal function value in similar vein to the steepest descent method or

Newton's method for optimization[JN00].

The problem with Newton's method when used for optimization is that it requires the Hessian of the function that is to be optimized. The Hessian of such a function can be extremely difficult to compute and may consist of unreasonably long expressions that require a lot of computer operations to calculate explicitly, let alone find. The idea of a Quasi-Newton method is to approximate the Hessian well enough to be able to use what is essentially Newton's Method without requiring the user to explicitly supply the Hessian.[JN00]

One of these is the BFGS method, named after Broyden, Fletcher, Goldfarb, and Shanno who invented it.

To derive it, we can start with the quadratic model of the objective function at the current iterate  $x_k$ [JN00]

$$m_k(p^k) = f^k + \nabla f^k p^k + \frac{1}{2} p^k B^k p^k \quad (1.15)$$

Where  $B^k$  is a positive definite approximation that is updated at every iteration  $k$ . The minimizer of the above model can be written as:

$$p^k = -(B^k)^{-1} \nabla f^k \quad (1.16)$$

Instead of computing  $B^k$  at every iteration, Davidson proposed updating it using the next iterate  $x^{k+1}$  using the following quadratic model[JN00]:

$$m_{k+1}(p^k) = f^{k+1} + \nabla f^{k+1} p^k + \frac{1}{2} p^k B^{k+1} p^k \quad (1.17)$$

We require of of  $B^{k+1}$ , based on the Wolfe conditions used for line-searching, that:

$$\nabla m_{k+1}(p^k) = \nabla f^{k+1} - \alpha^k B^{k+1} p^k = \nabla f^k \quad (1.18)$$

By rearrangement we get:

$$\alpha^k B^{k+1} \alpha^k p^k = \nabla f^{k+1} - \nabla f^k \quad (1.19)$$

This is known as the secant equation. At this point, to simplify the notation, the following vectors are defined:

$$h^k = \alpha^k p^k, \quad y^k = \nabla f^{k+1} - \nabla f^k \quad (1.20)$$

Thus the secant equation becomes, with the second version being particularly important for BFGS:

$$B^{k+1} h^k = y^k \Leftrightarrow H^{k+1} y^k = h^k \quad (1.21)$$

Where  $H^k = \frac{1}{B^k}$ .

The secant equation requires that  $H^{k+1}$  maps  $y^k$  into  $h^k$ . For this to be possible, they must satisfy the curvature condition:

$$h^k y^k > 0 \quad (1.22)$$

In order to make sure that this condition is fulfilled, it needs to be enforced. By imposing the restrictions on the line search mentioned in the previous chapter, the curvature condition is guaranteed to hold [JN00] because of the curvature condition imposed as part of the Wolfe Conditions:

$$y^k h^k \geq (c_2 - 1) \alpha^k \nabla f^k p^k \quad (1.23)$$

This admits for a number of different solutions. In order to choose a specific unique solution, an additional condition of closeness is specified. Denoting the new iterate simply  $H$ , we want the new iterate to be positive definite and symmetric, and we want the minimal distance between the two, so the following optimization problem is solved.

$$\min_H \|H - H^k\|$$

$$\text{subject to} \quad H = H^T, \quad H y^k = s^k$$

This problem has the unique solution which is precisely the BFGS updating method[JN00]:

$$H^{k+1} = \left(I - \frac{1}{y^k h^k} h^k y^k\right) H^k \left(I - \frac{1}{y^k h^k} y^k h^k\right) + \frac{1}{y^k h^k} h^k h^k \quad (1.24)$$

The only remaining problem is figuring out where to start. Since this is only an update formula - what does it update from the first time?

There appears to be no "magic trick" for this, but starting with an evaluation of the true  $H_0$  would work well. However, there is a useful heuristic which is to take a steepest descent direction for the first step and, afterwards, set [JN00]:

$$H^0 = \frac{y^k h^k}{y^k y^k} I \quad (1.25)$$

Thus, the entire algorithm is now:

1. Begin with a starting guess  $x^0$  on the at least locally convex function  $f \in \mathcal{C}^2 : \mathbb{R}^n \rightarrow \mathbb{R}$  where  $n$  is the number of input dimensions in the function.
2. Set  $H^0 = I$ .
3. Sample the function value  $f(x^0)$ , the gradient  $\nabla f(x^0)$ .
4. While the first order optimality conditions are not yet met:
  - (a) Find  $p^k = -\frac{\nabla f(x^k)}{\nabla^2 f(x^k)}$
  - (b) Use the line search technique to find a value  $\alpha^k$  which satisfies the line-search conditions.
  - (c) Take the step:  $x^{k+1} = x^k + \alpha^k p^k$
  - (d) Define  $h^k$  and  $y^k$  in accordance with (19). Check the secant equation.
  - (e) Compute  $H^{k+1}$  in accordance with (24). If this is the first iteration, use (25).
  - (f) Increment  $k$ .
  - (g) Sample the function value  $f(x^k)$ , the gradient  $\nabla f(x^k)$ .

A final thing that is very important to note: As this algorithm updates the inverse Hessian, that means it is no longer needed to solve a linear system. Instead, a simple multiplication of a matrix and a vector is all that is needed. This is huge for computer performance. [\[Mey00\]](#)[\[Net12\]](#)[\[ABB+99\]](#)

## 1.6 Finite Difference

In many cases even differentiating a function once, let alone twice, can sometimes be difficult, even if it can be proven that the function in question is continuously differentiable. A forward differentiation method is included and can be used. Forward difference approximation is just the definition of derivative without taking the limit as  $h$  approaches 0. Rather,  $h$  is set to a specific, small value, which causes an error of size  $O(h)$  [\[Sch07\]](#).

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h) \tag{1.26}$$

It should be noted that setting  $h$  to a negative number immediately causes backwards difference approximation.

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h) \tag{1.27}$$

The overall result of using both a Quasi-Newton method as well as finite difference is that only the function  $f$  itself needs to be given explicitly, although it is still required that  $f \in \mathcal{C}^2$ . This can lead to very large reductions in the work required by the user.

## CHAPTER 2

# Installing BLAS and LAPACK in Ubuntu

---

Switching gears to the more practical implementation, it will be necessary to install BLAS in order to use the optimization toolbox, because the optimization toolbox is written using BLAS libraries for multiplications involving vectors and matrices.

This guide and software is written for use primarily with Ubuntu 12.04 (but it should work on many POSIX-compliant systems), and will take you step-by-step to getting the solution running on your system. If you are not using Ubuntu, the setup steps may differ.

## 2.1 Installing Eclipse, BLAS & LAPACK

This guide assumes we are using Ubuntu. We are going to be using ATLAS, which is a mostly hardware-independent and very efficient variety of the BLAS and (some of the) LAPACK packages. However it does not implement all LAPACK functions, so LAPACK is installed separately.[\[Net12\]](#)

We also need a good IDE, like Eclipse. Alternatively, it is possible to use a text

editor such as gedit, VIM or emacs. For Eclipse+ATLAS, open a terminal and execute:

```
1 apt-get install eclipse-cdt libatlas-dev liblapack-dev build-essential
```



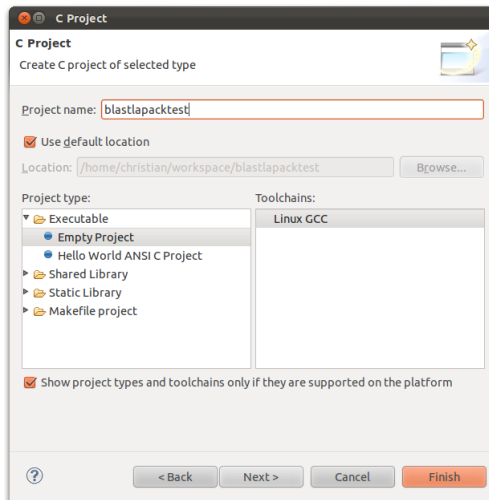
**Figure 2.1:** Installing Eclipse, GCC, BLAS, LAPACK

Wait for Ubuntu's package manager to finish the process. Once it is finished, you may close the terminal. Eclipse for C, a C compiler, BLAS and LAPACK are now installed.

## 2.2 Creating a new project

1. This is only relevant if you use Eclipse.
2. Go into File -> New -> C project.
3. Type in a project name, and select Empty Project. Select the toolchain Linux GCC. Then press Finish.





**Figure 2.2:** Creating a new project in Eclipse-cdt

4. Now you should include a new file. Go into File -> New -> Source File
5. Call the file main.c, as the tool-chain will always attempt to compile this file at first.

## 2.3 Importing, using linking BLAS & LAPACK

In order to include BLAS, simply type `#include <cblas.h>` in the preamble of whichever file you need it in. Then open the project properties.

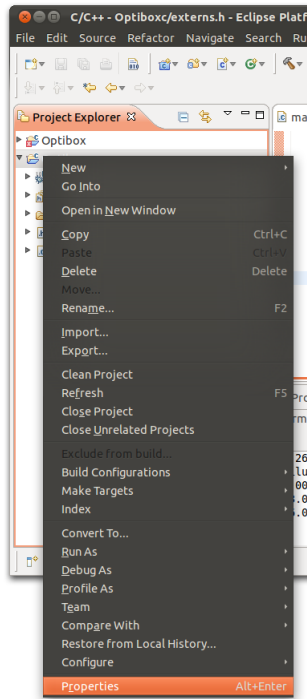


Figure 2.3: Accessing Project Properties

Expand GCC Build, then click on Settings. In the new selection panel under Tool Settings, go into the GCC linker and select Libraries, and add the libraries "m", "lapack" and "blas" without the quotes. Do not add any Library Paths. In order to use the library tests, also include rt.h. On Mac OS X, add Accelerate.framework.

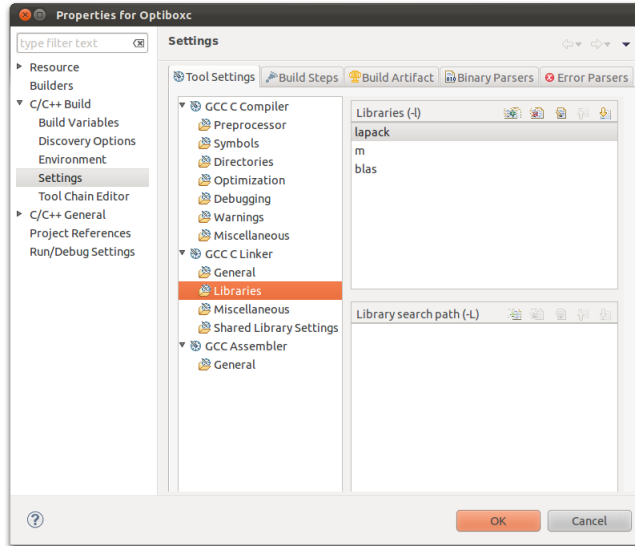


Figure 2.4: Linking the compiler to BLAS and LAPACK

The equivalent can be performed by calling "gcc main.c -O3 -o blastest.out -lblas -llapack -lm" in the terminal without the quotes.

## 2.4 First Matrix-multiplications

### 2.4.1 Matrix-Vector

In order to perform a Matrix-Matrix multiplication, I need to use the blas library. For this purpose, the cblas.h library is used. Column Major order is preferred since this saves computer resources, given that the functions I are using were written in FORTRAN, and thus the matrices use FORTRAN order, or Column Major.

All of the functions have logical names to them: Generally, they take a form starting with the name of the library, then underscore, ending with the function name as explained on netlib.[\[Net12\]](#) They are put into 3 categories: vector-vector operations are level 1, matrix-vector operations are level 2, matrix-matrix operations are level 3.

In order to do a matrix-vector multiplication, which is a level 2 subroutine, something like this

```
1 cblas_dgemv(CblasColMajor, CblasNoTrans, A m, A n, alpha, A, A n, x, numcols x
  , beta, y, numcols y);
```

It will perform the following operation:

$$y = \alpha \cdot A \cdot x + \beta \cdot y$$

Here is an example of using it:

```
1 //Some actual BLAS-testing!
2 int Mlrow = 3;
3 double Ml[] = {
4     3, 1, 3,
5     1, 5, 9,
6     2, 6, 5
7 }; //Remember it's transposed!
8
9 double x[] = {-1, -1, 1};
10
11 double y[3] = {0.0};
12
13 cblas_dgemv(CblasColMajor, CblasNoTrans, 3, 3, 1.0, Ml, 3, x, 1, 0.0, y,
14     1); //matrix-vector mult
15 //y = alpha*A*x + beta*y
16 int i; //Print result
17 for (i=0; i<3; ++i) printf("%5.3f\n", y[i]);
```

It gives the result:

```
1 -2.000
2 0.000
3 -7.000
```

## 2.4.2 Matrix-Matrix

A level 3 function will generally perform the following operation:

$$y = \alpha \cdot A \cdot B + \beta \cdot y$$

And so the input format I use for the general case is:

```
1 cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, A[m], A[n], alpha, A,  
  numcols A, B, numcols B, beta, y, cols of output);
```

Here is an example:

```
1 int main()  
2 {  
3     int M1row = 3;  
4     double M1[] = {  
5         3, 1, 3,  
6         1, 5, 9,  
7         2, 6, 5  
8     }; //Remember it's transposed!  
9  
10    int M2col = 3;  
11    double M2[] = {  
12        1, 2, 3,  
13        4, 5, 6,  
14        7, 8, 9  
15    }; //Remember it's transposed!  
16  
17    double y[3] = {0.0};  
18  
19    int M3row = 3;  
20    double matrixout[9] = {0.0};  
21  
22    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, 3, 3, 3, 1.0, M1,  
23               M1row, M2, M2col, 0.0, matrixout, M3row); //matrix-matrix mult  
24  
25    int i;  
26    for (i=0; i<3; ++i) printf("%5.3f\n", y[i]);  
27    printf("\n");
```

Produces the following output:

```
1  11.000 29.000 36.000  
2  29.000 65.000 87.000  
3  47.000 101.000 138.000
```

Which is as expected.

## 2.5 Solving your first LA-equations

Despite the fact that the BFGS method does not need any LAPACK routines at all, working with LAPACK is still useful for further functions in the toolbox in the future.

Since there seems to be no decent, complete C wrapper for LAPACK for Linux, we'll just have to do it ourselves! Using the LAPACK documentation[[ABB+99](#)]. Before the main method, or in a separate file, you must import the proper functions into C.

Since I link to LAPACK, I can use the methods directly. There are a few things to keep in mind. In C, any FORTRAN77 function I import is appended with an underscore. This happens due to the way a FORTRAN77 library is compiled.[[Lin12](#)] Looking at the documentation I can see that dgtsv can do what I want. Here is an example of importing such a function into a more usable C format:

```

1  static long dgtsv(long N, long NRHS, double *DL, double *D, double *DU, double
    *B, long LDB)
2  {
3      extern void dgtsv_(const long *Np, const long *NRHSp, double *DL, double *
    D, double *DU, double *B, const long *LDBp, long *INFOp);
4      long info;
5      dgtsv_(&N, &NRHS, DL, D, DU, B, &LDB, &info);
6      return info;
7  }

```

This function will solve a system of linear equations in which only the diagonal, upper diagonal and lower diagonal are defined. Everything else is set to 0. Let us try to solve the system:

$$\begin{pmatrix} 3 & 6 & 0 \\ 6 & 1 & 4 \\ 0 & 4 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1 \\ 3 \\ -3 \end{pmatrix}$$

Here is an example of using it:

```

1      double z[] = { -1, 3, -3 };
2
3      int info, j;
4
5
6      double l[] = {6, 4};
7      double u[] = {6, 4};

```

```
8     double d[] = {3, 1, 3};
9
10    info = dgtsv(3, 1, 1, d, u, z, 3);
11    if (info != 0) printf("Failure with error: %d\n", info);
12
13    for (j=0; j<3; ++j) printf("%5.3f\n", z[j]);
```

It returns:

```
1  0.769
2 -0.551
3 -0.265
```

Next I wish to take a look at Cholesky factorization. Looking at the documentation, there is `dpbtrf`, which is Cholesky Factorization which takes the matrix as the first column, then the lower diagonal part of the second column, append with 0's to make the column as long as the first one, then the lower diagonal part of the next column, append with 0's and so on, which can be imported as such:

```
1  static long dpbtrf(char UPLO, long N, long KD, double* AB, long LDAB)
2  {
3      extern void dpbtrf_(char* UPLOp, long* Np, long* KDp, double* AB, long*
         LDABp, long* infop);
4      long info;
5      dpbtrf_(&UPLO, &N, &KD, AB, &LDAB, &info);
6      return info;
7  }
```

Let's say I wish to find the lower matrix such that  $L^T L = A$  for the positive definite symmetric matrix

$$A = \begin{pmatrix} 4 & -2 & -6 \\ -2 & 10 & 9 \\ -6 & 9 & 14 \end{pmatrix}$$

Then this is what I would do:

```
1 double m[] = {4, -2, -6, 10, 9, 0, 14, 0, 0};
2
3 info = dpbtrf('L', 3, 2, m, 3);
4 if (info != 0) printf("Failure with error: %d\n", info);
5
6 for (i=0; i<3; ++i)
7 {
8     for (j=0; j<3; ++j)
9         if (j > i)
10            printf(" %5.3f", 0.0f);
11         else
12            printf(" %5.3f", m[j*3+(i-j)]); //FORTRAN order...
13     printf("\n");
14 }
```

This will return the correct result:

```
1 2.000 0.000 0.000
2 -1.000 3.000 0.000
3 -3.000 2.000 1.000
```

Now that the system is set up, the real work can begin.



## CHAPTER 3

# Library documentation

---

## ucminf

Finds the minimizer to an unconstrained optimization problem given by fun() by using the Quasi-Newton BFGS method. Extra parameters may be sent using the void structure.

Call:

```
1  int opti_ucminf(void (*fun)(double *x, double *f, double *g, void *p),
    double *x, int n, double *opts, double *D, int *evalused, double *
    perfinfo, double *xiter, void *p);
```

Input:

- fun - A function handle to a function with the structure:
  - \*x - The address of an array of doubles representing the point to take the function at.
  - \*f - The address of the storage of the function evaluation

- \*g - The address of an array of doubles. Will be filled with the derivative of  $f(x)$ .
- \*p - A pointer to a void structure that can be anything.
- \*x - The starting guess  $x$  for the function. On exit, the optimal value for  $x$ .
- n - Number of variables given to the function
- \*opts - An array containing:
  - delta - Initial value for largest stepsize used by the linesearching.
  - tolg - The value which the norm of  $g$  is under when the 1st order KKT conditions are met.
  - tolx - If the stepsize is less than  $\text{tolx} * (\text{tolx} + \text{norm}(x, n, 2))$ , stop.
  - maxeval - The maximum number of tolerable iterations before stopping.
  - findiffh - Used in the finite difference approximation for  $\text{fun}()$ . If set to 0,  $g$  is expected. Warning: Using  $\text{findiffh} = 0$  and not supplying  $g$  in  $\text{fun}$  WILL result in errors.
  - Default: 1, 1e-4, 1e-8, 100, 0
- \*D - An array containing an initial estimation of the inverse Hessian to the function. Set to NULL to start with with I. On exit, a pointer to the Hessian. Warning: Points to unclaimed memory if \*D was set to NULL on exit!
- \*evalused - A pointer to an integer which will be filled with the number of evaluations. Warning: NOT ITERATIONS.
- \*perfinfo - A pointer that must point to an array of doubles of size  $6 * \text{maxeval}$ . Will contain performance information which can be printed using `printperf`.
- \*xiter - If null, nothing happens. Otherwise, contains a sequence of  $x$  for each iteration on exit. Must be preallocated to size  $n * \text{maxiter}$  (default  $n * 100$ )
- \*p - A void-pointer which can contain anything. Is passed through the function and all the way over to  $\text{fun}()$ .

Returns:

- -1 - memory allocation, error
- 0 - perfect starting guess, success
- 1 - Step size H became NaN, inf or -inf, error
- 2 - Stopped due to small derivative g, success
- 3 - Stopped due to small change in x, success
- 4 - Norm of step size became 0, success.
- 5 - Too many function evaluations, potential failure

Example usage:

```

1 //Requires C99 at least
2
3 //Standard ANSI C libraries for I/O
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <stddef.h>
7
8 //Include the library itself
9 #include <libopti.h>
10
11 #include <math.h>
12
13 void himmelblau(double* x, double* f, double* df, void *p)
14 {
15     *f = pow(x[0]*x[0] + x[1] - 11,2) + pow(x[0] + x[1]*x[1] - 7,2);
16
17     df[0] = 2*(2*x[0]*(x[0]*x[0] + x[1] - 11) + x[0] + x[1]*x[1] - 7);
18     df[1] = 2*(-11+x[0]*x[0] + x[1] + 2*x[1]*(-7 + x[0] + x[1]*x[1]));
19
20     return;
21 }
22
23 int main()
24 {
25     //Number of inputs to n, number of max iterations
26     const int maxiters = 100;
27     int n = 2;
28     double x[2] = {1, 1};
29
30     double perfinfo[maxiters*6];
31     double xinfo[n*maxiters];
32
33     double opts[] = {1.0, 0.00000001, 0.00000001, maxiters, 0};
34     int ucminfr = opti_ucminf(himmelblau, x, n, opts, NULL, NULL, perfinfo,
35                             xinfo, NULL);
36
37     int folderinfo = opti_printperf_to_folder(perfinfo, xinfo, maxiters, n, "
38         testhimmelblau");
39
40     printf("%i\n",ucminfr);
41     printf("(%f, %f)\n",x[0], x[1]);
42
43     return 0;
44 }

```

Resulted in:

```
1 Beale test function:  
2 Solve time: 0.000046  
3 Save to folder info: 0  
4 Solution return: 2
```

In other words, the program was minimized in 0.000046 seconds, and the algorithm stopped due to small size of derivative.

## line search

Calculates a step length of a step on a function, and updates  $x$ ,  $f$ ,  $g$ . Can save return performance information about the line search. Also takes options.

Call:

```
1  int opti_linesearch(void (*fun)(double *x, double *f, double *g, void *p),
    double *x, int n, double *f, double *g, double *h, double *opts, double
    *perf, void *p);
```

Input:

- fun - A function handle to a function with the structure:
  - \*x - The address of an array of doubles representing the point to take the function at.
  - \*f - The address of the storage of the function evaluation
  - \*g - The address of an array of doubles. Will be filled with the derivative of  $f(x)$ .
  - \*p - A pointer to a void structure that can be anything.
- \*x - Pointer to the variables given to the function
- n - Number of variables given to the function
- \*f - The address of the storage of the function evaluation
- \*g - The address of an array of doubles. Will be filled with the derivative of  $f(x)$ .
- \*h - The search direction, the steplength multiplier,
- \*opts- An array with a few options:
  - choice: 1 = soft linesearch, 0 = exact linesearch
  - cp1: Constant 1 timed onto derivative of  $f$  at initial point (not used if choice = 0)
  - cp2: Constant 2 timed onto derivative of  $f$  at initial point
  - maxeval: Maximum number of evaluation before quitting.
  - amax: Maximum tolerable search direction multiplier

- Default: 0, 1e-3, 1e-3, 10, 10. Giving NULL to opts will load these values.
  
- \*perf- Some simple performance information from linesearch. Contains: search direction multiplier, final slope, number of evaluations
  
  
  
  
  
  
  
  
  
  
- \*p - A pointer to a void structure that can be anything.

Returns:

- 0 - Successful return
  
  
  
  
  
  
  
  
  
  
- -1 - Dynamic memory allocation failed.
  
  
  
  
  
  
  
  
  
  
- 1 - h is not downhill or it is so large and, maxeval so small, that a better point was not found.

Example call: Used internally by `ucminf`. See appendix [A.10](#).

---

## interpolate

Finds the minimizer of a 1D parabola which is given by two points  $a$  and  $b$ .

```
1 double opti_interpolate(double *xfd, int n);
```

Input:

- $xfd$  - An array consisting of  $xfd = a, b, f(a), f(b), f'(a)$ . The function value at  $a$  and the function value at  $b$ . The derivative of the function value at  $a$ .
  
- $n$  - The dimension of the problem original problem within which the parabola is to be constructed.

Output: The minimizer of the parabola.

Example call: See appendix [A.6](#). Line search uses it for refining the interval.

## findiff

Finds  $f$  and  $g$  directly if  $h == 0$ . Otherwise, approximates  $g$  with forward difference approximation using an  $O(n+1)$  algorithm. If  $h$  is negative, uses backward difference.

Call:

```
1  int opti_findiff(void (*fun)(double *x, double *f, double *g, void *p),
    double *x, int n, double *f, double *g, double h, void *p);
```

Input:

- $fun$  - A function handle to a function with the structure:
  - $*x$  - The address of an array of doubles representing the point to take the function at.
  - $*f$  - The address of the storage of the function evaluation
  - $*g$  - The address of an array of doubles. Will be filled with the derivative of  $f(x)$ .
  - $*p$  - A pointer to a void structure that can be anything.
- $*x$  - Pointer to the variables given to the function
- $n$  - Number of variables given to the function
- $*f$  - The address of the storage of the function evaluation
- $*g$  - The address of an array of doubles. Will be filled with the derivative of  $f(x)$ .
- $h$  - The stepsize of the forward approximation. Can be any real number. If  $h$  is 0,  $g$  is expected to be handed directly.
- $*p$  - A pointer to a void structure that can be anything.

Returns:

- 0 - Successfully returned the function and its derivative.
- -1 - Dynamic memory allocation failed.



---

## printperf\_tofile

Prints performance information to a file in such a way that the file has the iterations listed among the columns:

- Number of evaluations of f done in the iteration.
- The value of f at the iteration
- The norm of the derivative of the function at the iteration
- The linesearch limit delta at the value of the iteration
- The scaling factor for the step as given by the linesearch
- The slope of the function at the point in the direction of the step

Call:

```
1 int opti_printperf_tofile(double* perf, int maxiters, char* filename);
```

Input:

- \*perf - The performance array given by ucminf
- maxiters - The maximum allowed iterations. Must be the same number passed to opts. Set to negative number to default to 100.
- filename - A string containing the filename.

Returns:

- 0 - Successfully saved the data to file.
- -2 - Failed to open/write file. Nothing is saved.

## printperf\_tofolder

Prints performance information to a specified folder at the location the program is run. Makes 7 subfiles named:

- evals.txt - Number of evaluations of f done in the iteration.
- fun.txt - The value of f at the iteration
- ng.txt - The norm of the derivative of the function at the iteration
- delta.txt - The linesearch limit delta at the value of the iteration
- am.txt - The scaling factor for the step as given by the linesearch
- finalslope.txt - The slope of the function at the point in the direction of the step
- x.txt - The contents of x at each iteration

Call:

```
1 int opti_printperf_tofolder(double* perf, double* xiter, int maxiters, int n,
   char* foldername);
```

Input:

- \*perf - The performance array given by ucminf
- maxiters - The maximum allowed iterations. Must be the same number passed to opts. Set to negative number to default to 100.
- foldername - A string containing the foldername from the location your program is run.

Returns:

- 0 - Successfully saved the data to file.
- -2 - Failed to open/write file. Nothing is saved.
- -3 - File existed with name of folder to be created. Nothing saved.
- -4 - Folder could not be created. Check permissions or drive

## 3.1 Usage example

In this example we seek to solve the famous Rosenbrock problem[Ros60].

The Rosenbrock Function is a non-convex function invented by Howard H. Rosenbrock in 1960. It has a long, narrow parabolic-shaped valley with only a single minimum. To find the valley is easy, but to find the minimum is quite difficult.

$$f(x) = (1 - x_1)^2 + 100 (x_2 - x_1^2)^2 \quad (3.1)$$

In order to do this, the library needs to be added onto the system. The compiled form of the library is called libopti.so.

If the library is not already compiled, a makefile is provided inside the Release folder of the project folder which compiles it. It is required that ATLAS and GCC is installed for this to work. This will build the shared object libopti.so.

To include the object in your own program, GCC must be told to link with it, and where the object is.

- If the object is in /usr/bin/ or /usr/local/bin then GCC only needs to be told that the library should be added using the -lopti argument.
- If the object elsewhere, GCC also needs to be told where the library is located. In addition to the above, use the -L/path/name/to/library argument.

Next, the header file of the library needs to be included in your project. This is done by adding the line, assuming libopti.h is in the same folder as your code.

```
1 #include "libopti.h"
```

If libopti.h is in /usr/include or /usr/local/include, you can type:

```
1 #include <libopti.h>
```

in the file. Afterwards, the library can be freely used, such as with this example minimizing the Rosenbrock function.

```
1 //Requires C99 at least
2
3 //Standard ANSI C libraries for I/O
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <stddef.h>
7
8 //Include the library itself
9 #include <libopti.h>
10
11 #include <math.h>
12
13 void rosenbrock(double* x, double* f, double* df, void *p)
14 {
15     *f = pow(1 - x[0],2) + 100*pow(x[1] - x[0]*x[0],2);
16
17     df[0] = -2 + 2*x[0] - 400*(x[1] - x[0]*x[0])*x[0];
18     df[1] = 200*x[1] - 200*x[0]*x[0];
19
20     return;
21 }
22
23 int main()
24 {
25     //Number of inputs to n, number of max iterations
26     int n = 2;
27     double x[2] = {4, 2};
28
29     double opts[] = {1.0, 0.0001, 0.00000001, 100, 0};
30     int ucminfr = opti_ucminf(rosenbrock, x, n, opts, NULL, NULL, NULL, NULL,
31                             NULL);
32
33     printf("%i\n",ucminfr);
34     printf("(%.1f, %.1f)\n",x[0], x[1]);
35
36     return 0;
37 }
```

Resulting in the following output when run:

```
1 2
2 (1.000000, 0.999999)
```

## CHAPTER 4

# Mixing FORTRAN and C

---

It may also be useful to be able to run the library from multiple different programming languages. Therefore, it is going to be essential to be able to run this library from multiple different languages.

Since C++ and Objective C are supersets of C, the ability to call any C function within them is given for free.

However, FORTRAN needs to be implemented explicitly.

On the next page is a comparison between the different datatypes in FORTRAN and C, respectively.[\[Lin12\]](#)

A subroutine in FORTRAN is basically the same as a function call in C, but it doesn't have a return type, which in C can be represented as the return type void. All FORTRAN functions have their input arguments passed by reference. In other words, we must give it an address which points to the location of our variable rather than the variable itself.

It is important to remember that arrays already get passed by reference to C programs, and as a result it is not necessary to use the address-of symbol when working with arrays.

byte	unsigned char
integer*2	short
integer	long
integer A(m,n)	int A(n,m)
logical	int
real	float
real*8	double
real*16	real*16
complex z	structfloat realnum; float imagnum; z;
double complex z	structdouble realnum; double imagnum; z;
character	char
character str(m)	char* str[m]

**Table 4.1:** Table of data types in FORTRAN and C

In addition, all subroutine names must be written in lower case letters because a compiled FORTRAN program contains only lowercase letters as they are all converted. In addition, it will append an underscore at the end of the function name.[\[Lin12\]](#)

This means that these two function calls are equivalent:

```
CALL MMADD(A, B, m, n)
```

and

```
mmadd_(A, B, &m, &n);
```

And finally, C arrays are in row major, FORTRAN arrays are in column major. This means that for example the following array:[\[Lin12\]](#)

```
double A[2][3] = {{a, b, c}, {d, e, f}};
```

Which can be interpreted as a flat array, is read by C programs like this:

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

But writing the basically equivalent statement in FORTRAN sees it like this:

$$\begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$$

To call a function from C there are a set of things that need to be done.

Make sure the correct software installed. For this, `build-essential` and `fort77` are vital. They can be installed in Ubuntu with:

```
1 sudo apt-get install build-essential fort77.
```

Write the function. This will assume general knowledge of FORTRAN 77. Perhaps the function is already written. The function must have the inputs specified by the relevant optimization procedure. Remember that everything is passed to FORTRAN subroutines by reference, so normal header declarations will ensure compliance with the requirement for pointers.

Compile the subroutines(s) into a library using the following command, making sure you are in the same directory you saved the compiled file(s):

```
1 f77 -c filename.F
```

Generate a library file by collecting all the subroutines in a single file. To do this, type the following command, remaining in the same folder:

```
1 ar rcv libmylib.a *.o
```

Generate an overview of the included subroutines with the following command:

```
1 ranlib libmylib.a
```

Over in C, add a new library: `mylib` (The name of the library file created minus the `lib` in front and the appended `.a`) Next, add a library search path which is exactly the path where you saved the library.

Create a C wrapper for the function and make it return the whichever type you wish by defining e.g. (Addition of two matrices  $A^{m \times n}$  and  $B^{m \times n}$ ):

```
1 static void mmadd(double* a, double* b, long m, long n)
2 {
3     extern void mmadd_(double* a, double* b, long* mp, long* np);
4     mmadd_(a, b, &m, &n);
5     return;
6 }
```





## CHAPTER 5

# Using the library in ...

---

This library also comes with bindings for several popular languages. Using the knowledge given above about how to work with FORTRAN and C together, the below lists documentation and examples of how to use it.

## 5.1 FORTRAN

It should be noted that while it is possible to code problems in FORTRAN and import those into C, and then use libopti to optimize, the reverse is not possible up until FORTRAN 2003. Since FORTRAN 2003 was not in the discussion above and very few people use the newer versions of FORTRAN, this problem is placed outside the scope of this report.

However, it is still possible to perform an optimization as part of a MATLAB program, and this is how.

Given the project is set up in C as explained in section 5, the following can be done.

Make an example of Rosenbrock coded in FORTRAN with the appropriate header for use with the UCMINF method.

```

1      SUBROUTINE ROSENBROCKF(X, F, DF)
2      c =====PARAMETERS=====
3      c Scalar arguments:
4          INTEGER N
5          DOUBLE PRECISION X(N), F, DF(N)
6
7      c Written by: Christian Wichmann Moesgaard
8      c This FORTRAN77 program is specifically for demonstration purposes.
9      c This implements the Rosenbrock function and is used for an example
10     c demonstrating how to optimize FORTRAN functions with the optimization
11     c toolbox.
12         N = 2
13         F = (1-X(1))**2 + 100*(x(2)-x(1)**2)**2
14
15         DF(1) = -2 + 2*X(1) - 400*(X(2)-X(1)**2)*X(1)
16         DF(2) = 200*X(2) - 200*(X(1))**2
17
18     RETURN
19     END

```

Then, compile the subroutine as demonstrated in chapter 6.1 and include it, then create the following wrapper function in the C program:

```

1 void rosenbrockf(double* x, double* f, double* df)
2 {
3     extern void rosenbrockf_(double* x, double* f, double* df);
4     rosenbrockf_(x, f, df);
5 }

```

And finally the minimum may be found using the same method as in section 5, replacing rosenbrock inside the ucminf call with rosenbrockf.

Doing the above should correctly return the result  $x \approx (1, 1)$ .

Furthermore, it is possible to make this function return something back to FORTRAN. The general idea here is to:

1. Program a FORTRAN model
2. Compile it, then load it into C as above.
3. Use it in a C function which takes the inputs as desired and compile it, remembering to respect the rules for interoperability.
4. Use this function in FORTRAN.

For example, to optimize the above subroutine, one could do the following:

```

1  //Requires C99 at least
2
3  //Standard ANSI C libraries for I/O
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <stddef.h>
7
8  //Include the library itself
9  #include <libopti.h>
10
11 #include <math.h>
12
13 void rosenbrockf(double* x, double* f, double* df)
14 {
15     extern void rosenbrockf_(double* x, double* f, double* df);
16     rosenbrockf_(x, f, df);
17 }
18
19 void optirosenbrock_(double *x, double *np)
20 {
21     int n = *np;
22
23     double opts[] = {1.0, 0.0001, 0.00000001, 100, 0};
24     opti_ucminf(rosenbrock, x, n, opts, NULL, NULL, NULL, NULL, NULL);
25 }

```

Compiling this function, the Rosenbrock function, which was originally written in FORTRAN, can now be optimized by libopti and the optimized values can be returned back to FORTRAN again using the following:

```

1      SUBROUTINE OPTIMBROCK(X, F, DF)
2  c =====PARAMETERS=====
3  c Scalar arguments:
4      INTEGER N
5      DOUBLE PRECISION X(N), F, DF(N)
6
7      N = 2
8      X(1) = 3
9      X(2) = 2
10
11     CALL OPTIROSENBROCK(X, N)
12
13     CALL ROSENBROCKF(X, F, DF)
14
15     RETURN
16     END

```

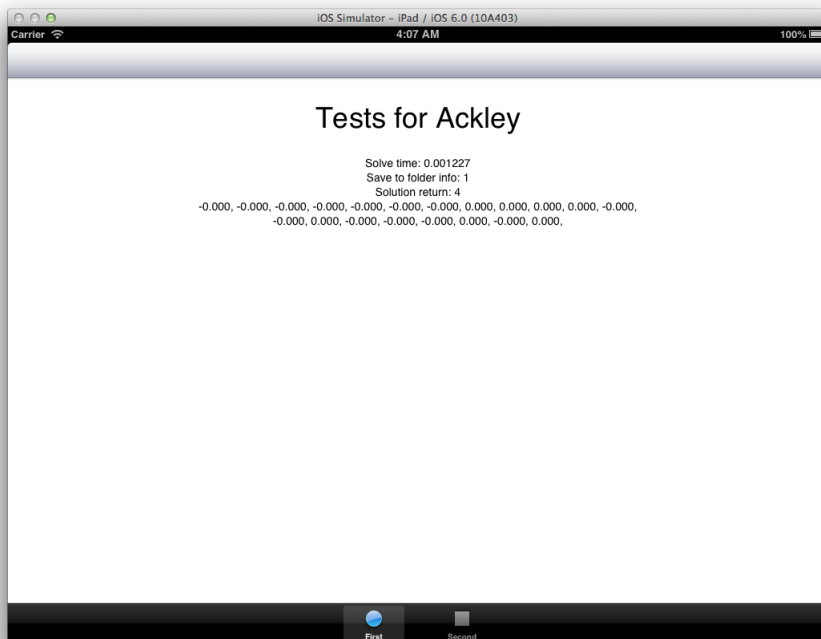
## 5.2 Mac OS X and iOS

The library is already built with OS X in mind, and will correctly attempt to use Accelerate instead of ATLAS. The test-cases will use MACH functions instead of POSIX functions where appropriate.

The library can be fully compiled as part of an Objective C application. This is because Objective C is an extension of C.[\[Inc12a\]](#)

This also implies that the library works on an iPhone or iPad, since it uses Apple's built-in Accelerate framework, which is available since iOS 4.0.[\[Inc12b\]](#)

Here is an image of an iPad Simulator having correctly solved a 20-variable Ackley optimization problem from pseudo-random numbers between -3 and 3 (more on the Ackley optimization problem in the next chapter):



**Figure 5.1:** Solution to Ackley problem on iPad

## CHAPTER 6

# Performance testing

---

For the performance testing, 6 different functions have been selected. They were all found on the the webpage: [\[Hed12a\]](#) Except Himmelblau, which was acquired through CampusNet. All of these functions are problems of the form:

$$\min_x f(x)$$

Where  $f(x) \in \mathcal{C}^2 : \mathbb{R}^n \rightarrow \mathbb{R}$ .

These 6 functions will be run through 4 different and common optimization software packages as well as my own. These software packages are:

1. IMM Optibox UCMINF, latest version\*
2. Maple 16 Optimization Toolbox using a the function "NLPsolve" internal to the Optimization Assistant, latest version\*
3. MATLAB's Optimization Toolbox using FMINUNC, latest version\*
4. minlbfgs for C++ version 2.6.0

\*as of this writing.

Each problem will be run 10 times and the average time will be calculated, except in cases where the computation time is so long that it is inconceivable that this algorithm will win out.

The performance of these software packages will be compared with the performance of the version of UCMINF made alongside this report, and their computation time will be measured in real time (seconds) on the following set-up:

- Intel Core i7-950 @ 3.07 GHz (4 cores, hyperthreading)
- 8GB PC3-12800 CL9 RAM
- Ubuntu 12.10 64-bit
- Using ATLAS, GCC 4.6.3, Optimize level 3

However, before one even gets to that, there's an important thing to note about licensing.

1. IMM Optibox is developed internally at DTU and is, as such, accessible to projects within DTU. However, it builds upon MATLAB, which is an expensive piece of proprietary software. The source code is readily visible. The dependence on MATLAB means that it is using a scripting language and thus cannot be used in embedded devices due to restrictions. The software does run on GNU Octave, but issues arise with porting to iOS or Android or similar.
2. The Maple 16 optimization algorithms are proprietary, and build upon the proprietary piece of software, Maple. The same problems apply as with MATLAB.
3. MATLAB's Optimization Toolbox is proprietary software built for proprietary MATLAB. It features DRM and is, like MATLAB, not suitable for embedded devices.
4. minlbfgs is free software, but it is also copyleft under the GNU GPLv2. It does offer dual licensing, though, but that is expensive.[\[ALG12a\]](#)

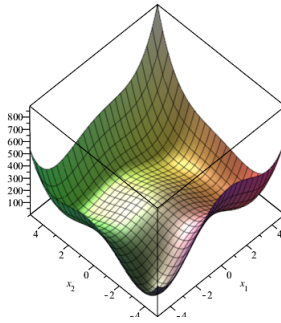
This is worth keeping in mind before even looking at performance comparisons between libopti and the other solutions.

## 6.1 Himmelblau

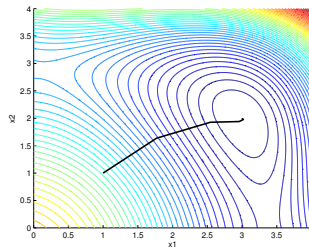
The first problem that will be looked at is Himmelblau's function. This problem is famous for its relative simplicity, making it easy to understand for students. However, the 1st and especially 2nd hand derivatives become somewhat complicated. It is named after David Mautner Himmelblau who introduced it in 1972. [Him72]

$$f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \quad (6.1)$$

Here is a plot of the relevant area of the function:



**Figure 6.1:** 4 global minima at  $x \approx \{(3, 2), (-2.8, 3.1), (-3.8, -3.3), (3.6, -1.8)\}$

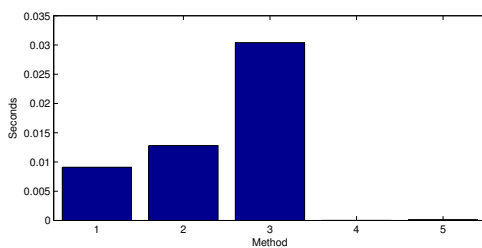


**Figure 6.2:** libopti's path

First, a comparison of the CPU times, where the starting guess is (1, 1) and the algorithm converges towards the minimum at (3, 2). All times are measured in seconds.

Attempt	UCMINF	FMINUNC	Maple 16	minlbfgs	libopti
1	0.0121	0.0114	0.0314	0.000029	0.000072
2	0.0097	0.0126	0.0302	0.000007	0.000010
3	0.0076	0.0138	0.0296	0.000004	0.000009
4	0.0096	0.0135	0.0296	0.000004	0.000009
5	0.0075	0.0116	0.0296	0.000004	0.000009
6	0.0096	0.0135	0.0296	0.000004	0.000009
7	0.0075	0.0131	0.0296	0.000006	0.000009
8	0.0096	0.0132	0.0296	0.000006	0.000009
9	0.0077	0.0112	0.0296	0.000007	0.000009
10	0.0099	0.0144	0.0296	0.000007	0.000009
Average	0.0091	0.0128	0.0304	0.000008	0.000015

**Table 6.1:** Performance on Himmelblau



**Figure 6.3:** Average, methods are ordered in the same way as the table

The Maple and MATLAB solutions are completely out of the question. They are based on scripting languages. minlbfgs takes the lead here, but not particularly much. In fact, the difference was unnoticeable.

The code used to take these samples can be found in [Appendix C.2](#).

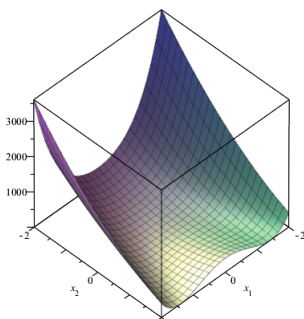


## 6.2 Rosenbrock

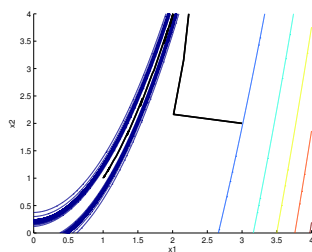
The Rosenbrock Function is a non-convex function invented by Howard H. Rosenbrock in 1960. It has a long, narrow parabolic-shaped valley with only a single minimum. To find the valley is easy, but to find the minimum is quite difficult. [Ros60]

$$f(x) = (1 - x_1)^2 + 100 (x_2 - x_1^2)^2 \quad (6.2)$$

Here is the relevant area of the function:



**Figure 6.4:** Minimum at  $x = (1, 1)$

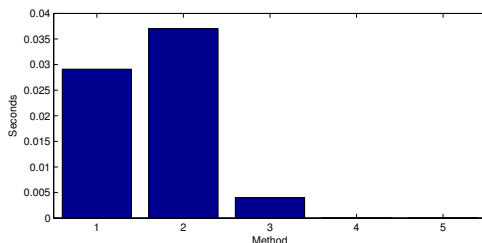


**Figure 6.5:** libopti's path

First, a comparison of the CPU times, where the starting guess is  $(3, 3)$  and the algorithm converges towards the minimum at  $(1, 1)$ . All times are measured in seconds.

Attempt	UCMINF	FMINUNC	Maple 16	minlbfgs	libopti
1	0.0550	0.0462	0.004*	0.000046	0.000134
2	0.0318	0.0416		0.000022	0.000020
3	0.0262	0.0436		0.000020	0.000018
4	0.0242	0.0422		0.000026	0.000018
5	0.0263	0.0436		0.000034	0.000018
6	0.0243	0.0439		0.000025	0.000018
7	0.0262	0.0427		0.000020	0.000018
8	0.0244	0.0441		0.000020	0.000018
9	0.0264	0.0418		0.000020	0.000018
10	0.0263	0.0435		0.000020	0.000018
Average	0.0291	0.0370	< 0.004	0.000025	0.000030

**Table 6.2:** Performance on Rosenbrock, \* = Maple could not measure lower time deltas



**Figure 6.6:** Average, methods are ordered in the same way as the table

The MATLAB solutions are completely out of the question. They are based on scripting languages. minlbfgs takes the lead here by quite a vast amount.

Something interesting happened here, however. libopti was faster in most cases, but in the first optimization it was so significantly slower that the entire average was shifted in favor of minlbfgs. These results persisted through multiple test-runs of the same code.

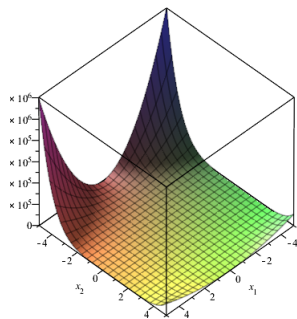
The code used to take these samples can be found in [Appendix C.3](#).

## 6.3 Beale

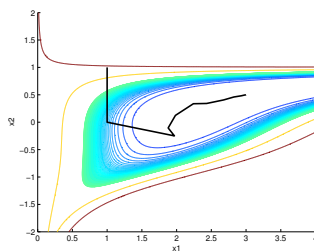
The second problem is Beale's function. This problem was introduced by E. Beale in 1958 and is famous for having a minimum in an otherwise almost flat area surrounded by very steep "hills" all around the sides. This can prove tricky for functions who do not use line search or trust regions. [Bea58]

$$f(x) = (1.5 - x_1(1 - x_2))^2 + (2.25 - x_1(1 - x_2)^2)^2 + (2.625 - x_1(1 - x_2)^3)^2 \quad (6.3)$$

Here is a plot of the relevant area of the function:



**Figure 6.7:** Minimum at  $x = (3, 0.5)$

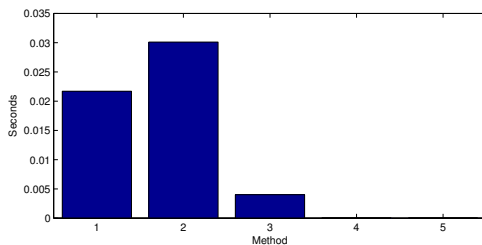


**Figure 6.8:** libopti's path

First, a comparison of the CPU times, where the starting guess is  $(4, 2)$  and the algorithm converges towards the minimum at  $(3, 0.5)$ . All times are measured in seconds.

Attempt	UCMINF	FMINUNC	Maple 16	minlbfgs	libopti
1	0.0199	0.0392	0.012001	0.000060	0.000076
2	0.0206	0.0258	0.004*	0.000024	0.000017
3	0.0230	0.0269		0.000021	0.000017
4	0.0207	0.0250		0.000021	0.000017
5	0.0203	0.0270		0.000021	0.000016
6	0.0205	0.0250		0.000021	0.000017
7	0.0224	0.0271		0.000021	0.000017
8	0.0209	0.0270		0.000021	0.000016
9	0.0270	0.0241		0.000021	0.000017
10	0.0215	0.0271		0.000020	0.000017
Average	0.0217	0.0301	0.004800	0.000025	0.000023

**Table 6.3:** Performance on Beale, \* = Maple could not measure lower time deltas



**Figure 6.9:** Average, methods are ordered in the same way as the table

Once again we see the scripting based solutions fall significantly behind. In this case, interestingly, Maple also does quite well. However, Maple does not give precise enough time measurements once again. This time, like with Rosenbrock, libopti was slowest on the first attempt, but in this case it was fast enough for the other attempts to slightly win out on the average of 10 runs.

All algorithms converged towards the minimum.

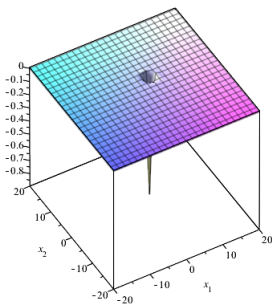
The code used to take these samples can be found in [Appendix C.4](#).

## 6.4 Easom

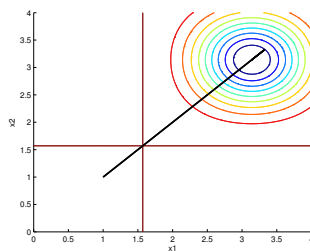
The third problem is Easom's function. This problem was introduced by E. Easom in 1990 and has a characteristic where it is very flat, except just around its one and only minimum at  $(\pi, \pi)$ , where it falls very sharply. This problem is hard to solve with poor computer precision as the algorithm may stop due to due to its first order conditions. [Eas90]

$$f(x) = -\cos(x_1) \cos(x_2) e^{-(x_1-\pi)^2 - (x_2-\pi)^2} \quad (6.4)$$

Here is a plot of the relevant area of the function:



**Figure 6.10:** Minimum at  $x = (\pi, \pi)$

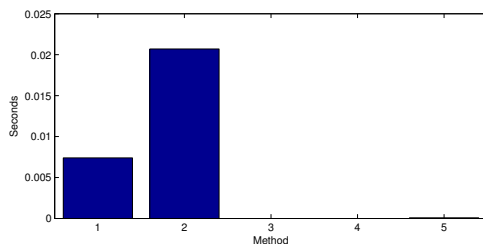


**Figure 6.11:** libopti's path

First, a comparison of the CPU times, where the starting guess is  $(4, 2)$  and the algorithm converges towards the minimum at  $(3, 0.5)$ . All times are measured in seconds.

Attempt	UCMINF	FMINUNC	Maple 16	minlbfgs	libopti
1	0.0160	0.0190			0.000689
2	0.0058	0.0212			0.000595
3	0.0057	0.0196			0.000593
4	0.0077	0.0202			0.000592
5	0.0058	0.0194			0.000613
6	0.0076	0.0251			0.000592
7	0.0058	0.0202			0.001141
8	0.0058	0.0202			0.001536
9	0.0078	0.0195			0.001287
10	0.0057	0.0221			0.001234
Average	0.0074	0.0207			0.000793

**Table 6.4:** Performance on Easom



**Figure 6.12:** Average, methods are ordered in the same way as the table

UCMINF had problems. Both `tolx` and `tolg` had to be set to at least  $10^{-8}$  before any iterations were performed, but after doing that it converged just fine. This happened because the first step had had a norm of its gradient of  $8.2617 \cdot 10^{-5}$ . The exact same happened with `libopti`. Maple could not solve the problem at all, stopping immediately.

`minlbfgs` was also unable to solve the problem, stalling at virtually any starting guess to any of the 4 stopping criteria regardless of what they were set to. It succeeded at the starting guess (3,3), but this would have been an unfair comparison.

Among the optimizers that could solve it, of which there were only 3, `libopti` was by far the fastest because it isn't based on scripting.

The code used to take these samples can be found in [Appendix C.5](#).

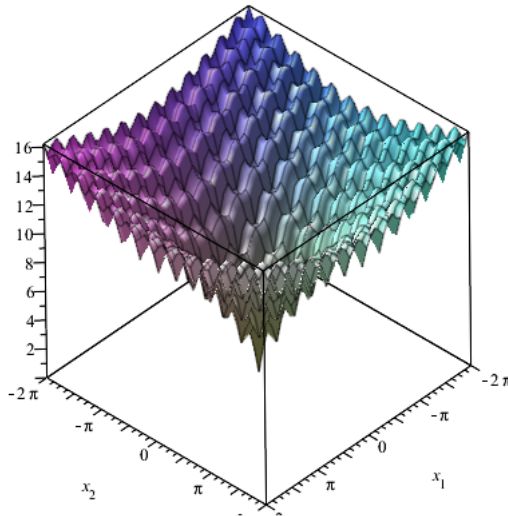
## 6.5 Ackley

Ackley's problem was defined by D. H. Ackley in 1987 in his paper about evolutionary algorithms and genetics.[Ack87] Originally, it was a 2-dimensional problem, pictured below, but was generalized by T. Bäck in 1996 to an  $n$ -dimensional problem.[Bäck96] This problem is difficult to solve due to its single global minimum but also cosine-periodic structure giving it an infinity of local minima in all directions. It is very easy for an algorithm to get "trapped" in one of these local minima.

This report deals with the relatively large problem where  $n = 5000$ .

$$f(x) = 20 + e - 20e^{-1/5} \sqrt{\frac{\sum_{i=1}^n x_i^2}{n}} - e \frac{\sum_{i=1}^n \cos(2\pi x_i)}{n} \quad (6.5)$$

Here is a plot of the relevant area of the function when  $n = 2$ :



**Figure 6.13:** Minimum at  $x_i = 0$

First, a comparison of the CPU times, where the starting guess is a sequence of 5000 random numbers between -3 and 3, and the algorithm converges towards the minimum at all zeros. All times are measured in seconds.

Attempt	UCMINF	FMINUNC	Maple 16	minlbfgs	libopti
1			3691.6		37.891406
2					37.551338
3					37.103554
4					38.802811
5					39.229581
6					38.581886
7					38.993073
8					38.768691
9					37.615502
10					36.913864
Average					38.145171

**Table 6.5:** Performance on Easom

The only algorithm that seemed able to converge correctly was libopti. UCMINF stopped itself in various valleys, but libopti did not. This happens because of the line search in UCMINF for MATLAB, which stops the algorithm because a potential precision error is detected. libopti does not check for this and carries on.

Maple took a very long time to solve the problem and converged to the wrong point, namely  $x_i \approx 0.9685$ . Even if it did converge correctly with a better starting guess, it would still take a very long time. Minlbfgs performs the same check that UCMINF does and stops, and FMINUNC does likewise.

The code used to take these samples can be found in [Appendix C.6](#).



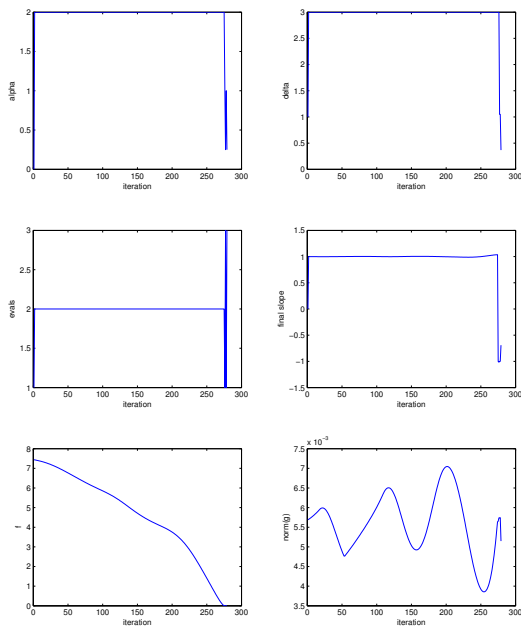


Figure 6.14: Solution of Ackley

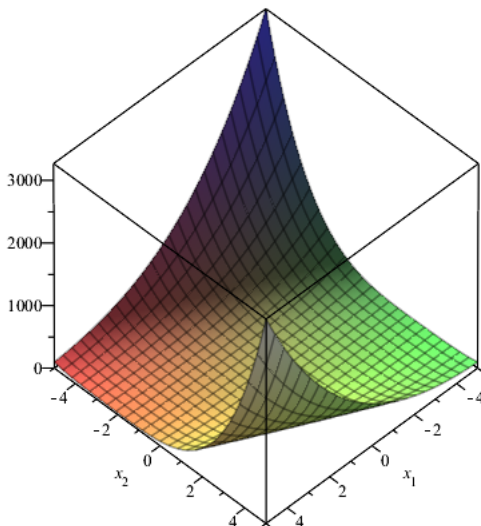
As can be seen from these graphs (generated for the problem with  $n = 1000$  as  $n = 5000$  segfaulted due to too large memory requirements!), the solver constantly kept converging to lower values of  $x$ . The tolerance for the norm of  $\nabla f(x)$  had to be set to a low number as it was often quite low.

## 6.6 Zakharov

Zakharov's function is of unknown origin to me, but I chose to feature it because it has an interesting phenomenon of a very flat valley along every second direction and power growth in the other directions, meaning that converging into this valley is trivial, but finding the actual minimum is very difficult, especially for higher dimensional problems and, unlike Beale's function, it is generalized to  $n$  dimensions. The function was found at the University of Kyoto's web-page, and proper referencing was not provided.[[Hed12b](#)]

$$f(x) = \sum_{i=1}^n x_i^2 + \left( \sum_{i=1}^n 1/2 i x_i \right)^2 + \left( \sum_{i=1}^n 1/2 i x_i \right)^4 \quad (6.6)$$

Here is a plot of the relevant area of the function when  $n = 2$ :



**Figure 6.15:** Minimum at  $x_i = 0$

First, a comparison of the CPU times, where the starting guess is a sequence of 1000 ones, and the algorithm converges towards the minimum at all zeros. All times are measured in seconds.

Attempt	UCMINF	FMINUNC	Maple 16	minlbfgs	libopti
1	0.1927	1.1757	11384s*		
2	0.1886	1.1584			
3	0.1890	1.1485			
4	0.1895	1.1479			
5	0.1879	1.1708			
6	0.1899	1.1527			
7	0.1880	1.1523			
8	0.1880	1.1600			
9	0.1988	1.2187			
10	0.2144	1.2022			
Average	0.1927	1.1687			

**Table 6.6:** Performance on Zakharov

Only the MATLAB algorithms were capable of finishing this, most likely due to a special implementation, which is proprietary, of the norm function not present anywhere else. In the case of the MATLAB algorithms, they both solved the problem. UCMINF was fastest by far, roughly a factor of 10. Maple 16 was able to start, but took so long I gave up.

The reason why libopti failed was because of the norm function. On the first iteration,  $\Delta$  is 1, and so the step should get normalized. However, we had no initial approximation of  $H$ , so we take the steepest descent direction. It scaled so badly that the step became so large, that when their powers of two were summed up, the sum wrapped around itself, going up through the negatives again, and the total result was about  $1.5 \cdot 10^0$ . Thus the step was not normalized at all. This could have been avoided with an initial approximation of the inverse Hessian, but for a 1000-dimensional problem it is virtually impossible to find accurately on a computer due to other scaling issues.

This problem gets passed on to the line search, which sees the massive step and ends up giving up because  $\alpha$  becomes too small and takes no step at all. This means that the step length became 0, and the algorithm terminates. Setting  $\Delta_0$  to a large number didn't help due to rounding errors.

The reason why minlbfgs failed is unknown - however the results it returned were all NaN, Inf, or -Inf.

The code used to take these samples can be found in [Appendix C.7](#).



# Conclusion

---

It is quite clear that libopti is a fast library. On the performance tests for small problems it stood up to an optimization library which claims to be 7-8 times faster than the GNU Scientific Library[ALG12b]. In addition, for all test problems but Zakharov it was much faster than the scripting based solutions. It could not solve Zakharov due to scaling problems, which mysteriously disappear in MATLAB. "Mysteriously" because it is a built-in non-viewable function.

One of the things I learned from this project is that C is fast. The reason why this library is so fast is precisely because it was written in C. C is one of the most efficient languages available and is a leading industry standard along with C++. The other reason is that it is based upon the MATLAB implementations from Hans Bruun Nielsen, which are supposedly also very fast compared to similar software available in MATLAB, according to John Bagterp Jørgensen.

But the implementation carries other benefits as well. It is portable and can be used on embedded devices, depending only on BLAS and standard UNIX functions to run correctly. I also learned that, in some cases, separating system specific calls with generic calls that can be replaced depending on OS is quite important and very useful.

However, the implementation has a big problem: It uses function pointers. One of the things I've learned from this project is that function pointers is a very C-specific feature that does not have direct counterparts in most other languages. This made running the algorithm from other languages impossible in most cases, possible in some cases with a bit of hacking.

Therefore one of the primary goals with this project were lost. This project is not older versions of FORTRAN, but it is possible to create a library that uses a routine from either of these two languages and calls it, then returns the result back to the language. It is directly callable from languages which have C as a subset of their own feature set, like C, Objective-C and Objective-C++, which was a nice little free feature.

Overall, I feel the project is a success. Despite its shortcomings in terms of porting to other languages it is still possible to do so and the speed and portability alone are enough to justify its use. The simple structure lays a strong foundation for the project to grow to include other types of optimization algorithms.

However, it can still be improved. Future ideas could be vectorization for more performance on multi-core processors, as well as enhancements to the algorithm in general and, of course, more optimization algorithms to use.

## APPENDIX A

# C library

---

### A.1 Library Header

```
1  /*
2  * libopti.h
3  *
4  * Created on: Aug 15, 2012
5  * Author: christian
6  */
7
8  #ifndef LIBOPTI_H_
9  #define LIBOPTI_H_
10
11 #ifndef __cplusplus //Ensure compatibility with C++.
12 extern "C" {
13 #endif /* cplusplus */
14
15 //Written by: Christian Wichmann Moesgaard, Sep. 2012
16 //Finds the minimizer to an unconstrained optimization problem given by fun()
17 //by using
18 //the Quasi-Newton BFGS method. Extra parameters may be sent using the void
19 //structure.
20 //Input: fun - A function handle to a function with the structure:
21 //          *x - The address of an array of doubles representing the
22 //              point to take the function at.
23 //          *f - The address of the storage of the function
24 //              evaluation
25 //          *g - The address of an array of doubles. Will be filled
26 //              with
27 //              the derivative of f(x).
```

```

24 //          *p - A pointer to a void structure that can be anything.
25 //          *x - The starting guess x for the function. On exit, the optimal
           value for x.
26 //          n - Number of variables given to the function
27 //          *opts - An array containing:
28 //               $\Delta$  - Initial value for largest stepsize used by the
           linesearching.
29 //              tolg - The value which the norm of g is under when the 1st
           order KKT conditions are met.
30 //              tolx - If the stepsize is less than tolx*(tolx + opti_norm
           (x, n, 2)), stop.
31 //              maxeval - The maximum number of tolerable iterations before
           stopping.
32 //              findiffh - Used in the finite difference approximation for fun
           (). If set to 0, g is expected.
33 //              Warning: Using findiffh = 0 and not supplying g in
           fun WILL result in errors.
34 //              Default: 1, 1e-4, 1e-8, 100, 0
35 //          *D - An array containing an initial estimation of the
           inverse Hessian to the function. Set to NULL to start with
36 //              with I. On exit, a pointer to the Hessian.
37 //              Warning: Points to unclaimed memory if *D was set to NULL!
38 //          *evalused - A pointer to an integer which will be filled with the number
           of evaluations.
39 //              Warning: NOT ITERATIONS.
40 //          *perfinfo - A pointer that must point to an array of doubles of size 6*
           maxeval.
41 //              Will contain performance information which can be printed
           using opti_printperf.
42 //          *xiter - If null, nothing happens. Otherwise, contains a sequence of x
           for each iteration on exit.
43 //              Must be preallocated to size n*maxiter (default n*100)
44 //          *p - A void-pointer which can contain anything. Is passed through
           the function and
45 //              all the way over to fun().
46 //          //Returns: -2 - BLAS failed to initialize. Error.
47 //              -1 - memory allocation, error
48 //              0 - perfect starting guess, success
49 //              1 - Step size H became NaN, inf or -inf, error
50 //              2 - Stopped due to small derivative g, success
51 //              3 - Stopped due to small change in x, success
52 //              4 - Norm of stepsize became 0, success.
53 //              5 - Too many function evaluations, potential failure
54 //          int opti_ucminf(void (*fun)(double *x, double *f, double *g, void *p),
55 //              double *x,
56 //              int n,
57 //              double *opts,
58 //              double *D,
59 //              int *evalused,
60 //              double *perfinfo,
61 //              double *xiter,
62 //              void *p);
63 //
64 //
65 //
66 //Written by: Christian Wichmann Moesgaard, Sep. 2012
67 //Calculates a steplength of a step on a function, and puts the new point
68 //one ends in x.
69 //Input: fun - A function handle to a function with the structure:
70 //          *x - The address of an array of doubles representing the
71 //              point to take the function at.
72 //          *f - The address of the storage of the function
           evaluation
73 //          *g - The address of an array of doubles. Will be filled
           with
74 //          the derivative of f(x).

```



```

75 //          *p - A pointer to a void structure that can be anything.
76 //          *x - Pointer to the variables given to the function
77 //          n - Number of variables given to the function
78 //          *f - The address of the storage of the function evaluation
79 //          *g - The address of an array of doubles. Will be filled with
80 //              the derivative of f(x).
81 //          *h - The search direction, the steplength multiplier,
82 //          *opts- An array with a few options:
83 //              choice: 1 = soft linesearch, 0 = exact linesearch
84 //              cp1:   Constant 1 timed onto derivative of f at initial point
85 //              (not used if choice = 0)
86 //              cp2:   Constant 2 timed onto derivative of f at initial point
87 //              maxeval: Maximum number of evaluation before quitting.
88 //              amax:   Maximum tolerable search direction multiplier
89 //              Default: 0, 1e-3, 1e-3, 10, 10
90 //          *perf- Some simple performance information from linesearch.
91 //              Contains: search direction multiplier, final slope, number of
92 //              evaluations
93 //          *p - A pointer to a void structure that can be anything.
94 //Returns: 0 - Successful return
95 //        -1 - Dynamic memory allocation failed.
96 //        1 - h is not downhill or it is so large and maxeval
97 //          so small, that a better point was not found.
98 int opti_linesearch(void (*fun)(double *x, double *f, double *g, void *p),
99                   double *x,
100                  int n,
101                  double *f,
102                  double *g,
103                  double *h,
104                  double *opts,
105                  double *perf,
106                  void *p);
107 //Written by: Christian Wichmann Moesgaard, Aug. 2012
108 //Finds the minimizer of a 1D parabola which is given by:
109 //A point a, and a point b.
110 //The function value at a and the function value at b.
111 //The derivative of the function value at a
112 //Input: xfd - An array consisting of xfd = {a, b, f(a), f(b), f'(a)}
113 //        n - The dimension of the problem original problem within
114 //            which the parabola is to be constructed.
115 //Output: The minimizer of the parabola.
116 double opti_interpolate(double *xfd,
117                        int n);
118 //Written by: Christian Wichmann Moesgaard, Sep. 2012
119 //Finds f and g directly if h == 0. Otherwise, approximates g with forward
120 //difference approximation using an O(n+1) algorithm. If h is negative, uses
121 //backward difference.
122 //Input: fun - A function handle to a function with the structure:
123 //          *x - The address of an array of doubles representing the
124 //              point to take the function at.
125 //          *f - The address of the storage of the function
126 //              evaluation
127 //          *g - The address of an array of doubles. Will be filled
128 //              with
129 //              the derivative of f(x).
130 //          *p - A pointer to a void structure that can be anything.
131 //          *x - Pointer to the variables given to the function
132 //          n - Number of variables given to the function
133 //          *f - The address of the storage of the function evaluation
134 //          *g - The address of an array of doubles. Will be filled with
135 //              the derivative of f(x).

```

```

135 //          h - The stepsize of the forward approximation. Can be any
        real number.
136 //          If h is 0, g is expected to be handed directly.
137 //          *p - A pointer to a void structure that can be anything.
138 //Returns: 0 - Successfully returned the function and its derivative.
139 //          -1 - Dynamic memory allocation failed.
140 int opti_finddiff(void (*fun)(double *x, double *f, double *g, void *p),
141                 double *x,
142                 int n,
143                 double *f,
144                 double *g,
145                 double h,
146                 void *p);
147
148
149 //Written by: Christian W. Moesgaard, Sep. 2012
150 //Prints performance information to a file in such a way that the file
151 //has the iterations listed among the columns.
152 //
153 //Number of evaluations of f done in the iteration.
154 //The value of f at the iteration
155 //The norm of the derivative of the function at the iteration
156 //The linesearch limit  $\Delta$  at the value of the iteration
157 //The scaling factor for the step as given by the linesearch
158 //The slope of the function at the point in the direction of the step
159 //
160 //Input:*perf - The performance array given by ucminf
161 //   maxiters - The maximum allowed iterations.
162 //           Must be the same number passed to opts.
163 //           Set to negative number to default to 100.
164 //   filename - A string containing the filename.
165 //Returns: 0 - Successfully saved the data to file.
166 //          -2 - Failed to open/write file. Nothing is saved.
167 int opti_printperf_tofile(double* perf,
168                          int maxiters,
169                          char* filename);
170
171 //Written by: Christian W. Moesgaard, Sep. 2012
172 //Prints performance information to a specified folder at the location the
173 //the program is run. Makes 7 subfiles named:
174 //
175 //evals.txt - Number of evaluations of f done in the iteration.
176 //fun.txt - The value of f at the iteration
177 //ng.txt - The norm of the derivative of the function at the iteration
178 //Δ.txt - The linesearch limit  $\Delta$  at the value of the iteration
179 //am.txt - The scaling factor for the step as given by the linesearch
180 //finalslope.txt - The slope of the function at the point in the direction of
        the step
181 //x.txt - The contents of x at each iteration
182 //
183 //Input:*perf - The performance array given by ucminf
184 //   maxiters - The maximum allowed iterations.
185 //           Must be the same number passed to opts.
186 //           Set to negative number to default to 100.
187 //   foldername - A string containing the foldername from the location your
        program is run.
188 //Returns: 0 - Successfully saved the data to file.
189 //          -2 - Failed to open/write file. Nothing is saved.
190 //          -3 - File existed with name of folder to be created. Nothing saved.
191 //          -4 - Folder could not be created. Check permissions or drive
192 int opti_printperf_tofolder(double* perf,
193                          double* xiter,
194                          int maxiters,
195                          int n,
196                          char* foldername);

```

```

197
198 //=====
199 //INTERNAL FUNCTIONS. NOT INTENDED FOR USAGE
200 //=====
201 //Takes a pointer to preallocated n*n array M and puts sets values such that
202 //it contains I.
203 void opti_deye(double *M, int n);
204 //Calculates the R^n-norm of the n-dimensional array x.
205 //Type can be 0 for inf-norm, or p ≥ 1 for p-norm.
206 double opti_norm(double *x, int n, int type);
207
208
209 #ifdef __cplusplus
210 }
211 #endif /* cplusplus */
212
213
214 #endif /* LIBOPTI_H_ */

```

## A.2 BLAS usage

```

1 /*
2  * use_blas.h
3  *
4  * Created on: Sep 23, 2012
5  * Author: christian
6  */
7
8 #ifndef USE_BLAS_H_
9 #define USE_BLAS_H_
10
11 int opti_initblas();
12
13 int opti_exitblas();
14
15 void opti_multmv(int dim, double scalar, double *A, double *x, double *y);
16
17 void opti_multmm(int rowA, int colA, double scalar, double *A, double *B,
18                 double *C);
19
20 double opti_multvv(int dim, double *x, double *y);
21
22 #endif /* USE_BLAS_H_ */

```

## A.3 Create I matrix

```

1 /*
2  * opti_deye.c
3  *
4  * Created on: Nov 2, 2012

```

```

5  *      Author: christian
6  */
7
8  #ifdef __cplusplus
9  extern "C" {
10 #endif
11
12 //Written by: Christian W. Moesgaard, Sep. 2012
13 //Takes a pointer to an n-by-n array M and fills it with
14 //contents such that M is the identity matrix.
15 //Returns M by reference.
16 //
17 //Input:    M - The performance array given by ucminf
18 //          n - The size of n
19 //Output:   void
20 void opti_deye(double *M, int n)
21 {
22     int i;
23     for (i = 0; i < n*n; ++i)
24         if (i%n == i/n)
25             M[i] = 1;
26         else
27             M[i] = 0;
28     return;
29 }
30
31 #ifdef __cplusplus
32 }
33 #endif

```

## A.4 Forward difference

```

1  /*
2  * opti_findiff.c
3  *
4  * Created on: Nov 2, 2012
5  *      Author: christian
6  */
7
8
9  #ifdef __cplusplus
10 extern "C" {
11 #endif
12
13 #include <stdlib.h>
14
15 //Written by: Christian Wichmann Moesgaard, Sep. 2012
16 //Finds f and g directly if h == 0. Otherwise, approximates g with forward
17 //difference approximation using an O(n+1) algorithm. If h is negative, uses
18 //backward difference.
19 //Input:  fun - A function handle to a function with the structure:
20 //          *x - The address of an array of doubles representing the
21 //              point to take the function at.
22 //          *f - The address of the storage of the function
23 //              evaluation
24 //          *g - The address of an array of doubles. Will be filled
25 //              with
26 //              the derivative of f(x).

```

```

25 //          *p - A pointer to a void structure that can be anything.
26 //          *x - Pointer to the variables given to the function
27 //          n - Number of variables given to the function
28 //          *f - The address of the storage of the function evaluation
29 //          *g - The address of an array of doubles. Will be filled with
30 //              the derivative of f(x).
31 //          h - The stepsize of the forward approximation. Can be any real
           number.
32 //          If h is 0, g is expected to be handed directly.
33 //          *p - A pointer to a void structure that can be anything.
34 //Returns: 0 - Successfully returned the function and its derivative.
35 //        -1 - Dynamic memory allocation failed.
36 int opti_findiff(void (*fun)(double *x, double *f, double *g, void *p), double
           *x, int n, double *f, double *g, double h, void *p)
37 {
38     int i; //Iterator
39     double fplush; //Returned value for f
40     double *gfplush = malloc(n*sizeof(double)); //g (unused)
41
42     if (gfplush == NULL) //Memory allocation failure
43     {
44         free(gfplush);
45         return -1;
46     }
47
48     fun(x, f, g, p);
49     if (h == 0) //If h is set to 0 we expect useful value from g.
50     {
51         free(gfplush);
52         return 0;
53     }
54
55     for (i = 0; i < n; ++i) //For each direction
56     {
57         x[i] += h; //Changing x in a new direction
58
59         fun(x, &fplush, gfplush, p); //fplush has useful value!
60
61         g[i] = (fplush - *f)/h; //Calculating finite difference
62
63         x[i] -= h; //Going back from that direction again.
64
65     } //x should now be exactly what it was, and g has been approximated!
66
67     fun(x, f, gfplush, p); //Get proper value of f.
68
69     return 0;
70 }
71
72 #ifdef __cplusplus
73 }
74 #endif

```

## A.5 Quadratic Interpolation

```

1  /*
2  * opti_interpolate.c
3  *

```

```

4  *   Created on: Nov 2, 2012
5  *   Author: christian
6  */
7
8  #ifndef __cplusplus
9  extern "C" {
10 #endif
11
12 #include <math.h>
13
14 //Written by: Christian Wichmann Moesgaard, Aug. 2012
15 //Finds the minimizer of a 1D parabola which is given by:
16 //A point a, and a point b.
17 //The function value at a and the function value at b.
18 //The derivative of the function value at a
19 //Input: xfd - An array consisting of xfd = {a, b, f(a), f(b), f'(a)}
20 //      n - The dimension of the problem original problem within
21 //      which the parabola is to be constructed.
22 //Output: The minimizer of the parabola.
23 double opti_interpolate(double *xfd, int n)
24 {
25     //Minimizer of parabola given by xfd.
26     double a = xfd[0], b = xfd[1], d = b - a, df = xfd[4];
27     double C = (xfd[3] - xfd[2]) - (d*df);
28     if (C ≥ 5*n*pow(2.220446049250313,-16)*b) //Minimizer exists
29     {
30         double A = a - 0.5*df*((d*d)/C); d = 0.1*d;
31         double t = a + d > A ? a + d : A;
32         return t < b - d ? t : b - d; //Ensure significant resuction
33     }
34     else
35     {
36         return (a+b)/2;
37     }
38 }
39
40 #ifndef __cplusplus
41 }
42 #endif

```

## A.6 Line-search

```

1  /*
2  *   opti_linesearch.c
3  *
4  *   Created on: Nov 2, 2012
5  *   Author: christian
6  */
7  #ifndef __cplusplus
8  extern "C" {
9  #endif
10
11 #include "../headers/libopti.h"
12
13 #include <math.h>
14 #include <string.h>
15 #include <stddef.h>
16 #include <stdlib.h>

```

```

17
18 #include "../headers/use_blas.h"
19
20 //Written by: Christian Wichmann Moesgaard, Sep. 2012
21 //Calculates a steplength of a step on a function, and puts the new point
22 //(scaled) point in x.
23 //Input: fun - A function handle to a function with the structure:
24 //          *x - The address of an array of doubles representing the
25 //              point to take the function at.
26 //          *f - The address of the storage of the function
27 //              evaluation
28 //          *g - The address of an array of doubles. Will be filled
29 //              with
30 //                  the derivative of f(x).
31 //          *p - A pointer to a void structure that can be anything.
32 //          *x - Pointer to the variables given to the function
33 //          n - Number of variables given to the function
34 //          *f - The address of the storage of the function evaluation
35 //          *g - The address of an array of doubles. Will be filled with
36 //              the derivative of f(x).
37 //          *h - The search direction, the steplength multiplier,
38 //          *opts- An array with a few options:
39 //                  choice: 1 = soft linesearch, 0 = exact linesearch
40 //                  cp1: Constant 1 timed onto derivative of f at initial point
41 //                  (not used if choice = 0)
42 //                  cp2: Constant 2 timed onto derivative of f at initial point
43 //                  maxeval: Maximum number of evaluation before quitting.
44 //                  amax: Maximum tolerable search direction multiplier
45 //                  Default: 0, 1e-3, 1e-3, 10, 10
46 //          *perf- Some simple performance information from linesearch.
47 //                  Contains: search direction multiplier, final slope, number of
48 //                  evaluations
49 //          *p - A pointer to a void structure that can be anything.
50 //Returns: 0 - Successful return
51 //          -1 - Dynamic memory allocation failed.
52 //          1 - h is not downhill or it is so large and maxeval
53 //              so small, that a better point was not found.
54 int opti_linesearch(void (*fun)(double *x, double *f, double *g, void *p),
55                   double *x,
56                   int n,
57                   double *f,
58                   double *g,
59                   double *h,
60                   double *opts,
61                   double *perf,
62                   void *p)
63 {
64     //Define all data structures
65
66     int i; //Iterator for simple for-loops
67
68     //Initialization
69     double f0 = *f; //Starting point function value
70     double df0 = opti_multvv(n, h, g); //Starting point derivative function value in LS direction
71     double slope0, slopethr; //Linesearch stopping criteria information
72     int info = 0; //Number of iterations and final slope value
73     double eps = pow(2.220446049250313, -16); //Epsilon value for type double
74     double *xn, *gn; //Allocation for new x and new g

```

```

71
72 //Opts variables
73 double choice = 0, cp1 = 1e-3, cp2 = 1e-3, maxeval = 10, amax = 10,
    findiffh = 0.01; //Default options for
    stopping criteria
74
75 //Get an initial interval for am
76 double a = 0, fa = f0, dfa = df0;
    //am starts at 0, here f(a) = f0, df(a) = df0
77 double b = amax < 1 ? amax : 1; //Picking min value //b is
    either amax or 1, depending on which is smaller
78 int stop = 0;
    //Used to mark stopping criteria found
79
80 //Declaring variables outside first while scope.
81 double fb, *xplusbtimesh, dfb;
82
83 //Declare variables outside second (refinement) while scope.
84 double c, fc, *xplusctimesh;
85
86 //Declare variables
87 xn = malloc(n*sizeof(double));
88 gn = malloc(n*sizeof(double));
89 xplusbtimesh = malloc(n*sizeof(double));
90 xplusctimesh = malloc(n*sizeof(double));
91
92 if (xn == NULL || gn == NULL ||
93     xplusbtimesh == NULL || xplusctimesh == NULL)
94 { //If yes, deallocate memory and exit
95     free(xn);
96     free(gn);
97     free(xplusbtimesh);
98     free(xplusctimesh);
99
100     return -1; //Allocation failed, not enough space for allocation
101 }
102
103 memcpy(xn,x,n*sizeof(double));
104 memcpy(gn,g,n*sizeof(double));
105
106 if (opts != NULL)
107 {
108     choice = opts[0];
109     cp1 = opts[1];
110     cp2 = opts[2];
111     maxeval = opts[3];
112     amax = opts[4];
113     findiffh = opts[5];
114 }
115
116 perf[0] = 0;
117 perf[1] = 1;
118 perf[2] = 0;
119
120 //Simple checks for descent condition
121 if (df0 ≥ -10*eps*opti_norm(h, n, 2)*opti_norm(g, n, 2))
122 {
123     free(xn);
124     free(gn);
125     free(xplusbtimesh);
126     free(xplusctimesh);
127
128     perf[2] = 0;
129     return 1;
130

```



```

131     }
132
133     if (choice == 1) //Soft linesearch
134     {
135         slope0 = cp1*df0;
136         slopethr = cp2*df0;
137     }
138     else //Exact linesearch
139     {
140         slope0 = 0;
141         slopethr = cp1*fabs(df0);
142     }
143
144     while (!stop)
145     {
146         info++;
147
148         for (i = 0; i < n; ++i) xplusbtimesh[i] = x[i] + b*h[i];
149
150         int findinfo = opti_finddiff(fun, xplusbtimesh, n, &fb, g, finddiffh, p)
151         ;
152         perf[2]++;
153
154         if (findinfo == -1)
155         {
156             free(xn);
157             free(gn);
158             free(xplusbtimesh);
159             free(xplusctimesh);
160
161             return -1; //Allocation failed.
162         }
163
164         dfb = opti_multvv(n, g, h);
165
166         if (fb < f0 + slope0*b) //New lower bound
167         {
168             perf[0] = b;
169             perf[1] = dfb/df0;
170
171             if (choice)
172             {
173                 a = b;
174                 fa = fb;
175                 dfa = dfb;
176             }
177             for (i = 0; i < n; ++i) xn[i] = x[i] + b*h[i];
178             memcpy(gn, g, n*sizeof(double));
179
180             if (dfb < ((slopethr ≤ 0)*slopethr) && (info < maxeval) && (b <
181                 amax))
182             {
183                 //Augment right hand end
184                 if (!choice)
185                 {
186                     a = b;
187                     fa = fb;
188                     dfa = dfb;
189                 }
190                 if (2.5*b ≥ amax) b = amax;
191                 else b = 2*b;
192             }
193             else stop = 1;
194         }
195     }

```

```

194     else stop = 1;
195
196
197
198     } //phase 1: expand interval
199
200     if (stop ≥ 0) //Everything OK so far. Check stopping criteria
201         stop = ((info ≥ maxeval) || ((b ≥ amax) && (dfb < slopethr)) //Cannot
                improve
                || (choice && ((a > 0) && (dfb ≥ slopethr))))); //OK
202
203
204     //If the above determines we must stop...
205     if (stop)
206     {
207         memcpy(x,xn,n*sizeof(double));
208         memcpy(g,gn,n*sizeof(double));
209         *f = fb;
210
211         free(xn);
212         free(gn);
213         free(xplusbtimesh);
214         free(xplusctimesh);
215
216         return info;
217     }
218
219
220     //Refine interval. Use auxiliary array xfd
221
222     double xfd[9] = {a, b, b, fa, fb, fb, dfa, dfb, dfb};
223
224     while (!stop)
225     {
226         info++;
227         double xfd_temp[5] = {xfd[0], xfd[1], xfd[3], xfd[4], xfd[6]};
228         c = opti_interpolate(xfd_temp, n);
229         for (i = 0; i < n; ++i) xplusctimesh[i] = x[i] + c*h[i];
230
231         int findinfo = opti_findiff(fun, xplusctimesh, n, &fc, g, findiffh, p)
                ;
232         perf[2]++;
233
234         if (findinfo == -1)
235         {
236             free(xn);
237             free(gn);
238             free(xplusbtimesh);
239             free(xplusctimesh);
240
241             return -1; //Allocation failed.
242         }
243
244
245         xfd[2] = c;
246         xfd[5] = fc;
247         xfd[8] = opti_multvv(n, g, h);
248         if (choice) //soft linesearch
249             if (fc < f0 + slope0*c) //New lower bound
250             {
251                 perf[0] = c;
252                 perf[1] = xfd[8]/df0;
253                 for (i = 0; i < n; ++i) xn[i] += c*h[i];
254                 memcpy(gn,g,n*sizeof(double));
255
256                 for (i = 0; i < 3; ++i) xfd[3*i] = xfd[3*i+2];

```

```

257         stop = (xfd[8] > slopethr);
258     }
259     else //New upper bound
260         for (i = 0; i < 3; ++i) xfd[3*i+1] = xfd[3*i+2];
261     else //Exact line search
262     {
263         if (fc < (*f))
264         {
265             perf[0] = c;
266             perf[1] = xfd[8]/df0;
267             for (i = 0; i < n; ++i) xn[i] += c*h[i];
268             memcpy(gn,g,n*sizeof(double));
269             *f = fc;
270         }
271         if (xfd[8] < 0) for (i = 0; i < 3; ++i) xfd[3*i] = xfd[3*i+2];
272         else for (i = 0; i < 3; ++i) xfd[3*i+1] = xfd[3*i
273             +2];
274         stop = ((fabs(xfd[8]) ≤ slopethr) || (fabs(xfd[0] - xfd[1]) <
275             (cp2*xfd[1])));
276     }
277     stop = (stop || info ≥ maxeval);
278 } //End refinement
279 memcpy(x,xn,n*sizeof(double));
280 memcpy(g,gn,n*sizeof(double));
281
282 *f = fc;
283
284 free(xn);
285 free(gn);
286 free(xplusbtimesh);
287 free(xplusctimesh);
288
289 return 0;
290
291 }
292
293
294
295
296
297
298
299
300
301 #ifdef __cplusplus
302 }
303 #endif

```

## A.7 Norm calculation

```

1  /*
2  * opti_norm.c
3  *
4  * Created on: Nov 2, 2012
5  * Author: christian
6  */

```

```

7
8 #ifndef __cplusplus
9 extern "C" {
10 #endif
11
12 #include <math.h>
13
14 //Written by: Christian W. Moesgaard, Sep. 2012
15 //Finds the norm of an n-length array x of double values.
16 //The third argument is the type of the norm, 0 is infinity norm.
17 //
18 //Input:   x - The input array
19 //         n - The length of x.
20 //         type -
21 //Returns: A double with the result, should be positive.
22 double opti_norm(double *x, int n, int type)
23 {
24     int i;
25     double result = 0;
26     if (type == 0) //Infinity norm
27     {
28         for (i = 0; i < n; ++i)
29             if (result < fabs(x[i]))
30                 result = fabs(x[i]);
31     }
32     else
33     {
34         for (i = 0; i < n; ++i)
35             result += pow(fabs(x[i]), type);
36         result = pow(result, 1.0/(double) n);
37     }
38     return result;
39 }
40 }
41
42
43
44 #ifndef __cplusplus
45 }
46 #endif

```

## A.8 Print performance to file

```

1 /*
2  * opti_printperf_tofile.c
3  *
4  * Created on: Nov 2, 2012
5  * Author: christian
6  */
7
8 #ifndef __cplusplus //Ensure compatibility with C++.
9 extern "C" {
10 #endif /* cplusplus */
11
12 #include <stdio.h>
13 #include <stddef.h>
14 #include <stdlib.h>
15 #include <unistd.h>

```

```

16 #include <sys/types.h>
17 #include <sys/stat.h>
18 #include <errno.h>
19 #include <stddef.h>
20 #include <string.h>
21
22 //Written by: Christian W. Moesgaard, Sep. 2012
23 //Prints performance information to a file in such a way that the file
24 //has the iterations listed among the columns.
25 //
26 //Number of evaluations of f done in the iteration.
27 //The value of f at the iteration
28 //The norm of the derivative of the function at the iteration
29 //The linesearch limit  $\Delta$  at the value of the iteration
30 //The scaling factor for the step as given by the linesearch
31 //The slope of the function at the point in the direction of the step
32 //
33 //Input: *perf - The performance array given by ucminf
34 //      maxiters - The maximum allowed iterations.
35 //      Must be the same number passed to opts.
36 //      Set to negative number to default to 100.
37 //      filename - A string containing the filename.
38 //Returns: 0 - Successfully saved the data to file.
39 //      -2 - Failed to open/write file. Nothing is saved.
40 int opti_printperf_tofile(double* perf,
41                          int maxiters,
42                          char* filename)
43 {
44     int iters = (int) perf[0];
45
46     if (maxiters < 0) maxiters = 100;
47
48     FILE *perffile;
49     perffile = fopen(filename, "w+");
50     if (perffile == NULL)
51         return -2;
52
53     fprintf(perffile, "Contents of rows: Evaluations, function value, norm of
54         g, Delta, steplength multiplier, slope towards step at new x\n");
55
56     int perftype, iter;
57
58     for (perftype = 0; perftype < 6; ++perftype)
59     {
60         for (iter = 0; iter < iters-2; ++iter)
61         {
62             if (perftype == 0 && iter == 0)
63                 fprintf(perffile, "%f, ", 1.0);
64             else
65                 fprintf(perffile, "%f, ", perf[maxiters*perftype + iter]);
66             fprintf(perffile, "%f.\n", perf[maxiters*perftype + iter]);
67         }
68
69         fclose(perffile);
70
71         return 0;
72     }
73 }
74
75 #ifdef __cplusplus
76 }
77 #endif /* cplusplus */

```

## A.9 Print performance to folder

```

1  /*
2  * opti_printperf_tofolder.c
3  *
4  *   Created on: Nov 2, 2012
5  *   Author: christian
6  */
7
8  #ifdef __cplusplus //Ensure compatibility with C++.
9  extern "C" {
10 #endif /* cplusplus */
11
12 #include <stdio.h>
13 #include <stddef.h>
14 #include <stdlib.h>
15 #include <unistd.h>
16 #include <sys/types.h>
17 #include <sys/stat.h>
18 #include <errno.h>
19 #include <string.h>
20 #include <string.h>
21
22 //Written by: Christian W. Moesgaard, Sep. 2012
23 //Prints performance information to a specified folder at the location the
24 //program is run. Makes 7 subfiles named:
25 //
26 //evals.txt - Number of evaluations of f done in the iteration.
27 //fun.txt - The value of f at the iteration
28 //ng.txt - The norm of the derivative of the function at the iteration
29 //Δ.txt - The linesearch limit Δ at the value of the iteration
30 //am.txt - The scaling factor for the step as given by the linesearch
31 //finalslope.txt - The slope of the function at the point in the direction of
   the step
32 //x.txt - The contents of x at each iteration
33 //
34 //Input:*perf - The performance array given by ucminf
35 //   maxiters - The maximum allowed iterations.
36 //           Must be the same number passed to opts.
37 //           Set to negative number to default to 100.
38 // foldername - A string containing the foldername from the location your
   program is run.
39 //Returns:  0 - Successfully saved the data to file.
40 //          -2 - Failed to open/write file. Nothing is saved.
41 //          -3 - File existed with name of folder to be created. Nothing saved.
42 //          -4 - Folder could not be created. Check permissions or drive
43 int opti_printperf_tofolder(double* perf,
44     double* xiter,
45     int maxiters,
46     int n,
47     char* foldername)
48 {
49     struct stat sb;
50     int e;
51     extern int errno;
52
53     //Get info on the folder/file
54     e = stat(foldername, &sb);
55
56     //If something with this name already exists...
57     if (e == 0)
58     {

```

```

59     if (sb.st_mode & 0100000) //If it's a file
60         return -3; //File exists with foldername. Abort.
61     }
62     else
63     {
64         if (errno == ENOENT)
65         {
66             e = mkdir(foldername, S_IRWXU);
67             if (e != 0)
68                 return -4; //Folder could not be created. Check permissions or
69                             //drive
69         }
70     }
71
72     //Before we can use strcpy and strcat, we need the length of the string
73     size_t pathlen = strlen(foldername);
74
75     //Setting up iterations sequence
76     int iters = (int) perf[0];
77     if (maxiters < 0) maxiters = 100;
78
79     //Contents of rows:
80     //Evaluations,
81     //function value,
82     //norm of g,
83     //Delta,
84     //steplength multiplier,
85     //slope towards step at new x
86     //x as horizontal array
87     int perftype, iter;
88     char path[7][pathlen + 20];
89
90     for (perftype = 0; perftype < 7; ++perftype)
91         strcpy(path[perftype], foldername);
92
93     //Names of files in new folder
94     strcat(path[0], "/evals.txt");
95     strcat(path[1], "/fun.txt");
96     strcat(path[2], "/ng.txt");
97     strcat(path[3], "/Δ.txt");
98     strcat(path[4], "/am.txt");
99     strcat(path[5], "/finalslope.txt");
100    strcat(path[6], "/x.txt");
101
102
103
104
105    for (perftype = 0; perftype < 6; ++perftype)
106    {
107        FILE *perffile;
108        perffile = fopen(path[perftype], "w+");
109        if (perffile == NULL)
110            return -2; //Failed to open subfile
111
112        for (iter = 0; iter < iters-2; ++iter)
113        {
114            if (perftype == 0 && iter == 0)
115                fprintf(perffile, "%f ", 1.0);
116            else
117                fprintf(perffile, "%f ", perf[maxiters*perftype + iter]);
118        }
119        fprintf(perffile, "%f", perf[maxiters*perftype + iter]);
120
121        fclose(perffile);
122    }

```

```

123
124     if (xiter != NULL)
125     {
126         FILE *perffile;
127         perffile = fopen(path[6], "w+");
128         if (perffile == NULL)
129             return -2; //Failed to open subfile
130
131         int i;
132
133         for (iter = 0; iter < iters-1; ++iter)
134         {
135             for (i = 0; i < n; ++i)
136             {
137                 fprintf(perffile, "%f ", xiter[iter*n + i]);
138             }
139             fprintf(perffile, "\n");
140         }
141
142         fclose(perffile);
143     }
144
145     return 0;
146 }
147
148 #ifdef __cplusplus
149 }
150 #endif /* cplusplus */

```

## A.10 Unconstrained BFGS minimization

```

1 //
2 // opti_ucminf.c
3 // Optiboxmac
4 //
5 // Created by Christian on 09/10/12.
6 //
7
8 #ifdef __cplusplus
9 extern "C" {
10 #endif
11
12 #include "../headers/libopti.h"
13
14 #include <math.h>
15 #include <float.h>
16 #include <stdio.h>
17 #include <string.h>
18 #include <stddef.h>
19 #include <stdlib.h>
20
21 #include "../headers/use_blas.h"
22
23
24 // Get machine-dependent NAN and INFINITY values.
25 #include <bits/nan.h>
26 #include <bits/inf.h>

```



```

27
28 //Written by: Christian Wichmann Moesgaard, Sep. 2012
29 //Finds the minimizer to an unconstrained optimization problem given by fun()
    by using
30 //the Quasi-Newton BFGS method. Extra parameters may be sent using the void
    structure.
31 //Input: fun - A function handle to a function with the structure:
32 //          *x - The address of an array of doubles representing the
33 //              point to take the function at.
34 //          *f - The address of the storage of the function
    evaluation
35 //          *g - The address of an array of doubles. Will be filled
    with
36 //              the derivative of f(x).
37 //          *p - A pointer to a void structure that can be anything.
38 //          *x - The starting guess x for the function. On exit, the optimal
    value for x.
39 //          n - Number of variables given to the function
40 //          *opts - An array containing:
41 //              Δ - Initial value for largest stepsize used by the
    linesearching.
42 //              tolg - The value which the norm of g is under when the 1st
    order KKT conditions are met.
43 //              tolx - If the stepsize is less than tolx*(tolx + opti_norm
    (x, n, 2)), stop.
44 //              maxeval - The maximum number of tolerable iterations before
    stopping.
45 //              findiffh - Used in the finite difference approximation for fun
    (). If set to 0, g is expected.
46 //              Warning: Using findiffh = 0 and not supplying g in
    fun WILL result in errors.
47 //              Default: 1, 1e-4, 1e-8, 100, 0
48 //          *D - An array containing an initial estimation of the
    inverse Hessian to the function. Set to NULL to start with
49 //              with I. On exit, a pointer to the Hessian.
50 //              Warning: Points to unclaimed memory if *D was set to NULL!
51 //          *evaluated - A pointer to an integer which will be filled with the number
    of evaluations.
52 //              Warning: NOT ITERATIONS.
53 //          *perfinfo - A pointer that must point to an array of doubles of size 6*
    maxeval.
54 //              Will contain performance information which can be printed
    using opti_printperf.
55 //          *xiter - If null, nothing happens. Otherwise, contains a sequence of x
    for each iteration on exit.
56 //              Must be preallocated to size n*maxiter (default n*100)
57 //          *p - A void-pointer which can contain anything. Is passed through
    the function and
58 //              all the way over to fun().
59 //Returns: -2 - BLAS failed to initialize. Error.
60 //          -1 - memory allocation, error
61 //          0 - perfect starting guess, success
62 //          1 - Stepsize H became NaN, inf or -inf, error
63 //          2 - Stopped due to small derivative g, success
64 //          3 - Stopped due to small change in x, success
65 //          4 - Norm of stepsize became 0, success.
66 //          5 - Too many function evaluations, potential failure
67 //
68 int opti_ucminf(void (*fun)(double *x, double *f, double *g, void *p),
69               double *x,
70               int n,
71               double *opts,
72               double *D,
73               int *evaluated,
74               double *perfinfo,
75               double *xiter,

```

```

76         void *p)
77 {
78     //Define all data structures
79     int i, j;

        //Iterator for simple for-loops
80     int neval = 0, iter = 0;

        //Number of
        evaluations and iterations done
81     double eps = pow(2.220446049250313,-16);

        //Definition of epsilon,
        smallest difference in doubles
82
83     double Delta = 1, tolg = 1e-4, tolx = 1e-8; int maxeval = 100;
        //Options
84     double f;

        //Function values
85     double *g = malloc(n*sizeof(double)), *h = malloc(n*sizeof(double));
        //Values of derivative and steplength
86     double *xp = malloc(n*sizeof(double)), *gp = malloc(n*sizeof(double));
        //Data for the internal parts of the loop
87     double nx;

        //Norm of x
88
89     double *tempD = NULL;

        //
        Potentially necessary inverse hessian workspace
90     int fst = 0;

        //Checking first iteration, 1 = yes, BOOL
91
92     double ng;

        //Infinity norm of g
93     double nh = 0;

        //Infinity norm of h
94     int stop = 0, red = 0, more = 1;

        //Stopping criteria
        measurement
95     double ngs[3];

        //Norm of gs_k, gs_k-1, gs_k-2
96
97     double lsopts[6] = {1, 0.05, 0.99, 5, 2, 0.01};
        //Linesearch options
98     double perf[3] = {0, 0, 0};

        //Linesearch
        performance
99     int linfo = 0;

        //Info about number of iterations in the linesearch
100
101     double *y = malloc(n*sizeof(double));

        //y = x - xp
102     double yh;

        //norm of y
103     double *v = malloc(n*sizeof(double)), *w = malloc(n*sizeof(double));
        //Part of the BFGS update
104     double a, yv;

        //ddot(y,v)

```

```

105
106     double thrx;
           //Used for 1st order KKT measurements
107
108     double findiffh = 0;
           //h in
           finite difference approximation
109
110     //Initialize BLAS
111     int blasinfo = opti_initblas();
112     if (blasinfo) //Check if BLAS init failed
113     {
114         free(g);
115         free(h);
116         free(xp);
117         free(gp);
118         free(y);
119         free(v);
120         free(w);
121
122         return -2; //BLAS init fail
123     }
124
125
126     //Test allocation
127     if (g == NULL || h == NULL || xp == NULL ||
128         gp == NULL || y == NULL || v == NULL || w == NULL)
129     { //If yes, deallocate memory and exit
130         free(g);
131         free(h);
132         free(xp);
133         free(gp);
134         free(y);
135         free(v);
136         free(w);
137
138         return -1; //Allocation failed, not enough space for allocation
139     }
140
141
142     //Check options parameters and set them if necessary
143     if (opts != NULL)
144     {
145         Delta = opts[0];
146         tolg = opts[1];
147         tolx = opts[2];
148         maxeval = opts[3];
149         findiffh = opts[4];
150         lsopts[5] = findiffh;
151     }
152
153     linfo = opti_findiff(fun, x, n, &F, g, findiffh, p);
154     neval++;
155
156     if (xiter != NULL)
157     {
158         for (i = 0; i < n; ++i)
159             xiter[i + n*xiter] = x[i];
160     }
161
162     if (linfo == -1)
163     {
164         free(g);
165         free(h);

```

```

166     free(xp);
167     free(gp);
168     free(y);
169     free(v);
170     free(w);
171     return -1; //Allocation failed.
172 }
173
174
175
176 //Finish initialization of g
177 ng = opti_norm(g, n, 0);
178
179 //Setup approximate inverse Hessian if missing.
180 if (D == NULL)
181 {
182     tempD = malloc(n*n*sizeof(double));
183     if (tempD == NULL)
184     {
185         free(g);
186         free(h);
187         free(xp);
188         free(gp);
189         free(y);
190         free(v);
191         free(w);
192         return -1; //Allocation failed
193     }
194
195     opti_deye(tempD,n);
196     D = tempD;
197     fst = 1;
198 }
199
200 //Update performance log
201 if (perfinfo != NULL)
202 {
203     perfinfo[iter          ] = 0;
204     perfinfo[iter + maxeval*1] = f;
205     perfinfo[iter + maxeval*2] = ng;
206     perfinfo[iter + maxeval*3] = Delta;
207     perfinfo[iter + maxeval*4] = 0;
208     perfinfo[iter + maxeval*5] = 0;
209 }
210
211
212 if (ng ≤ tol) //Check optimality condition
213 {
214     free(g);
215     free(h);
216     free(xp);
217     free(gp);
218     free(y);
219     free(v);
220     free(w);
221     free(tempD);
222     return 0; //Started optimal
223 }
224 else
225 {
226     memset(h,0,n*sizeof(double));
227     for (i = 0; i < 3; ++i) ngs[i] = ng;
228 }
229
230 //Start main loop

```

```

231     while (!(stop) && more)
232     {
233         iter++;
234
235         //Set previous values
236         memcpy(xp,x,n*sizeof(double));
237         memcpy(gp,g,n*sizeof(double));
238         nx = opti_norm(x, n, 2);
239
240         //Keep ngs up-to-date
241         ngs[0] = ngs[1]; ngs[1] = ngs[2]; ngs[2] = ng;
242
243         //Get step.
244         opti_multmv(n, -1.0, D, g, h);
245         //Check h for NAN's or other oddities.
246         for (i = 0; i < n; ++i)
247         {
248             if (h[i] != h[i] || h[i] == INFINITY || h[i] == -INFINITY) //Check
                if inf, or if NaN.
249             {
250                 free(g);
251                 free(h);
252                 free(xp);
253                 free(gp);
254                 free(y);
255                 free(v);
256                 free(w);
257                 free(tempD);
258                 return 1;
259             }
260         }
261
262         nh = opti_norm(h, n, 2);
263         red = 0;
264
265         //Check optimality
266         if (nh ≤ tolx*(tolx + nx)) stop = 2;
267         else
268         {
269             if (fst || nh > Delta)
270             {
271                 for (i = 0; i < n; ++i) h[i] *= Delta/nh;
272                 nh = Delta;
273                 fst = 0; red = 1;
274             }
275
276             //Linesearch
277             linfo = opti_linesearch(fun, x, n, &f, g, h, lsopts, perf, p);
278             if (xiter != NULL)
279             {
280                 for (i = 0; i < n; ++i)
281                     xiter[i + n*xiter] = x[i];
282             }
283             neval += perf[2]; //Counting evals
284
285             if (perf[0] < 1) //Reduce Delta
286                 Delta *= 0.35;
287             else if (red && (fabs(perf[1]) > 0.7)) //Expand Δ
288                 Delta *= 3;
289
290             //Update ||g||
291             ng = opti_norm(g, n, 0);
292             //Update h
293             for (i = 0; i < n; ++i) h[i] = x[i] - xp[i];
294             nh = opti_norm(h, n, 2);

```

```

295     if (nh == 0)
296         stop = 3;
297     else
298     {
299         for (i = 0; i < n; ++i) y[i] = g[i] - gp[i];
300         yh = opti_multvv(n, y, h);
301         if (yh > sqrt(eps) * nh * opti_norm(y, n, 2))
302         {
303             //Update D using BFGS
304             opti_multmv(n, 1.0, D, y, v);
305             yv = opti_multvv(n, y, v);
306             a = (1 + yv/yh)/yh;
307             for (i = 0; i < n; ++i) w[i] = (a/2)*h[i] - v[i]/yh;
308
309             //Calculating wh and hw and adding it to D, BFGS update
310             for (i = 0; i < n; ++i)
311                 for (j = 0; j < n; ++j)
312                     {
313                         D[i*n + j] += w[i]*h[j] + h[i]*w[j];
314                     }
315
316         }
317
318         //Update performance benchmark
319         if (perfinfo != NULL)
320         {
321             perfinfo[iter          ] = perf[2];
322             perfinfo[iter + maxeval*1] = f;
323             perfinfo[iter + maxeval*2] = ng;
324             perfinfo[iter + maxeval*3] = Delta;
325             perfinfo[iter + maxeval*4] = perf[0];
326             perfinfo[iter + maxeval*5] = perf[1];
327         }
328
329         //Check stopping criteria
330         thrx = tol*(tolx + opti_norm(x, n, 2));
331         if (ng ≤ tol) stop = 1;
332         else if (nh ≤ thrx) stop = 2;
333         else if (neval ≥ maxeval) stop = 4;
334         else Delta = Delta > 2*thrx ? Delta : 2*thrx;
335     }
336 }
337 }
338
339 //De-allocate variables
340 free(g);
341 free(h);
342 free(xp);
343 free(gp);
344 free(y);
345 free(v);
346 free(w);
347
348 iter++; //Fix off-of-one error
349
350 //State number of iterations
351 if (evalused != NULL)
352     *evalused = neval;
353
354 //Insert number of iterations into perfinfo.
355 if (perfinfo != NULL)
356     perfinfo[0] = iter;
357
358 //Initialize BLAS
359 blasinfo = opti_exitblas();

```

```
360     if (blasinfo) //Check if BLAS exit failed
361     {
362         free(g);
363         free(h);
364         free(xp);
365         free(gp);
366         free(y);
367         free(v);
368         free(w);
369
370         return -2; //BLAS exit fail
371     }
372
373     //Successful run
374     return (stop + 1);
375
376 }
377
378
379
380
381
382
383
384
385
386 #ifdef __cplusplus
387 }
388 #endif
```





## APPENDIX B

# Wrappers to external libraries

---

## B.1 Use Apple Accelerate

```
1  /*
2  * use_accelerate.c
3  *
4  * Created on: Nov 2, 2012
5  * Author: christian
6  */
7
8  #ifdef __MACH__
9  #include <Accelerate/Accelerate.h>
10
11 int opti_initblas()
12 {
13     return 0;
14 }
15
16 int opti_exitblas()
17 {
18     return 0;
19 }
20
21 void opti_multmv(int dim, double scalar, double *A, double *x, double *y)
22 {
23     cblas_dgemv(CblasColMajor, CblasNoTrans, dim, dim, scalar, A, dim, x, 1,
24                0.0, y, 1);
25     return;
```

```

26
27 void opti_multmm(int rowA, int colA, double scalar, double *A, double *B,
    double *C)
28 {
29     cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, rowA, rowA, colA,
        scalar, A, rowA, B, colA, 0.0, C, rowA);
30     return;
31 }
32
33 double opti_multvv(int dim, double *x, double *y)
34 {
35     return cblas_ddot(dim, x, 1, y, 1);
36 }
37 #endif /*__MACH__*/

```

## B.2 Use ATLAS

```

1 /*
2  * use_atlas.c
3  *
4  * Created on: Nov 2, 2012
5  * Author: christian
6  */
7
8 #ifndef __MACH__
9 #ifndef USE_CUDA_OPTI
10 #include <cblas.h>
11
12 int opti_initblas()
13 {
14     return 0;
15 }
16
17 int opti_exitblas()
18 {
19     return 0;
20 }
21
22 void opti_multmv(int dim, double scalar, double *A, double *x, double *y)
23 {
24     cblas_dgemv(CblasColMajor, CblasNoTrans, dim, dim, scalar, A, dim, x, 1,
        0.0, y, 1);
25     return;
26 }
27
28 void opti_multmm(int rowA, int colA, double scalar, double *A, double *B,
    double *C)
29 {
30     cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, rowA, rowA, colA,
        scalar, A, rowA, B, colA, 0.0, C, rowA);
31     return;
32 }
33
34 double opti_multvv(int dim, double *x, double *y)
35 {
36     return cblas_ddot(dim, x, 1, y, 1);
37 }
38 #endif /*USE_CUDA_OPTI*/

```

```
39 #endif /*__MACH__*/
```



## APPENDIX C

# Test runs

---

### C.1 Apple Test

```
1 //
2 // FirstViewController.m
3 // Optiboxios
4 //
5 // Created by Christian on 23/10/12.
6 // Copyright (c) 2012 Christian. All rights reserved.
7 //
8
9 #import "FirstViewController.h"
10 #import "liboptibox.h"
11 #import <math.h>
12
13 //Ackley function that must be called.
14 void ackley(double* x, double* f, double* df, void *p)
15 {
16
17     double a, b, c; //Constants
18     int i, n; //Iterator and dimensions
19
20     a = *((double*) p);
21     b = *((double*) p+1);
22     c = *((double*) p+2);
23     n = *((double*) p+3);
24
25
26     //Calculating function value
27     double sum1 = 0, sum2 = 0; //Defining sums
28
```

```

29     for (i = 0; i < (int) n; ++i)
30     {
31         sum1 += x[i]*x[i];
32         sum2 += cos(c*x[i]);
33     }
34
35     *f = -a*exp(-b*sqrt((1.0/n)*sum1)) - exp((1.0/n)*sum2) + a + exp(1.0);
36
37     //Calculating derivatives
38     int j;
39
40     for (j = 0; j < (int) n; ++j)
41     {
42         df[j] = 0; //Initializing to 0.
43
44         df[j] += a*b*x[j]*exp(-(1.0/n)*b*sqrt(n*sum1))/sqrt(n*sum1);
45
46         df[j] += -1.0/n*sin(c*x[j])*exp(1.0/n*sum2);
47     }
48
49
50
51     return;
52 }
53
54 //=====
55 //Needs to be put into alltests!
56 #include <time.h>
57 #include <sys/time.h>
58 #ifdef __MACH__ //If on a Mac, use Mach libraries
59 #include <mach/clock.h>
60 #include <mach/mach.h>
61 #endif //__MACH__
62
63 double opti_get_time()
64 {
65     struct timespec ts;
66 #ifdef __MACH__ // OS X does not have clock_gettime, use clock_get_time
67     clock_serv_t cclock;
68     mach_timespec_t mts;
69     host_get_clock_service(mach_host_self(), CALENDAR_CLOCK, &cclock);
70     clock_get_time(cclock, &mts);
71     mach_port_deallocate(mach_task_self(), cclock);
72     ts.tv_sec = mts.tv_sec;
73     ts.tv_nsec = mts.tv_nsec;
74 #else
75     clock_gettime(CLOCK_REALTIME, &ts);
76 #endif
77     return (double) ts.tv_sec + (double) ts.tv_nsec * 1e-9;
78 }
79 //=====
80
81 @interface FirstViewController ()
82
83 @end
84
85 @implementation FirstViewController
86
87 @synthesize UIMainText;
88 - (void) viewDidLoad
89 {
90     [super viewDidLoad];
91     // Do any additional setup after loading the view, typically from a nib.
92
93     //Generate random starting guess

```

```

94     unsigned int  iseed = (unsigned int)time(NULL);
95     srand (iseed);
96
97     //Number of inputs to n, number of max iterations
98     int n = 20;
99     int maxiters = 1000;
100    double x[n];
101    int i;
102
103    //Randomly generate numbers
104    for (i = 0; i < n; ++i)
105    {
106        x[i] = (rand()%600 - 300)/100.0;
107    }
108
109    //Define array with params
110    double pi = 3.141592654;
111
112    double p[] = {20, 0.2, 2*pi, (double) n};
113
114
115    double perfinfo[maxiters*6];
116
117    //Setting options
118    double opts[] = {1.0, 0.0001, 0.00000001, maxiters, 0};
119
120    //The function shall be called on the function, using x as starting guess,
121    //with n values, opts shall be used for options, no initial approximation
122    //of inverse Hessian,
123    //want number of iterations used (not printed), want the performance info,
124    //we want all x iterations, and parameters p shall be passed.
125    double xinfo[n*maxiters];
126    double t1 = opti_get_time();
127    int ucminfr = opti_ucminf(ackley, x, n, opts, NULL, NULL, perfinfo, xinfo,
128    (void*) p);
129    double t2 = opti_get_time();
130
131    //Results printed to folder
132    //int folderinfo = opti_printperf_tofolder(perfinfo, xinfo, maxiters, n, "
133    testsackley");
134    int folderinfo = 1;
135
136    NSString * printthisfoo = [NSString stringWithFormat:
137    @"Solve time: %f\nSave to folder info: %i\
138    nSolution return: %i\n",
139    t2 - t1, folderinfo, ucminfr];
140
141
142
143    for (i = 0; i < n; ++i)
144    {
145        printthisfoo = [printthisfoo stringByAppendingString:[NSString
146        stringWithFormat:@"%5.3f, ", x[i]]];
147    }
148
149    UILabelMainText.text = printthisfoo;
150 }
151
152 - (void)didReceiveMemoryWarning
153 {
154     [super didReceiveMemoryWarning];
155     // Dispose of any resources that can be recreated.
156 }
157
158 @end

```

## C.2 Himmelblau

### C.2.1 MATLAB test

```

1  for i = 1:10
2      tic,
3      y(i,:) = ucminf(@himmelblau2,[1, 1]');
4      t1(i) = toc;
5  end
6  time1 = sum(t1)/10;
7
8  for i = 1:10
9      tic,
10     x(i,:) = fminunc(@himmelblau,[1, 1]');
11     t2(i) = toc;
12 end
13 time2 = sum(t2)/10;
14
15 t1, time1, t2, time2

```

### C.2.2 minlbfgs test

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <cmath>
5  #include "ap.h"
6  #include "minlbfgs.h"
7
8  using namespace std;
9
10 //Needs to be put into alltests!
11 #include <time.h>
12 #include <sys/time.h>
13 #ifdef __MACH__ //If on a Mac, use Mach libraries
14 #include <mach/clock.h>
15 #include <mach/mach.h>
16 #endif //__MACH__
17
18 double get_time()
19 {
20     struct timespec ts;
21 #ifndef __MACH__ // OS X does not have clock_gettime, use clock_get_time
22     clock_serv_t cclock;
23     mach_timespec_t mts;
24     host_get_clock_service(mach_host_self(), CALENDAR_CLOCK, &cclock);
25     clock_get_time(cclock, &mts);
26     mach_port_deallocate(mach_task_self(), cclock);
27     ts.tv_sec = mts.tv_sec;
28     ts.tv_nsec = mts.tv_nsec;
29 #else
30 #ifdef CLOCK_MONOTONIC
31     clock_gettime(CLOCK_MONOTONIC, &ts);
32 #else

```



```

33     clock_gettime(CLOCK_REALTIME, &ts);
34 #endif
35 #endif
36     return (double) ts.tv_sec + (double) ts.tv_nsec * 1e-9;
37 }
38
39 void himmelblau(double* x, double* f, double* df, void *p)
40 {
41     *f = pow(x[0]*x[0] + x[1] - 11,2) + pow(x[0] + x[1]*x[1] - 7,2);
42
43     df[0] = 2*(2*x[0]*(x[0]*x[0] + x[1] - 11) + x[0] + x[1]*x[1] - 7);
44     df[1] = 2*(-11+x[0]*x[0] + x[1] + 2*x[1]*(-7 + x[0] + x[1]*x[1]));
45
46     return;
47 }
48
49 //=====
50
51 int main(int argc, char **argv)
52 {
53     int n = 2;
54     int m = 1;
55     minlbfgsstate state;
56     minlbfgsreport rep;
57     ap::real_ld_array s;
58     double x[n];
59
60     double time = 0;
61
62     n = 2;
63     m = 1;
64     s.setlength(n);
65
66
67     //f and g
68     double f;
69     double g[n];
70
71     for (int i = 0; i < 10; ++i)
72     {
73         s(0) = 1;
74         s(1) = 1;
75         double t1 = get_time();
76         minlbfgscreate(n, m, s, state);
77         minlbfgssetcond(state, 0.0, 0.0, 0.0001, 0);
78         while(minlbfgsiteration(state))
79         {
80             if( state.needfg )
81             {
82                 for (int k = 0; k < n; ++k)
83                     x[k] = state.x(k);
84
85                 himmelblau(x, &f, g, NULL);
86
87                 state.f = f;
88
89                 for (int k = 0; k < n; ++k)
90                     state.g(k) = g[k];
91             }
92         }
93         minlbfgsresults(state, s, rep);
94
95         double t2 = get_time();
96
97         time += t2 - t1;

```

```

98     printf("Solve time: %f\n", t2 - t1);
99     }
100
101     printf("Result: (%f, %f)\n", state.x(0), state.x(1));
102     printf("Average solve time: %f\n", time /= 10);
103
104     return 0;
105 }
106

```

### C.2.3 libopti test

```

1  /*
2  * himmelblau.h
3  *
4  * Created on: Nov 3, 2012
5  * Author: christian
6  */
7
8  #ifndef HIMMELBLAU_H_
9  #define HIMMELBLAU_H_
10
11 #include <math.h>
12 #include <stddef.h>
13
14 void himmelblau(double* x, double* f, double* df, void *p)
15 {
16     *f = pow(x[0]*x[0] + x[1] - 11,2) + pow(x[0] + x[1]*x[1] - 7,2);
17
18     df[0] = 2*(2*x[0]*(x[0]*x[0] + x[1] - 11) + x[0] + x[1]*x[1] - 7);
19     df[1] = 2*(-11+x[0]*x[0] + x[1] + 2*x[1]*(-7 + x[0] + x[1]*x[1]));
20
21     return;
22 }
23
24 //Example of using the algorithm.
25 double TestHimmelblau()
26 {
27
28     //Number of inputs, number of max iterations
29     int n = 2, maxiters = 100;
30
31     //Setting options.
32     //      Δ - Initial value for largest stepsize used by the
33     //      linesearching.
34     //      tolx - The value which the norm of g is under when the
35     //      1st order KKT conditions are met.
36     //      tolx - If the stepsize is less than tolx*(tolx +
37     //      opti_norm(x, n, 2)), stop.
38     //      maxeval - The maximum number of tolerable iterations
39     //      before stopping.
40     //      findiffh - Used in the finite difference approximation for
41     //      fun(). If set to 0, df is expected.
42     //      Warning: Using findiffh = 0 and not supplying g
43     //      in fun WILL result in errors.
44     //      Default: 1, 1e-4, 1e-8, 100, 0
45     double opts[] = {1.0, 1e-4, 1e-8, maxiters, 0};
46
47     //The function shall be called on the function, using x as starting guess,

```

```

42     //with n values, opts shall be used for options, no initial approximation
        of inverse Hessian,
43     //want number of iterations used (not printed), want the performance info,
44     //we want all x iterations, no parameters.
45
46     //Starting guess 1
47     double x1[2] = {1, 1};
48     double t1 = opti_get_time();
49     int ucminfr = opti_ucminf(himmelblau, x1, n, opts, NULL, NULL, NULL, NULL,
        NULL);
50     double t2 = opti_get_time();
51
52     return t2 - t1;
53 }
54
55 #endif /* HIMMELBLAU_H_ */

```

## C.3 Rosenbrock

### C.3.1 MATLAB test

```

1  for i = 1:10
2      tic,
3      y(i,:) = ucminf(@rosenbrock2,[4, 2]');
4      t1(i) = toc;
5  end
6  time1 = sum(t1)/10;
7
8  for i = 1:10
9      tic,
10     x(i,:) = fminunc(@rosenbrock,[4, 2]');
11     t2(i) = toc;
12 end
13 time2 = sum(t2)/10;
14
15
16 t1, time1, t2, time2

```

### C.3.2 minlbfgs test

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <cmath>
5  #include "ap.h"
6  #include "minlbfgs.h"
7
8  using namespace std;
9

```

```

10 //Needs to be put into alltests!
11 #include <time.h>
12 #include <sys/time.h>
13 #ifdef __MACH__ //If on a Mac, use Mach libraries
14 #include <mach/clock.h>
15 #include <mach/mach.h>
16 #endif //__MACH__
17
18 double get_time()
19 {
20     struct timespec ts;
21 #ifdef __MACH__ // OS X does not have clock_gettime, use clock_get_time
22     clock_serv_t cclock;
23     mach_timespec_t mts;
24     host_get_clock_service(mach_host_self(), CALENDAR_CLOCK, &cclock);
25     clock_get_time(cclock, &mts);
26     mach_port_deallocate(mach_task_self(), cclock);
27     ts.tv_sec = mts.tv_sec;
28     ts.tv_nsec = mts.tv_nsec;
29 #else
30 #ifdef CLOCK_MONOTONIC
31     clock_gettime(CLOCK_MONOTONIC, &ts);
32 #else
33     clock_gettime(CLOCK_REALTIME, &ts);
34 #endif
35 #endif
36     return (double) ts.tv_sec + (double) ts.tv_nsec * 1e-9;
37 }
38
39 void rosenbrock(double* x, double* f, double* df, void *p)
40 {
41     *f = pow((double) (1 - x[0]), (double) 2) + 100 * pow((double) (x[1] - pow(
42         ((double) x[0], (double) 2)), (double) 2);
43     df[0] = -2 + 2 * x[0] - 400 * (x[1] - pow((double) x[0], (double) 2)) * x
44         [0];
45     df[1] = 200 * x[1] - 200 * pow((double) x[0], (double) 2);
46     return;
47 }
48
49 //=====
50
51 int main(int argc, char **argv)
52 {
53     int n = 2;
54     int m = 1;
55     minlbfgsstate state;
56     minlbfgsreport rep;
57     ap::real_ld_array s;
58     double x[n];
59
60     double time = 0;
61
62     n = 2;
63     m = 1;
64     s.setlength(n);
65
66
67     //f and g
68     double f;
69     double g[n];
70
71     for (int i = 0; i < 10; ++i)
72     {

```

```

73     s(0) = 4;
74     s(1) = 2;
75     double t1 = get_time();
76     minlbfgscreate(n, m, s, state);
77     minlbfgssetcond(state, 0.0, 0.0, 0.0001, 0);
78     while(minlbfgsiteration(state))
79     {
80         if( state.needfg )
81         {
82             for (int k = 0; k < n; ++k)
83                 x[k] = state.x(k);
84
85             rosenbrock(x, &f, g, NULL);
86
87             state.f = f;
88
89             for (int k = 0; k < n; ++k)
90                 state.g(k) = g[k];
91         }
92     }
93     minlbfgsresults(state, s, rep);
94
95     double t2 = get_time();
96
97     time += t2 - t1;
98
99     printf("Solve time: %f\n", t2 - t1);
100 }
101
102 printf("Result: (%f, %f)\n", state.x(0), state.x(1));
103 printf("Average solve time: %f\n", time /= 10);
104
105 return 0;
106 }

```

### C.3.3 libopti test

```

1  /*
2  * rosenbrock.h
3  *
4  * Created on: Aug 18, 2012
5  * Author: christian
6  */
7
8  #ifndef ROSENBRACK_H_
9  #define ROSENBRACK_H_
10
11 #include <math.h>
12 #include <stddef.h>
13
14 void rosenbrock(double* x, double* f, double* df, void *p)
15 {
16     *f = pow((double) (1 - x[0]), (double) 2) + 100 * pow((double) (x[1] - pow
17         ((double) x[0], (double) 2)), (double) 2);
18
19     df[0] = -2 + 2 * x[0] - 400 * (x[1] - pow((double) x[0], (double) 2)) * x
20         [0];
21     df[1] = 200 * x[1] - 200 * pow((double) x[0], (double) 2);
22 }

```

```

21     return;
22 }
23
24 //Example of using the algorithm.
25 double TestRosenbrock()
26 {
27
28     //Number of inputs to n, number of max iterations
29     int maxiters = 100;
30     int n = 2;
31     double x[2] = {4, 2};
32
33     //Setting options
34     double opts[] = {1.0, 0.0001, 0.00000001, maxiters, 0};
35
36     //The function shall be called on the function, using x as starting guess,
37     //with n values, opts shall be used for options, no initial approximation
38     //of inverse Hessian,
39     double t1 = opti_get_time();
40     int ucmnfr = opti_ucminf(beale, x, n, opts, NULL, NULL, NULL, NULL, NULL)
41     ;
42     double t2 = opti_get_time();
43
44     return t2 - t1;
45 }
46 #endif /* ROSENBROCK_H_ */

```

## C.4 Beale

### C.4.1 MATLAB test

```

1  for i = 1:10
2      tic,
3      y(i,:) = ucmnfr(@beale2,[4, 2]');
4      t1(i) = toc;
5  end
6  time1 = sum(t1)/10;
7
8  for i = 1:10
9      tic,
10     x(i,:) = fminunc(@beale,[4, 2]');
11     t2(i) = toc;
12 end
13 time2 = sum(t2)/10;
14
15
16 t1, time1, t2, time2

```

### C.4.2 minlbfgs test

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <cmath>
5  #include "ap.h"
6  #include "minlbfgs.h"
7
8  using namespace std;
9
10 //Needs to be put into alltests!
11 #include <time.h>
12 #include <sys/time.h>
13 #ifdef __MACH__ //If on a Mac, use Mach libraries
14 #include <mach/clock.h>
15 #include <mach/mach.h>
16 #endif //__MACH__
17
18 double get_time()
19 {
20     struct timespec ts;
21     #ifdef __MACH__ // OS X does not have clock_gettime, use clock_get_time
22     clock_serv_t cclock;
23     mach_timespec_t mts;
24     host_get_clock_service(mach_host_self(), CALENDAR_CLOCK, &cclock);
25     clock_get_time(cclock, &mts);
26     mach_port_deallocate(mach_task_self(), cclock);
27     ts.tv_sec = mts.tv_sec;
28     ts.tv_nsec = mts.tv_nsec;
29     #else
30     #ifdef CLOCK_MONOTONIC
31     clock_gettime(CLOCK_MONOTONIC, &ts);
32     #else
33     clock_gettime(CLOCK_REALTIME, &ts);
34     #endif
35     #endif
36     return (double) ts.tv_sec + (double) ts.tv_nsec * 1e-9;
37 }
38
39 void beale(double* x, double* f, double* df, void *p)
40 {
41     *f = pow(0.15e1 - x[0] + x[0] * x[1], 0.2e1) + pow(0.225e1 - x[0] + x[0] *
42         pow(x[1], 0.2e1), 0.2e1) + pow(0.2625e1 - x[0] + x[0] * pow(x[1],
43         0.3e1), 0.2e1);
44
45     df[0] = 0.2e1 * (0.15e1 - x[0] + x[0] * x[1]) * (-0.1e1 + x[1]) + 0.2e1 *
46         (0.225e1 - x[0] + x[0] * pow(x[1], 0.2e1)) * (-0.1e1 + pow(x[1], 0.2
47         e1)) + 0.2e1 * (0.2625e1 - x[0] + x[0] * pow(x[1], 0.3e1)) * (-0.1e1
48         + pow(x[1], 0.3e1));
49     df[1] = 0.2e1 * (0.15e1 - x[0] + x[0] * x[1]) * x[0] + 0.4e1 * (0.225e1 -
50         x[0] + x[0] * pow(x[1], 0.2e1)) * x[0] * x[1] + 0.6e1 * (0.2625e1 - x
51         [0] + x[0] * pow(x[1], 0.3e1)) * x[0] * pow(x[1], 0.2e1);
52
53     return;
54 }
55
56 //=====
57
58 int main(int argc, char **argv)
59 {
60     int n = 2;
61     int m = 1;
62     minlbfgsstate state;
63     minlbfgsreport rep;
64     ap::real_ld_array s;

```

```

58     double x[n];
59
60     double time = 0;
61
62     n = 2;
63     m = 1;
64     s.setlength(n);
65
66
67     //f and g
68     double f;
69     double g[n];
70
71     for (int i = 0; i < 10; ++i)
72     {
73         s(0) = 4;
74         s(1) = 2;
75         double t1 = get_time();
76         minlbfgscreate(n, m, s, state);
77         minlbfgssetcond(state, 0.0, 0.0, 0.0001, 0);
78         while(minlbfgsiteration(state))
79         {
80             if( state.needfg )
81             {
82                 for (int k = 0; k < n; ++k)
83                     x[k] = state.x(k);
84
85                 beale(x, &f, g, NULL);
86
87                 state.f = f;
88
89                 for (int k = 0; k < n; ++k)
90                     state.g(k) = g[k];
91             }
92         }
93         minlbfgsresults(state, s, rep);
94
95         double t2 = get_time();
96
97         time += t2 - t1;
98
99         printf("Solve time: %f\n", t2 - t1);
100     }
101
102     printf("Result: (%f, %f)\n", state.x(0), state.x(1));
103     printf("Average solve time: %f\n", time /= 10);
104
105     return 0;
106 }

```

### C.4.3 libopti test

```

1  /*
2  * beale.h
3  *
4  * Created on: Aug 18, 2012
5  * Author: christian
6  */
7

```



```

 8  #ifndef BEALE_H_
 9  #define BEALE_H_
10
11  #include <math.h>
12
13  void beale(double* x, double* f, double* df, void *p)
14  {
15      *f = pow(0.15e1 - x[0] + x[0] * x[1], 0.2e1) + pow(0.225e1 - x[0] + x[0] *
          pow(x[1], 0.2e1), 0.2e1) + pow(0.2625e1 - x[0] + x[0] * pow(x[1],
          0.3e1), 0.2e1);
16
17      df[0] = 0.2e1 * (0.15e1 - x[0] + x[0] * x[1]) * (-0.1e1 + x[1]) + 0.2e1 *
          (0.225e1 - x[0] + x[0] * pow(x[1], 0.2e1)) * (-0.1e1 + pow(x[1], 0.2
          e1)) + 0.2e1 * (0.2625e1 - x[0] + x[0] * pow(x[1], 0.3e1)) * (-0.1e1
          + pow(x[1], 0.3e1));
18      df[1] = 0.2e1 * (0.15e1 - x[0] + x[0] * x[1]) * x[0] + 0.4e1 * (0.225e1 -
          x[0] + x[0] * pow(x[1], 0.2e1)) * x[0] * x[1] + 0.6e1 * (0.2625e1 - x
          [0] + x[0] * pow(x[1], 0.3e1)) * x[0] * pow(x[1], 0.2e1);
19
20      return;
21  }
22
23  //Example of using the algorithm.
24  double TestBeale()
25  {
26
27      //Number of inputs, number of max iterations
28      int n = 2, maxiters = 100;
29
30      //Setting options.
31      //      Δ - Initial value for largest stepsize used by the
          linesearching.
32      //      tolg - The value which the norm of g is under when the
          1st order KKT conditions are met.
33      //      tolx - If the stepsize is less than tolx*(tolx +
          opti_norm(x, n, 2)), stop.
34      //      maxeval - The maximum number of tolerable iterations
          before stopping.
35      //      findiffh - Used in the finite difference approximation for
          fun(). If set to 0, df is expected.
36      //      Warning: Using findiffh = 0 and not supplying g
          in fun WILL result in errors.
37      //      Default: 1, 1e-4, 1e-8, 100, 0
38      double opts[] = {1.0, 1e-4, 1e-8, maxiters, 0};
39
40      //The function shall be called on the function, using x as starting guess,
41      //with n values, opts shall be used for options, no initial approximation
          of inverse Hessian,
42
43      //Starting guess 1
44      double x1[2] = {4, 2};
45      double t1 = opti_get_time();
46      int ucminfr = opti_ucminf(beale, x1, n, opts, NULL, NULL, NULL, NULL, NULL
          );
47      double t2 = opti_get_time();
48
49      return t2 - t1;
50  }
51
52  #endif /* BEALE_H_ */

```

## C.5 Easom

### C.5.1 MATLAB test

```

1  for i = 1:10
2      tic,
3      [y, info, perf] = ucmintf(@easom2,[1, 1]', [1 1e-4 1e-8 100]);
4      t1(i) = toc;
5  end
6  time1 = sum(t1)/10;
7
8  for i = 1:10
9      tic,
10     x(i,:) = fminunc(@easom,[1, 1]');
11     t2(i) = toc;
12 end
13 time2 = sum(t2)/10;
14
15
16 t1, time1, t2, time2

```

### C.5.2 minlbfgs test

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <cmath>
5  #include "ap.h"
6  #include "minlbfgs.h"
7
8  using namespace std;
9
10 //Needs to be put into alltests!
11 #include <time.h>
12 #include <sys/time.h>
13 #ifdef __MACH__ //If on a Mac, use Mach libraries
14 #include <mach/clock.h>
15 #include <mach/mach.h>
16 #endif //__MACH__
17
18 double get_time()
19 {
20     struct timespec ts;
21     #ifdef __MACH__ // OS X does not have clock_gettime, use clock_get_time
22     clock_serv_t cclock;
23     mach_timespec_t mts;
24     host_get_clock_service(mach_host_self(), CALENDAR_CLOCK, &cclock);
25     clock_get_time(cclock, &mts);
26     mach_port_deallocate(mach_task_self(), cclock);
27     ts.tv_sec = mts.tv_sec;
28     ts.tv_nsec = mts.tv_nsec;
29     #else
30     #ifdef CLOCK_MONOTONIC
31     clock_gettime(CLOCK_MONOTONIC, &ts);

```

```

32 #else
33     clock_gettime(CLOCK_REALTIME, &ts);
34 #endif
35 #endif
36     return (double) ts.tv_sec + (double) ts.tv_nsec * 1e-9;
37 }
38
39 void easom(double* x, double* f, double* df, void *p)
40 {
41     double pi = 3.14159265358979323846;
42     *f = -cos(x[0])*cos(x[1])*exp(-pow((x[0]-pi),2) - pow((x[1] - pi),2));
43     df[0] = sin(x[0])*cos(x[1])*exp(-pow((x[0]-pi),2) - pow((x[1] - pi),2))
44           + -cos(x[0])*cos(x[1])*(-2*x[0] + 2*pi)*exp(-pow((x[0]-pi),2) - pow
45               ((x[1] - pi),2));
46     df[1] = cos(x[0])*sin(x[1])*exp(-pow((x[0]-pi),2) - pow((x[1] - pi),2))
47           + -cos(x[0])*cos(x[1])*(-2*x[1] + 2*pi)*exp(-pow((x[0]-pi),2) - pow
48               ((x[1] - pi),2));
49 }
50 //=====
51 int main(int argc, char **argv)
52 {
53     int n = 2;
54     int m = 1;
55     minlbfgsstate state;
56     minlbfgsreport rep;
57     ap::real_ld_array s;
58     double x[n];
59
60     double time = 0;
61
62     n = 2;
63     m = 1;
64     s.setlength(n);
65
66
67     //f and g
68     double f;
69     double g[n];
70
71     for (int i = 0; i < 10; ++i)
72     {
73         s(0) = 1;
74         s(1) = 1;
75         double t1 = get_time();
76         minlbfgscreate(n, m, s, state);
77         minlbfgssetcond(state, 1e-10, 1e-10, 1e-10, 100);
78         while(minlbfgsiteration(state))
79         {
80             if( state.needfg )
81             {
82                 for (int k = 0; k < n; ++k)
83                     x[k] = state.x(k);
84
85                 easom(x, &f, g, NULL);
86
87                 state.f = f;
88
89                 for (int k = 0; k < n; ++k)
90                     state.g(k) = g[k];
91             }
92         }
93         minlbfgsresults(state, s, rep);
94

```

```

95     double t2 = get_time();
96
97     time += t2 - t1;
98
99     printf("Solve time: %f\n", t2 - t1);
100 }
101
102     printf("Result: (%f, %f)\n", state.x(0), state.x(1));
103     printf("Result: (%f, %f)\n", state.g(0), state.g(1));
104     printf("Result: %f\n", state.f);
105     printf("Average solve time: %f\n", time /= 10);
106
107     return 0;
108 }

```

### C.5.3 libopti test

```

1  /*
2  * easom.h
3  *
4  * Created on: Aug 18, 2012
5  * Author: christian
6  */
7
8  #ifndef EASOM_H_
9  #define EASOM_H_
10
11 #include <math.h>
12 #include <stddef.h>
13 #include <stdio.h>
14
15 void easom(double* x, double* f, double* df, void *p)
16 {
17     double pi = 3.14159265358979323846;
18     *f = -cos(x[0])*cos(x[1])*exp(-pow((x[0]-pi),2) - pow((x[1] - pi),2));
19     df[0] = sin(x[0])*cos(x[1])*exp(-pow((x[0]-pi),2) - pow((x[1] - pi),2))
20         + -cos(x[0])*cos(x[1])*(-2*x[0] + 2*pi)*exp(-pow((x[0]-pi),2) - pow
21             ((x[1] - pi),2));
22     df[1] = cos(x[0])*sin(x[1])*exp(-pow((x[0]-pi),2) - pow((x[1] - pi),2))
23         + -cos(x[0])*cos(x[1])*(-2*x[1] + 2*pi)*exp(-pow((x[0]-pi),2) - pow
24             ((x[1] - pi),2));
25 }
26
27 //Example of using the algorithm.
28 double TestEasom()
29 {
30     //Number of inputs, number of max iterations
31     int n = 2, maxiters = 100;
32
33     //Setting options.
34     // Δ - Initial value for largest stepsize used by the
35     // linesearching.
36     // tolg - The value which the norm of g is under when the
37     // 1st order KKT conditions are met.
38     // tolx - If the stepsize is less than tolx*(tolx +
39     // opti_norm(x, n, 2)), stop.
40     // maxeval - The maximum number of tolerable iterations
41     // before stopping.

```

```

37 //          findiffh - Used in the finite difference approximation for
          fun(). If set to 0, df is expected.
38 //          Warning: Using findiffh = 0 and not supplying g
          in fun WILL result in errors.
39 //          Default: 1, 1e-4, 1e-8, 100, 0
40 double opts[] = {1.0, 1e-8, 1e-8, maxiters, 0};
41
42 //The function shall be called on the function, using x as starting guess,
43 //with n values, opts shall be used for options, no initial approximation
          of inverse Hessian,
44
45 //Starting guess 1
46 double x1[2] = {1, 1};
47 double t1 = opti_get_time();
48 int ucminfr = opti_ucminf(easom, x1, n, opts, NULL, NULL, NULL, NULL, NULL
          );
49 double t2 = opti_get_time();
50
51 double df[2], f;
52 easom(x1, &f, df, NULL);
53 printf("%f, %f\n", x1[0],x1[1]);
54 return t2 - t1;
55 }
56
57 #endif /* EASOm_H_ */

```

## C.6 Ackley

### C.6.1 MATLAB test

```

1 for i = 1:10
2     tic,
3     yin = (rand([5000,1])-0.5)*6;
4     [y, info, perf] = ucminf(@ackley2,yin, [1 1e-8 1e-8 1000]);
5     [f g] = ackley2(y);
6     f, h = norm(g),
7     t1(i) = toc;
8 end
9 time1 = sum(t1)/10;
10
11 for i = 1:10
12     tic,
13     xin = (rand([5000,1])-0.5)*6;
14     x = fminunc(@ackley,xin);
15     t2(i) = toc;
16 end
17 time2 = sum(t2)/10;
18
19
20 t1, time1, t2, time2

```

## C.6.2 minlbfgs test

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <cmath>
4  #include "ap.h"
5  #include "minlbfgs.h"
6
7  using namespace std;
8
9  //Needs to be put into alltests!
10 #include <time.h>
11 #include <sys/time.h>
12 #ifdef __MACH__ //If on a Mac, use Mach libraries
13 #include <mach/clock.h>
14 #include <mach/mach.h>
15 #endif //__MACH__
16
17 double get_time()
18 {
19     struct timespec ts;
20 #ifdef __MACH__ // OS X does not have clock_gettime, use clock_get_time
21     clock_serv_t cclock;
22     mach_timespec_t mts;
23     host_get_clock_service(mach_host_self(), CALENDAR_CLOCK, &cclock);
24     clock_get_time(cclock, &mts);
25     mach_port_deallocate(mach_task_self(), cclock);
26     ts.tv_sec = mts.tv_sec;
27     ts.tv_nsec = mts.tv_nsec;
28 #else
29 #ifdef CLOCK_MONOTONIC
30     clock_gettime(CLOCK_MONOTONIC, &ts);
31 #else
32     clock_gettime(CLOCK_REALTIME, &ts);
33 #endif
34 #endif
35     return (double) ts.tv_sec + (double) ts.tv_nsec * 1e-9;
36 }
37
38 void ackley(double* x, double* f, double* df, void *p)
39 {
40     double a, b, c; //Constants
41     int i, n; //Iterator and dimensions
42
43     c = 2*M_PI;
44     n = 5000;
45
46
47     //Calculating function value
48     double sum1 = 0, sum2 = 0; //Defining sums
49
50     for (i = 0; i < (int) n; ++i)
51     {
52         sum1 += x[i]*x[i];
53         sum2 += cos(c*x[i]);
54     }
55
56     *f = -a*exp(-b*sqrt((1.0/n)*sum1)) - exp((1.0/n)*sum2) + a + exp(1.0);
57
58     //Calculating derivatives
59     int j;
60
61     for (j = 0; j < (int) n; ++j)

```

```

62     {
63         df[j] = 0; //Initializing to 0.
64
65         df[j] += a*b*x[j]*exp(-(1.0/n)*b*sqrt(n*sum1))/sqrt(n*sum1);
66
67         df[j] += -1.0/n*sin(c*x[j])*exp(1.0/n*sum2);
68     }
69
70
71
72     return;
73 }
74
75 //=====
76
77 int main(int argc, char **argv)
78 {
79     int n = 5000;
80     int m = 1;
81     minlbfgsstate state;
82     minlbfgsreport rep;
83     ap::real_1d_array s;
84     double x[n];
85
86     double timet = 0;
87
88     s.setlength(n);
89
90
91     //f and g
92     double f;
93     double g[n];
94
95     for (int i = 0; i < 10; ++i)
96     {
97         //Generate random starting guess
98         unsigned int iseed = (unsigned int) time(NULL);
99         srand (iseed);
100        //Randomly generate numbers
101        for (int k = 0; k < n; ++k) s(k) = (rand()%600 - 300)/100.0;
102
103        double t1 = get_time();
104        minlbfgscreate(n, m, s, state);
105        minlbfgssetcond(state, 1e-10, 1e-10, 1e-10, 1000);
106        while(minlbfgsiteration(state))
107        {
108            if( state.needfg )
109            {
110                for (int k = 0; k < n; ++k)
111                    x[k] = state.x(k);
112
113                ackley(x, &f, g, NULL);
114
115                state.f = f;
116
117                for (int k = 0; k < n; ++k)
118                    state.g(k) = g[k];
119            }
120        }
121        minlbfgsresults(state, s, rep);
122
123        double t2 = get_time();
124
125        timet += t2 - t1;
126

```

```

127     printf("Solve time: %f\n", t2 - t1);
128     }
129
130     printf("Result: (%f, %f)\n", state.x(0), state.x(1));
131     printf("Result: (%f, %f)\n", state.g(0), state.g(1));
132     printf("Result: %f\n", state.f);
133     printf("Average solve time: %f\n", timet /= 10);
134
135     return 0;
136 }

```

### C.6.3 libopti test

```

1  /*
2  * Ackley.h
3  *
4  * Created on: Aug 17, 2012
5  * Author: christian
6  */
7
8  #ifndef ACKLEY_H_
9  #define ACKLEY_H_
10
11 #include <math.h>
12
13 void ackley(double* x, double* f, double* df, void *p)
14 {
15
16     double a, b, c; //Constants
17     int i, n; //Iterator and dimensions
18
19     a = *((double*) p);
20     b = *((double*) p+1);
21     c = *((double*) p+2);
22     n = *((double*) p+3);
23
24
25     //Calculating function value
26     double sum1 = 0, sum2 = 0; //Defining sums
27
28     for (i = 0; i < (int) n; ++i)
29     {
30         sum1 += x[i]*x[i];
31         sum2 += cos(c*x[i]);
32     }
33
34     *f = -a*exp(-b*sqrt((1.0/n)*sum1)) - exp((1.0/n)*sum2) + a + exp(1.0);
35
36     //Calculating derivatives
37     int j;
38
39     for (j = 0; j < (int) n; ++j)
40     {
41         df[j] = 0; //Initializing to 0.
42
43         df[j] += a*b*x[j]*exp(-1.0/n)*b*sqrt(n*sum1)/sqrt(n*sum1);
44
45         df[j] += -1.0/n*sin(c*x[j])*exp(1.0/n*sum2);
46     }

```



```

47
48
49
50     return;
51 }
52
53 //Example of using the algorithm.
54 double TestAckley()
55 {
56     //Generate random starting guess
57     unsigned int  iseed = (unsigned int) time(NULL);
58     srand (iseed);
59
60     //Number of inputs to n, number of max iterations
61     int n = 5000;
62     int maxiters = 10000;
63     double x[n];
64     int i;
65
66     //Randomly generate numbers
67     for (i = 0; i < n; ++i) x[i] = (rand()%600 - 300)/100.0;
68
69     //Define array with params
70     double pi = 3.141592654;
71
72     double p[] = {20, 0.2, 2*pi, (double) n};
73
74
75     //Setting options
76     double opts[] = {1.0, 1e-4, 1e-8, maxiters, 0};
77
78     //The function shall be called on the function, using x as starting guess,
79     //with n values, opts shall be used for options, no initial approximation of
80     //inverse Hessian,
81     double xinfo[n*maxiters];
82     double t1 = opti_get_time();
83     int ucminfr = opti_ucminf(ackley, x, n, opts, NULL, NULL, NULL, NULL, (void
84     *) p);
85     double t2 = opti_get_time();
86
87     double f, df[n];
88
89     ackley(x, &f, df, (void*) p);
90
91     printf("%f\n", f);
92     return t2 - t1;
93 }
94 #endif /* ACKLEY_H_ */

```

## C.7 Zakharov

### C.7.1 MATLAB test

```

1 for i = 1:10

```

```

2     tic,
3     yin = ones(1000,1);
4     y = ucminf(@zakh2,yin, [1 1e-10, 1e-10, 1000]);
5     [f g] = zakh2(y);
6     f, h = norm(g),
7     t1(i) = toc;
8 end
9 time1 = sum(t1)/10;
10
11 for i = 1:10
12     tic,
13     xin = ones(1000,1);
14     x = fminunc(@zakh,xin);
15     t2(i) = toc;
16 end
17 time2 = sum(t2)/10;
18
19
20 t1, time1, t2, time2

```

## C.7.2 minlbfgs test

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <cmath>
4 #include "ap.h"
5 #include "minlbfgs.h"
6
7 using namespace std;
8
9 //Needs to be put into alltests!
10 #include <time.h>
11 #include <sys/time.h>
12 #ifdef __MACH__ //If on a Mac, use Mach libraries
13 #include <mach/clock.h>
14 #include <mach/mach.h>
15 #endif //__MACH__
16
17 double get_time()
18 {
19     struct timespec ts;
20 #ifdef __MACH__ // OS X does not have clock_gettime, use clock_get_time
21     clock_serv_t cclock;
22     mach_timespec_t mts;
23     host_get_clock_service(mach_host_self(), CALENDAR_CLOCK, &cclock);
24     clock_get_time(cclock, &mts);
25     mach_port_deallocate(mach_task_self(), cclock);
26     ts.tv_sec = mts.tv_sec;
27     ts.tv_nsec = mts.tv_nsec;
28 #else
29 #ifdef CLOCK_MONOTONIC
30     clock_gettime(CLOCK_MONOTONIC, &ts);
31 #else
32     clock_gettime(CLOCK_REALTIME, &ts);
33 #endif
34 #endif
35     return (double) ts.tv_sec + (double) ts.tv_nsec * 1e-9;
36 }
37

```

```

38 void zakharov(double* x, double* f, double* df)
39 {
40
41     int n, i;
42
43     n = 1000;
44
45     *f = 0;
46
47     double fpart23 = 0;
48
49     for (i = 0; i < n; ++i)
50     {
51         *f += x[i]*x[i];
52         fpart23 += 0.5*i*x[i];
53     }
54
55     *f += pow((fpart23),2) + pow((fpart23),4);
56
57
58 }
59
60 //=====
61
62 int main(int argc, char **argv)
63 {
64     int n = 1000;
65     int m = 1;
66     minlbfgsstate state;
67     minlbfgsreport rep;
68     ap::real_1d_array s;
69     double x[n];
70
71     double timet = 0;
72
73     s.setlength(n);
74
75
76     //f and g
77     double f;
78     double g[n];
79
80     for (int i = 0; i < 10; ++i)
81     {
82         for (int k = 0; k < n; ++k) s(k) = 1;
83
84         double t1 = get_time();
85         minlbfgscreate(n, m, s, state);
86         minlbfgssetcond(state, 1e-10, 1e-10, 1e-10, 1000);
87         while(minlbfgsiteration(state))
88         {
89             if( state.needfg )
90             {
91                 for (int k = 0; k < n; ++k)
92                     x[k] = state.x(k);
93
94                 zakharov(x, &f, g);
95
96                 state.f = f;
97
98                 for (int k = 0; k < n; ++k)
99                     state.g(k) = g[k];
100             }
101         }
102     minlbfgsresults(state, s, rep);

```

```
103     double t2 = get_time();
104
105     timet += t2 - t1;
106
107     printf("Solve time: %f\n", t2 - t1);
108     }
109
110     printf("Average solve time: %f\n", timet /= 10);
111
112     return 0;
113 }
114 }
```

### C.7.3 libopti test

```
1  /*
2  * zakharov.h
3  *
4  * Created on: Aug 18, 2012
5  * Author: christian
6  */
7
8  #ifndef ZAKHAROV_H_
9  #define ZAKHAROV_H_
10
11 #include <math.h>
12 #include <stddef.h>
13 #include <stdio.h>
14
15 void zakharov(double* x, double* f, double* df, void *p)
16 {
17
18     int n, i;
19
20     n = *((int*) p);
21
22     *f = 0;
23
24     double fpart23 = 0;
25
26     for (i = 0; i < n; ++i)
27     {
28         *f += x[i]*x[i];
29         fpart23 += 0.5*i*x[i];
30     }
31
32     *f += pow((fpart23),2) + pow((fpart23),4);
33
34
35 }
36
37 //Example of using the algorithm.
38 double TestZakharov()
39 {
40
41     //Number of inputs, number of max iterations
42     int n = 10, maxiters = 1000;
43
44     //Setting options.
```

```

45     //          Δ - Initial value for largest stepsize used by the
46     //          linesearching.
47     //          tolg - The value which the norm of g is under when the
48     //          1st order KKT conditions are met.
49     //          tolx - If the stepsize is less than tolx*(tolx +
50     //          opti_norm(x, n, 2)), stop.
51     //          maxeval - The maximum number of tolerable iterations
52     //          before stopping.
53     //          findiffh - Used in the finite difference approximation for
54     //          fun(). If set to 0, df is expected.
55     //          Warning: Using findiffh = 0 and not supplying g
56     //          in fun WILL result in errors.
57     //          Default: 1, 1e-4, 1e-8, 100, 0
58     double opts[] = {1.0, 1e-8, 1e-8, maxiters, 0.0001};
59
60     //The function shall be called on the function, using x as starting guess,
61     //with n values, opts shall be used for options, no initial approximation
62     //of inverse Hessian,
63
64     double perfinfo[maxiters*6];
65     double xinfo[n*maxiters];
66
67     //Starting guess 1
68     double x[n];
69     int i; for (i = 0; i < n; ++i) x[i] = 1;
70     double t1 = opti_get_time();
71     int ucminfr = opti_ucminf(zakharov, x, n, opts, NULL, NULL, perfinfo,
72     xinfo, (void*) &n);
73     double t2 = opti_get_time();
74
75     double df[2], f;
76     zakharov(x, &f, df, (void*) &n);
77     printf("size x = %f\n", x[5]);
78     return t2 - t1;
79 }
80 #endif /* ZAKHAROV_H_ */

```

## C.8 Plot Ackley performance

```

1  clear all
2  load testackley/am.txt
3  load testackley/Δ.txt
4  load testackley/evals.txt
5  load testackley/finalslope.txt
6  load testackley/fun.txt
7  load testackley/ng.txt
8
9
10 subplot(3,2,1);
11 plot(1:length(am),am);
12 xlabel iteration, ylabel alpha
13
14 subplot(3,2,2);
15 plot(1:length(am),Δ);
16 xlabel iteration, ylabel Δ,
17
18 subplot(3,2,3);

```

```
19 plot(1:length(am),evals);
20 xlabel iteration, ylabel evals,
21
22 subplot(3,2,4);
23 plot(1:length(am),finalslope);
24 xlabel iteration, ylabel 'final slope',
25
26 subplot(3,2,5);
27 plot(1:length(am),fun);
28 xlabel iteration, ylabel f,
29
30 subplot(3,2,6);
31 plot(1:length(am),ng);
32 xlabel iteration, ylabel norm(g),
```

# Bibliography

---

- [ABB<sup>+</sup>99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [Ack87] D. H. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer, Boston, 1987.
- [ALG12a] ALGLIB. Commercial support for alglib. <http://www.alglib.net/commercial.php>, 2012.
- [ALG12b] ALGLIB. Unconstrained optimization: L-bfgs and cg. <http://www.alglib.net/optimization/lbfgsandcg.php>, 2012.
- [Bäc96] T. Bäck. *Evolutionary algorithms in theory and practice*. Oxford University Press, 1996.
- [Bea58] E. Beale. On an iterative method for finding a local minimum of a function of more than one variable. Technical report, Statistical Techniques Research Group, Princeton University, 1958.
- [Eas90] E. Easom. A survey of global optimization techniques. Technical report, University of Louisville, 1990.
- [Fou12] Free Software Foundation. Gnu compiler collection. <http://gcc.gnu.org/>, 2012.
- [Han08] Vagn Lundsgaard Hansen. *Entrance to Advanced Mathematics: The metric foundations of modern analysis*. Institut for Matematik, DTU, 2008.

- [HBN10] K. Madsen H. B. Nielsen. *Introduction to Optimization and Data Fitting*. DTU Informatics, IMM, 2010.
- [Hed12a] Abdel-Rehman Hedar. Global optimization test problems. [http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar\\_files/TestGO.htm](http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO.htm), 2012.
- [Hed12b] Dr. Abdel-Rahman Hedar. Zakharov function. [http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar\\_files/TestGO\\_files/Page3088.htm](http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page3088.htm), 2012.
- [HEJK06] P. Hjorth H. E. Jensen, S. Markvorsen and W. Kliem. *Matematisk Analyse 1*. Institut for Matematik, DTU, third edition, 2006.
- [Him72] D. Himmelblau. *Applied nonlinear programming*. McGraw-Hill, 1972.
- [Inc12a] Apple Inc. The objective-c programming language. <https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html>, 2012.
- [Inc12b] Apple Inc. What's new in ios4.0. <http://developer.apple.com/library/ios/#releasenotes/General/WhatsNewIniPhoneOS/Articles/iPhoneOS4.html>, 2012.
- [JN00] Steven Wright Jorge Nocedal. *Numerical Optimization*. Springer, second edition, 2000.
- [Kre89] Erwin Kreyszig. *Introductory Functional Analysis with Applications*. Wiley, 1989.
- [Lin12] Yo Linux. Tutorial - using c/c++ and fortran together. <http://www.yolinux.com/TUTORIALS/LinuxTutorialMixingFortranAndC.html>, 2012.
- [Mey00] Carl D. Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, 2000.
- [Net12] Netlib. Blas (basic linear algebra subprograms). <http://www.netlib.org/blas/index.html>, 2012.
- [oF12] Aalto University of Finland. History of optimization. <http://www.mitrikitti.fi/opthist.html>, 2012.
- [Ros60] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 1960.
- [Sch07] Gert Schomacker. *Gyldendals Gymnasiematematik A*. Gyldendal, 2007.