Aalto University
School of Science
Degree programme in Engineering Physics and Mathematics

Comparison of line search methods in unconstrained optimization

Bachelor's Thesis
1.5.2019

Einari Tuukkanen

Aalto-yliopisto

AALTO-YLIOPISTO PERUSTIETEIDEN KORKEAKOULU PL 11000, 00076 Aalto http://www.aalto.fi	KANDIDAATINTYÖN TIIVISTELMÄ
Tekijä: Teppo Teekkari	
Työn nimi: Teekkarien valmistumistodennäköisyyden mallintaminen Markov-ketjuilla	
Tutkinto-ohjelma: Teknillisen fysiikan ja matematiikan tutkinto-ohjelma	
Pääaine: Systeemitieteet	Pääaineen koodi: F3010
Vastuuopettaja(t): Prof. Ansio Akateemikko Ohjaaja(t): DI Otto Operaatioinsinööri	
Tiivistelmä: Tämä on LaTeX-pohjaan kuuluva esimerkki tiivistelmäsivusta. Korvaa tämä tiedosto omalla tiivistelmälläsi, jonka olet tallentanut esim. Microsoft Officesta tai LibreOfficesta PDF-muotoon. Uusimman tiivistelmäpohjan löydät TFM-kandidaatintyön ja seminaarin Noppa-sivuilta.	

Päivämäärä: 11.11.2011	Kieli: suomi	Sivumäärä: 4+1
Avainsanat: esimerkki, lorem ipsum		

Contents

1 Introduction

There are a number of sources for real-life unconstrained nonlinear optimization problems from different fields of sciences and technologies. In general an unconstrained optimization problem can be formulated as

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}). \tag{1}$$

Depending on the methods used, there may also be additional requirements for f, such as being continuously differentiable up to n times or being convex i.e. having one unique global minimum. While there are a number of real-life problems that do satisfy these conditions they are often extremely complex. Therefore artificial problems are often used to develop and test optimization algorithms. [1]

The optimization algorithms are iterative processes that abuse the properties of the target function combined with the raw processing powers of computers. The commonly used methods are based on the same principle of having some termination condition checking if the minimum was found, and if not, continuing by choosing a direction and a step size and moving to the next point and repeating. The step size is often chosen by a separate algorithm called a line search, which transforms the target function into one-dimensional function of step size λ and using the new function to calculate the optimal step size for each iteration of the main method. Similarly to the main methods, there are a number of different algorithms for finding the optimal step size, the simplest being just using a constant λ for every iteration.

In this thesis we will be exploring the effects of line search method selection on the overall performance of the optimization method. Due to the limiting factor of the scope of the bachelor's thesis, this will be in no way a comprehensive statistical research. Instead the main goal is to provide insight on few use cases, primarily on the ones introduced on the Nonlinear Optimization (2018-2019) course in the Aalto University.

Should I mention the target functions, main methods and line search methods explored already here on intro?

General points: should I write in past or present tense? How about writing "I" or "we", or rather use passive?
What is the recommended style for e.g. marking vectors?
Which equations should I numerate?

2 Theory

2.1 Target Functions

2.1.1 Choosing target functions

One of the biggest factors effecting the performance of an optimization algorithm is the complexity of the target function we are using the method for. In this thesis we are going to focus on two different functions. Both of the functions are convex and differentiable since some of the methods we will be comparing require these properties. To increase the complexity of the problems a bit, we are going to use 50-dimensional versions of the two functions. Therefore we are going to pick functions that are defined in $\mathbb{R}^n$ or similar domain and then set $n = 50$.

2.1.2 Step size function

Each optimization method involves a step in form of $\text{argmin}_{\lambda \in \mathbb{R}} f(x_i + \lambda d_i)$. To programmatically produce this step, we are going to define step size function $g(\lambda)$ related to each of the target functions. Then on each iteration of the main optimization method, a new step size function is seeded with the updated point $\mathbf{x}$ and direction vector $\mathbf{d}$.

2.1.3 Matrix Square Sum

The Matrix Square Sum is an extended version of the regular Sum Squares Function, which is defined as

$$f(\mathbf{x}) = f(x_1, ..., x_n) = \sum_{i=1}^{n} i x_i^2. \tag{2}$$

To make Sum Squares problem less trivial, we are going modify it by adding three constants A, b and c. Together they form what we are going to call a Matrix Square Sum function, defined as

$$f(\mathbf{x}) = \|A\mathbf{x} + \mathbf{b}\|^2 + c\|\mathbf{x}\|^2, \tag{3}$$

where A is a positive definite (PD) $n \times n$ matrix, $\mathbf{x}$ and $\mathbf{b}$ are vectors of length n, and the scalar c is a positive constant.

While A could in theory be any $m \times n$ matrix, we are limiting it to a square PD-matrices only to ensure that the function is convex and has an unique global minimum. The matrix A is formed in few steps by first generating an initial $n \times n$

matrix A' with random values $a_{ij} \in [-0.5, 0.5] \ \forall \ i, j = 1 \dots n$. Then we attempt to make a PD-matrix by $A = 0.5 * (A' + A'^{\top})$. Finally, if A does not happen to be a PD-matrix yet, it is modified by the formula

$$A = A + (|\lambda_{min}| + d)I, \tag{4}$$

where λ_{min} is the minimal eigenvalue of A and d is a constant. The value of d also determines the A's condition number defined as $\frac{\lambda_{max}}{\lambda_{min}}$. Smaller values of d make the function more elliptic and harder for gradient methods to solve, while larger values produce more circular gradient curves, which are easy for methods like Gradient Descent to solve. The value of d used in the thesis is 5, resulting in condition number of around 6.5.

Some of the optimization methods require access to the function's gradient, Hessian matrix and the step size function with its first two derivatives, so let's calculate them in advance. While at it, we should also derive the formula for calculating the correct optima so we can use it to determine whether our algorithms produce the correct solution. Lets begin by expanding the function into more easily differentiable form:

$$\begin{aligned} f(\mathbf{x}) &= \|A\mathbf{x} + \mathbf{b}\|^2 + c\|\mathbf{x}\|^2 \\ &= (A\mathbf{x} + \mathbf{b})^{\top}(A\mathbf{x} + \mathbf{b}) + c\mathbf{x}^{\top}\mathbf{x} \\ &= (\mathbf{x}^{\top}A^{\top} + \mathbf{b}^{\top})(A\mathbf{x} + \mathbf{b}) + c\mathbf{x}^{\top}\mathbf{x} \\ &= \mathbf{x}^{\top}A^{\top}A\mathbf{x} + \mathbf{x}A^{\top}\mathbf{b} + \mathbf{b}^{\top}A\mathbf{x} + \mathbf{b}^{\top}\mathbf{b} + c\mathbf{x}^{\top}\mathbf{x} \\ &= \mathbf{x}^{\top}A^{\top}A\mathbf{x} + 2\mathbf{b}^{\top}A\mathbf{x} + \mathbf{b}^{\top}\mathbf{b} + c\mathbf{x}^{\top}\mathbf{x}. \end{aligned}$$

Now let's calculate the gradient and then Hessian matrix:

$$\begin{aligned} \nabla f(\mathbf{x}) &= \nabla(\mathbf{x}^{\top}A^{\top}A\mathbf{x}) + 2\nabla(\mathbf{b}^{\top}A\mathbf{x}) + \nabla(\mathbf{b}^{\top}\mathbf{b}) + \nabla(c\mathbf{x}^{\top}\mathbf{x}) \\ &= (\mathbf{x}^{\top}A^{\top}A)^{\top} + A^{\top}A\mathbf{x} + 2(\mathbf{b}^{\top}A)^{\top} + c(\mathbf{x}^{\top})^{\top} + c\mathbf{x} \\ &= 2A^{\top}A\mathbf{x} + 2A^{\top}\mathbf{b} + 2c\mathbf{x} \end{aligned} \tag{5}$$

$$\mathbf{H} = \nabla^2 f(\mathbf{x}) = 2A^{\top}A + 2cI \tag{6}$$

From the gradient we get the necessary optimality condition

$$(A^{\top}A + cI)\mathbf{x} = -A^{\top}\mathbf{b} \tag{7}$$

and because the function is convex (the Hessian is positive definite for all $\mathbf{x} \in \mathbb{R}^n$), we get the unique optimal solution

$$\mathbf{x} = -(A^{\top}A + cI)^{-1}A^{\top}\mathbf{b} \tag{8}$$

Let's now form the step size function and its derivatives:

$$\begin{aligned} g(\lambda) &= f(\mathbf{x}+\lambda\mathbf{d}) = \|A(\mathbf{x}+\lambda\mathbf{d})+\mathbf{b}\|^2 + c\|(\mathbf{x}+\lambda\mathbf{d})\|^2 \\ &= \lambda\mathbf{d}^\top A^\top(2A\mathbf{x}+\lambda A\mathbf{d}+2\mathbf{b}) + \mathbf{x}^\top A^\top(A\mathbf{x}+2\mathbf{b}) + \mathbf{b}^\top\mathbf{b} \\ &\quad + c(\mathbf{x}^\top\mathbf{x} + 2\lambda\mathbf{d}^\top\mathbf{x} + \lambda^2\mathbf{d}^\top\mathbf{d}) \end{aligned} \tag{9}$$

$$g'(\lambda) = \mathbf{d}^\top A^\top(2A\mathbf{x}+2\lambda A\mathbf{d}+2\mathbf{b}) + 2c\mathbf{d}^\top(\mathbf{x}+\lambda\mathbf{d}) \tag{10}$$

$$g''(\lambda) = 2\mathbf{d}^\top(A^\top A\mathbf{d} + c\mathbf{d}) \tag{11}$$

2.1.4 Negative Entropy

The second target function reviewed is a variation of entropy maximization problem, in which the maximation problem is inverted therefore making it a minimization problem. We will be calling this convex minimization problem a Negative Entropy problem and it can be formulated as follows:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i \log x_i, \tag{12}$$

where $\mathbf{x} \in \mathbb{R}^n_{>0}$. Since the domain is now limited we have to do some adjustments to our algorithms, which we will get into in a moment.

The global minima of the function can be found by calculating the zero point of the gradient:

$$\nabla f(\mathbf{x}) = \frac{\delta}{\delta x_i}\sum_{i=1}^{n} x_i \log x_i \tag{13}$$

$$= \sum_{i=1}^{n}(\log x_i + 1) = 0. \tag{14}$$

Because the $\log x_i + 1$ is truly positive, the answer is the same for every term x_i. Therefore we get the minima: $\log x_i = -1 \Leftrightarrow x_i = \frac{1}{e}$, for every $i = 1 \dots n$.

In addition to the gradient, we need to calculate the Hessian matrix and the one dimensional step size function $g(\lambda) = f(\mathbf{x}+\lambda\mathbf{d})$ with its first and second derivatives in regard to λ.

As we already know the gradient of f, the Hessian can be derived with ease since $\frac{\delta^2}{\delta x_i \delta x_j}\sum_{i=1}^{n} x_i \log x_i = 0$ for every i, j where $i \neq j$. For the diagonal cases, i.e. when $i = j$, we are left with the second derivative $\frac{1}{x_i}$. Therefore we get the following

Hessian matrix:

$$\mathbf{H} = \begin{bmatrix} x_1^{-1} & 0 & 0 & \dots & 0 \\ 0 & x_2^{-1} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & x_n^{-1} \end{bmatrix}. \tag{15}$$

Finally, the step size function for λ with its first two derivatives are as follows:

$$g(\lambda) = \sum_{i=1}^{n} (x_i + \lambda d_i) \log(x_i + \lambda d_i) \tag{16}$$

$$g'(\lambda) = \sum_{i=1}^{n} d_i (\log(x_i + \lambda d_i) + 1) \tag{17}$$

$$g''(\lambda) = \sum_{i=1}^{n} \frac{d_i^2}{x_i + \lambda d_i}. \tag{18}$$

Because the domain of this function is only positive numbers, we are going to have make some changes to our starting points, parameters and algorithms. Therefore the starting point must only have positive coordinates and the following domain limiter was added to the line search algorithms which update the starting point so that it could otherwise go negative:

Algorithm 1 Domain Limiter

```
initialize γ ∈ [0, 1]
while min (x + λd) <= 0 do
    λ = λγ
end while
return λ
```

The domain limiter will be called from all of the line search methods before any evaluation of $f(\lambda)$ to reduce the step size to such value that it prevents evaulation of negative logarithms. In the context of this thesis we are going to set $d = 0.99$, but it could also be added as a parameter for each of the line search functions as some methods will require more iterations on the while-loop than others.

2.2 Main Methods

The main methods as I call them in the scope of this thesis are methods for finding minimums for unconstrained, differentiable and convex functions. While some of

the methods or line search methods do not require all these properties, they are included anyway for this analysis.

2.2.1 Newton's Method

Newton's method is commonly used for finding global minimums for differentiable and convex functions. The algorithm is a more general version of its line search version, utilizing the gradient and hessian matrix of the function instead of its first and second single derivatives.

source?

Algorithm 2 Newton's Method

1: **initialize** $l > 0, k = 0$, $k_{max} \in \mathbb{N}$, $\mathbf{x} = \mathbf{x_0} \in \mathbb{R}^n$, step size function $g_{\mathbf{x},\mathbf{d}}(\lambda)$ and line search method L.
2: **while** $\|\nabla\theta(\mathbf{x})\| > l$ **and** $k < k_{max}$ **do**
3: $\quad \mathbf{d} = -H(\mathbf{x})^{-1}\nabla\theta(\mathbf{x})$
4: $\quad \lambda = L(g_{\mathbf{x},\mathbf{d}})$
5: $\quad \mathbf{x} = \mathbf{x} + \lambda\mathbf{d}$
6: $\quad k = k + 1$
7: **end while**
8: **return** λ

2.2.2 Gradient Descent Method

Gradient descent method can be interpret as a simpler version of Newton's method as it only advances one gradient at a time. The algorithm is exactly the same as in Newton's method except for line 3 where the descent direction d is calculated in a little bit simpler manner.

Algorithm 3 Gradient Descent Method

1: **initialize** $l > 0, k = 0$, $k_{max} \in \mathbb{N}$, $\mathbf{x} = \mathbf{x_0} \in \mathbb{R}^n$, step size function $g_{\mathbf{x},\mathbf{d}}(\lambda)$ and line search method L.
2: **while** $\|\nabla\theta(\mathbf{x})\| > l$ **and** $k < k_{max}$ **do**
3: $\quad \mathbf{d} = -\nabla\theta(\mathbf{x})$
4: $\quad \lambda = L(g_{\mathbf{x},\mathbf{d}})$
5: $\quad \mathbf{x} = \mathbf{x} + \lambda\mathbf{d}$
6: $\quad k = k + 1$
7: **end while**
8: **return** λ

2.2.3 Conjugate Gradient Method

Conjugate gradient method uses the information from the previous and current iterations to calculate ratio of gradients to calculate optimal next point. The method should theoretically require less steps than gradient descent method. source?

The algorithm involves a second loop which runs for n times per every iteration on the main loop. The n is equal to the dimensions of the problem (i.e. $\dim(\mathbf{x})$) and also the inner iterations are calculated towards the total iterations.

Algorithm 4 Conjugate Gradient Method

initialize $l > 0, k = 0$, $k_{max} \in \mathbb{N}$, $\alpha \in \mathbb{R}$, $\mathbf{x} = \mathbf{x_0} \in \mathbb{R}^n$, step size function $g_{\mathbf{x},\mathbf{d}}(\lambda)$ and line search method L.
$\mathbf{d} = -\nabla\theta(\mathbf{x})$
while $\|\nabla\theta(\mathbf{x})\| > l$ **and** $k < k_{max}$ **do**
  $\mathbf{y} = \mathbf{x}$
  for $i = 1 \dots n$ **do**
    $\lambda = L(g_{\mathbf{y},\mathbf{d}})$
    $\mathbf{y_{prev}} = \mathbf{y}, \ \mathbf{y} = \mathbf{y} + \lambda\mathbf{d}$
    $\alpha = \frac{\nabla\theta(\mathbf{y})^2}{\nabla\theta(\mathbf{y_{prev}})^2}$
    $\mathbf{d} = -\nabla\theta(\mathbf{y}) + \alpha\mathbf{d}$
    $k = k + 1$
  end for
  $\mathbf{x} = \mathbf{y}, \ \mathbf{d} = \nabla\theta(\mathbf{x})$
end while
return λ

2.2.4 Heavy Ball Method

Heavy ball method is basically an extension of gradient descent as only adds one term for the direction vector calculation step. The name of the method comes from the physical representation of momentum of a ball rolling downhill. The algorithm for the method is basically the same as Gradient descent with a single additional term with multiplier β in descent direction d. source?

Algorithm 5 Heavy Ball Method

initialize $l > 0, k = 0$, $k_{max} \in \mathbb{N}$, $\beta \in \mathbb{R}$, $\mathbf{x} = \mathbf{x_{prev}} = \mathbf{x_0} \in \mathbb{R}^n$, step size function $g_{\mathbf{x},\mathbf{d}}(\lambda)$ and line search method L.
while $\|\nabla\theta(\mathbf{x})\| > l$ **and** $k < k_{max}$ **do**
  $\mathbf{d} = -\nabla\theta(\mathbf{x}) + \beta(\mathbf{x} - \mathbf{x_{prev}})$
  $\lambda = L(g_{\mathbf{x},\mathbf{d}})$
  $\mathbf{x_{prev}} = \mathbf{x}, \ \mathbf{x} = \mathbf{x} + \lambda\mathbf{d}$
  $k = k + 1$
end while
return λ

2.3 Line Search Methods

Most line search methods are based on a single idea, which we are going to call the theorem 1:

Theorem 1: Let $\theta : \mathrm{R} \to \mathrm{R}$ be strictly quasiconvex over the interval $[a, b]$, and let λ, $\mu \in [a, b]$ such that $\lambda < \mu$. If $\theta(\lambda) > \theta(\mu)$, then $\theta(z) \geq \theta(\mu)$ for all $z \in [a, \lambda]$. If $\theta(\lambda) \leq \theta(\mu)$, then $\theta(z) \leq \theta(\lambda)$ for all $z \in [\mu, b]$. [3]

2.3.1 Constant Step Size

While not actually a search, the use of single step size value λ is sometimes enough to provide correct and high performance solutions for the main methods. In this thesis I will be considering constant λ as one of the search methods to provide comprehensive comparison.

Table 1: Parameters tested for ConstantSearch

Parameter	MatrixSquareSum	NegativeEntropy
λ	0.0001, 0.1, 0.25, 0.5, 0.9	0.1, 0.25, 0.5

2.3.2 Golden Section Search

While golden section and fibonacci numbers are closely related, the golden section search does take much more straight forward way for calculating the new step size bounds using the approximation of $\phi = 0.618$.

is this approx good enough or should I add more digits?

Algorithm 6 Golden Section Search

initialize tolerance $l > 0$, $\alpha = 0.618$, $k = 0$, $k_{max} \in \mathbb{N}$, $(a, b) \in \mathbb{R}$.
$\lambda = a + (1 - \alpha)(b - a)$, $\mu = a + \alpha(b - a)$.
while $b - a > l$ **and** $k < k_{max}$ **do**
  if $\theta(\lambda) > \theta(\mu)$ **then**
    $a = \lambda,\ \ \lambda = \mu,\ \ \mu = a + \alpha(b - a)$
  else
    $b = \mu,\ \ \mu = \lambda,\ \ \lambda = a + (1 - \alpha)(b - a)$
  end if
  $k = k + 1$
end while
return $\lambda = \frac{a+b}{2}$

Table 2: Parameters tested for GoldenSectionSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	10, 5	10, 5
l	0.0001, 1e-07	0.0001, 1e-07

2.3.3 Bisection Search

Bisection method is commonly used line search method for differentiable functions. source? It basically narrows down the optimal step size range using the first derivative of the target function as a direction of which bound to move next.

Algorithm 7 Bisection Search

initialize $l > 0, k = 0$, $k_{max} \in \mathbb{N}$, $(a, b) \in \mathbb{R}$.
while $|b - a| > l$ **and** $k < k_{max}$ **do**
  $\lambda = \frac{b+a}{2}$
  if $\theta'(\lambda) = 0$ **then**
    return λ
  else if $\theta'(\lambda) > 0$ **then**
    $b = \lambda$
  else
    $a = \lambda$
  end if
  $k = k + 1$
end while
return $\lambda = \frac{a+b}{2}$

Table 3: Parameters tested for BisectionSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	10, 5	10, 5
l	0.0001, 1e-07	0.0001, 1e-07

2.3.4 Dichotomous Search

Dichotomous search is an example of sequentical searches. It uses the information of the previous evaulation of θ resulting in higher performance than e.g. uniform search.

Algorithm 8 Dichotomous Search

initialize $l > 0, k = 0$, $k_{max} \in \mathbb{N}$, $(a, b) \in \mathbb{R}$.
while $b - a > l$ **and** $k < k_{max}$ **do**
  $\lambda = \frac{b+a}{2} - \epsilon, \ \mu = \frac{b+a}{2} + \epsilon$
  if $\theta(\lambda) < \theta(\mu)$ **then**
    $b = \mu$
  else
    $a = \lambda$
  end if
  $k = k + 1$
end while
return $\lambda = \frac{a+b}{2}$

Table 4: Parameters tested for DichotomousSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	10, 5	5, 10
ϵ	1e-07, 1e-08	1e-06, 1e-07
l	0.0001	0.0001, 1e-05

2.3.5 Fibonacci Search

Fibonacci is an interesting numerical method for providing optimal step sizes using fibonacci numbers when calculating the bounds for the step size. [better explanation of why this works?] The calculation may be somewhat heavy due to the requirement of calculating n fibonacci numbers, where n is determined by the required accuracy i.e. tolerance. However, in some real life applications these fibonacci numbers may of course be precalculated. In this thesis I will not be using a table for the fibonacci values but instead actually calculate them using a while loop. The iteration count for finding the fibonacci numbers will not be included in the performance comparison, but the time taken to find the values is included.

is this fine, or should I also report the fibonacci iteration count and time?

or what term to use for methods that do not use derivative?

Algorithm 9 Fibonacci Search

initialize $l > 0, \epsilon > 0, k = 0,\ k_{max} \in \mathbb{N},\ (a, b) \in \mathbb{R}, n = \min_n\{\frac{b-a}{F_n} \leq l\}$.
$\lambda = a + \frac{F_{n-2}}{F_n}(b-a),\ \ \mu = a + \frac{F_{n-1}}{F_n}(b-a)$
while $k \leq n-1$ **and** $k < k_{max}$ **do**
  if $\theta(\lambda) < \theta(\mu)$ **then**
    $a = \lambda,\ \ \lambda = \mu,\ \mu = a + \frac{F_{n-k-1}}{F_{n-k}}(b-a)$
  else
    $b = \mu,\ \ \lambda = \mu,\ \lambda = a + \frac{F_{n-k-2}}{F_{n-k}}(b-a)$
  end if
  $k = k + 1$
end while
$\lambda = \theta(\lambda)$
if $\theta(\lambda) > \theta(\mu + \epsilon)$ **then**
  $a = \lambda$
else
  $b = \mu$
end if
return $\lambda = \frac{a+b}{2}$

The F_n represents n:th Fibonacci number. To optimize algorithm performance, the values of F_n could be pre-evaluated but for our use case we are just going to generate the Fibonacci numbers using single loop algorithm of time complexity $O(n)$.

Table 5: Parameters tested for FibonacciSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	10, 5	10, 5
ϵ	0.01, 1e-05, 1e-08	0.01, 1e-05, 1e-08
l	0.0001, 1e-07	0.0001, 1e-07

2.3.6 Uniform Search

Uniform search works by dividing a constrained section of a function into intervals, then choosing the interval with the lowest value and performing the same procedure again for it. To increase the precision of the method we also increase the number of intervals we divide the newly selected interval into.

More formally: break $[a, b]$ into n uniform intervals of size δ, which leads to $n + 1$ grid points $a_0 + k\delta$ with $b = a_n = a_0 + n\delta,\ k = 0, ..., n$. [3]

In addition we are going to define n as the interval count and m as the interval count multiplier.

Algorithm 10 Uniform Search

initialize $l > 0, k = 0, (k_{max}, n) \in \mathbb{N}, (a, b, m) \in \mathbb{R}$
$s = \frac{b-a}{n},\ p_{min} = a$
while $s > l$ **and** $k < k_{max}$ **do**
  for $i = 0, \ldots, n$ **do**
    $x = a + is$
    if $\theta(x) < \theta(p_{min})$ **then**
      $p_{min} = x$
    end if
    $k = k + 1$
  end for
  $a = p_{min} - s,\ b = p_{min} + s$
  $n = \lfloor nm \rfloor,\ s = \frac{b-a}{n}$
end while
return $\lambda = \frac{a+b}{2}$

Table 6: Parameters tested for UniformSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	10, 5	10, 5
n	10, 100	10, 100
m	1, 1.5	1, 1.5
l	0.0001, 1e-07	0.0001, 1e-07

2.3.7 Newton's Search

Newton's method is another very common example of line search methods. Similarily to bisection method, Newton's method requires the target function to be differentiable up to two times as it uses the ratio of first and second derivatives to move the optimal step size forward.

Algorithm 11 Newton's Search

initialize $l > 0, k = 0$, $k_{max} \in \mathbb{N}$, $\lambda \in \mathbb{R}$
while $|\theta'(\lambda)| > l$ **and** $k < k_{max}$ **do**
    $\lambda = \lambda - \frac{\theta'(\lambda)}{\theta''(\lambda)}$
    $k = k + 1$
end while
return λ

Table 7: Parameters tested for NewtonsSearch

Parameter	MatrixSquareSum	NegativeEntropy
λ	1, 5, 10, 50	0.5, 1.0
l	1e-07, 1e-08	1e-06, 1e-07

2.3.8 Armijo's Search

Armijo's search is the only backtracking method of the ones included in this thesis.

include more in depth explanation of backtracking algorithms?

Algorithm 12 Armijo's Search

initialize $l > 0, k = 0$, $k_{max} \in \mathbb{N}$, $(\lambda, \alpha, \beta) \in \mathbb{R}$
$\theta_0 = \theta(0)$, $\theta'_0 = \theta'(0)$
while $\theta(\lambda) > \theta_0 + \alpha\lambda\theta'_0$ **and** $k < k_{max}$ **do**
    $\lambda = \lambda\beta$
    $k = k + 1$
end while
return λ

3 Methods

Our goal is to determine how does the selection of line search method effect the overall performance of an optimization method. To answer this question with the resources described in theory section, we are going to run each of the target function, main method and line search method permutations multiple times. In addition we need to take into account all the different parameters for each of the algorithms.

Choosing the best parameters for each method for each target function would be a high level problem of its own if performed accurately. Since that is something we are not especially interested in within the scope of this thesis, we are going to manually test and select few different values for each parameters for each algorithm. Next, we are going to programmatically generate all the parameter permutations for each algorithm and test them with m starting points of dimension n, where n is the same value used for the actual overall performance testing. In addition we are going to have to limit the absolute value of coordinates into some specific area. For all the parameter selection and performance testing we used the same setup by generating 100 points x_0 of dimension $n = 50$ with each $x_i \in [-10, 10]$. Because negative entropy problem has the domain $\mathbb{R}_{>0}$, we are going to generate a separate set of 100 points otherwise similar to the first one but with positive values only: $x_i \in]0, 10]$. In addition, as the matrix square sum problem has the option for randomly generated constants A, b and c, we are going to use that in our favor and generate 100 randomized versions of that function in advance as well. This way we can in the best way compensate for our relatively small sample size. Another way to compensate for the sample size is to generate the starting points more evenly by controlling the randomness of the points. Once again, because this is something of a completely different research topic to handle perfectly, we ended up using non-perfect but simple algorithm for generating our starting point distribution.

Algorithm 13 Generating Even Distribution of Random Starting Points

initialize $(d_{min}, x_{min}, x_{max}) \in \mathbb{R}, (n, p, q) \in \mathbb{N}, Z = \{\mathbf{z}_1, \ldots, \mathbf{z}_m\}, i = 0, q = 1, \mathbf{z}_1 = \mathbf{random}_n(x_{min}, x_{max})$
while $i < p$ **do**
  $\mathbf{x} = \mathbf{random}_n(x_{min}, x_{max})$
  ok = **true**
  for $j = 1 \ldots q$ **do**
    if $\|\mathbf{z}_j - \mathbf{x}\| < d_{min}$ **then**
      ok = **false**
      break
    end if
  end for
  if ok **then**
    $q = q + 1, \;\; i = i + 1$
    $\mathbf{z}_q = \mathbf{x}$
  end if
end while
return Z

The algorithm generates $p = 100$ points of dimension n. Function $\mathbf{random}_n(x_{min}, x_{max})$ generates vector $\mathbf{x}$ of size n filled with random values $x_i \in [x_{min}, x_{max}], i = 1 \dots n$. We then compare distance of newly generated point x and every point generated before i.e. $z_j, z < q$ and if for any pair of points the Euclidean distance is less than d_{min} we skip to the next iteration and generate a new candidate x. In theory the algorithm could get stuck as it depends on random values on the exit condition, but using small enough d_{min} will result in fast and even distribution of random numbers within targeted area of values. The points will likely never get optimally distributed but it is still better than completely randomized points where portions of the targeted area could be left completely empty of starting points.

3.0.1 Comparing Performance

Performance comparison is going to be based mostly on total iterations meaning iterations of main method plus total iterations of the line search methods that the main method ran. The iterations of main method and line search methods are treated as equally valuable. Execution times are also recorded but not used as primary tool for performance tracking since with such little repetitions and the value of n certain randomness plays too big of a part in the duration parameter. In addition, duration is heavily dependant on the programming language and style used so while the results could be comparable within the same language, they might behave differently under another setup. In comparison iterations are somewhat universal and thus make a better measure for performance. Another aspect that is recorded is the total number of function calls n_f performed by both the main method and line search together. All the calls of θ, its gradient and Hessian matrix, as well as the function calls of the step size function $f(\mathbf{x} + \lambda\mathbf{d})$ and its derivatives made by the line search method are calculated towards the same total count. This number is used as a secondary tool for comparison of the algorithms' performance.

4 Results

Table 8: Average performances of NewtonsMethod when minimizing MatrixSquareSum using different line search methods and 1000 randomly generated starting points

Line Search Name	s (%)	t (ms)	k	f_n	t_{LS} (ms)	k_{LS}
ConstantSearch	100.0	445.8	1.0	7.0	0.0	0.0
GoldenSectionSearch	100.0	445.0	1.0	101.0	2.6	47.0
BisectionSearch	99.6	535.9	1.0	35.0	89.0	28.0
DichotomousSearch	100.0	892.1	2.0	91.0	2.0	40.0
FibonacciSearch	100.0	453.2	1.0	119.0	4.5	55.0
UniformSearch	100.0	455.0	1.0	156.0	7.2	148.0
NewtonsSearch	100.0	443.5	1.0	8.0	0.5	0.0
ArmijoSearch	99.7	3525.2	8.0	59.0	4.8	0.0

Table 9: Average performances of GradientDescentMethod when minimizing MatrixSquareSum using different line search methods and 1000 randomly generated starting points

Line Search Name	s (%)	t (ms)	k	f_n	t_{LS} (ms)	k_{LS}
ConstantSearch	98.9	4681.2	2942.0	8828.9	141.2	0.0
GoldenSectionSearch	99.4	283.7	33.1	1784.1	155.4	841.0
BisectionSearch	99.4	823.5	33.0	696.6	749.8	594.5
DichotomousSearch	99.4	270.1	33.1	1491.1	142.6	694.4
FibonacciSearch	99.4	354.3	33.1	2615.3	238.6	1223.5
UniformSearch	99.4	389.2	33.1	3508.2	282.3	3372.9
NewtonsSearch	99.4	453.3	33.1	241.2	355.7	35.3
ArmijoSearch	99.6	209.3	25.8	343.2	97.9	185.5

Table 10: Average performances of ConjugateGradientMethod when minimizing MatrixSquareSum using different line search methods and 1000 randomly generated starting points

Line Search Name	s (%)	t (ms)	k	f_n	t_{LS} (ms)	k_{LS}
ConstantSearch	99.6	989.3	300.1	1216.2	23.9	0.0
GoldenSectionSearch	100.0	429.5	50.0	2722.9	198.9	1258.5
BisectionSearch	100.0	1154.9	50.0	1106.0	1003.4	900.0
DichotomousSearch	100.0	426.3	50.0	2306.0	177.1	1050.0
FibonacciSearch	100.0	517.8	50.0	4006.0	298.5	1850.0
UniformSearch	100.0	558.2	50.0	5356.0	348.9	5100.0
NewtonsSearch	100.0	552.8	50.6	364.0	347.5	35.0
ArmijoSearch	100.0	405.3	50.0	914.8	157.6	558.8

Table 11: Average performances of HeavyBallMethod when minimizing MatrixSquareSum using different line search methods and 1000 randomly generated starting points

Line Search Name	s (%)	t (ms)	k	f_n	t_{LS} (ms)	k_{LS}
ConstantSearch	98.9	4711.5	2941.9	8828.8	134.5	0.0
GoldenSectionSearch	99.4	227.7	24.3	1303.4	127.1	613.8
BisectionSearch	99.4	666.6	24.4	514.9	607.0	438.8
DichotomousSearch	99.4	209.4	24.4	904.9	99.3	414.4
FibonacciSearch	99.4	353.2	32.7	2585.0	239.5	1209.3
UniformSearch	99.4	322.4	24.4	2587.4	232.0	2486.9
NewtonsSearch	99.4	375.1	24.4	179.6	295.0	26.4
ArmijoSearch	99.6	217.3	25.8	343.1	99.6	185.4

Table 12: Average performances of NewtonsMethod when minimizing NegativeEntropy using different line search methods and 1000 randomly generated starting points

Line Search Name	s (%)	t (ms)	k	f_n	t_{LS} (ms)	k_{LS}
ConstantSearch	100.0	3718.1	10.7	45.7	1.0	0.0
GoldenSectionSearch	100.0	2038.3	5.6	426.9	29.4	200.7
BisectionSearch	100.0	2191.6	5.6	116.9	161.4	91.5
DichotomousSearch	100.0	2231.2	6.0	265.5	52.0	119.2
FibonacciSearch	100.0	2100.7	5.6	738.5	59.0	350.9
UniformSearch	100.0	2316.0	6.1	510.3	132.5	476.7
NewtonsSearch	100.0	2266.0	5.7	192.6	348.6	53.7
ArmijoSearch	100.0	2206.9	6.3	52.6	3.1	5.5

Table 13: Average performances of GradientDescentMethod when minimizing NegativeEntropy using different line search methods and 1000 randomly generated starting points

Line Search Name	s (%)	t (ms)	k	f_n	t_{LS} (ms)	k_{LS}
ConstantSearch	100.0	13.1	16.3	51.8	0.8	0.0
GoldenSectionSearch	100.0	53.4	8.6	400.1	47.9	185.7
BisectionSearch	100.0	92.2	8.6	158.4	87.0	129.7
DichotomousSearch	100.0	54.7	8.6	289.3	48.9	130.2
FibonacciSearch	100.0	58.4	8.6	610.8	52.6	282.4
UniformSearch	100.0	121.8	10.9	889.6	114.6	842.8
NewtonsSearch	100.0	458.8	8.8	687.6	453.8	216.5
ArmijoSearch	100.0	11.6	10.2	96.5	5.6	32.0

Table 14: Average performances of ConjugateGradientMethod when minimizing NegativeEntropy using different line search methods and 1000 randomly generated starting points

Line Search Name	s (%)	t (ms)	k	f_n	t_{LS} (ms)	k_{LS}
ConstantSearch	99.9	4393.4	163.2	663.3	4248.0	0.0
GoldenSectionSearch	100.0	539.7	100.0	4681.2	451.2	2136.6
NewtonsSearch	100.0	2801.7	100.0	4147.0	2716.9	1213.0
ArmijoSearch	100.0	198.4	99.8	1119.4	112.0	412.8
BisectionSearch	100.0	759.5	100.0	1228.3	672.9	777.8
DichotomousSearch	100.0	576.4	102.7	3496.1	485.5	1538.6
FibonacciSearch	100.0	623.7	100.0	7155.1	534.0	3273.5
UniformSearch	100.0	6371.1	100.0	8208.0	6277.0	7700.0

Table 15: Average performances of HeavyBallMethod when minimizing NegativeEntropy using different line search methods and 1000 randomly generated starting points

Line Search Name	s (%)	t (ms)	k	f_n	t_{LS} (ms)	k_{LS}
ConstantSearch	100.0	8.3	14.0	45.0	0.3	0.0
GoldenSectionSearch	100.0	92.4	17.3	808.4	80.5	376.8
BisectionSearch	100.0	131.7	11.8	223.7	124.6	185.4
DichotomousSearch	100.0	68.8	11.8	486.5	60.9	224.1
FibonacciSearch	100.0	72.1	11.8	843.4	64.3	390.8
UniformSearch	100.0	134.9	12.9	1051.1	126.6	996.3
NewtonsSearch	100.0	541.0	12.8	806.1	533.7	250.6
ArmijoSearch	100.0	15.5	14.4	113.2	6.9	23.6

4.1 Analysis

There are multiple metrics we could look into when analyzing the results. Since our goal is to compare the effect in performance of the main methods caused by the line search methods, one natural way of analyzing the results is finding the overall best line search method. This is done by firstly prioritizing success rate and secondly the average duration in which the solutions were found. After sorting the line search methods with these two metrics per target function and per main method, we can then score them. For scoring we are going to use two methods:

Table 16: Ranking scores of line search methods per target function. Data is colored column-wise so that darker green is better and darker red is worse.

LineSearchMethod	SUM MSS	SUM NE	SUM TOTAL
ConstantSearch	10	24	34
GoldenSectionSearch	25	18	43
BisectionSearch	6	13	19
DichotomousSearch	24	22	46
FibonacciSearch	19	16	35
UniformSearch	16	13	29
NewtonsSearch	18	11	29
ArmijoSearch	26	27	53

first is to give each line method a rank so that the fastest method gets 8 points, the second fastest 7 points and so on until each line method has gotten 1 to 8 points, and then repeat this separately for each combination of target functions and main methods. The pros of this method is that it takes the success rate into account unlike the second method we are using which is calculating the total sum of average durations for each line search.

Table 17: Ranking scores of line search methods per main method. Data is colored column-wise so that darker green is better and darker red is worse.

LineSearchmethod	NM	GDM	CGM	HBM
ConstantSearch	14	8	3	9
GoldenSectionSearch	8	12	13	10
BisectionSearch	4	5	5	5
DichotomousSearch	8	12	13	13
FibonacciSearch	7	9	10	9
UniformSearch	11	6	5	7
NewtonsSearch	14	4	7	4
ArmijoSearch	6	16	16	15

Table 18: Average durations of line search methods per target function. Data is colored column-wise so that darker green is better and darker red is worse.

LineSearchMethod	SUM MSS	SUM NE	SUM TOTAL
ConstantSearch	10 993,40	8 132,90	19 126,30
GoldenSectionSearch	1 385,90	2 723,80	4 109,70
BisectionSearch	3 015,30	3 175,00	6 190,30
DichotomousSearch	1 797,90	2 931,10	4 729,00
FibonacciSearch	1 678,50	2 854,90	4 533,40
UniformSearch	1 724,80	8 943,80	10 668,60
NewtonsSearch	1 824,70	6 067,50	7 892,20
ArmijoSearch	4 357,10	2 432,40	6 789,50

Table 19: Average durations of line search methods per main method. Data is colored column-wise so that darker green is better and darker red is worse.

LineSearchmethod	NM	GDM	CGM	HBM
ConstantSearch	4 163,90	4 694,30	5 548,30	4 719,80
GoldenSectionSearch	2 483,30	337,10	969,20	320,10
BisectionSearch	2 727,50	915,70	1 748,80	798,30
DichotomousSearch	3 123,30	324,80	1 002,70	278,20
FibonacciSearch	2 553,90	412,70	1 141,50	425,30
UniformSearch	2 771,00	511,00	6 929,30	457,30
NewtonsSearch	2 709,50	912,10	3 354,50	916,10
ArmijoSearch	5 732,10	220,90	603,70	232,80

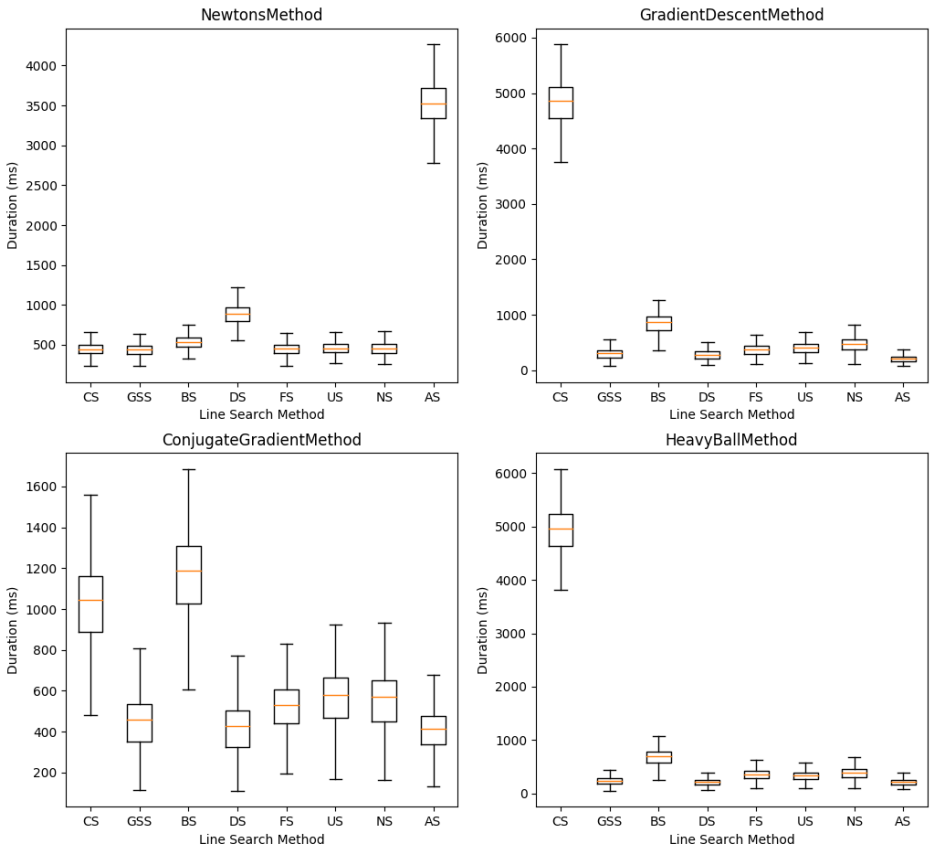

Figure 1: Box plot of solution times per line search and main method for MatrixSquareSum target function.

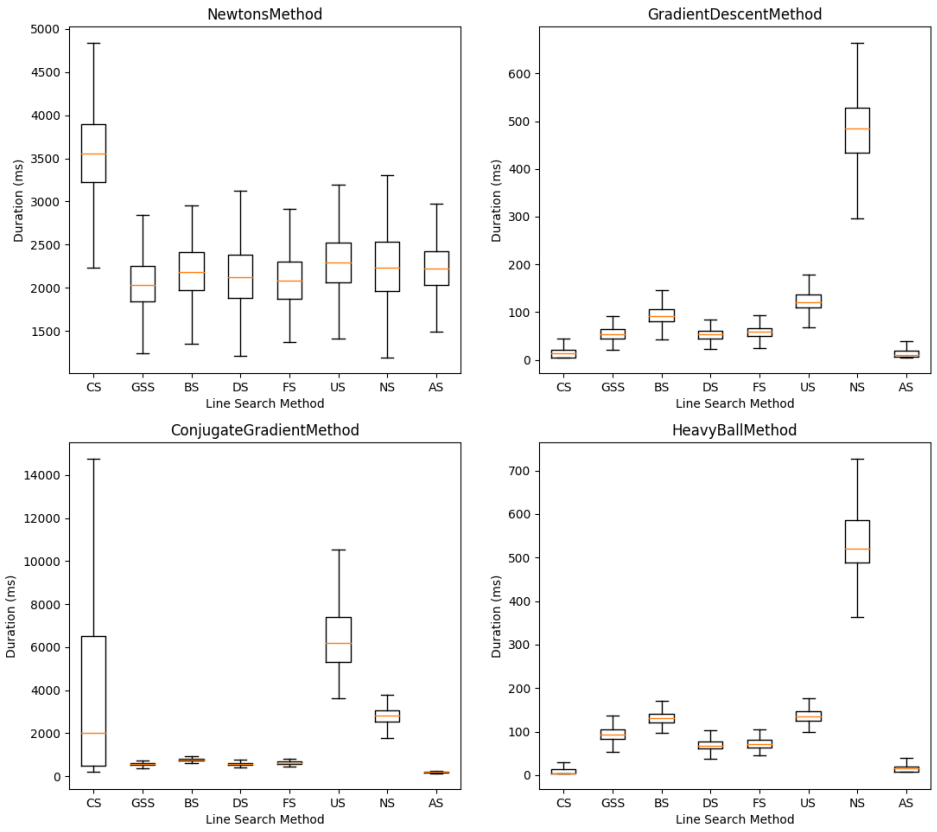

Figure 2: Box plot of solution times per line search and main method for NegativeEntropy target function.

5 Conclusion and Future Research

To make the results completely comprehensive we should consider all existing functions as possible target functions. However as this is practically impossible, one substantial way to improve the coverage would be increasing the number of target functions: other similar research has used up to 80 different problems to calculate the performance for [2],

Another major factor in increasing the reliability of the results are the starting points. In this research we limited the area of where the points were generated to $[-10, 10]$ or $]0, 10]$ depending on the domain of the target function. In a perfect

world we could test the algorithms for every single starting point. Even though that is practically impossible, a wider range of points could be utilized. Anyways the selection of starting point is very target function related and should always be considered per function. In addition the number of starting points tested could be raised within the computing power available.

References

[1] N. Andrei. An unconstrained optimization test functions collection. *Advanced Modeling and Optimization, Volume 10, Number 1, 2008*, 2000.

[2] W. W. H. HONGCHAO ZHANG. A nonmonotone line search technique and its application to unconstrained optimization. *SIAM J. OPTIM. Vol. 14, No. 4, pp. 1043–1056*, 2004.

[3] F. Oliveira. Ms-e2122 - nonlinear optimisation. *Operations Research, Aalto University*, 2018.

6 Attachments