

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

The document can be stored and made available to the public on the open internet pages of Aalto University.

All other rights are reserved.

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: Systemitieteet		Pääaineen koodi: F3010	
Vastuuopettaja(t): Prof. Ansio Akateemikko			
Ohjaaja(t): DI Otto Operaatioinsinööri			
<p>Tiivistelmä:</p> <p>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.</p>			
Päivämäärä: 11.11.2011		Kieli: suomi	Sivumäärä: 4+1
Avainsanat: esimerkki, lorem ipsum			

Contents

1	Introduction	1
2	Theory	3
2.1	Target Functions	3
2.1.1	Choosing target functions	3
2.1.2	Step size function	3
2.1.3	Matrix Square Sum	3
2.1.4	Negative Entropy	5
2.2	Main Methods	7
2.2.1	Newton's Method	7
2.2.2	Gradient Descent Method	7
2.2.3	Conjugate Gradient Method	8
2.2.4	Heavy Ball Method	9
2.3	Line Search Methods	9
2.3.1	Constant Step Size	9
2.3.2	Golden Section Search	10
2.3.3	Bisection Search	11
2.3.4	Dichotomous Search	12
2.3.5	Fibonacci Search	12
2.3.6	Uniform Search	14
2.3.7	Newton's Search	15
2.3.8	Armijo's Search	15
3	Methods	16
3.1	Parameter Selection	16
3.2	Starting Point Selection	17
3.3	Randomizing Target Function	18
3.4	Comparing Performance	18
3.4.1	Parameter Selection	19
4	Results	19
4.1	Analysis	23
4.2	Assumptions	32
5	Conclusion and Future Research	33
A	Appendix	36

1 Introduction

Unconstrained nonlinear optimization has numerous real-life applications in different fields of science and technology, such as in computational biology, machine learning, and finance. In general, an unconstrained optimization problem can be formulated as

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

where $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is called an *objective function* (or a *target function*), and we are looking for a *solution* vector $x^* \in \mathbb{R}^n$ that minimizes the value of f such that $f(x^*) \leq f(x)$ for all $x \in \mathbb{R}^n$. There are several different optimization algorithms (or methods) that can be used to solve the problem (1), e.g., Newton's method and ... However, the performances of different optimization methods typically vary significantly for different objective functions f , and it can be difficult to find the best solution method for a given problem instance. For example, we may use a different method when f is convex compared to when f is non-convex and has multiple local optima. Moreover, some methods are feasible only when f satisfies certain properties, such as being continuously differentiable up to n times. While there are countless real-life problems that satisfy these conditions, they are often extremely complex and finding a global minimizing solution can be difficult. Therefore, artificial problems are often used instead to develop and test optimization algorithms. [1]

In general, optimization algorithms are iterative processes that harness the raw processing power of computers efficiently by exploiting the properties of the target function f in order to converge to an optimal solution. The commonly used optimization methods typically start from a given point, choose a direction and a step size, and move to a new point. Then, different stopping criteria can be used to check if a minimizing solution has been found, and if not, the process is repeated until a minimum solution is reached or some other condition, such as maximum time limit or iteration count, is exceeded. The step size is often computed by a separate *line search* algorithm. After computing the target function value $f(x')$ at a given point $x' \in \mathbb{R}$ and a new direction $d \in \mathbb{R}^n$, the line search function is of the form $f(x' + \lambda d)$ which is a one-dimensional function of the step size $\lambda \in \mathbb{R}$. Line search algorithms try to find the value of λ that minimizes this function, i.e., they try to solve the problem

$$\min_{\lambda \in \mathbb{R}} f(x' + \lambda d) \quad (2)$$

either approximately or exactly. Thus, line search algorithms are basically univariate optimization methods. Similarly to the multidimensional optimization methods, there are a number of different algorithms for finding the optimal step size, the simplest being just using a constant step size λ for every iteration.

In this thesis, we will investigate the effects of different line search algorithms on the overall performance of different optimization methods. The thesis focuses primarily on the use cases presented in the Nonlinear Optimization course that was lectured in Fall 2018 in Aalto University. The main goal of this thesis is to investigate how different line search algorithms affect the performance of optimization methods for unconstrained nonlinear optimization problems. In addition, we will carefully analyze the performance differences caused by the different line search methods with the goal of providing a suggestions on how the line search methods could be ranked.

Abbreviations and symbols used

Table 1: Abbreviations used in this thesis, and their explanations.

Abbreviation	Explanation
MM	main method, an unconstrained optimization method
LS	line search, provides step sizes for main methods
NM	Newton's Method
GDM	Gradient Descent Method
CGM	Conjugate Gradient Method
HBM	Heavy Ball Method
CS	Constant search i.e. constant step size
GSS	Golden Section line search
BS	Bisection line search
DS	Dichotomous line search
FS	Fibonacci line search
US	Uniform line search
NS	Newton's line search
AS	Armijo's line search
MSS	Matrix Square Sum target function
NE	Negative Entropy target function

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 of course the problem we are solving with it. In this thesis we are going to focus on two different problems or target functions. Both of the functions are convex and differentiable since some of the methods tested require these two properties. To increase the complexity of the problems a bit, we are going solve both problems in 50-dimensions i.e. setting $n = 50$ for the functions when they're defined in \mathbb{R}^n or similar domain.

2.1.2 Step size function

Each optimization method involves a step in form of $\operatorname{argmin}_{\lambda \in \mathbb{R}} f(\mathbf{x} + \lambda \mathbf{d})$. In practice we are going to define a step size function $g_{\mathbf{x}, \mathbf{d}}(\lambda) = f(\mathbf{x} + \lambda \mathbf{d})$ related to each of the target functions. Then on each iteration of the main optimization method, a new step size function is generated with the updated point \mathbf{x} and direction vector \mathbf{d} .

2.1.3 Matrix Square Sum

The Matrix Square Sum is an extension to the Sum Squares Function, which is defined as

$$f(\mathbf{x}) = f(x_1, \dots, x_n) = \sum_{i=1}^n i x_i^2. \quad (3)$$

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

$$f(\mathbf{x}) = \|A\mathbf{x} + \mathbf{b}\|^2 + c\|\mathbf{x}\|^2, \quad (4)$$

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

$$A = A + (|\lambda_{min}| + d)I, \quad (5)$$

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.

The values of b and c are also randomized so that for each element of b applies $b_i \in [-0.5, 0.5]$ and c is a scalar randomly generated from range $[-0.5, 0.5]$.

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:

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

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

$$\begin{aligned} \nabla f(\mathbf{x}) &= \nabla(\mathbf{x}^\top \mathbf{A}^\top \mathbf{Ax}) + 2\nabla(\mathbf{b}^\top \mathbf{Ax}) + \nabla(\mathbf{b}^\top \mathbf{b}) + \nabla(c\mathbf{x}^\top \mathbf{x}) \\ &= (\mathbf{x}^\top \mathbf{A}^\top \mathbf{A})^\top + \mathbf{A}^\top \mathbf{Ax} + 2(\mathbf{b}^\top \mathbf{A})^\top + c(\mathbf{x}^\top)^\top + c\mathbf{x} \\ &= 2\mathbf{A}^\top \mathbf{Ax} + 2\mathbf{A}^\top \mathbf{b} + 2c\mathbf{x} \end{aligned} \quad (6)$$

$$\mathbf{H} = \nabla^2 f(\mathbf{x}) = 2\mathbf{A}^\top \mathbf{A} + 2cI \quad (7)$$

From the gradientz we get the necessary optimality condition

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

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

$$\mathbf{x} = -(A^\top A + cI)^{-1} A^\top \mathbf{b} \quad (9)$$

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

$$\begin{aligned} g(\lambda) &= f(\mathbf{x} + \lambda \mathbf{d}) = \|A(\mathbf{x} + \lambda \mathbf{d}) + \mathbf{b}\|^2 + c\|\mathbf{x} + \lambda \mathbf{d}\|^2 \\ &= \lambda \mathbf{d}^\top A^\top (2A\mathbf{x} + \lambda A\mathbf{d} + 2\mathbf{b}) + \mathbf{x}^\top A^\top (A\mathbf{x} + 2\mathbf{b}) + \mathbf{b}^\top \mathbf{b} \\ &\quad + c(\mathbf{x}^\top \mathbf{x} + 2\lambda \mathbf{d}^\top \mathbf{x} + \lambda^2 \mathbf{d}^\top \mathbf{d}) \end{aligned} \quad (10)$$

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

$$g''(\lambda) = 2\mathbf{d}^\top (A^\top A\mathbf{d} + c\mathbf{d}) \quad (12)$$

2.1.4 Negative Entropy

The second target function reviewed in this thesis is an inverted entropy maximization problem, which converts it into a minimization problem [3]. We will be calling this convex minimization problem a Negative Entropy problem and it can be formulated as follows:

$$f(\mathbf{x}) = \sum_{i=1}^n x_i \log x_i, \quad (13)$$

where $\mathbf{x} \in \mathbb{R}_{>0}^n$. 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:

$$\nabla f(\mathbf{x}) = \frac{\delta}{\delta x_i} \sum_{i=1}^n x_i \log x_i \quad (14)$$

$$= \sum_{i=1}^n (\log x_i + 1) = 0. \quad (15)$$

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:

$$\mathbf{H} = \begin{bmatrix} x_1^{-1} & 0 & 0 & \dots & 0 \\ 0 & x_2^{-1} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & x_n^{-1} \end{bmatrix}. \quad (16)$$

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

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

$$g'(\lambda) = \sum_{i=1}^n d_i (\log(x_i + \lambda d_i) + 1) \quad (18)$$

$$g''(\lambda) = \sum_{i=1}^n \frac{d_i^2}{x_i + \lambda d_i}. \quad (19)$$

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

```

1: initialize  $\gamma \in [0, 1]$ 
2: while  $\min(\mathbf{x} + \lambda \mathbf{d}) \leq 0$  do
3:    $\lambda = \lambda \gamma$ 
4: end while
5: return  $\lambda$ 

```

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 evaluation 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

In this thesis we are going to call all nonlinear unconstrained optimization methods as main methods. The main methods usually follow the same formula: start from some point x_0 , select a step direction d_k and a step size λ_k , then update the next point $x_{k+1} = x_k + \lambda_k d_k$ and repeat until a method specific termination condition is reached. Choosing the step direction λ is an optimization problem of its own and its solved by the line search methods introduced in the section 2.3. [2]

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:    $\mathbf{d} = -H(\mathbf{x})^{-1}\nabla\theta(\mathbf{x})$ 
4:    $\lambda = L(g_{\mathbf{x},\mathbf{d}})$ 
5:    $\mathbf{x} = \mathbf{x} + \lambda\mathbf{d}$ 
6:    $k = k + 1$ 
7: end while
8: return  $\lambda$ 

```

2.2.2 Gradient Descent Method

Gradient descent method can be interpret as a first order of the Newton's method as it only uses the first gradient to determine step direction. The algorithm itself is exactly the same as in Newton's method except for the line 3 where the descent direction d is calculated.

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:    $\mathbf{d} = -\nabla\theta(\mathbf{x})$ 
4:    $\lambda = L(g_{\mathbf{x},\mathbf{d}})$ 
5:    $\mathbf{x} = \mathbf{x} + \lambda\mathbf{d}$ 
6:    $k = k + 1$ 
7: end while
8: return  $\lambda$ 

```

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

```

1: 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  $L$ .
2:  $\mathbf{d} = -\nabla\theta(\mathbf{x})$ 
3: while  $\|\nabla\theta(\mathbf{x})\| > l$  and  $k < k_{max}$  do
4:    $\mathbf{y} = \mathbf{x}$ 
5:   for  $i = 1 \dots n$  do
6:      $\lambda = L(g_{\mathbf{y},\mathbf{d}})$ 
7:      $\mathbf{y}_{prev} = \mathbf{y}, \mathbf{y} = \mathbf{y} + \lambda\mathbf{d}$ 
8:      $\alpha = \frac{\nabla\theta(\mathbf{y})^2}{\nabla\theta(\mathbf{y}_{prev})^2}$ 
9:      $\mathbf{d} = -\nabla\theta(\mathbf{y}) + \alpha\mathbf{d}$ 
10:     $k = k + 1$ 
11:   end for
12:    $\mathbf{x} = \mathbf{y}, \mathbf{d} = \nabla\theta(\mathbf{x})$ 
13: end while
14: return  $\lambda$ 

```

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

```

1: 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  $L$ .
2: while  $\|\nabla\theta(\mathbf{x})\| > l$  and  $k < k_{max}$  do
3:    $\mathbf{d} = -\nabla\theta(\mathbf{x}) + \beta(\mathbf{x} - \mathbf{x}_{prev})$ 
4:    $\lambda = L(g_{\mathbf{x},\mathbf{d}})$ 
5:    $\mathbf{x}_{prev} = \mathbf{x}, \mathbf{x} = \mathbf{x} + \lambda\mathbf{d}$ 
6:    $k = k + 1$ 
7: end while
8: return  $\lambda$ 

```

Add HBM parameters table

2.3 Line Search Methods

Most line search methods are based on similar ideas: pick a section and begin reducing it until some stopping condition is satisfied. There are also other approaches like backtracking search and different methods exploiting the derivatives of the target function.

There are also multiple small variations of the same methods available on different sources in the internet and print. Different sources also often tend to suggest different recommended values for the line search parameters. Because of limited resources available for optimizing the parameter selection, we decided to go with self evaluated options for the parameters with little to no inspiration from the external sources. [3] [2]

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 2: Parameters tested for ConstantSearch

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

Because Constant Search has some convergence issues with some of the configurations with the test values listed in table 2, a different set of values must be used for some main methods. For Matrix Square Sum and Newton’s Method the test values for parameter λ are $\{0.5, 0.9, 1.0, 1.1, 1.5\}$. For Negative Entropy different values are used for Newton’s Method and Conjugate Gradient Methods, respectively $\{0.1, 0.25, 0.5, 0.9\}$ and $\{0.0001, 0.1, 0.15, 0.2\}$.

2.3.2 Golden Section Search

Golden Section Search is a sequential line search meaning that it utilizes the previous iterations when calculating the next value for the step size. The Golden Section method’s name comes from it’s unique reduction factor of $\phi = 0.618$. [2]

The parameters tested for Golden Section Search are displayed in table 3.

Algorithm 6 Golden Section Search

```

1: initialize tolerance  $l > 0$ ,  $\alpha = 0.618$ ,  $k = 0$ ,  $k_{max} \in \mathbb{N}$ ,  $(a, b) \in \mathbb{R}$ .
2:  $\lambda = a + (1 - \alpha)(b - a)$ ,  $\mu = a + \alpha(b - a)$ .
3: while  $b - a > l$  and  $k < k_{max}$  do
4:   if  $\theta(\lambda) > \theta(\mu)$  then
5:      $a = \lambda$ ,  $\lambda = \mu$ ,  $\mu = a + \alpha(b - a)$ 
6:   else
7:      $b = \mu$ ,  $\mu = \lambda$ ,  $\lambda = a + (1 - \alpha)(b - a)$ 
8:   end if
9:    $k = k + 1$ 
10: end while
11: return  $\lambda = \frac{a+b}{2}$ 

```

Table 3: Parameters tested for GoldenSectionSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	5, 10	5, 10
l	1e-04, 1e-07	1e-04, 1e-07

2.3.3 Bisection Search

Bisection search is a line search that uses derivatives to progress. This requires the problems to be pseudoconvex and therefore differentiable within some closed bounds. [2].

The parameters tested for Bisection Search are displayed in table 4.

Algorithm 7 Bisection Search

```

1: initialize  $l > 0, k = 0, k_{max} \in \mathbb{N}, (a, b) \in \mathbb{R}$ .
2: while  $|b - a| > l$  and  $k < k_{max}$  do
3:    $\lambda = \frac{b+a}{2}$ 
4:   if  $\theta'(\lambda) = 0$  then
5:     return  $\lambda$ 
6:   else if  $\theta'(\lambda) > 0$  then
7:      $b = \lambda$ 
8:   else
9:      $a = \lambda$ 
10:  end if
11:   $k = k + 1$ 
12: end while
13: return  $\lambda = \frac{a+b}{2}$ 

```

Table 4: Parameters tested for BisectionSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	5, 10	5, 10
l	1e-04, 1e-07	1e-04, 1e-07

2.3.4 Dichotomous Search

Similarly to Golden Section search, Dichotomous search is a sequential search meaning it uses the past iterations to proceed. Dichotomous search progresses by selecting some closed bounds $[a, b]$ and selecting its mid-point, then selecting new bounds ϵ away from the midpoint. [2]

Algorithm 8 Dichotomous Search

```

1: initialize  $l > 0, k = 0, k_{max} \in \mathbb{N}, (a, b) \in \mathbb{R}$ .
2: while  $b - a > l$  and  $k < k_{max}$  do
3:    $\lambda = \frac{b+a}{2} - \epsilon, \mu = \frac{b+a}{2} + \epsilon$ 
4:   if  $\theta(\lambda) < \theta(\mu)$  then
5:      $b = \mu$ 
6:   else
7:      $a = \lambda$ 
8:   end if
9:    $k = k + 1$ 
10: end while
11: return  $\lambda = \frac{a+b}{2}$ 

```

Table 5: Parameters tested for DichotomousSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	5, 10	5, 10
ϵ	1e-07, 1e-10	1e-06, 1e-07
l	1e-05, 1e-09	1e-04, 1e-05

Initial tests show that Dichotomous Search has problems converging with some of the test values displayed in table 5 when using Newton's Method with Negative Entropy problem. Because of this, a slight adjustment to test parameters is done by giving a a set value of -10 for NE + NM combination.

2.3.5 Fibonacci Search

In many ways, Fibonacci Search is similar to previous sequential searches such as Golden Section search. It also functions over a closed bounds reduces the interval

with reduction factors. Instead of a constant reduction factor, Fibonacci search instead calculates a new factor for each iteration using Fibonacci numbers. [2]

The parameters tested for Fibonacci Search are displayed in table 6.

Algorithm 9 Fibonacci Search

```

1: initialize  $l > 0, \epsilon > 0, k = 0, k_{max} \in \mathbb{N}, (a, b) \in \mathbb{R}, n = \min_n \{ \frac{b-a}{F_n} \leq l \}$ .
2:  $\lambda = a + \frac{F_{n-2}}{F_n}(b - a), \mu = a + \frac{F_{n-1}}{F_n}(b - a)$ 
3: while  $k \leq n - 1$  and  $k < k_{max}$  do
4:   if  $\theta(\lambda) < \theta(\mu)$  then
5:      $a = \lambda, \lambda = \mu, \mu = a + \frac{F_{n-k-1}}{F_{n-k}}(b - a)$ 
6:   else
7:      $b = \mu, \lambda = \mu, \lambda = a + \frac{F_{n-k-2}}{F_{n-k}}(b - a)$ 
8:   end if
9:    $k = k + 1$ 
10: end while
11:  $\lambda = \theta(\lambda)$ 
12: if  $\theta(\lambda) > \theta(\mu + \epsilon)$  then
13:    $a = \lambda$ 
14: else
15:    $b = \mu$ 
16: end if
17: return  $\lambda = \frac{a+b}{2}$ 

```

The F_n represents n :th Fibonacci number. To optimize algorithm's performance, the values of F_n should be pre-evaluated. After running some quick tests, it seems that the pre-evaluation doesn't effect performance significantly and so for our implementation we decided to generate the Fibonacci numbers run-time using a single loop algorithm of time complexity $O(n)$.

Table 6: Parameters tested for FibonacciSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	5, 10	5, 10
ϵ	1e-08, 1e-10, 1e-12	1e-08, 1e-10, 1e-12
l	1e-06, 1e-10	1e-06, 1e-12

2.3.6 Uniform Search

Uniform search divides a bounded section of a function into intervals, then chooses the interval with the lowest value and repeats the process again for it. To increase the precision of the method we may also increase the number of intervals to divide the new interval into. [2]

More specifically, choose a range $[a, b]$ and split it into n sections, with one section's size being $\delta = \frac{b-a}{n}$. We end up with $n + 1$ subsections, from which we pick a $a_0 + k\delta$ with lowest value, where $k = 0 \dots n$. We then increase the section count by multiplying n with $m \geq 1$ and repeat the process with the new section. [4]

The parameters tested for Uniform Search are displayed in table 7.

Algorithm 10 Uniform Search

```

1: initialize  $l > 0, k = 0, (k_{max}, n) \in \mathbb{N}, (a, b, m) \in \mathbb{R}$ 
2:  $s = \frac{b-a}{n}, p_{min} = a$ 
3: while  $s > l$  and  $k < k_{max}$  do
4:   for  $i = 0, \dots, n$  do
5:      $x = a + is$ 
6:     if  $\theta(x) < \theta(p_{min})$  then
7:        $p_{min} = x$ 
8:     end if
9:      $k = k + 1$ 
10:  end for
11:   $a = p_{min} - s, b = p_{min} + s$ 
12:   $n = \lfloor nm \rfloor, s = \frac{b-a}{n}$ 
13: end while
14: return  $\lambda = \frac{a+b}{2}$ 

```

Table 7: Parameters tested for UniformSearch

Parameter	MatrixSquareSum	NegativeEntropy
a	-10, -5	-10, -5
b	5, 10	5, 10
n	5, 10, 100	5, 10, 20
m	1, 1.5, 2	1, 1.5, 2
l	1e-06, 1e-08	1e-05, 1e-06

2.3.7 Newton's Search

Newton's search is another line search that uses the derivatives of the target function. It exploits the function's quadratic approximations which yields an equation of second order differential. This means that the target function must be differentiable twice. [2]

Algorithm 11 Newton's Search

```

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

```

Table 8: Parameters tested for NewtonsSearch

Parameter	MatrixSquareSum	NegativeEntropy
λ	0.5, 1, 5, 10	0.1, 0.5, 1, 5
l	1e-07, 1e-08	1e-07, 1e-08

In addition to the parameters listed in table 8, the max iteration count was added as an extra parameter for Negative Entropy function with options being either 100 or 1000 iterations.

2.3.8 Armijo's Search

Armijo's search is an example of an inexact line search meaning that the step size is chosen to approximately minimize the target function. Armijo's search is also the only backtracking method of the ones included in this thesis. Backtracking means that the initial step size is reduced on every iteration until we satisfy the stopping condition. [3]

The parameters tested for Armijo's Search are displayed in table 9.

Algorithm 12 Armijo's Search

```

1: initialize  $l > 0, k = 0, k_{max} \in \mathbb{N}, (\lambda, \alpha, \beta) \in \mathbb{R}$ 
2:  $\theta_0 = \theta(0), \theta'_0 = \theta'(0)$ 
3: while  $\theta(\lambda) > \theta_0 + \alpha\lambda\theta'_0$  and  $k < k_{max}$  do
4:    $\lambda = \lambda\beta$ 
5:    $k = k + 1$ 
6: end while
7: return  $\lambda$ 

```

Table 9: Parameters tested for ArmijoSearch

Parameter	MatrixSquareSum	NegativeEntropy
λ	0.9, 1, 1.1	0.9, 1, 1.1
α	0.1, 0.25, 0.5	0.1, 0.25, 0.5
β	0.5, 0.75, 0.9	0.5, 0.75, 0.9
l	1e-07	1e-07

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 introduced in the theory section, we first need to tackle the problem of choosing parameters.

3.1 Parameter Selection

Optimization method parameter selection is a whole large topic of its own, and it is not our main focus on this thesis. Therefore a very straightforward method for picking the parameters is going to be appropriate.

Using the same setup as when we are comparing the performances of different method combinations, we can also compare the effects of different parameters. Using the manually picked parameter values shown in the theory section, we can programmatically generate all the permutations for each parameter and optimization method combination.

3.2 Starting Point Selection

So we are going to compare the performances of each line search method and main method combination with their related parameter options. We are running the tests with 100 starting points of dimension $n = 50$. For practical reasons we are also going to use a limited space for the points to be randomized from.

Let a single starting point be $x = \{x_1, x_2, \dots, x_{50}\}$. For MatrixSquareSum the points are generated so that for each point applies $x_i \in [-10, 10] \forall i$. Because NegativeEntropy's domain is $\mathbb{R}_{>0}$, the same rule applies but with the bounds being $]0, 10]$ instead.

Since we are interested in overall performance, we prefer starting points that are random but evenly distributed so that no two point are the same, or too close to each other. To generate a point distribution which is random but evenly filled with points, we use the following algorithm.

Algorithm 13 Generating Even Distribution of Random Starting Points

```

1: initialize  $(d_{min}, x_{min}, x_{max}) \in \mathbb{R}, (n, p) \in \mathbb{N}, \mathbf{x}_1 = \mathbf{random}_n(x_{min}, x_{max}), q = 1$ 
2: while  $i < p$  do
3:    $\mathbf{y} = \mathbf{random}_n(x_{min}, x_{max})$ 
4:    $\mathbf{ok} = \mathbf{true}$ 
5:   for  $j = 1 \dots q$  do
6:     if  $\|\mathbf{x}_j - \mathbf{y}\| < d_{min}$  then
7:        $\mathbf{ok} = \mathbf{false}$ 
8:       break
9:     end if
10:  end for
11:  if  $\mathbf{ok}$  then
12:     $q = q + 1, i = i + 1$ 
13:     $\mathbf{x}_q = \mathbf{y}$ 
14:  end if
15: end while
16: return  $\{x_1, x_2, \dots, x_p\}$ 

```

The algorithm generates p points of dimension n . The generated points are stored in variables x_i . The function $\mathbf{random}_n(x_{min}, x_{max})$ generates a single point of dimension n filled with random values from the range $[x_{min}, x_{max}]$. On every iteration, we test to see if the newly generated point is closer than d_{min} to some existing point and if so, we discard the new point and try again.

This is by means not a perfect algorithm since its performance heavily depends on the d_{min} value used. The minimum distance selection depends on the point count and the available point space: too high values cause the algorithm to end up on infinite loop and too low values do not provide the even distribution we are looking for. Via trial and error the values presented in table 10 are picked.

Table 10: The minimum Eucleidean distances used for different target functions on distributions of 10 and 1000 points.

Point count	MSS d_{min}	NE d_{min}
10	64	32
1000	48	24

After successfully generating the starting points once with the minimum distances shown in table 10, we are going to save them into a file from which they are loaded during future runs. This way we maintain the same level of randomness for each run.

3.3 Randomizing Target Function

In addition to randomizing the starting points, we are going to use randomized parameters for Matrix Square Sum target function. The values of parameters A , b and c are going to be randomized as described in section 2.1.3 but for each starting point separately. To control the randomness between runs, we give the function a random seed of the index of the current point. This way we end up with number of random functions equal to the number of starting points used. Combined with the saving of points, we can guarantee to get the same setup for each run.

3.4 Comparing Performance

There are multiple metrics that we could analyze, like algorithm success rate, main method or line search duration, iterations or function calls. The most important metric we are going to focus on is the success rate i.e. from how many of the 1000 starting points did the algorithm reach correct optimum. The solution is interpret as correct if the first 8 decimals of the found solution and correct one match. Since it is excepted that most of the algorithms find the solution correctly every time, we also need a secondary comparison metric. Here we choose to use the total duration of the algorithm, since it is often the most interesting value for real life use cases and it is comparable within different methods unlike for example total iteration

count. The other metrics will also be recorded and listed in the results section, but they will not be taken into consideration when scoring the methods.

3.4.1 Parameter Selection

4 Results

The results are going to be represented in a table with average performances of each line search method for each permutation of main methods and target functions. The performances are calculated using the best parameters found for each setup as described in the previous section. The algorithms are run with a similar setup as when calculating the best parameters, but this time with 1000 evenly randomized starting points instead of 100. Again for each starting point a unique target function configuration is generated in the case of MatrixSquareSum using the same randomizing rules as described in the previous section. The random seeds were again starting point indices i.e. integers from 0 to 1000.

After running the performance tests, we are left with the results introduced in the following tables 11 to 18. The row's of the table show the line search method used, the success rate of the algorithm, average duration, average main method iteration count and function calls of the algorithm. The function call count includes all calls to the function and to its gradients or step size functions. In addition to these, the last two columns show average duration and iterations used by the line search algorithm alone. A table including all eight line search methods with their representative performances is generated for each main method and target function combination.

Table 11: 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	429.1	1.0	77.0	3.3	35.0
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	100.0	424.2	1.0	10.0	0.9	0.0

Table 12: 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 13: 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 14: 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 15: 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 16: 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 17: 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
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
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

Table 18: 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 performances of the main methods in relation to the choice of line search method, we always want to analyze the data per line search method. We therefore group our data by line search method and either main method or target problem. The most important metric for performance is the actual success rate of the algorithm. However, since success rates seem to have lots of ties, we need a secondary metric, which we choose to be the average duration

of the main method. While not universal within different implementations or problems, this provides us a decent comparison method of the line search methods within a specific main method or target problem. This means that all the values are mainly comparable within the same column in the following tables 19 to 22. To visualize the column-wise differences, we apply a per column color scale so that darker red imply worse performance and darker greens imply better performance.

Table 19: Average duration of the algorithms using different line search methods for each main method when finding minimum for Matrix Square Sum function. Colored per column so that darker red is worse (slower) and darker green is better (faster).

Line Search Method	NM	GDM	CGM	HBM
ConstantSearch	445,80	4 681,20	989,30	4 711,50
GoldenSectionSearch	445,00	283,70	429,50	227,70
BisectionSearch	535,90	823,50	1 154,90	666,60
DichotomousSearch	429,10	270,10	426,30	209,40
FibonacciSearch	453,20	354,30	517,80	353,20
UniformSearch	455,00	389,20	558,20	322,40
NewtonsSearch	443,50	453,30	552,80	375,10
ArmijoSearch	424,20	209,30	405,30	217,30

Table 20: Average duration of the algorithms using different line search methods for each main method when finding minimum for Negative Entropy function. Colored per column so that darker red is worse (slower) and darker green is better (faster).

Line Search Method	NM	GDM	CGM	HBM
ConstantSearch	3 718,10	13,10	4 393,40	8,30
GoldenSectionSearch	2 038,30	53,40	539,70	92,40
BisectionSearch	2 191,60	92,20	759,50	131,70
DichotomousSearch	2 231,20	54,70	576,40	68,80
FibonacciSearch	2 100,70	58,40	623,70	72,10
UniformSearch	2 316,00	121,80	6 371,10	134,90
NewtonsSearch	2 266,00	458,80	2 801,70	541,00
ArmijoSearch	2 206,90	11,60	198,40	15,50

Table 21: The average of the durations for finding the minimum for both Matrix Square Sum and Negative Entropy Functions combined and listed separately for each main method. Colored per column so that darker red is worse (slower) and darker green is better (faster).

Line Search Method	NM	GDM	CGM	HBM
ConstantSearch	2 081,95	2 347,15	2 691,35	2 359,90
GoldenSectionSearch	1 241,65	168,55	484,60	160,05
BisectionSearch	1 363,75	457,85	957,20	399,15
DichotomousSearch	1 330,15	162,40	501,35	139,10
FibonacciSearch	1 276,95	206,35	570,75	212,65
UniformSearch	1 385,50	255,50	3 464,65	228,65
NewtonsSearch	1 354,75	456,05	1 677,25	458,05
ArmijoSearch	1 315,55	110,45	301,85	116,40

Table 22: The average of the durations for finding the minimum with all the main methods listed separately for each target function. Colored per column so that darker red is worse (slower) and darker green is better (faster).

Line Search Method	MSS	NE	OVERALL
ConstantSearch	2 706,95	2 033,23	2 370,09
GoldenSectionSearch	346,48	680,95	513,71
BisectionSearch	795,23	793,75	794,49
DichotomousSearch	333,73	732,78	533,25
FibonacciSearch	419,63	713,73	566,68
UniformSearch	431,20	2 235,95	1 333,58
NewtonsSearch	456,18	1 516,88	986,53
ArmijoSearch	314,03	608,10	461,06

While the primary metric for comparison is the success rate, it does not offer as interesting differences as the durations above. However, it is an important piece of information to be read along with the duration comparison to understand the relations of different line search methods better. The successrates are listed in the following tables 23 and 24 in a similar manner colored column-wise but this time coloring only the unoptimal results with red.

Table 23: The success rates of the main methods using each line search method for finding the minimum of Matrix Square Sum function. Darker red color means lower success rate.

Line Search Method	NM	GDM	CGM	HBM
ConstantSearch	100,00	98,90	99,60	98,90
GoldenSectionSearch	100,00	99,40	100,00	99,40
BisectionSearch	99,60	99,40	100,00	99,40
DichotomousSearch	100,00	99,40	100,00	99,40
FibonacciSearch	100,00	99,40	100,00	99,40
UniformSearch	100,00	99,40	100,00	99,40
NewtonsSearch	100,00	99,40	100,00	99,40
ArmijoSearch	100,00	99,60	100,00	99,60

Table 24: The success rates of the main methods using each line search method for finding the minimum of Negative Entropy function. Darker red color means lower success rate.

Line Search Method	NM	GDM	CGM	HBM
ConstantSearch	100,00	100,00	99,90	100,00
GoldenSectionSearch	100,00	100,00	100,00	100,00
BisectionSearch	100,00	100,00	100,00	100,00
DichotomousSearch	100,00	100,00	100,00	100,00
FibonacciSearch	100,00	100,00	100,00	100,00
UniformSearch	100,00	100,00	100,00	100,00
NewtonsSearch	100,00	100,00	100,00	100,00
ArmijoSearch	100,00	100,00	100,00	100,00

In addition to comparing the average performance, an important factor to consider is the consistency of the results. The boxplots in figures 1 and 2 give us valuable

insight of the variance of the algorithm's performance. The box plot is generated so that half of the test points are included in the box area and the best and worse 25 % are shown as whiskers above and below the box. The orange line within the box shows the actual average duration.

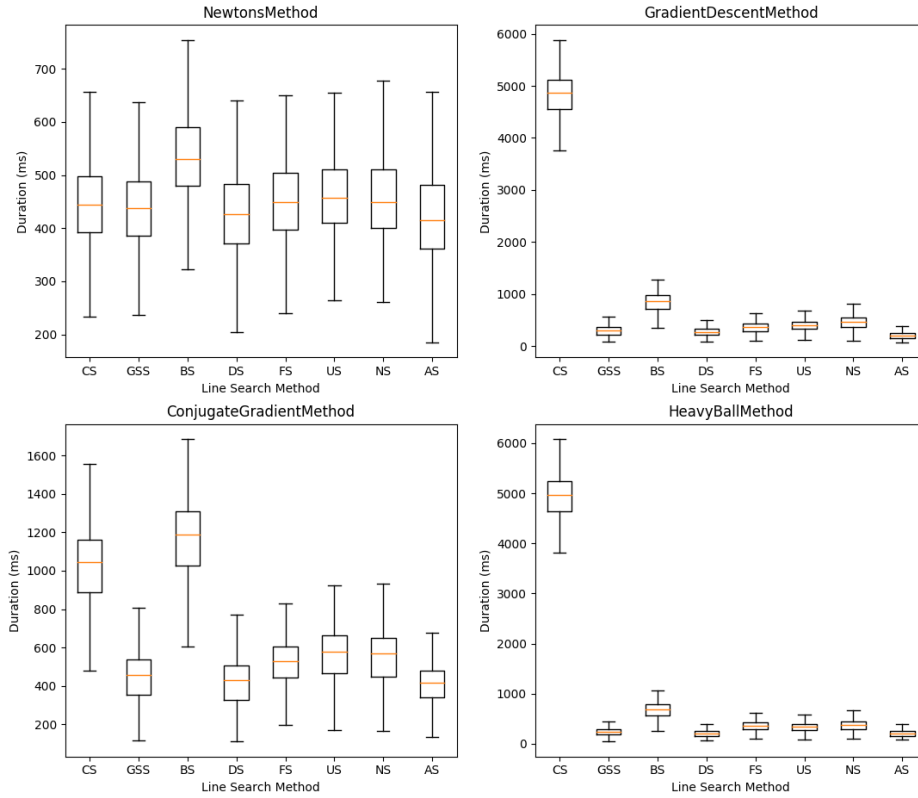


Figure 1: Box plot of solution times per line search and main method for MatrixSquareSum target function.

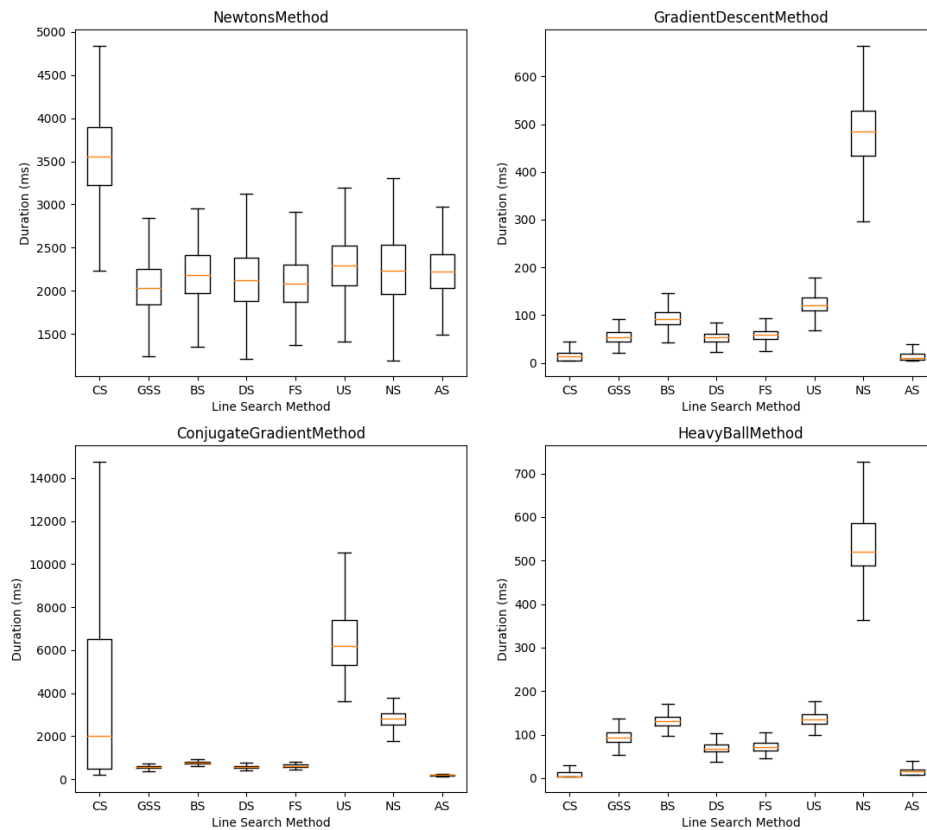


Figure 2: Box plot of solution times per line search and main method for NegativeEntropy target function.

Explain the box plot method. Apparently the box contains 50% of the dataset and the lines rest or smthng like that?

Should I make separate boxplots with the best searches or something to improve the scale for their comparison?

Constant Search

Since Constant Search simply provides constant step sizes it should provide a good baseline for comparing other line search methods. The execution of Constant Search does basically cost zero time, so the main stress remains on the main

method. This way we may be able to conclude something about the baseline performance of the main methods as well.

Looking at the overall durations of Constant Search in table 22 and the success rates at tables 23 and 24 we can quickly see that the Constant Search is the worse performing method as expected. The only scenario where constant step sizes prove themselves as rather effective are when optimizing the Negative Entropy function using Gradient Descent or Heavy Ball methods. In those cases, Constant Search was actually the best or second to best performer. This is due to the overall effectivity of gradient descent based methods for the said problem in general, and since the Constant Search is the fastest to execute, it provides a slightly better performance. This can be proven to be the case when looking at the overall duration and line search duration columns in tables 16 and 18: most of the overall duration is from the line searches.

Looking at the box plots in 1 and 2 Constant Search is clearly the worst performer by not just absolute durations but also due to the high variance of results. The only exception to this are the two previously mentioned cases when optimizing Negative Entropy with Gradient Descent based methods, where Constant Search scores one of the lowest variances.

Golden Section Search

Comparing the overall results in tables 21 and 22, we immediately notice from the solid green colored rows that Golden Section Search is one of the best performers. The success rates of this methods are also great as it does, like most methods tested, find the correct solution in almost all of the test cases.

The Golden Section Search has average variances as can be seen from box plots, which combined with one of the lowest average durations make this method one of the best of the tested line searches.

Bisection Search

Bisection Search does perform rather poorly in our test setup. It scores the second lowest total success rate, though only losing by 0.4 percentile units to others in the case of Matrix Square Sum optimized with Newton's Method. The performance is especially poor when optimizing Matrix Square Sum function and barely average with other setups, causing the overall score to be the average of the tested methods.

Judging by the box plots, the Bisection Method does also show a bit higher variances in performance than other methods especially in the case of Matrix Square Sum.

Dichotomous Search

In our first tests Dichotomous Search was one of the slower methods due to not reaching the correct minimums within the set max iterations. However, lowering the precision slightly proved to be especially helpful for this method, which ultimately turned out as one of the stronger methods, by even outperforming Golden Section Search in Matrix Square Sum optimization and scoring very similar overall scores. The success rates are also identical and reviewing variance from box plots shows no significant differences between the two methods, which share the second best place in our comparison.

Fibonacci Search

Fibonacci Search scores a slightly above average results in our performance tests, and performs much better in Negative Entropy optimization when compared to other line searchers' performances. The success rates are average and variances are at the same level with Dichotomous and Golden Section Searches.

Our implementation of Fibonacci Search does compute each Fibonacci number live without storing them. This proved to not be an issue since the largest Fibonacci reached was around F_{50} . A non-official test was also performed with precomputed Fibonacci numbers, and it only improved the overall performance by approximately 10%. For real life applications precomputation should definitely be added if possible.

Uniform Search

Uniform Search is known to be slow and not very sophisticated method. This also shows in the performance as it scores averagely on Matrix Square Sum and below average for Negative Entropy. Uniform Search also struggles especially with Negative Entropy when using Conjugate Gradient Method, where it often reaches max iteration limits. This is likely due to the way we limit the domain using the Domain Limiter (algorithm 1). Overall, Uniform Search scored second lowest performances with average success rates and above average variances.

Newton's Search

Similarly to Uniform Search, Newton's Search has average performance when optimizing Matrix Square Sum but quickly falls behind in the case of Negative Entropy. Newton's Search also causes multiple warnings of max iterations reached regardless of the max iteration limit used. However, the method does eventually end up providing good enough step sizes so that success rates stayed average.

It seems that the method does not converge properly when combined with the Domain Limiter which alters the step size given by the derivatives. Because of this, a way smaller max iteration limit of just 100 or 1000 iterations proved much more effective in the case of Negative Entropy. Due to this and the use of domain limiter, the results of Newton's Search for Negative Entropy are more unreliable than in case of Matrix Square Sum. This is confirmed by the box plots as well: Newton's Search has some of the largest variances in performance of optimizing the Negative Entropy function.

Armijo's Search

Armijo's Search proves to be the overall best performing method of the tested line searches. It has the best performance in five categories out of eight, and is right on par with the top performers in the rest of the categories as well. It also has a slightly better success rate than any other method on Matrix Square Sum optimization when using Gradient Descent or Heavy Ball Method. The overall performance is also approximately 10% better than the next best overall performance of Golden Section Search. There are some differences in the variances: Armijo's Search has a slightly higher variance in Matrix Square Sum using Newton's Method but it outperforms most other method's variances when using other methods for the same function. For Negative Entropy the Armijo's Search has clearly lower variances than other methods.

4.2 Assumptions

The results introduced in this thesis contain a number of assumptions as all research usually does. The assumptions made in this paper are discussed here briefly.

One major factor to take into account when considering the results shown in this thesis is that there are only two types of different target functions that the methods are used for. Even though some variety is added by the randomization of parameters in Matrix Square Sum, both of the target functions remain relatively simple. The randomization of MSS parameters is also one factor to take into account. Randomized values are generated using different random seeds of 0 to 999 matching the index of starting points. The randomized values are selected from the range $[-0.5, 0.5]$ and the way multiplier matrix A is confirmed to be positive-definite only yields certain type of matrices. Therefore this can not be held as a totally comprehensive study of a single type of target function either.

Another major factor to discuss is the method we use for picking values for the optimization methods. Since the problem of picking the absolute best parameters is really complex in itself, the method of picking a set of options and then comparing

the performance of each permutation separately for each setup, is justified. However, by implementing a better parameter selection, such as increasing the number of options and test cases, we can likely reach better results for each optimization methods. The tradeoff would be a higher computation time to find the optimal parameters, and while the results would highly likely improve, we can not know by how much.

Some factors in performance are also extremely difficult to review accurately. These include the actual implementation of the methods which varies in different sources. Most notably one difference seems to be the positioning of the stopping condition within the main methods: some sources suggest that it is better to check the condition before updating the point but in our implementation the check is always done just after the update. Another optimization that could be done is the pre-calculation of line search values. This means that we pre-evaluate and save the values for each different input for each line search so that instead of running the algorithm during runtime we simply get the pre-evaluated value from our dictionary. [3] [2]

In addition, factors like choice programming language and CPU or GPU performance may effect the optimal results. The tests within this thesis are performed with Python 3.7.3 using a multiprocessing library and an Intel i5-8500K CPU. Different implementations and programming languages, such as moving the heavy calculations to GPU's, may for example optimize matrix calculations differently causing potentially unexpected differences within otherwise same test setups. For this reason, if the goal is to fully optimize the process, one is suggested to always run the performance testing separately for each test setup in use.

5 Conclusion and Future Research

To conclude with a strict ranking for different line search methods, we should consider many more scenarios making the computation exponentially more complex. However, what we can conclude is the answer to the question introduced in the very beginning of this thesis: the choice of line search method seems to have a varying size of impact on the performance of an optimization method. Comparing the known to be worst line search methods with the best ones, there are differences of up to a single magnitude of ten. The difference in performance for the best and second best method is around 11 % and the fourth is around 23 % slower than the fastest one.

Continuing from here with a goal of generalizing the results further, I would definitely focus on increasing the number of target functions first. The only issue in

this would be that the optimal parameters are likely to change with the target functions, forcing us to re-run the comparisons and find new values for all the parameters. Since this requires very high computation times, a better method for parameter selection should be implemented prior to adding new target functions. This could be something simple such as deciding a strict set of few options for each parameter but then the final results could be highly interfered by the limited parameter values.

To conclude, I would say it could be worth it to perform further research using the methods we introduce in this thesis but only if the targeted problem is something worth spending a while just to reach a potentially very slight optimization. In turn, if we would like to compare for example another set of line search methods, main methods or target functions, we should probably focus on finding better solutions to the choice of optimal parameters problems first.

References

- [1] N. Andrei. An unconstrained optimization test functions collection. *Advanced Modeling and Optimization, Volume 10, Number 1, 2008*, 2000.
- [2] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms, 3rd Edition*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2006.
- [3] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.
- [4] F. Oliveira. Ms-e2122 - nonlinear optimisation. *Operations Research, Aalto University*, 2018.

A Appendix

Constant Search

Figure 3: Best parameter permutations for different main methods using ConstantSearch for optimizing the MatrixSquareSum function with 10 sample points.

NewtonMethod		
(λ)	s (%)	t (ms)
(1.0,)	100.0	436.3
(0.9,)	100.0	4072.7
(1.1,)	100.0	4389.6
(1.5,)	100.0	11768.2
(0.5,)	100.0	13857.0

(a) Top 5 permutations with NewtonMethod

GradientDescentMethod		
(λ)	s (%)	t (ms)
(0.0001,)	100.0	8299.9
(0.5,)	0.0	162.8
(0.9,)	0.0	169.7
(0.1,)	0.0	187.9
(0.25,)	0.0	193.4

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(λ)	s (%)	t (ms)
(0.0001,)	100.0	1054.7
(0.1,)	0.0	192.4
(0.5,)	0.0	270.2
(0.25,)	0.0	280.1
(0.9,)	0.0	298.0

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
(λ, β_{HBM})	s (%)	t (ms)
(0.0001, 0.1)	100.0	6437.8
(0.0001, 0.5)	100.0	6580.6
(0.0001, 10.0)	100.0	6714.6
(0.0001, 5.0)	100.0	6784.6
(0.0001, 1.0)	100.0	6838.7

(d) Top 5 permutations with HeavyBallMethod

Figure 4: Best parameter permutations for different main methods using ConstantSearch for optimizing the NegativeEntropy function with 10 sample points.

NewtonsMethod		
(λ)	s (%)	t (ms)
(0.9,)	100.0	3379.7
(0.5,)	100.0	7239.4
(0.25,)	100.0	17055.7
(0.1,)	100.0	46295.4

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(λ)	s (%)	t (ms)
(0.5,)	100.0	17.8
(0.25,)	100.0	57.5
(0.1,)	100.0	226.9

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(λ)	s (%)	t (ms)
(0.1,)	100.0	2361.4
(0.0001,)	100.0	4281.1
(0.15,)	100.0	4969.4
(0.2,)	100.0	6062.7

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
(λ, β_{HBM})	s (%)	t (ms)
(0.5, 0.1)	100.0	9.1
(0.5, 0.5)	100.0	14.7
(0.25, 0.1)	100.0	15.3
(0.25, 0.5)	100.0	15.8
(0.25, 1.0)	100.0	18.2

(d) Top 5 permutations with HeavyBallMethod

Golden Section Search

Figure 5: Best parameter permutations for different main methods using GoldenSectionSearch for optimizing the MatrixSquareSum function with 10 sample points.

NewtonsMethod		
(a, b, l)	s (%)	t (ms)
$(-10, 5, 1e-07)$	100.0	515.6
$(-10, 10, 1e-07)$	100.0	868.6
$(-10, 10, 0.0001)$	100.0	887.5
$(-5, 5, 1e-07)$	100.0	891.0
$(-5, 10, 0.0001)$	100.0	914.0

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(a, b, l)	s (%)	t (ms)
$(-10, 5, 0.0001)$	100.0	279.4
$(-5, 10, 0.0001)$	100.0	318.0
$(-5, 5, 0.0001)$	100.0	344.3
$(-10, 5, 1e-07)$	100.0	371.1
$(-10, 10, 0.0001)$	100.0	379.6

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, l)	s (%)	t (ms)
$(-5, 10, 0.0001)$	100.0	436.7
$(-10, 10, 0.0001)$	100.0	467.8
$(-5, 5, 0.0001)$	100.0	489.2
$(-10, 5, 0.0001)$	100.0	529.8
$(-10, 10, 1e-07)$	100.0	533.9

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
(a, b, l, β_{HBM})	s (%)	t (ms)
$(-5, 10, 0.0001, 10.0)$	100.0	230.6
$(-10, 5, 0.0001, 10.0)$	100.0	236.7
$(-10, 10, 0.0001, 10.0)$	100.0	243.6
$(-5, 10, 0.0001, 5.0)$	100.0	244.8
$(-5, 5, 0.0001, 10.0)$	100.0	252.1

(d) Top 5 permutations with HeavyBallMethod

Figure 6: Best parameter permutations for different main methods using Golden-SectionSearch for optimizing the NegativeEntropy function with 10 sample points.

NewtonsMethod		
(a, b, l)	s (%)	t (ms)
$(-5, 5, 1e-07)$	100.0	1758.0
$(-10, 5, 0.0001)$	100.0	1905.6
$(-10, 10, 0.0001)$	100.0	1946.7
$(-10, 10, 1e-07)$	100.0	1947.3
$(-5, 5, 0.0001)$	100.0	1968.5

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(a, b, l)	s (%)	t (ms)
$(-5, 5, 0.0001)$	100.0	57.1
$(-5, 5, 1e-07)$	100.0	68.4
$(-5, 10, 0.0001)$	100.0	85.4
$(-5, 10, 1e-07)$	100.0	85.8
$(-10, 5, 1e-07)$	100.0	109.9

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, l)	s (%)	t (ms)
$(-5, 5, 0.0001)$	100.0	591.3
$(-10, 5, 0.0001)$	100.0	644.1
$(-5, 5, 1e-07)$	100.0	644.4
$(-5, 10, 0.0001)$	100.0	648.5
$(-10, 5, 1e-07)$	100.0	672.1

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
(a, b, l, β_{HBM})	s (%)	t (ms)
$(-5, 5, 0.0001, 0.1)$	100.0	72.3
$(-10, 5, 0.0001, 0.1)$	100.0	75.0
$(-5, 10, 0.0001, 0.1)$	100.0	75.2
$(-5, 5, 1e-07, 0.1)$	100.0	79.7
$(-5, 10, 1e-07, 0.1)$	100.0	80.4

(d) Top 5 permutations with HeavyBallMethod

Bisection Search

Figure 7: Best parameter permutations for different main methods using BisectionSearch for optimizing the MatrixSquareSum function with 10 sample points.

NewtonMethod		
(a, b, l)	s (%)	t (ms)
$(-10, 5, 1e-07)$	100.0	555.4
$(-5, 10, 1e-07)$	100.0	597.9
$(-5, 5, 1e-07)$	100.0	962.4
$(-10, 5, 0.0001)$	100.0	1025.9
$(-10, 10, 0.0001)$	100.0	1049.4

(a) Top 5 permutations with NewtonMethod

GradientDescentMethod		
(a, b, l)	s (%)	t (ms)
$(-10, 5, 0.0001)$	100.0	836.1
$(-5, 5, 0.0001)$	100.0	911.2
$(-10, 10, 0.0001)$	100.0	921.0
$(-5, 10, 0.0001)$	100.0	956.1
$(-5, 5, 1e-07)$	100.0	1094.3

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, l)	s (%)	t (ms)
$(-10, 10, 0.0001)$	100.0	1152.1
$(-5, 5, 0.0001)$	100.0	1153.7
$(-10, 5, 0.0001)$	100.0	1157.4
$(-5, 10, 0.0001)$	100.0	1175.7
$(-5, 10, 1e-07)$	100.0	1575.7

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
(a, b, l, β_{HBM})	s (%)	t (ms)
$(-10, 10, 0.0001, 10.0)$	100.0	581.2
$(-5, 10, 0.0001, 5.0)$	100.0	627.6
$(-5, 10, 0.0001, 10.0)$	100.0	676.6
$(-5, 5, 0.0001, 10.0)$	100.0	706.7
$(-5, 5, 0.0001, 5.0)$	100.0	714.3

(d) Top 5 permutations with HeavyBallMethod

Figure 8: Best parameter permutations for different main methods using BisectionSearch for optimizing the NegativeEntropy function with 10 sample points.

NewtonsMethod		
(a, b, l)	s (%)	t (ms)
$(-10, 10, 0.0001)$	100.0	1753.5
$(-10, 5, 0.0001)$	100.0	2006.3
$(-5, 5, 1e-07)$	100.0	2044.7
$(-5, 10, 0.0001)$	100.0	2075.4
$(-5, 5, 0.0001)$	100.0	2162.0

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(a, b, l)	s (%)	t (ms)
$(-5, 5, 0.0001)$	100.0	105.5
$(-10, 5, 0.0001)$	100.0	114.6
$(-5, 10, 0.0001)$	100.0	115.6
$(-10, 5, 1e-07)$	100.0	128.9
$(-5, 5, 1e-07)$	100.0	136.3

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, l)	s (%)	t (ms)
$(-5, 5, 0.0001)$	100.0	883.4
$(-10, 10, 0.0001)$	100.0	891.5
$(-10, 5, 0.0001)$	100.0	893.8
$(-5, 10, 0.0001)$	100.0	916.2
$(-10, 5, 1e-07)$	100.0	1092.6

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
(a, b, l, β_{HBM})	s (%)	t (ms)
$(-10, 10, 0.0001, 0.1)$	100.0	138.9
$(-5, 5, 0.0001, 0.1)$	100.0	142.4
$(-10, 5, 0.0001, 0.1)$	100.0	153.4
$(-5, 10, 0.0001, 0.1)$	100.0	154.3
$(-5, 5, 0.0001, 0.5)$	100.0	182.9

(d) Top 5 permutations with HeavyBallMethod

Dichotomous Search

Figure 9: Best parameter permutations for different main methods using DichotomousSearch for optimizing the MatrixSquareSum function with 10 sample points.

NewtonsMethod			GradientDescentMethod		
(a, b, ϵ, l)	s (%)	t (ms)	(a, b, ϵ, l)	s (%)	t (ms)
$(-5, 10, 1e-10, 1e-09)$	100.0	388.3	$(-10, 5, 1e-07, 1e-05)$	100.0	234.8
$(-10, 10, 1e-10, 1e-09)$	100.0	436.4	$(-10, 5, 1e-08, 0.0001)$	100.0	241.2
$(-5, 5, 1e-10, 1e-09)$	100.0	439.6	$(-5, 5, 1e-07, 0.0001)$	100.0	252.5
$(-10, 5, 1e-10, 1e-09)$	100.0	455.0	$(-5, 10, 1e-08, 0.0001)$	100.0	252.9
$(-5, 10, 1e-07, 1e-05)$	100.0	840.8	$(-10, 5, 1e-08, 1e-05)$	100.0	260.4

(a) Top 5 permutations with NewtonsMethod

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod			HeavyBallMethod		
(a, b, ϵ, l)	s (%)	t (ms)	$(a, b, \epsilon, l, \beta_{HBM})$	s (%)	t (ms)
$(-5, 10, 1e-07, 1e-05)$	100.0	354.2	$(-5, 5, 1e-07, 0.0001, 10.0)$	100.0	173.6
$(-5, 10, 1e-07, 0.0001)$	100.0	382.0	$(-10, 10, 1e-07, 0.0001, 10.0)$	100.0	185.4
$(-5, 5, 1e-07, 0.0001)$	100.0	397.3	$(-5, 10, 1e-08, 0.0001, 10.0)$	100.0	193.9
$(-10, 10, 1e-08, 1e-05)$	100.0	406.2	$(-10, 5, 1e-08, 1e-05, 5.0)$	100.0	202.6
$(-10, 5, 1e-08, 0.0001)$	100.0	406.7	$(-5, 5, 1e-07, 1e-05, 10.0)$	100.0	205.2

(c) Top 5 permutations with ConjugateGradientMethod

(d) Top 5 permutations with HeavyBallMethod

Figure 10: Best parameter permutations for different main methods using DichotomousSearch for optimizing the NegativeEntropy function with 10 sample points.

NewtonsMethod			GradientDescentMethod		
(a, b, ϵ, l)	s (%)	t (ms)	(a, b, ϵ, l)	s (%)	t (ms)
$(-10, 10, 1e-07, 1e-05)$	100.0	1756.0	$(-5, 5, 1e-07, 0.0001)$	100.0	54.3
$(-10, 5, 1e-06, 1e-05)$	100.0	1846.8	$(-5, 10, 1e-06, 0.0001)$	100.0	61.7
$(-10, 10, 1e-06, 0.0001)$	100.0	1882.9	$(-5, 10, 1e-06, 1e-05)$	100.0	63.8
$(-10, 5, 1e-06, 0.0001)$	100.0	1893.5	$(-5, 5, 1e-07, 1e-05)$	100.0	64.6
$(-10, 10, 1e-06, 1e-05)$	100.0	1923.0	$(-5, 10, 1e-07, 0.0001)$	100.0	65.4

(a) Top 5 permutations with NewtonsMethod

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod			HeavyBallMethod		
(a, b, ϵ, l)	s (%)	t (ms)	$(a, b, \epsilon, l, \beta_{HBM})$	s (%)	t (ms)
$(-5, 5, 1e-06, 0.0001)$	100.0	640.6	$(-5, 5, 1e-06, 1e-05, 0.1)$	100.0	70.4
$(-10, 5, 1e-06, 0.0001)$	100.0	642.7	$(-5, 5, 1e-06, 0.0001, 0.1)$	100.0	70.8
$(-5, 5, 1e-07, 0.0001)$	100.0	645.3	$(-10, 5, 1e-07, 0.0001, 0.1)$	100.0	73.3
$(-5, 10, 1e-07, 0.0001)$	100.0	657.0	$(-5, 10, 1e-06, 0.0001, 0.1)$	100.0	74.0
$(-5, 5, 1e-06, 1e-05)$	100.0	680.9	$(-5, 5, 1e-07, 0.0001, 0.1)$	100.0	74.0

(c) Top 5 permutations with ConjugateGradientMethod

(d) Top 5 permutations with HeavyBallMethod

Fibonacci Search

Figure 11: Best parameter permutations for different main methods using FibonacciSearch for optimizing the MatrixSquareSum function with 10 sample points.

NewtonsMethod			GradientDescentMethod		
(a, b, ϵ, l)	s (%)	t (ms)	(a, b, ϵ, l)	s (%)	t (ms)
$(-5, 5, 1e-10, 1e-10)$	100.0	443.2	$(-5, 10, 1e-12, 1e-06)$	100.0	355.6
$(-5, 5, 1e-12, 1e-10)$	100.0	452.3	$(-5, 10, 1e-10, 1e-06)$	100.0	356.0
$(-5, 5, 1e-08, 1e-10)$	100.0	459.6	$(-5, 5, 1e-12, 1e-06)$	100.0	366.8
$(-10, 5, 1e-10, 1e-10)$	100.0	460.9	$(-10, 10, 1e-12, 1e-06)$	100.0	369.2
$(-10, 5, 1e-12, 1e-10)$	100.0	469.7	$(-10, 5, 1e-12, 1e-06)$	100.0	369.4

(a) Top 5 permutations with NewtonsMethod

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod			HeavyBallMethod		
(a, b, ϵ, l)	s (%)	t (ms)	$(a, b, \epsilon, l, \beta_{HBM})$	s (%)	t (ms)
$(-5, 10, 1e-10, 1e-06)$	100.0	473.8	$(-10, 5, 1e-12, 1e-06, 10.0)$	100.0	251.4
$(-10, 10, 1e-10, 1e-06)$	100.0	485.0	$(-5, 5, 1e-12, 1e-06, 10.0)$	100.0	268.5
$(-5, 10, 1e-12, 1e-06)$	100.0	490.3	$(-5, 5, 1e-08, 1e-06, 10.0)$	100.0	273.0
$(-10, 5, 1e-08, 1e-06)$	100.0	508.6	$(-10, 10, 1e-12, 1e-06, 10.0)$	100.0	273.5
$(-10, 10, 1e-12, 1e-06)$	100.0	516.0	$(-5, 5, 1e-10, 1e-06, 10.0)$	100.0	273.5

(c) Top 5 permutations with ConjugateGradientMethod

(d) Top 5 permutations with HeavyBallMethod

Figure 12: Best parameter permutations for different main methods using FibonacciSearch for optimizing the NegativeEntropy function with 10 sample points.

NewtonMethod			GradientDescentMethod		
(a, b, ϵ, l)	s (%)	t (ms)	(a, b, ϵ, l)	s (%)	t (ms)
(-5, 5, 1e-12, 1e-12)	100.0	1834.0	(-5, 5, 1e-12, 1e-06)	100.0	74.5
(-10, 5, 1e-08, 1e-06)	100.0	1908.9	(-5, 10, 1e-08, 1e-06)	100.0	75.8
(-10, 10, 1e-12, 1e-06)	100.0	1946.8	(-5, 5, 1e-10, 1e-06)	100.0	78.0
(-5, 10, 1e-10, 1e-06)	100.0	1957.9	(-5, 5, 1e-08, 1e-06)	100.0	78.3
(-10, 10, 1e-10, 1e-06)	100.0	1967.3	(-10, 5, 1e-12, 1e-12)	100.0	78.6

(a) Top 5 permutations with NewtonMethod

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod			HeavyBallMethod		
(a, b, ϵ, l)	s (%)	t (ms)	$(a, b, \epsilon, l, \beta_{HBM})$	s (%)	t (ms)
(-5, 5, 1e-08, 1e-06)	100.0	682.8	(-5, 5, 1e-12, 1e-06, 0.1)	100.0	76.4
(-5, 5, 1e-12, 1e-06)	100.0	683.5	(-5, 5, 1e-10, 1e-06, 0.1)	100.0	82.2
(-5, 5, 1e-10, 1e-06)	100.0	701.5	(-10, 5, 1e-08, 1e-06, 0.1)	100.0	85.1
(-10, 5, 1e-10, 1e-06)	100.0	728.0	(-5, 10, 1e-10, 1e-06, 0.1)	100.0	86.9
(-5, 10, 1e-08, 1e-06)	100.0	729.5	(-5, 5, 1e-08, 1e-06, 0.1)	100.0	87.6

(c) Top 5 permutations with ConjugateGradientMethod

(d) Top 5 permutations with HeavyBallMethod

Uniform Search

Figure 13: Best parameter permutations for different main methods using Uniform-Search for optimizing the MatrixSquareSum function with 10 sample points.

NewtonsMethod		
(a, b, n, m, l)	s (%)	t (ms)
$(-10, 5, 5, 1.5, 1e-06)$	100.0	424.0
$(-5, 5, 10, 1.5, 1e-06)$	100.0	424.5
$(-5, 5, 10, 1, 1e-08)$	100.0	449.1
$(-10, 10, 5, 1.5, 1e-08)$	100.0	459.5
$(-5, 10, 5, 1, 1e-06)$	100.0	463.1

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(a, b, n, m, l)	s (%)	t (ms)
$(-10, 5, 5, 1, 1e-06)$	100.0	231.0
$(-5, 5, 5, 1, 1e-06)$	100.0	254.0
$(-5, 10, 5, 1, 1e-06)$	100.0	287.5
$(-5, 10, 10, 1, 1e-06)$	100.0	289.1
$(-5, 5, 5, 1, 1e-08)$	100.0	381.7

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, n, m, l)	s (%)	t (ms)
$(-5, 10, 5, 1, 1e-06)$	100.0	328.1
$(-10, 10, 5, 1, 1e-06)$	100.0	364.8
$(-5, 5, 5, 1, 1e-06)$	100.0	371.5
$(-10, 5, 5, 1, 1e-06)$	100.0	378.8
$(-5, 5, 10, 1, 1e-06)$	100.0	529.3

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
$(a, b, n, m, l, \beta_{HBM})$	s (%)	t (ms)
$(-10, 10, 5, 1, 1e-06, 10.0)$	100.0	137.3
$(-5, 10, 5, 1, 1e-06, 10.0)$	100.0	155.6
$(-5, 5, 5, 1, 1e-06, 10.0)$	100.0	170.2
$(-10, 10, 5, 1, 1e-06, 5.0)$	100.0	185.4
$(-5, 5, 5, 1, 1e-06, 5.0)$	100.0	200.0

(d) Top 5 permutations with HeavyBallMethod

Figure 14: Best parameter permutations for different main methods using Uniform-Search for optimizing the NegativeEntropy function with 10 sample points.

NewtonsMethod		
(a, b, n, m, l)	s (%)	t (ms)
(0, 5, 5, 1, 1e-05)	100.0	1339.2
(-5, 5, 5, 1, 1e-05)	100.0	2047.3
(-5, 10, 5, 1, 1e-05)	100.0	2059.8
(0, 5, 5, 1, 1e-06)	100.0	2068.5
(0, 5, 10, 1, 1e-06)	100.0	2125.3

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(a, b, n, m, l)	s (%)	t (ms)
(0, 5, 10, 1, 1e-05)	100.0	139.2
(0, 5, 10, 1.5, 1e-06)	100.0	159.3
(0, 5, 10, 1.5, 1e-05)	100.0	160.6
(0, 5, 10, 1, 1e-06)	100.0	161.2
(0, 5, 10, 2, 1e-06)	100.0	169.0

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, n, m, l)	s (%)	t (ms)
(0, 5, 10, 1, 1e-05)	100.0	6718.5
(0, 5, 5, 1, 1e-05)	100.0	7035.3
(-5, 5, 10, 1, 1e-05)	100.0	7444.3
(-5, 5, 5, 1, 1e-05)	100.0	7807.4
(0, 5, 5, 1.5, 1e-05)	100.0	8299.8

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
$(a, b, n, m, l, \beta_{HBM})$	s (%)	t (ms)
(0, 5, 10, 1, 1e-05, 0.1)	100.0	138.5
(0, 5, 5, 1.5, 1e-05, 0.1)	100.0	154.5
(0, 5, 5, 1, 1e-05, 0.1)	100.0	166.8
(0, 5, 10, 1, 1e-06, 0.1)	100.0	177.9
(0, 5, 10, 1.5, 1e-05, 0.1)	100.0	178.0

(d) Top 5 permutations with HeavyBallMethod

Newton's Search

Figure 15: Best parameter permutations for different main methods using NewtonsSearch for optimizing the MatrixSquareSum function with 10 sample points.

NewtonsMethod		
(λ, l)	s (%)	t (ms)
(1, 1e-07)	100.0	411.3
(10, 1e-08)	100.0	435.8
(1, 1e-08)	100.0	445.1
(5, 1e-08)	100.0	447.9
(5, 1e-07)	100.0	449.2

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(λ, l)	s (%)	t (ms)
(1, 1e-08)	100.0	412.0
(5, 1e-07)	100.0	415.6
(10, 1e-07)	100.0	430.2
(5, 1e-08)	100.0	460.3
(10, 1e-08)	100.0	470.8

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(λ, l)	s (%)	t (ms)
(1, 1e-07)	100.0	502.4
(0.5, 1e-07)	100.0	537.2
(10, 1e-08)	100.0	554.9
(5, 1e-07)	100.0	569.9
(1, 1e-08)	100.0	592.0

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
$(\lambda, l, \beta_{HBM})$	s (%)	t (ms)
(10, 1e-07, 10.0)	100.0	338.8
(5, 1e-07, 10.0)	100.0	360.2
(10, 1e-08, 5.0)	100.0	361.5
(0.5, 1e-07, 10.0)	100.0	361.9
(1, 1e-08, 10.0)	100.0	382.9

(d) Top 5 permutations with HeavyBallMethod

Figure 16: Best parameter permutations for different main methods using NewtonsSearch for optimizing the NegativeEntropy function with 10 sample points.

NewtonsMethod		
(λ, l)	s (%)	t (ms)
(0.5, 1e-08, 100)	100.0	1659.8
(1, 1e-08, 100)	100.0	1662.6
(1, 1e-07, 100)	100.0	1671.8
(5, 1e-07, 100)	100.0	1707.0
(0.5, 1e-07, 100)	100.0	1723.9

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(λ, l)	s (%)	t (ms)
(0.5, 1e-07, 100)	100.0	476.2
(0.5, 1e-08, 100)	100.0	501.1
(1, 1e-08, 100)	100.0	501.8
(0.1, 1e-08, 100)	100.0	520.6
(5, 1e-07, 100)	100.0	521.0

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(λ, l)	s (%)	t (ms)
(0.1, 1e-08, 100)	100.0	2779.5
(0.1, 1e-07, 100)	100.0	2789.7
(0.5, 1e-07, 100)	100.0	2921.1
(0.5, 1e-08, 100)	100.0	2985.0
(1, 1e-07, 100)	100.0	3050.7

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
$(\lambda, l, \beta_{HBM})$	s (%)	t (ms)
(1, 1e-07, 100, 0.1)	100.0	586.3
(0.1, 1e-08, 100, 0.1)	100.0	591.6
(1, 1e-08, 100, 0.1)	100.0	602.4
(0.5, 1e-07, 100, 0.1)	100.0	609.2
(0.1, 1e-07, 100, 0.1)	100.0	614.9

(d) Top 5 permutations with HeavyBallMethod

Armijo's Search

Figure 17: Best parameter permutations for different main methods using ArmijoSearch for optimizing the MatrixSquareSum function with 10 sample points.

NewtonsMethod		
(λ, α, β)	s (%)	t (ms)
(1, 0.1, 0.5)	100.0	429.7
(1, 0.25, 0.75)	100.0	484.5
(1, 0.25, 0.5)	100.0	493.8
(1, 0.25, 0.9)	100.0	493.9
(1, 0.1, 0.9)	100.0	498.7

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(λ, α, β)	s (%)	t (ms)
(1.1, 0.25, 0.5)	100.0	143.0
(1.1, 0.5, 0.5)	100.0	211.3
(1, 0.25, 0.5)	100.0	215.6
(0.9, 0.25, 0.5)	100.0	216.4
(0.9, 0.5, 0.5)	100.0	236.4

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(λ, α, β)	s (%)	t (ms)
(1.1, 0.25, 0.5)	100.0	262.6
(1, 0.25, 0.5)	100.0	339.0
(1.1, 0.1, 0.5)	100.0	353.9
(1, 0.25, 0.75)	100.0	361.5
(0.9, 0.25, 0.5)	100.0	376.6

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
$(\lambda, \alpha, \beta, \beta_{HBM})$	s (%)	t (ms)
(1.1, 0.25, 0.5, 0.1)	100.0	142.5
(0.9, 0.25, 0.5, 10.0)	100.0	151.8
(1, 0.25, 0.5, 10.0)	100.0	165.7
(1, 0.25, 0.5, 0.5)	100.0	170.4
(1.1, 0.5, 0.75, 10.0)	100.0	173.8

(d) Top 5 permutations with HeavyBallMethod

Figure 18: Best parameter permutations for different main methods using ArmijoSearch for optimizing the NegativeEntropy function with 10 sample points.

NewtonsMethod		
(λ, α, β)	s (%)	t (ms)
(1.1, 0.5, 0.9)	100.0	2094.1
(1, 0.5, 0.75)	100.0	2457.5
(1, 0.5, 0.5)	100.0	2659.1
(1, 0.5, 0.9)	100.0	2756.4
(1.1, 0.25, 0.75)	100.0	2854.2

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(λ, α, β)	s (%)	t (ms)
(1.1, 0.5, 0.75)	100.0	11.2
(1.1, 0.5, 0.9)	100.0	12.6
(0.9, 0.5, 0.9)	100.0	12.7
(1.1, 0.25, 0.75)	100.0	13.9
(1.1, 0.25, 0.5)	100.0	15.4

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(λ, α, β)	s (%)	t (ms)
(0.9, 0.5, 0.5)	100.0	221.7
(1.1, 0.5, 0.5)	100.0	242.8
(0.9, 0.5, 0.75)	100.0	252.2
(1.1, 0.5, 0.75)	100.0	255.0
(1, 0.5, 0.5)	100.0	262.5

(c) Top 5 permutations with ConjugateGradientMethod

HeavyBallMethod		
$(\lambda, \alpha, \beta, \beta_{HBM})$	s (%)	t (ms)
(1, 0.5, 0.5, 0.1)	100.0	11.4
(1.1, 0.25, 0.5, 0.1)	100.0	14.1
(0.9, 0.5, 0.75, 0.1)	100.0	14.5
(0.9, 0.25, 0.5, 0.1)	100.0	14.5
(1, 0.5, 0.9, 0.1)	100.0	15.8

(d) Top 5 permutations with HeavyBallMethod