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 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

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

$$\min\nolimits_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}), \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is called an *objective function* (or a *target function*), and we are looking for a *solution* vector $\mathbf{x}^* \in \mathbb{R}^n$ that minimizes the value of f such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{R}^n$. There are several different optimization algorithms (or methods) that can be used to solve the problem (1), e.g., Newton's and conjugate gradient methods [2]. 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 in developing and benchmarking optimization algorithms. [1]

In general, optimization algorithms are iterative processes that harness the processing power of computers efficiently by exploiting the properties of the target function f in order for it 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 $f(\mathbf{x}')$ at a given point $\mathbf{x}' \in \mathbb{R}$, and a new direction $\mathbf{d} \in \mathbb{R}^n$, the line search function takes the form $f(\mathbf{x}' + \lambda\mathbf{d})$ 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

$$\operatorname*{argmin}_{\lambda\in\mathbb{R}} f(\mathbf{x}' + \lambda\mathbf{d}) \tag{2}$$

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 optimization methods investigated are mostly based on the algorithms represented in the Nonlinear Optimization course that was lectured in Fall 2018 and 2019 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 on a general level. In addition, we will analyze the performance differences caused by the different line search methods with the goal of finding metrics that give us the best overall view of the algorithm's performance.

1.1 Organization of this thesis

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 affecting the performance of an optimization algorithm is the target function of the problem we are solving. In this thesis we are going to examine two different problems with convex and differentiable target functions $f : \mathbb{R}^n \to \mathbb{R}$ of dimension $n = 50$.

2.1.2 Step size function

Each optimization method involves computing a step at each iteration which is an optimization problem of the form (2). In practice, at iteration k we are at a point $\mathbf{x}^k$ and have computed the new direction vector $\mathbf{d}^k$. To simplify notation, we can define a step size function $g_{\mathbf{x}^k,\mathbf{d}^k}(\lambda) = f(\mathbf{x}^k + \lambda \mathbf{d}^k)$ for each of the target functions. Then at each iteration k of the main optimization method, a new step size function is generated with the updated point $\mathbf{x}^k$ and direction vector $\mathbf{d}^k$.

2.1.3 Matrix Square Sum

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

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

To make Sum of Squares problem more difficult to solve, 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

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

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 be a PD square matrix to ensure that the function is strictly convex and has a unique global minimum. The matrix A is formed in a 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 transform A into a PD-matrix by $A = 0.5(A' + A'^{\top})$. Finally, if A is still not PD, it is modified by the formula

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

where λ_{min} is the minimum eigenvalue of A and d is a constant. The value of d also determines A's condition number defined as $\frac{\lambda_{max}}{\lambda_{min}}$ [5]. Smaller values of d make the function more elliptic and harder for gradient methods to solve, while larger values produce higher condition numbers and more circular gradient curves, which makes the problem easier for certain methods like Gradient Descent [6]. The value of d used in the thesis is 5, resulting in condition numbers of around 6.5. The values of $b = (b_1, \dots, b_n) \in \mathbb{R}^n$ and $c \in \mathbb{R}$ are also randomly generated so that $b_i \in [-0.5, 0.5]$ and $c \in [-0.5, 0.5]$.

Some of the optimization methods need to compute the function's gradient and Hessian matrix at every iteration. Also, some step size algorithms need the first and second derivatives of the step size function. In this case, we can compute the gradient and the Hessian of both function in advance. We should also derive the formula for calculating the correct optima so we can use it to determine whether our algorithms produce the correct solutions. Let us begin by expanding the function into a 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 us calculate the gradient $\nabla f(\mathbf{x})$ and the Hessian matrix $\mathbf{H}$:

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

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

From the gradient equation (6) we get the necessary optimality condition

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

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

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

Let us finally form the step size function and its derivatives:

$$\begin{aligned} g(\lambda) &= f(\mathbf{x} + \lambda\mathbf{d}) = \|A(\mathbf{x} + \lambda\mathbf{d}) + \mathbf{b}\|^2 + c\|(\mathbf{x} + \lambda\mathbf{d})\|^2 \\ &= \lambda\mathbf{d}^\top A^\top (2A\mathbf{x} + \lambda A\mathbf{d} + 2\mathbf{b}) + \mathbf{x}^\top A^\top (A\mathbf{x} + 2\mathbf{b}) + \mathbf{b}^\top \mathbf{b} \qquad (10) \\ &\quad + c(\mathbf{x}^\top \mathbf{x} + 2\lambda\mathbf{d}^\top \mathbf{x} + \lambda^2 \mathbf{d}^\top \mathbf{d}) \\ g'(\lambda) &= \mathbf{d}^\top A^\top (2A\mathbf{x} + 2\lambda A\mathbf{d} + 2\mathbf{b}) + 2c\mathbf{d}^\top (\mathbf{x} + \lambda\mathbf{d}) \qquad (11) \\ g''(\lambda) &= 2\mathbf{d}^\top (A^\top A\mathbf{d} + c\mathbf{d}) \qquad (12) \end{aligned}$$

2.1.4 Negative Entropy

The second target function examined in this thesis is an inverted entropy maximization problem, which converts it into a minimization problem [3]. We will call this convex minimization problem a Negative Entropy problem and its target function is of the form:

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

where $\mathbf{x} \in \mathbb{R}^n_{>0}$. Since the domain is now limited, we have to make some adjustments to our algorithms, which are described at the end of this section.

As the function (13) is convex, its global minimum can be found by calculating the zero point of the gradient:

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

$$= \log x_i + 1 = 0. \tag{15}$$

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 $i = 1 \ldots n$.

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 with 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:

$$\mathbf{H} = \begin{bmatrix} x_1^{-1} & 0 & 0 & \ldots & 0 \\ 0 & x_2^{-1} & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & x_n^{-1} \end{bmatrix}. \tag{16}$$

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

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

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

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

Because the domain of the function (13) has only positive numbers, we have to make some changes to our starting points, parameters and algorithms. Therefore, the iteration points must have only positive coordinates. To enforce this, a domain limiter (see algorithm 1) was added to the line search algorithms to prevent the iteration points becoming 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 the line search methods before evaluating $f(\mathbf{x} + \lambda\mathbf{d})$ 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 in the while-loop than others.

2.2 Optimization Methods

In this thesis, all nonlinear unconstrained optimization methods are called *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 is solved by the line search methods introduced in the section 2.3. [2]

2.2.1 Newton's Method

Newton's method (see algorithm 2) is a more general version of its line search (or univariate) version (see 2.3.7), utilizing the gradient and hessian matrix of the function instead of its first and second single derivatives. Algorithm 2 terminates if (i) the euclidean norm of the gradient $||\nabla f(x)||$ is less than some small tolerance $l > 0$, or (ii) the max iteration count $k_{\max}$ is exceeded.

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 $\|\nabla f(\mathbf{x})\| > l$ and $k < k_{max}$ do
3:    $\mathbf{d} = -H(\mathbf{x})^{-1}\nabla f(\mathbf{x})$
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 (algorithm 3) can be interpreted as a first order Newton's method, as it uses only the gradient to determine the step direction. The algorithm is otherwise exactly the same as Newton's method except for the computation of the descent direction d in line 3.

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 $\|\nabla f(\mathbf{x})\| > l$ and $k < k_{max}$ do
3:    $\mathbf{d} = -\nabla f(\mathbf{x})$
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 (algorithm 4) uses information from the previous and current iterations to compute the next iteration point. To compute a good direction, it tries to obtain an approximation of the inverse Hessian using a number of successive gradient ratio calculations. The method should theoretically require less steps than gradient descent method. [6]

The algorithm involves a second loop which runs for n times per every iteration inside the main loop, where n is equal to the dimension of the problem (i.e., $n = \dim(\mathbf{x})$). Each of these inner iterations is counted as an iteration of the algorithm within the performance tests in this thesis.

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

2.2.4 Heavy Ball Method

Heavy ball method (algorithm 5) is basically an extension of the Gradient descent with an extra term in the direction vector calculation step. The name of the method comes from an analogue to the physical representation of a heavy sphere in a potential field [7]. The Heavy Ball method is similar to the Gradient descent but it has an additional term with a multiplier $\beta \in [0, 1)$ in the descent direction d in line 3 [4].

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 $\|\nabla f(\mathbf{x})\| > l$ and $k < k_{max}$ do
3:     $\mathbf{d} = -\nabla f(\mathbf{x}) + \beta(\mathbf{x} - \mathbf{x_{prev}})$
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 $\lambda$
```

Table 2: Parameters tested for heavy ball method.

Parameter	MatrixSquareSum	NegativeEntropy
β_{HBM}	0.1, 0.5, 1.0, 5.0, 10.0	0.1, 0.5, 1.0

As seen in the parameters table (table 2), also values outside of the regular range of $\beta \in [0, 1)$ were tested as some of them performed better in our performance tests than the more common values.

2.3 Line Search Methods

Most line search methods are based on a similar idea: choose a range of values and reduce it based on function evaluations until some stopping condition is satisfied. There are also approximate approaches like backtracking search, while some line search methods exploit 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 comparison with 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 the constant step size λ is considered as one of the search methods to provide a more comprehensive comparison.

Table 3: Parameters tested for constant step size.

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

Because constant step size causes some convergence issues with some of the configurations with the test values listed in table 3, 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: $(0.1, 0.25, 0.5, 0.9)$ and $(0.0001, 0.1, 0.15, 0.2)$, respectively.

2.3.2 Golden Section Search

Golden section search (algorithm 6) is a sequentical line search meaning that it utilizes the previous iterations when calculating the next value for the step size. The name of this method comes from its unique reduction factor of $\phi = 0.618$. [2]

The parameters tested for golden section search are displayed in table 4.

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 4: Parameters tested for golden section search.

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

Note that the version of the algorithm that was used for the performance testing performed two function calls per iteration even though only one is really necessary. This caused the function call number to be almost twice larger than the actual optimum for golden section search. However, this had little to no impact on the performance metrics we used i.e. the overall duration of the optimization method.

2.3.3 Bisection Search

Bisection search (algorithm 7) is a line search that uses derivatives to progress. This requires the target functions to be pseudoconvex and therefore differentiable within some closed bounds. [2].

The parameters tested for bisection search are displayed in table 5.

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 5: Parameters tested for bisection search.

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

Similarily to golden section search, dichotomous search (algorithm 8) is a sequentical 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 6: Parameters tested for dichotomous search

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 6 when using Newton's method for the 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 (algorithm 9) 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 seach instead calculates a new factor for each iteration using Fibonacci numbers. [2]

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 does not effect performance significantly and so for our implementation we decided to generate the Fibonacci numbers during run-time using a single loop algorithm of time complexity $O(n)$.

The parameters tested for Fibonacci search are displayed in table 7.

Table 7: Parameters tested for Fibonacci search.

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 (algorithm 10) divides a bounded section of a function into intervals, then chooses the interval with the lowest value and repeats the process. To increase the precision of the method we may also increase the number of intervals to divide the new interval into. [2]

More specfically, 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.

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 8: Parameters tested for uniform search

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

The parameters tested for uniform search are displayed in table 8.

2.3.7 Newton's Search

Newton's search (algorithm 2) 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 9: Parameters tested for Newton's search

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 9, 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 (algorithm 12) is the only backtracking method of the algorithms included in this thesis. In this context, backtracking is performed by reducing the initial step size on every iteration until it satisfies the stopping condition. [3]

The parameters tested for Armijo's search are displayed in table 10.

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 10: Parameters tested for Armijo's search

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 Problem Parameter Settings

In this section we are discussing the choice and effect of different parameters such as the starting points and the actual algorithm's parameters.

3.1 Parameter Selection

Optimization method's parameter selection is a whole topic of its own and not our primary focus in this thesis. Therefore a straightforward approach for picking the parameters is justified.

Using a similar setup as when comparing the performances of the methods, we can also compare the outcomes with different parameters. Using the manually chosen parameter values presented in the theory section, we can programmatically generate all the permutations for each combination of a parameter, a line search method, the main method, and a target function.

Displaying all the results for each parameter combination would require too much space, and therefore just a concise overview of the best parameters for each setup is found from the appendix A.

3.2 Starting Point Selection

To get reliable results with the different parameter and method combinations, we need to also take into account the starting points used.

For parameter selection, we are going to use just ten starting points since the number of parameter options already increases the required iteration count by a lot. For the actual performance tests, we are going to use 1000 starting points generated similarily.

We also need to decide some range from which the points are generated. Thus, let a single starting point be $x = (x_1, x_2, \dots x_{50})$. For matrix square sum problem the points are generated so that for each point x applies $x_i \in [-10, 10] \ \forall \ i = 1 \dots 50$. Because negative entropy problem'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 want the starting points to be random but evenly distributed so that no two points are the same, or too close to each other. To generate a point distribution which is random but evenly packed with points, we use a method shown in algorithm 13.

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:     ok = true
5:     for $j = 1 \ldots q$ do
6:         if $\|\mathbf{x}_j - \mathbf{y}\| < d_{min}$ then
7:             ok = false
8:             break
9:         end if
10:    end for
11:    if ok then
12:        $q = q + 1, \ \ i = i + 1$
13:        $\mathbf{x}_q = \mathbf{y}$
14:    end if
15: end while
16: return $(x_1, x_2, ..., x_p)$
```

The algorithm generates p points of dimension n. The generated points are stored in variables $x_q, q = 1 \ldots p$. 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 it is, we discard the point and try again.

The algorithm used is not perfect 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 an infinite loop, and too low values do not provide the even distribution we are expecting. By trial and error the d_{min} values presented in table 11 are found.

Table 11: 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 generating the points successfully, they are saved into a file and used for every run with the specific points count, maintaining an equal level of randomness between measurements.

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 a 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.

4 Results

We are going to represent the results in a table with the average overall performances of each line search method for each permutation of the main methods and target functions. The performances are calculated using the best parameters found for each setup as described in section 3.1. Again for each starting point, a unique target function configuration is generated in the case of matrix square sum problem using the same rules described in section 2.1.3. The random seeds used are integers from 0 to 999 matching the starting point indices.

After running the performance tests, we are left with the results introduced in the following tables 12 to 19. The row's of the table show the line search method used, the success rate of the algorithm, average total duration, average main method iteration count, and function calls of the algorithm. The function call count includes all calls to the function, its gradients and step size functions. In addition to these, the last two columns show average duration and iterations used by the line search algorithm alone. We generate a table with all line search methods and their representative performances for each main method and target function combination.

Table 12: Average performances of Newton's method when minimizing matrix square sum 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}
Constant search	100.0	445.8	1.0	7.0	0.0	0.0
Golden section search	100.0	445.0	1.0	101.0	2.6	47.0
Bisection search	99.6	535.9	1.0	35.0	89.0	28.0
Dichotomous search	100.0	429.1	1.0	77.0	3.3	35.0
Fibonacci search	100.0	453.2	1.0	119.0	4.5	55.0
Uniform search	100.0	455.0	1.0	156.0	7.2	148.0
Newton's search	100.0	443.5	1.0	8.0	0.5	0.0
Armijo's search	100.0	424.2	1.0	10.0	0.9	0.0

Table 13: Average performances of gradient descent method when minimizing matrix square sum 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}
Constant search	98.9	4681.2	2942.0	8828.9	141.2	0.0
Golden section search	99.4	283.7	33.1	1784.1	155.4	841.0
Bisection search	99.4	823.5	33.0	696.6	749.8	594.5
Dichotomous search	99.4	270.1	33.1	1491.1	142.6	694.4
Fibonacci search	99.4	354.3	33.1	2615.3	238.6	1223.5
Uniform search	99.4	389.2	33.1	3508.2	282.3	3372.9
Newton's search	99.4	453.3	33.1	241.2	355.7	35.3
Armijo's search	99.6	209.3	25.8	343.2	97.9	185.5

Table 14: Average performances of conjugate gradient method when minimizing matrix square sum 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}
Constant search	99.6	989.3	300.1	1216.2	23.9	0.0
Golden section search	100.0	429.5	50.0	2722.9	198.9	1258.5
Bisection search	100.0	1154.9	50.0	1106.0	1003.4	900.0
Dichotomous search	100.0	426.3	50.0	2306.0	177.1	1050.0
Fibonacci search	100.0	517.8	50.0	4006.0	298.5	1850.0
Uniform search	100.0	558.2	50.0	5356.0	348.9	5100.0
Newton's search	100.0	552.8	50.6	364.0	347.5	35.0
Armijo's search	100.0	405.3	50.0	914.8	157.6	558.8

Table 15: Average performances of heavy ball method when minimizing matrix square sum 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}
Constant search	98.9	4711.5	2941.9	8828.8	134.5	0.0
Golden section search	99.4	227.7	24.3	1303.4	127.1	613.8
Bisection search	99.4	666.6	24.4	514.9	607.0	438.8
Dichotomous search	99.4	209.4	24.4	904.9	99.3	414.4
Fibonacci search	99.4	353.2	32.7	2585.0	239.5	1209.3
Uniform search	99.4	322.4	24.4	2587.4	232.0	2486.9
Newton's search	99.4	375.1	24.4	179.6	295.0	26.4
Armijo's search	99.6	217.3	25.8	343.1	99.6	185.4

Table 16: Average performances of Newton's method when minimizing negative entropy 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}
Constant search	100.0	3718.1	10.7	45.7	1.0	0.0
Golden section search	100.0	2038.3	5.6	426.9	29.4	200.7
Bisection search	100.0	2191.6	5.6	116.9	161.4	91.5
Dichotomous search	100.0	2231.2	6.0	265.5	52.0	119.2
Fibonacci search	100.0	2100.7	5.6	738.5	59.0	350.9
Uniform search	100.0	2316.0	6.1	510.3	132.5	476.7
Newton's search	100.0	2266.0	5.7	192.6	348.6	53.7
Armijo's search	100.0	2206.9	6.3	52.6	3.1	5.5

Table 17: Average performances of gradient descent method when minimizing negative entropy 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}
Constant search	100.0	13.1	16.3	51.8	0.8	0.0
Golden section search	100.0	53.4	8.6	400.1	47.9	185.7
Bisection search	100.0	92.2	8.6	158.4	87.0	129.7
Dichotomous search	100.0	54.7	8.6	289.3	48.9	130.2
Fibonacci search	100.0	58.4	8.6	610.8	52.6	282.4
Uniform search	100.0	121.8	10.9	889.6	114.6	842.8
Newton's search	100.0	458.8	8.8	687.6	453.8	216.5
Armijo's search	100.0	11.6	10.2	96.5	5.6	32.0

Table 18: Average performances of conjugate gradient method when minimizing negative entropy 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}
Constant search	99.9	4393.4	163.2	663.3	4248.0	0.0
Golden section search	100.0	539.7	100.0	4681.2	451.2	2136.6
Bisection search	100.0	759.5	100.0	1228.3	672.9	777.8
Dichotomous search	100.0	576.4	102.7	3496.1	485.5	1538.6
Fibonacci search	100.0	623.7	100.0	7155.1	534.0	3273.5
Uniform search	100.0	6371.1	100.0	8208.0	6277.0	7700.0
Newton's search	100.0	2801.7	100.0	4147.0	2716.9	1213.0
Armijo's search	100.0	198.4	99.8	1119.4	112.0	412.8

Table 19: Average performances of heavy ball method when minimizing negative entropy 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}
Constant search	100.0	8.3	14.0	45.0	0.3	0.0
Golden section search	100.0	92.4	17.3	808.4	80.5	376.8
Bisection search	100.0	131.7	11.8	223.7	124.6	185.4
Dichotomous search	100.0	68.8	11.8	486.5	60.9	224.1
Fibonacci search	100.0	72.1	11.8	843.4	64.3	390.8
Uniform search	100.0	134.9	12.9	1051.1	126.6	996.3
Newton's search	100.0	541.0	12.8	806.1	533.7	250.6
Armijo's search	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 concerning the choice of line search method, we want to display the data primarily per line search method and secondarily by main method or target function.

The primary metric for measuring algorithm's performance is the success rate of the algorithm. Since the success rates seem to have lots of ties, we also need a secondary metric, which we choose to be the average overall duration of the main method. While it is not universally comparable within different implementations or problems, it provides us a decent comparison of the line search methods within a specific scope of the main method and a target problem. This means that all the values are essentially comparable within the same column in the tables 20 to 23. To visualize the column-wise differences, we apply a per-column color scale in which darker red implies worse performance and darker green implies better performance.

Table 20: 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**
Constant search	445,80	4 681,20	989,30	4 711,50
Golden section search	445,00	283,70	429,50	227,70
Bisection search	535,90	823,50	1 154,90	666,60
Dichotomous search	429,10	270,10	426,30	209,40
Fibonacci search	453,20	354,30	517,80	353,20
Uniform search	455,00	389,20	558,20	322,40
Newton's search	443,50	453,30	552,80	375,10
Armijo's search	424,20	209,30	405,30	217,30

Table 21: 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**
Constant search	3 718,10	13,10	4 393,40	8,30
Golden section search	2 038,30	53,40	539,70	92,40
Bisection search	2 191,60	92,20	759,50	131,70
Dichotomous search	2 231,20	54,70	576,40	68,80
Fibonacci search	2 100,70	58,40	623,70	72,10
Uniform search	2 316,00	121,80	6 371,10	134,90
Newton's search	2 266,00	458,80	2 801,70	541,00
Armijo's search	2 206,90	11,60	198,40	15,50

Table 22: The average of the durations for finding the minimum for both Matrix Square Sum and Negative Entropy Functions. Results are listed separately for each main method and colored per column so that darker red is worse (slower) and darker green is better (faster).

Line Search Method	**NM**	**GDM**	**CGM**	**HBM**
Constant search	2 081,95	2 347,15	2 691,35	2 359,90
Golden section search	1 241,65	168,55	484,60	160,05
Bisection search	1 363,75	457,85	957,20	399,15
Dichotomous search	1 330,15	162,40	501,35	139,10
Fibonacci search	1 276,95	206,35	570,75	212,65
Uniform search	1 385,50	255,50	3 464,65	228,65
Newton's search	1 354,75	456,05	1 677,25	458,05
Armijo's search	1 315,55	110,45	301,85	116,40

Table 23: 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**
Constant search	2 706,95	2 033,23	2 370,09
Golden section search	346,48	680,95	513,71
Bisection search	795,23	793,75	794,49
Dichotomous search	333,73	732,78	533,25
Fibonacci search	419,63	713,73	566,68
Uniform search	431,20	2 235,95	1 333,58
Newton's search	456,18	1 516,88	986,53
Armijo's search	314,03	608,10	461,06

While the success rate is the primary metric for comparison, it does not offer as interesting differences as the average overall duration. However, it is an essential piece of information to be addressed along with the duration comparison to understand the relations of different line search methods better. The following tables 24 and 25 display the success rates with a similar column-wise coloring but this time with only the below-average cells colored as red.

Table 24: 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**
Constant search	100,00	98,90	99,60	98,90
Golden section search	100,00	99,40	100,00	99,40
Bisection search	99,60	99,40	100,00	99,40
Dichotomous search	100,00	99,40	100,00	99,40
Fibonacci search	100,00	99,40	100,00	99,40
Uniform search	100,00	99,40	100,00	99,40
Newton's search	100,00	99,40	100,00	99,40
Armijo's search	100,00	99,60	100,00	99,60

Table 25: 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**
Constant search	100,00	100,00	99,90	100,00
Golden section search	100,00	100,00	100,00	100,00
Bisection search	100,00	100,00	100,00	100,00
Dichotomous search	100,00	100,00	100,00	100,00
Fibonacci search	100,00	100,00	100,00	100,00
Uniform search	100,00	100,00	100,00	100,00
Newton's search	100,00	100,00	100,00	100,00
Armijo's search	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 algorithms' performance.

The durations in the box plots are scaled logarithmically to improve the readability of the figures in scenarios where some methods perform a lot differently from others. The box plot displays half of the datapoints within the main box area. The whiskers above and below the box are configured so that together with the main box body they cover 95% of the datapoints. The orange line inside the boxes shows the exact mean of the datapoints.

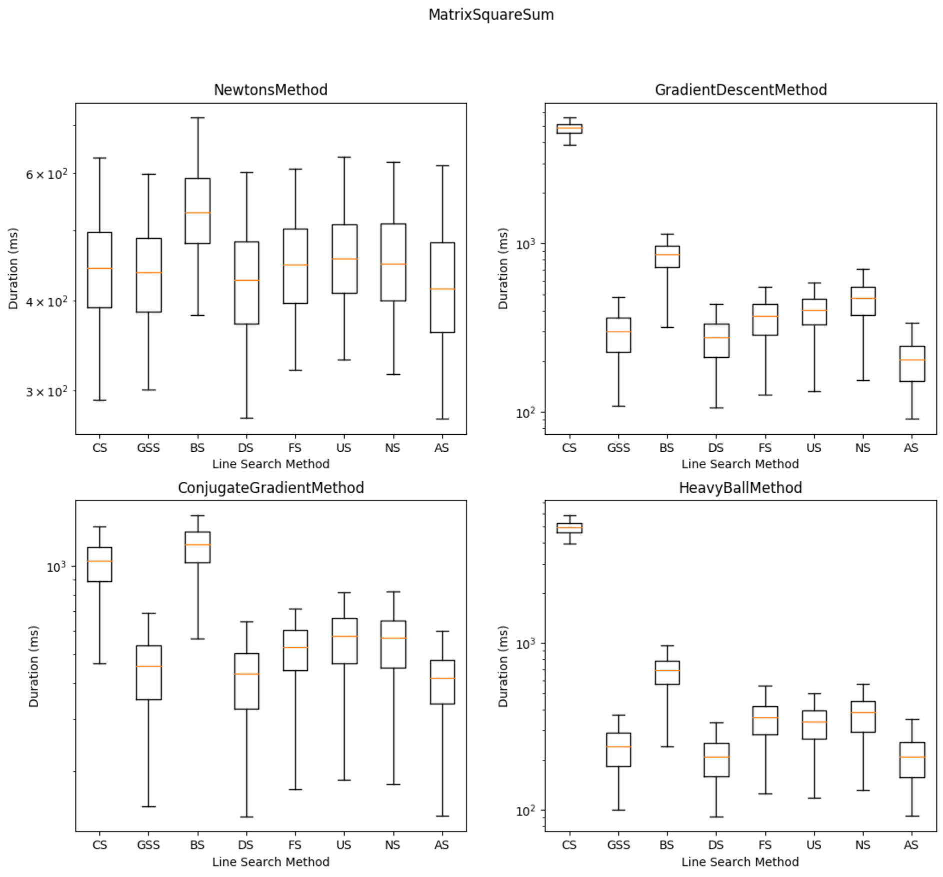

Figure 1: Box plot of solution times per line search and main method for MatrixSquareSum target function.

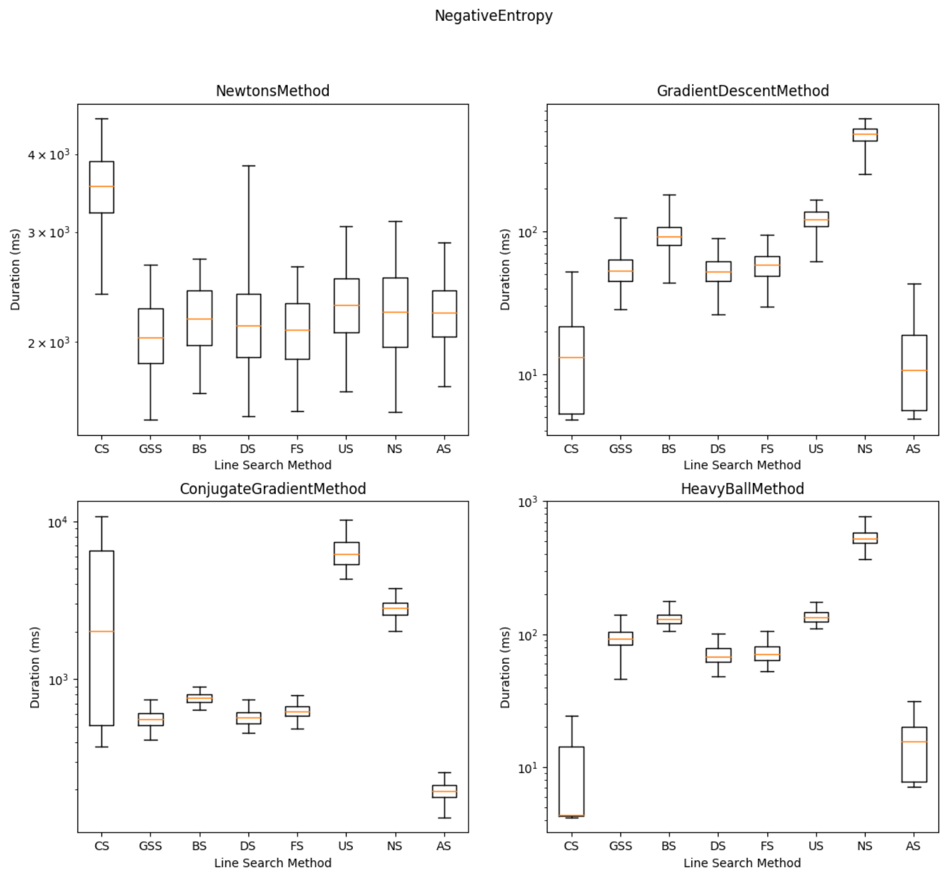

Figure 2: Box plot of solution times per line search and main method for NegativeEntropy target function.

Constant Search

Constant search always gives a constant step size and therefore provides an excellent baseline to compare the other line search methods with. The execution of constant search costs zero time, so the burden of optimization remains on the main method.

Looking at the overall durations of constant search in table 23 and the success rates in tables 24 and 25, we can see that the constant search is generally the worst-performing method as expected. The only scenarios where constant step size proves to be rather effective are when optimizing the negative entropy function using gradient descent or heavy ball methods. In those two cases, the constant search was the best or second-best performer. With NE problem gradient descent based methods seem to perform well in general, and since the Constant Search does not require any time to execute, it gives high overall performances as well. We can confirm this conclusion by looking at the overall and line search duration columns in tables 17 and 19, where most of the overall duration consists of the line search duration alone.

Looking at the box plots in figures 1 and 2 constant step size is again the worst performer by not just absolute durations but also by the high variance of results. The only two exceptions to this are the previously mentioned cases of NE+GD and NE+HBM, where constant search scores one of the lowest variances.

Golden Section Search

Comparing the overall results in tables 22 and 23, we immediately notice from the green-colored cells that golden section search is one of the best performing line searches. The success rates of this method are great, and it does, like most methods tested, find the correct solution in almost all of the test cases. The variances of golden section search's durations are approximately on the same level as the other line searches as can be seen from box plots.

In addition, as stated in the section 2.3.2, an unoptimal implementation of golden section search was used, so the function call count could be improved by a lot and depending on the target problem, small improvements on the overall performance are also possible.

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 points to others in the single case of MSS+NM. The overall performance is especially weak with MSS problem and barely average with other setups, causing it to land around the middle of the scale of line search methods in the overall score table. The box plots also show that the Bisection Method does have higher variances in performance than other methods, especially with the MSS problem.

Dichotomous Search

In our first tests, dichotomous search was one of the slower methods due to not reaching the correct solutions within the given max iteration limit. However, lowering the main method tolerance proved to be especially helpful for this method, which ultimately turned out as one of the stronger methods. It even outperforms golden section search in MSS optimization and scores almost equally in overall scores. The success rates are also identical and reviewing variance from box plots shows no significant differences between the two methods sharing the second-best place in our comparison.

Fibonacci Search

Fibonacci search scores above-average in overall results and much higher with the NE problem when compared to other line searches. The success rates are in order and variances are at the same level as dichotomous and golden section searches'.

Our implementation of Fibonacci search computes each Fibonacci number during runtime without memorizing them. The solution proved not to be an issue since the highest Fibonacci number reached was around F_{50}. A quick test with precomputed Fibonacci numbers showed at most 10% improvement in performance when using memorization. For real-life applications, precomputation should probably be added if possible.

Uniform Search

Uniform search is expected to be slow since it is pretty straight forward and inefficient algorithm. The inefficiency can also be seen in the results as the algorithm scores averagely on MSS problem and below average in NE. Uniform search also struggles to stay within the 10000 max iteration limit with NE problem when using conjugate gradient method. This is possibly due to the way we implemented the positive domain using the domain limiter (algorithm 1). Uniform search scores second-lowest in overall performance with average success rates and above-average variances.

Newton's Search

Similarily to uniform search, Newton's search has average performance when optimizing MSS problem but quickly falls behind in the case of NE. Newton's search also seems to hit the max iterations limit with NE, but the method ends up still providing good enough step sizes so that success rates stay high. It seems that the method does not fully converge when combined with the domain limiter, which alters the step size given by the derivatives. Because of this, even though it does not make sense in real life implementation, a way smaller max iteration limit of just 100 or 1000 iterations proved to be much more effective for the NE problem. Because of this, the results of Newton's search for NE are more unreliable than in the case of the MSS problem. The same effect is visible in the box plots as well: Newton's search's durations have an extremely high variance when optimizing the Negative Entropy function.

Armijo's Search

Armijo's search proves to be the best overall 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 MSS+GD and MSS+HBM test cases. Armijo's search has a slightly higher variance in some scenarios than other line searches, but still easily outperforms other methods most of the time. Especially for NE problem, the Armijo's search has lower variances than any other method.

4.2 Assumptions and Further Performance Improvements

The assumptions and choices made during the research are discussed here briefly. One major factor affecting the results is that there are only two types of different optimization problems considered. Even though the randomization of parameters adds some variety in Matrix Square Sum problem, both of the target functions remain relatively simple. Both of the problems are also convex and have a single global maximum, making the solving easier than if there were multiple local maxima present. The way MSS's parameters are randomized is also a factor to consider. The methods for randomization of the parameters and building a positive-definite matrix A described in section 2.1.3 only yield a limited variety of functions. In the case of Negative Entropy, a domain limiter is used to ensure that values stay within the correct domain, which potentially causes issues with the line searches. Few line searches even struggled to converge at all in some scenarios.

Another factor to consider is the starting points used. While, in theory, we could raise the point count indefinitely, the solution for evenly distributing the points should provide decent results even with lower point counts. However, the range from which we generate the points could have been set higher than ten.

Choosing the optimal parameter values is also a significant factor in performance comparison. Since the problem of choosing the absolute best parameters is complex in itself, it is justified to pick a set of options and then compare the performance of each permutation. However, by implementing a better parameter selection, such as increasing the number of options, test cases, and iterations used, we can likely reach better results for each optimization method. The tradeoff would naturally be a higher computation time required to find the optimal parameters.

Some factors in performance are difficult to review accurately. These include the actual implementation of the methods which varies in different sources. One difference, in particular, seems to be the position of the stopping condition within the main methods. Some sources suggest that it is better to check the condition before updating the point. However, in our implementation, the check is done just after the update. [3] [2]

Another possible optimization 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, we get the pre-evaluated value from our dictionary. [3]

Factors like the choice of programming language and CPU or GPU performance may affect the optimal results. We performed the tests in this thesis with Python 3.7.3 using a multiprocessing library and an Intel i5-8500K CPU. Different implementations and programming languages may, for example, optimize matrix calculations differently causing potentially unexpected differences within otherwise identical scenarios. For this reason, if the goal is to optimize the optimization process completely, it is suggested to run the performance testing separately for each test setup used.

5 Conclusion and Future Research

The choice of line search method seems to have a varying size of an impact on the performance of an optimization method. Comparing just the average overall durations for different line searches, it seems that the worst line search is several times slower than the best one. The difference in performance for the best and second-best method is around 11 %, and the third one is around 16 % slower than the fastest one. These numbers are not final by any means but instead, give an idea of the range of improvements we can look for when considering the choice of a line search algorithm.

If we were to perform further testing with the same goals, we would select a higher number and variety of target functions and a wider range of starting points. However, there is even more constraining issue we would need to tackle first: the optimal parameters are likely to change with the setups, forcing us to re-run the parameter comparisons and find new values for all the parameters. Since this requires high computation times, we should implement a better method for parameter selection before considering any new test scenarios. A simple solution could be to decide a well-reasoned set of options for each parameter based more strictly on existing literature around the topics.

Is it worth it to optimize the choice of line search method as an additional parameter for an unconstrained nonlinear optimization method? If one is already using a high-performance line search method such as Armijo's search, there is at most a minimal chance of gaining any benefit by performing a line search comparison.

However, we do believe that the subject could use further research to make the results more comprehensive and to build a framework for comparing and scoring line search methods in different scenarios.

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] E. Ghadimi, H. R. Feyzmahdavian, and M. Johansson. Global convergence of the heavy-ball method for convex optimization, 2014.

[5] D. B. Lloyd N. Trefethen. *Numerical Linear Algebra.* SIAM, 1997.

[6] A. E. Niclas Andréasson and M. Patriksson. *An Introduction to Continuous Optimization: Foundations and Fundamental Algorithms.* Studentlitteratur AB, 2005.

[7] B. T. Polyak. *Some methods of speeding up the convergence of iteration methods.* USSR Computational Mathematics and Mathematical Physics, 1964.

A Appendix

Constant Search

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

NewtonsMethod		
(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 NewtonsMethod

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 GoldenSectionSearch 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.

NewtonsMethod		
(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 NewtonsMethod

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		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 10, 1e-10, 1e-09)	100.0	388.3
(-10, 10, 1e-10, 1e-09)	100.0	436.4
(-5, 5, 1e-10, 1e-09)	100.0	439.6
(-10, 5, 1e-10, 1e-09)	100.0	455.0
(-5, 10, 1e-07, 1e-05)	100.0	840.8

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(a, b, ϵ, l)	s (%)	t (ms)
(-10, 5, 1e-07, 1e-05)	100.0	234.8
(-10, 5, 1e-08, 0.0001)	100.0	241.2
(-5, 5, 1e-07, 0.0001)	100.0	252.5
(-5, 10, 1e-08, 0.0001)	100.0	252.9
(-10, 5, 1e-08, 1e-05)	100.0	260.4

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 10, 1e-07, 1e-05)	100.0	354.2
(-5, 10, 1e-07, 0.0001)	100.0	382.0
(-5, 5, 1e-07, 0.0001)	100.0	397.3
(-10, 10, 1e-08, 1e-05)	100.0	406.2
(-10, 5, 1e-08, 0.0001)	100.0	406.7

(c) Top 5 permutations with ConjugateGradientMethod

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

(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		
(a, b, ϵ, l)	s (%)	t (ms)
(-10, 10, 1e-07, 1e-05)	100.0	1756.0
(-10, 5, 1e-06, 1e-05)	100.0	1846.8
(-10, 10, 1e-06, 0.0001)	100.0	1882.9
(-10, 5, 1e-06, 0.0001)	100.0	1893.5
(-10, 10, 1e-06, 1e-05)	100.0	1923.0

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 5, 1e-07, 0.0001)	100.0	54.3
(-5, 10, 1e-06, 0.0001)	100.0	61.7
(-5, 10, 1e-06, 1e-05)	100.0	63.8
(-5, 5, 1e-07, 1e-05)	100.0	64.6
(-5, 10, 1e-07, 0.0001)	100.0	65.4

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 5, 1e-06, 0.0001)	100.0	640.6
(-10, 5, 1e-06, 0.0001)	100.0	642.7
(-5, 5, 1e-07, 0.0001)	100.0	645.3
(-5, 10, 1e-07, 0.0001)	100.0	657.0
(-5, 5, 1e-06, 1e-05)	100.0	680.9

(c) Top 5 permutations with ConjugateGradientMethod

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

(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		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 5, 1e-10, 1e-10)	100.0	443.2
(-5, 5, 1e-12, 1e-10)	100.0	452.3
(-5, 5, 1e-08, 1e-10)	100.0	459.6
(-10, 5, 1e-10, 1e-10)	100.0	460.9
(-10, 5, 1e-12, 1e-10)	100.0	469.7

(a) Top 5 permutations with NewtonsMethod

GradientDescentMethod		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 10, 1e-12, 1e-06)	100.0	355.6
(-5, 10, 1e-10, 1e-06)	100.0	356.0
(-5, 5, 1e-12, 1e-06)	100.0	366.8
(-10, 10, 1e-12, 1e-06)	100.0	369.2
(-10, 5, 1e-12, 1e-06)	100.0	369.4

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 10, 1e-10, 1e-06)	100.0	473.8
(-10, 10, 1e-10, 1e-06)	100.0	485.0
(-5, 10, 1e-12, 1e-06)	100.0	490.3
(-10, 5, 1e-08, 1e-06)	100.0	508.6
(-10, 10, 1e-12, 1e-06)	100.0	516.0

(c) Top 5 permutations with ConjugateGradientMethod

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

(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.

NewtonsMethod		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 5, 1e-12, 1e-12)	100.0	1834.0
(-10, 5, 1e-08, 1e-06)	100.0	1908.9
(-10, 10, 1e-12, 1e-06)	100.0	1946.8
(-5, 10, 1e-10, 1e-06)	100.0	1957.9
(-10, 10, 1e-10, 1e-06)	100.0	1967.3

(a) Top 5 permutations with NewtonsMethod

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

(b) Top 5 permutations with GradientDescentMethod

ConjugateGradientMethod		
(a, b, ϵ, l)	s (%)	t (ms)
(-5, 5, 1e-08, 1e-06)	100.0	682.8
(-5, 5, 1e-12, 1e-06)	100.0	683.5
(-5, 5, 1e-10, 1e-06)	100.0	701.5
(-10, 5, 1e-10, 1e-06)	100.0	728.0
(-5, 10, 1e-08, 1e-06)	100.0	729.5

(c) Top 5 permutations with ConjugateGradientMethod

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

(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