Comparison of line search methods in unconstrained optimization

Einari Tuukkanen

School of Science

Bachelor's thesis

Espoo 31.8.2018

Supervisor

Asst. Prof. Fabricio Oliveira

Advisor

MSc Juho Andelmin

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the bachelor's thesis

Author Einari Tuukkanen		
Title Comparison of line search methods in unconstrained optimization		
Degree programme Engineering Physics and Mathematics		
Major Systems and operations research		**Code of major** SCI3055
Teacher in charge Asst. Prof. Fabricio Oliveira		
Advisor MSc Juho Andelmin		
Date 31.8.2018	**Number of pages** 52	**Language** English

Abstract
In this thesis, we are comparing the performance of 8 line search methods as part of four optimization methods. The line searches used are constant step size, Golden section, bisection, dichotomous, Fibonacci, uniform, Newton's and Armijo's searches. The optimization algorithms, in which the line searches are used for optimizing the step size, are Newton's, gradient descent, conjugate gradient, and heavy ball methods. All algorithms and their comparisons are implemented in Python.

The comparison of the methods is performed separately for two unconstrained nonlinear minimization problem. The first of these is a quadratic matrix problem and the second is an entropy minimization problem. Both of the problems are solved in the domain $\mathbb{R}^{50}$. The performance of the methods is measured as an average of thousand runs starting from different randomized points. For the comparison metrics, we use the solution time combined with the statistics of finding the correct minimum.

Before the actual performance comparison, parameters are selected for each problem, optimization algorithm, and line search combination separately. This is done by running the algorithms using ten random starting points and a set of pre-selected parameters, and selecting the parameter combination that yields the fastest average solving time.

From the final performance comparison, we find that some line searches perform much more efficiently than others. In particular, constant step size, Newton's and uniform searches perform the slowest almost without exception. In turn, Armjo's, golden section and dichotomous searches perform evenly well in almost all test scenarios. The rest of the line searches perform just slightly worse than the best ones on average. All of the optimization algorithms seem to find the correct minimums almost every time regardless of the line search choice.

In conclusion, we state that while there are differences between the line search methods, they are mostly expected differences. For example, the line searches that performed the best, are commonly used in literature and usually recognized as the best line searches of the bunch tested. On the other hand, the slowest searches were expected to perform poorly due to their over-simple design and straight-forward structure. However, there are some exceptions in the line search performances and thus we suggest that the line search selection could be an interesting additional parameter when optimizing a complex problem. In more general cases and simpler problems, sticking to Armijo's or golden section search is usually enough, since they provide good results and even in the rare cases where some other search would outperform them, the difference is likely to be fairly small.

Keywords line search, nonlinear, unconstrained, optimization, minimization, algorithm, performance

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Tekniikan kandidaatintyön tiivistelmä

Tekijä Einari Tuukkanen

Työn nimi Viivahakujen tehokkuuden vertailu rajoittamattomassa optimoinnissa

Koulutusohjelma Matematiikka ja systeemitieteet

Pääaine Systeemitieteet **Pääaineen koodi** SCI3055

Vastuuopettaja Apulaisprof. Fabricio Oliveira

Työn ohjaaja DI Juho Andelmin

Päivämäärä 31.8.2018 **Sivumäärä** 52 **Kieli** Englanti

Tiivistelmä

Työssä tarkastellaan kahdeksan viivahakualgoritmin tehokkuutta osana neljää tunnettua optimointialgoritmiä. Viivahakuja käytetään osana optimointialgoritmejä määrittämään kullakin iteraatiolla askelkoko eli matka, joka liikutaan algoritmin osoittamaan suuntaan. Työssä vertailtavat menetelmät ovat vakioaskelkoko, golden section, bisection, dichotomous, Fibonacci, uniform, Newtonin ja Armijon viivahakualgoritmit. Optimointialgoritmit, joiden osana viivahakuja käytetään, ovat gradient descent, conjugate gradient, heavy ball ja Newtonin menetelmät. Kaikki algoritmit sekä niiden tehokkuusvertailu toteutetaan itse Pythonilla.

Menetelmien vertailu suoritetaan erikseen kahdelle rajoittamattomalle epälineaariselle ja konveksille minimointiongelmalle, joista ensimmäinen on neliömäinen matriisimuotoinen ongelma ja toinen entropian minimointiongelma. Molempien ongelmien ulottuvuus on $\mathbb{R}^{50}$. Menetelmien tehokkuus mitataan tuhannen satunnaisista pisteistä aloitetun haun keskiarvona. Menetelmien tehokkuutta arvioidaan vertailemalla kuinka monesta satunnaisesta aloituspisteestä menetelmät löytävät oikean ratkaisun sekä kauanko sen löytäminen kestää.

Ennen varsinaista tehokkuusvertailua kullekin viivahaun ja optimointialgoritmin yhdistelmälle suoritetaan parametrien valinta kokeellisesti. Tämä tapahtuu suorittamalla tehokkuusvertailu jokaiselle algoritmille etukäteen valituilla parametriyhdistelmillä kymmenestä satunnaisesta aloituspisteestä. Parametreista valitaan varsinaiseen tehokkuusvertailuun ne, jotka yhdessä tuottavat keskiarvoltaan nopeimman suorituksen.

Tehokkuusvertailusta havaitaan, optimointialgoritmit päätyvät oikeisiin optimaalisiin ratkaisuihin lähes joka kerta kaikkien viivahakujen kohdalla. Tästä syystä tehokkuusvertailu suoritetaan pääasiassa ratkaisun keston perusteella. Tuloksista huomataan myös, että osa viivahauista on huomattavasti tehokkaampia kuin toiset. Erityisesti vakioaskelkoko, uniform haku sekä Newtonin viivahaku tuottavat lähes poikkeuksetta huonoimmat tulokset. Sen sijaan golden section, dichotomous ja Armijon viivahaut pärjäävät tehokkuusvertailussa tasaisen hyvin molemmissa tutkituissa ongelmissa kaikilla neljällä optimointialgoritmeillä. Loput viivahaut suoriutuvat keskimäärin vain vähän parhaita hitaammin. Optimointialgoritmit päätyvät oikeisiin optimaalisiin ratkaisuihin lähes joka kerta kaikkien viivahakujen kohdalla.

Tutkimuksen lopputuloksena todetaan, että viivahakujen tehokkuuksissa on eroavaisuuksia, mutta ne ovat pääasiassa odotusten mukaisia. Parhaiten menestyneet viivahaut ovat kirjallisuuslähteiden mukaan yleisesti käytettyjä ja tehokkaimpia viivahakumenetelmiä. Vastaavasti hitaimpien viivahakualgoritmien odotettiin olevan huonoimpia niiden raskaiden tai hitaasti suppenevien algoritmien vuoksi. Tutkimuksen perusteella viivahakujen vertailusta voi olla etua lähinnä erittäin monimutkaisissa ongelmissa, joissa sopiva räätälöity viivahakuvalinta voi tuottaa pienen edun useiden iteraatioiden myötä. Tutkimuksessa erot keskiverron ja parhaan menetelmän välillä ovat suhteellisen pieniä, joten tavallisesti viivahakujen valinta ongelmakohtaisesti ei kannata. Yleisesti on siis paras pysytellä tunnetuissa ja hyviksi todetuissa menetelmissä, kuten Armijon viivahaussa.

Avainsanat viivahaku, optimointialgoritmi, epälineaarinen, rajoittamaton, tehokkuus, minimointi, askelkoko

Contents

Abbreviations used

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

Abbreviation	Explanation
MM	main method, an unconstrained optimization methods
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 objective function
NE	Negative entropy objective function

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, we can formulate an unconstrained optimization problem as

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

where $f : \mathbb{R}^n \to \mathbb{R}$ is called an *objective 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 we can use 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 challenging 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 specific properties, such as being continuously differentiable up to n times.

While countless real-life problems satisfy these conditions, they are often extremely complex, and finding a global minimizing solution can be difficult. Therefore, we often use artificial problems while 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 objective 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. We then repeat these steps until we meet a method-specific stopping criterion. In case we do not find a minimum, we can also use additional conditions, such as maximum time limit or iteration count, to avoid infinite loops.

The step size is often computed by a separate *line search* algorithm. After computing the objective 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 univariate optimization methods. Similarly to the multidimensional optimization methods, there are several different algorithms for finding the optimal step size λ, and the simplest is just a constant step size 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 at 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. Also, we will analyze the performance differences caused by the different line search methods to find metrics and methods that give us the best overall view of the algorithm's performance.

1.1 Organization of the thesis

In section 2 Theory, we will be first introducing the problems we are using to grade the optimization methods. After that, we are going through the optimization methods used and their implementations in the form of pseudocodes. Finally, we will introduce the line search methods that we are comparing, with their respective pseudocode implementations and parameter choices.

Section 3 describes the setup we are using to test the performances. The section includes introducing the method for choosing the values for different optimization method parameters, generating an even but random starting point distribution, and finally, randomization of parameters for one of the problems.

Results are introduced and analyzed in section 4. The section also discusses the possible shortcomings and improvements that can we could make to the approach used to compare the methods in this paper. The section 5 Conclusion provides a brief conclusion and suggests changes if one would repeat the experiments in this paper.

We also include an appendix that contains a link to the source code used, as well as tables describing the best five parameter combinations for each optimization method and line search combination.

2 Theory

2.1 Objective functions

2.1.1 Choosing the objective functions

Since we are only considering unconstrained optimization problems, one of the most significant factors affecting the performance of an optimization algorithm is the complexity of the objective function of the problem we are solving. In this thesis we are going to examine two different problems with convex and differentiable objective 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 objective 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 first problem is an extension to the regular *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. The problem is convex, nonlinear, unconstrained and quadratic.

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 plays part in determining 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 lower 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 $b = (b_1, \ldots, b_n) \in \mathbb{R}^n$ 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 require the first and second derivatives of the step size function. In this case, we can compute the gradient and the Hessian of both functions 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 $\mathbf{H}$:

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

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

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

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

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

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

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 objective function examined in this thesis is an inverted entropy maximization problem. Inverting a maximization problem gives us a corresponding minimization problem [3]. We call this convex minimization problem *negative entropy* problem. Its objective function is of the form:

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

where $\mathbf{x} \in \mathbb{R}^n_{>0}$. Even though the domain is now limited, we can still consider the problem unconstrained. However, we have to make some adjustments to our algorithms to take the positivity constraint into account.

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

$$\begin{aligned} \nabla_{x_i} f(\mathbf{x}) &= \frac{\delta}{\delta x_i} \sum_{i=1}^{n} x_i \log x_i \qquad (14) \\ &= \log x_i + 1 = 0. \qquad (15) \end{aligned}$$

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:

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

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) only includes the positive numbers, we have to make some changes to our starting points, parameters, and algorithms. Therefore, the iteration points must have only positive coordinates. Adding a domain limiter (see algorithm 1) to the line search algorithms prevents the iteration points from 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 $\lambda$
```

We call the domain limiter 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* to more clearly separate them from line search 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 we reach a method-specific termination condition. Choosing the step direction λ is an optimization problem of its own and is solved by the line search methods introduced in 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.

As an additional note, we know that Newton's method is expected to converge in exactly one iteration for quadratic problems, such as the matrix square sum function. [6]

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 the same as Newton's method except for the computation of the descent direction d on line 3.

In the case of the matrix square sum problem, the performance of a gradient-based method will highly depend on the ellipticity or the condition number of the problem.

[6]

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 several successive gradient ratio calculations. The method should theoretically require fewer steps than a 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., $n = \dim(\mathbf{x})$). In terms of performance comparison, a single iteration of the algorithm means one iteration in the inner loop.

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 $i = 1 \dots n$ do
6:         $\lambda = L(g_{\mathbf{y},\mathbf{d}})$
7:         $\mathbf{y_{prev}} = \mathbf{y}, \ \mathbf{y} = \mathbf{y} + \lambda\mathbf{d}$
8:         $\alpha = \frac{\nabla f(\mathbf{y})^2}{\nabla f(\mathbf{y_{prev}})^2}$
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 an extension of the gradient descent with an additional term in the direction vector calculation step. The name of the method comes from an analog 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 on line 3 [4]. Note that there are multiple different variations of the method with slightly varying positioning of the multiplier, but the basic idea always remains the same [4] [7].

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	Matrix Square Sum	Negative Entropy
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 typical 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 objective function.

There are also multiple small variations of the same methods available on different sources on the internet and academic papers. Different sources also often tend to suggest different recommended values for the line search parameters. However, we pick the options for the parameters ourselves with little to no comparison with existing literature. [3] [2]

2.3.1 Constant step size

While not an actual algorithm, the use of a constant step size value λ can be sufficient in some scenarios with some optimization 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: Values tested for the constant step size.

Parameter	Matrix Square Sum	Negative Entropy
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 sequential 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	Matrix Square Sum	Negative Entropy
a	-10, -5	-10, -5
b	5, 10	5, 10
l	1e-04, 1e-07	1e-04, 1e-07

Note that the implementation above makes an error of calling the objective function twice per iteration, even though we only edit one of the limits per iteration. The mistake almost doubles the function calls in all our performance tests for the golden section search. The increase in performance after fixing this error would be around 10 %, which is significant but not groundbreaking within this paper.

2.3.3 Bisection search

Bisection search (algorithm 7) is a line search that uses derivatives to progress. This requires the objective 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	Matrix Square Sum	Negative Entropy
a	-10, -5	-10, -5
b	5, 10	5, 10
l	1e-04, 1e-07	1e-04, 1e-07

2.3.4 Dichotomous search

Dichotomous search (algorithm 8) is a sequential line search. It progresses by selecting some closed bounds $[a, b]$ and selecting the mid-point of the range and 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	Matrix Square Sum	Negative Entropy
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, we do a slight adjustment to test parameters and give a a value of -10 for the 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 bound and reduces the interval by some factor. However, instead of of a constant reduction factor, Fibonacci search calculates a new factor for each iteration using Fibonacci numbers [2]. The parameters tested for Fibonacci search are displayed in table 7.

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 tests, it seems that in our case the pre-evaluation does not effect performance too much and so we decided to generate the Fibonacci numbers during run-time using a single loop algorithm of time complexity $O(n)$.

Table 7: Parameters tested for Fibonacci search.

Parameter	Matrix Square Sum	Negative Entropy
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. We may also increase the number of intervals to divide the new interval into to increase the precision of the method. The parameters tested for uniform search are displayed in table 8. [2]

In practice, we 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	Matrix Square Sum	Negative Entropy
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 (algorithm 2) is another line search that uses the derivatives to find the optimal step size. It exploits the function's quadratic approximations, which yields an equation of second-order differential, and therefore the objective function must be differentiable twice. [2]

In addition to the parameters listed in table 9, the max iteration count is added as an extra parameter for negative entropy function with options being either 100 or 1000 iterations. This is because the domain limiter may cause problems with the convergence while using Newton's line search.

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	Matrix Square Sum	Negative Entropy
λ	0.5, 1, 5, 10	0.1, 0.5, 1, 5
l	1e-07, 1e-08	1e-07, 1e-08

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 means reducing the initial step size on every iteration until it satisfies the stopping condition. Armijo's search is also an inexact line search meaning it only attempts to find a good enough approximation to a minimization problem instead of finding the exact solution like many other searches introduced. The parameters tested for Armijo's search are displayed in table 10. [3]

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	Matrix Square Sum	Negative Entropy
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

The 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 an objective function.

Displaying all the results for each parameter combination would require too much space. Therefore just a concise overview of the best parameters for each setup is found from 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 similarly.

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 that is random but evenly packed with points, we use a method shown in algorithm 13.

Algorithm 13 Evenly Distributed 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 $j = 1 \dots q$ 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 $x_q, q = 1 \dots p$. 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, we find the d_{min} values presented in the table 11. After generating the points once, they are saved into a file and used for every run with the corresponding points count, thus maintaining an equal level of randomness between measurements.

Table 11: The minimum Euclidean distances used for different objective functions on distributions of 10 and 1000 points.

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

3.3 Randomizing objective function

In addition to randomizing the starting points, we are going to use randomized parameters for matrix square sum objective function. We are going to randomize the values of parameters A, b, and c as described in section 2.1.3 but for each starting point separately. Again, we want to maintain the randomness between runs, and thus always give the function a random seed equal to the index of the current point. This way, the number of randomized objective functions is equal to the number of starting points used. Combined with the saving of points, we are guaranteed to get the same test scenarios for each run.

4 Results

We measure the performances using the best parameters found for each scenario, as described in section 3.1. After running the performance tests, we get the results introduced in the tables 12 to 19. Each table contains eight rows, one for each line search method, and there is a separate table for each permutation of the main methods and objective functions. The columns show the line search method used, percentage of correct solutions or the success rate s (%), solution time t (ms), iteration count k, total function call count f_n, total time taken by line search algorithm alone t_{LS} (ms) and iterations made by the line search k_{LS}. The function call count includes all calls to the function, including its gradient, step size function and its derivatives.

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 Method	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 Method	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 Method	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 Method	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 Method	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 Method	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 Method	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 Method	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 comparisons we could make using the data we get. Since our goal is to compare the performances of the main methods concerning the choice of the line search method, we display the data primarily by the line search method and secondarily by the main method. We are also interested in spotting any differences between the two problems, so let us also include a comparison between the objective functions.

In addition to the different comparisons, there are also multiple metrics we could look into when analyzing the results. The most important metric for measuring the algorithm's performance is the success rate of solving the problem. However, since most of the methods have similar success rates and the differences in solution percentages are small, we need a different metric for a measure of performance. We choose to use the average solution duration of the main method as our primary comparison metric.

Even though the average solution times of the methods are not universally comparable within different implementations or problems, it provides us a decent comparison of the line search methods since we are using identical scenarios for each main method. Therefore the values in tables 20 to 23 are mainly comparable within the same column. To visualize the column-wise differences, we apply a per-column color scale in which red implies poor performance, and green implies good performance. The darker the color, the worse or better the value is within the column. White cells have their value around the average of the column.

Table 20: Average duration of the algorithms using different line search methods for each main method when finding minimum for the MSS problem. The color scale is applied to visualize the performance per column. The scale goes from dark red to dark green and respectively from the worst (slowest) to the best performance (fastest).

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 the NE problem. The color scale is applied to visualize the performance per column. The scale goes from dark red to dark green and respectively from the worst (slowest) to the best performance (fastest).

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 the MSS and NE problems. Results are listed separately for each main method. The color scale is applied to visualize the performance per column. The scale goes from dark red to dark green and respectively from the worst (slowest) to the best performance (fastest).

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 objective function. The color scale is applied to visualize the performance per column. The scale goes from dark red to dark green and respectively from the worst (slowest) to the best performance (fastest).

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 similar column-wise coloring, but this time with only the below-average cells colored with different shades of red.

Table 24: The success rates of the main methods using each line search method for finding the minimum of the MSS function. The color scale visualizes the success rate per column so that darker the red, the lower the success rate is.

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 the NE function. The color scale visualizes the success rate per column so that darker the red, the lower the success rate is.

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 essential factor to consider is the consistency of the results. The boxplots in figures 1 and 2 give us valuable insight into 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 drastically different from others. The box plot displays half of the data points within the main box area. Together with the box area, the whiskers above and belove the box cover 95% of the data points. The rest of the points are considered outliers and left out entirely. The orange line inside the boxes shows the exact mean of the data points.

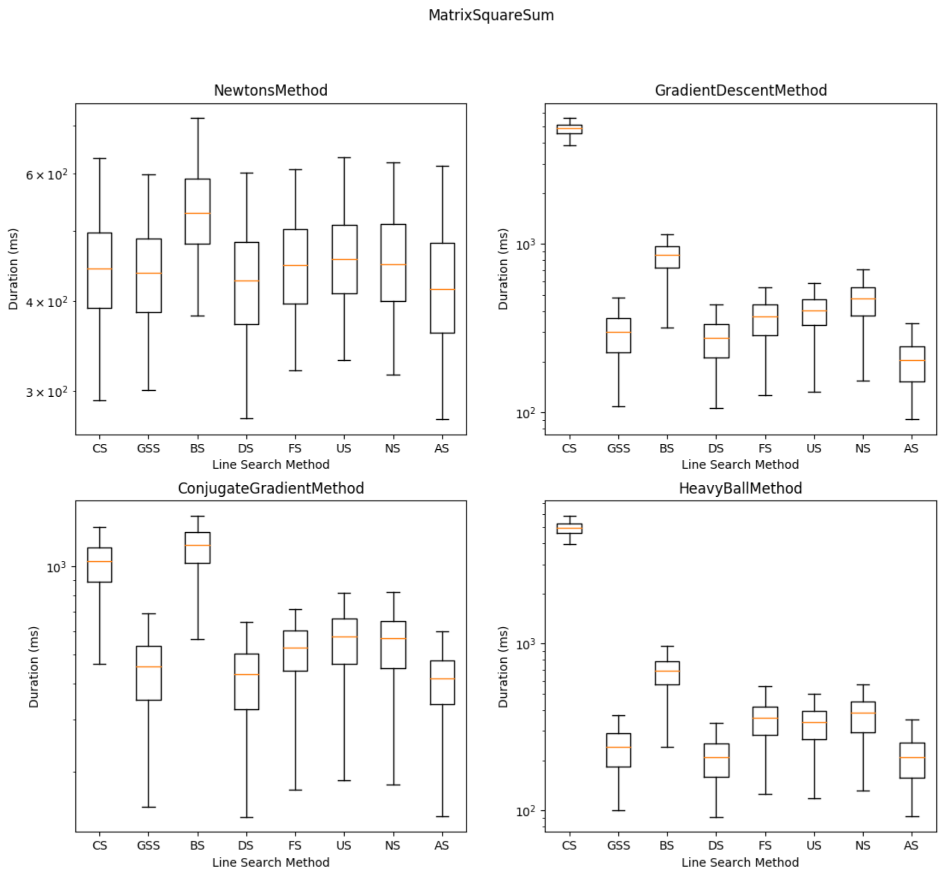

Figure 1: Box plot of solution times per line search and main method for the MSS problem.

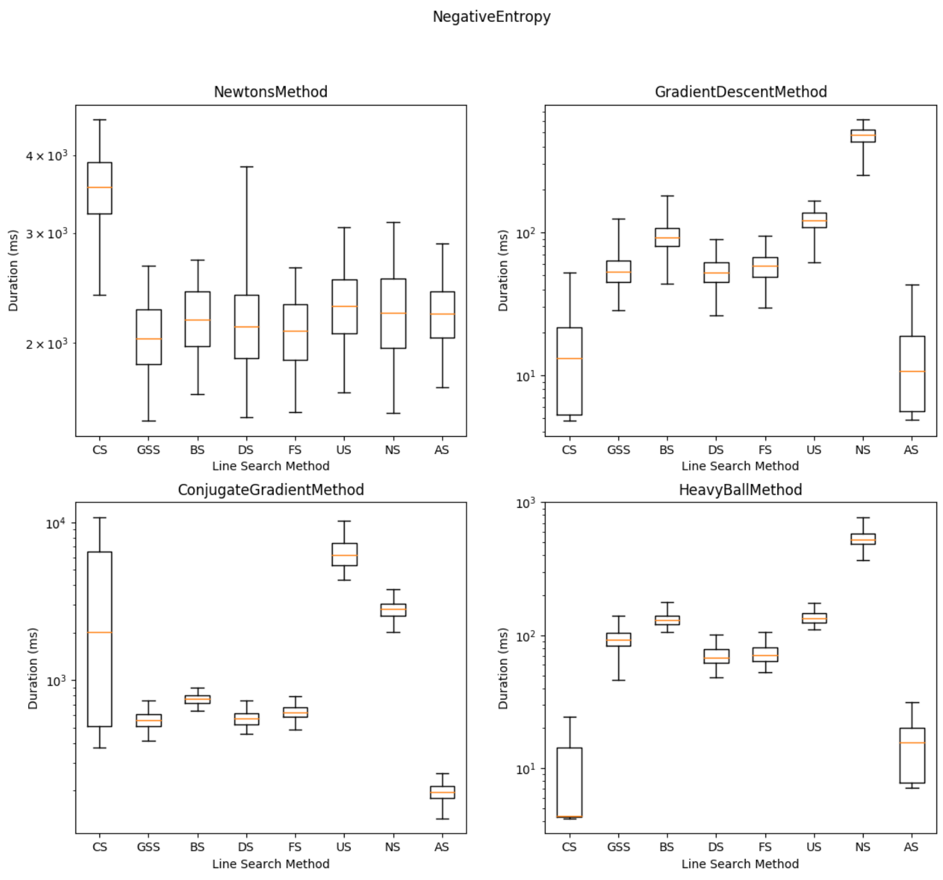

Figure 2: Box plot of solution times per line search and main method for the NE problem.

4.2 Optimization method comparison

In this paper, we mainly focus on comparing the line search methods, but it is crucial also to note a few things about the main optimization methods as well. For this comparison, we are mainly going to look at the performance data in tables 12 to 19, as well as the combined results in colored tables 20-25.

Newton's method

We know that Newton's method has the property that it solves any convex quadratic problem, such as the MSS problem, in one step [3]. We can see this behavior in table 12, where the iterations column has the value $k = 1$ for each line search. We can also see that some line searches have proceeded to execute several iterations, even though the main method would note require that. In this case, a constant step size should be the fastest option since it does not even make the unnecessary function call to the line search algorithm, but we can see that this is not the case. The unexpected difference of 20-30ms, which the methods mostly share between each other, can be explained by random variation. The fact that the bisection search is also able to cause Newton's method not to converge in 0.4% of the 1000 starting point setups implies that there is likely some improvements to be made to that line search at least.

Newton's method generally seems slower for the negative entropy problem. Since the problem is not quadratic, Newton's method needs several iterations with heavy calculations, such as calculating the Hessian, which is a possible cause for the added delay. Newton's method performs well on the MSS problem, but the performance on the NE problem is so weak that Newton's method ends up being the slowest main method in general, as seen in table 23.

Gradient descent method

Gradient descent method's performance is known to heavily depend on the condition number of the problem [6]. We discuss the MSS problem's condition number in detail in section 2.1.3. From the imperfect success rates with the MSS problem, we notice that there are few points in the randomized distribution from which the gradient descent method has problems converging. The effect that the condition number has on the performance is significant and therefore, should be accounted for when analyzing the gradient descent method's performance on such problems.

Even though the gradient method has some problematic starting points, it and the heavy ball method based on the same idea are the best performing optimization methods overall.

Conjugate gradient method

With our problems and the implementation of conjugate gradient method, we expect the algorithm to find a solution in $n = 50$ iterations [6]. From the performance tables, we can see that this is mostly the case for the MSS problem. The solution times are, in general, a little higher than with the other methods. The same effect can be seen with the NE problem, though in that case, the iterations required are approximately 100. The unexpectedly high iteration count may be due to the domain limiter causing unexpected inaccuracies within the algorithms. Nevertheless, the algorithm does find the correct solution in almost all cases. Conjugate gradient method has a reliable performance in general but faces serious issues when combined with some line searches and the NE problem in particular.

Heavy ball method

The heavy ball method is based on the gradient descent method and therefore shares its limitations and properties. In theory, the heavy ball method should, at worst, reach the performance of the gradient descent method. We notice that this is mostly the case in the MSS problem, where HBM mostly provides small improvement over the traditional gradient descent method. However, the solution times in the case of the NE problem are mostly slower, implying that the parameter configuration might have been faulty. Since the parameter options did not include option $\beta = 0$, which would have turned HBM into GDM, the values are accepted but not a sign of the method being generally worse.

4.3 Line search comparison

To compare the line search methods, we are going to use the same tables as with main optimization methods, but this time focus on the grouping by rows instead of columns.

Constant search

Constant search always gives a constant step size and therefore provides an excellent baseline to compare the other line search methods to. 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.

Looking at the box plots in figures 1 and 2, constant step size is again the worst performer by not just the 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 the golden section search is one of the best performing line searches when comparing the total solution times. The method does, like most methods tested, find the correct solution in almost all of the test cases. The variances displayed in the box plots are also average.

Besides, as stated in section 2.3.2, a non-optimal implementation of the golden section search was used, meaning that we could reduce the function call count by up to 50%. With the problems tested, this would lead to a noticeable but not groundbreaking difference in the method's performance. Even with this mistake, the golden section search proved as one of the three top performers in our tests.

Bisection search

Bisection search does perform rather poorly in our test setup. It scores the second to 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 the MSS problem and barely average with other setups. The box plots show that the bisection search does, however, have pretty average variances in performance.

Dichotomous search

In our initial tests, the dichotomous search was one of the slower methods due to not finding the correct solutions within the given maximum iteration limit. However, lowering the main method's tolerance proved to be especially helpful for this line search, which ultimately turned out surprisingly as one of the stronger methods. It even outperforms the golden section search in MSS optimization and scores almost equally in overall scores. The success rates are also identical, but reviewing the variance of the performance from box plots reveals some issues with the dichotomous search when used with Newton's method in particular. Overall, the dichotomous search proved as one of the three top performers in our tests.

Fibonacci search

The Fibonacci search ends up with an average ranking in our comparison in all measures: the raw performance, success rate, and variance. Our implementation of the Fibonacci search computes each Fibonacci number during runtime without saving them into memory. The highest Fibonacci number we reach is around F_{50}, so this is not a huge issue. However, for real-life applications, precomputation of the Fibonacci numbers would likely be a worthy addition since it showed around a 10% increase in performance with only a small increase in memory use.

Uniform search

We expect the uniform search to perform below average due to the inefficiency of the algorithm. From the performance tables, we can see that this is indeed the case, and the uniform search performs averagely on the MSS problem and clearly below average on the NE problem. Part of the poor overall performance in the NE problem is because the method sometimes fails to stay within the 10000 max iteration limit when used with the conjugate gradient method. There are many ways we could presumably relieve these issues, but since we know the uniform search's algorithm is inefficient in its core, it would not make sense to over-optimize its use case.

Newton's search

We find Newton's search to end up on the worse end of our averagely scoring methods on the MSS problem. However, the NE problem is where Newton's line search seems to lack in performance with twice the average overall execution time compared to the other average methods.

Solving the NE problem seems particularly tricky for Newton's method since the line search often hits the max iterations limit, yet it succeeds at finding the correct optimum. This behavior leads us to choose a low max iteration count that allows the method to fail faster when in the scenarios where it would get stuck. We suspect that this special treatment, together with the domain limiter, may be the cause for the method's terrible performance on the NE problem. The unreliability of this method is also visible from the high variances shown in the box plots.

Armijo's search

Armijo's search proves to be the overall best performing method of the tested line searches. It has the best performance measured in the solution times 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 higher success rate on MSS+GD and MSS+HBM scenarios compared to all other methods. There is an average amount of variance in the performances of Armijo's search. However, even after accounting for that, the performance is still clearly better or on-par with the other top performers.

4.4 Assumptions and further performance improvements

One major factor to consider when comparing optimization algorithms is the problems used to measure the methods. In this paper, we only used two different problems. The randomization of the parameters adds some variety in the MSS scenario, but still, both of the objective functions remain relatively simple. The randomization in MSS is also modest and does not provide us with many unique problems. Both of the problems considered are also convex and have a single global maximum, making the solving very different from when there also exist local maximums.

In the case of the NE problem, we use a domain limiter algorithm to ensure that values stay within the correct domain. As discussed earlier, this potentially causes some convergence issues with some of the line searches.

Another factor to consider is the starting point distribution. While we find the starting point count of 1000 to be reasonable, the range from which we generate the points could be set more extensive.

Choosing the optimal parameter values also plays a significant role in the performance comparison. Since choosing the best parameters is a complex problem in itself, we find our approach of finding the parameters by small scale simulations a justified solution for the scope of this thesis at least. However, by increasing the number of options for the values of parameters, the iterations, and the number of test cases, we can likely reach even better results. The tradeoff would naturally be a higher computation time required to find the optimal parameters.

Some performance factors are challenging to evaluate accurately. One example of such a factor is the actual implementation of an optimization algorithm, which seems to have small variances depending on the source. One difference, in particular, seems to be the position of the stopping condition within the main methods. Some sources suggest checking the stopping condition before updating the point, but in all of our implementations, the check is done just after the update before entering the next iteration. [3] [2]

Another possible implementation related optimization would be the pre-calculation of line search values. In some scenarios, it could be beneficial to pre-evaluate and save the step sizes for each different input instead of calculating them during run time. [3]

Factors like the choice of programming language, and CPU, GPU, and RAM performance may affect the optimal results as well. In this thesis, we do all of the performance testings using Python 3.7.3 with numpy, autograd, and multiprocessing libraries. One can find the source code used to produce the results from the appendix A. We run the software on 64bit Linux desktop PC with an Intel i5-8500K (6 cores @ 3.60GHz) CPU and 16 GB of RAM. Different implementations and programming languages may, for example, optimize matrix calculations differently, causing potentially unexpected differences within otherwise identical test scenarios. For this reason alone, we consider none of the results as universally accurate and instead suggest evaluating each problem and scenario separately.

5 Conclusion

The choice of line search method seems to have a varying size of an impact on the performance of an optimization method. For example, 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 top three methods, in turn, are around 10-20 % slower than the fastest one in average overall duration.

If we were to perform this research again and the performance comparisons again, we would use a higher number and variety of objective functions with a broader range of starting points. We would also like to spend more time justifying the different choices for the parameters of the optimization methods. For example, we could try to find proven values from existing literature or simply spend more time evaluating a more extensive range of different values. One change that would help in re-evaluating the parameters would be to drop out uninteresting line searches. Leaving out the slowest methods could offer significant time savings since evaluating algorithms that rarely converge is very time inefficient.

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 or golden section search, it is unlikely that any of the tested alternative algorithms would provide any improvement in performance. The only scenario in which there could potentially be a better alternative would be when the main optimization method solves the problem efficiently without a line search method, namely using a constant step size. However, this is a rare occurrence that would require a specific type of problem and optimization algorithm.

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

A.1 Source code

All source code in full is available in Github:
`https://github.com/EinariTuukkanen/line-search-comparison`

A.2 Parameter comparison results

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 ConjugateGradient-Method

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 4 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 3 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 4 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 Bisection-Search 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 ConjugateGradient-Method

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 Bisection-Search 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 ConjugateGradient-Method

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 ConjugateGradient-Method

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 ConjugateGradient-Method

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 UniformSearch 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 ConjugateGradient-Method

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