Extended Lock-in Feedback Applicable on Higher Dimensional Function Maximization

BSc AI Thesis

Radboud University Nijmegen

Tom Janssen Groesbeek *
Nijmegen, the Netherlands
tomjg@hotmail.nl

ABSTRACT

This thesis introduces two novel versions of the existing Lock-in Feedback algorithm. This algorithm is a means of performing stochastic optimization. The novel versions include alterations that make them applicable on higher dimensional function maximization problems. By running several simulation tests and examining the cumulative regret returned by each method, this thesis shows that the proposed extensions prove to be performing well on function maximization problems of two dimensions. Both versions are also applied on a function containing multiple maxima, in order to test their ability to deal with more complex maximization problems. By making adjustments to the Lock-in Feedback algorithm inspired on the Artificial Bee Colony algorithm, it was made sure that the method would uncover global in stead of local maxima.

Author Keywords

sequential function maximization; extended Lock-in Feedback; higher dimensional; simulator; continuous; stochastic optimization problem.

INTRODUCTION

This thesis addresses the topic of (sequential) function maximization. In other words: performing experiments to find the optimal value for maximizing the output of a function of interest. Where the value's level of optimality is assessed on how well it maximizes the output and where the function is unknown. An example challenge from the field of economics that illustrates this topic is the determination of a selling price for a product for which the product's revenue will be maximized [11] [3].

*T.J.G. acknowledges the support of Dr. Maurits Kaptein in conducting the research needed to write this thesis.

When selling products it is wise to perform some sort of market research in order to determine the best possible selling price. A simple study could already be performed by presenting customers with different prices at different times to subsequently observe the total amount of products sold for each price. After running this experiment for a certain period one could decide to stop experimenting and start exploiting the best observed selling price. Of course, reducing the total exploration time can be very interesting in many situations. But it might also be interesting to be able to perform a combination of exploration and exploitation, where the option currently being exploited is updated continuously [5]. Therefore, several literature on the exploration/exploitation trade off already exists (see, e.g., [12], [14]).

What is also widely covered in the literature is one-dimensional function maximization (see, e.g., [17], [4], [9], [16]). One-dimensional functions are functions dependend on a single controllable variable. Thus maximizing these functions still includes a certain exploration period, but only exploring the optimal value for this single variable. The economic example illustrates a scenario in which one is interested in picking, from a large set of options and at each moment in time, the one option that will lead to maximum total revenue. But, the option exists out of just one controllable variable, namely the selling price.

The current study, however, is interested in function maximization dealing with two controllable variables (higher-dimensional function maximization). A possible illustration of higher-dimensional function maximization is the case of a salesperson determining how many of the same product should be sold in a bundle while at the same time determining how much discount this bundle should get in order to obtain the highest possible revenue. Think for example of a grocery store selling two bags of the exact same potato chips for the price of just one bag of potato chips. Another possible scenario could be a salesperson thinking about introducing the concept of 'Group Buying' to increase the sales of a particular product. Group Buying is the concept of assigning a discount to a product only when a specific amount of people decide to buy the same product at once [2]. The salesperson could vary the amount of people needed to obtain this discount as well as the total discount itself.

What should be noted is that the current study is only interested in function maximization of sequential nature. Sequential means performing experiments, trying out different values for each controllable variable, at different moments in time. One experiment after the other. In a more formal sense, sequential function maximization boils down to finding $\vec{x}_{max} = arg\ max_{\vec{x}}\ f(\vec{x})$, where $arg\ max_{\vec{x}}\ f(\vec{x})$ is the set of values for $\vec{x}$ for which $f(\vec{x})$ will attain its maximum value. $\vec{x}$ represents a vector containing multiple controllable dimensions and which is used in the mostly unknown function $f(\vec{x})$. And each trial is indexed by $t \in \{1,\ldots,T\}$ in order to make sure the experiments occur sequentially.

There exist different kinds of methods to deal with sequential function maximization. Among several methods of interest for this thesis, which will all be introduced in the next section, stands a method originating from the field of physics and engineering called *Lock-in amplifier technique*. Kaptein and Iannuzzi adapted the technique so that it could function as a generic tool to find $\vec{x}_{max} = arg\ max_{\vec{x}}\ f(\vec{x})$ and named it LiF which is short for *Lock-in Feedback*. The LiF method relies on oscillating the controllable variable x at a fixed frequency and to look at the response of the dependent variable y at the same frequency via a *lock-in amplifier* [6]. Their focus was on cases where x is a scalar and they suggest that future experiments should also consider extending the original algorithm to make it applicable on cases where x is a vector.

Cases where x is a vector do exist as one of the previous examples stated. So the development of a method that is able to deal with higher dimensional cases could potentially result in several applications. Therefore the goal of this thesis is to extend the proposed LiF algorithm by Kaptein and Iannuzzi so that it can be applied on higher dimensional sequential fuction maximization. Meaning that it should be able to deal with multiple controllable variables ($\vec{x}$). The first research question of this thesis is: How can the existing Lock-in Feedback algorithm be altered to deal with higher-dimensional problems?

In the paper by Kaptein and Iannuzzi it is also stated that the current Lock-in Feedback algorithm is prone to getting stuck in a local maximum. This susceptibility may also remain present when working with altered versions of the original Lock-in Feedback method. So the second research question of this thesis is: How does an altered Lock-in Feedback algorithm perform when applied on functions with multiple maxima?

Next, this introduction will be followed by a literature review of a collection of relevant and related research on sequential function maximization. Most importantly the following section will elaborate more on the LiF algorithm and why it is of interest for this thesis. Subsequently, in the Method, the extended versions of LiF will be presented as well as the simulator which will be used to evaluate these novel algorithms as well as the existing optimization algorithms. The evaluation itself will mainly consist of comparing the cumulative regret of the used algorithms. The different existing approaches used in this thesis are the ε-first method as well as the Artificial Bee Colony algorithm ([10], [8]). Both approaches will be explained in more detail later on. The final sections of this thesis will contain an interpretation and a discussion of the results collected from the performed simulations.

LITERATURE REVIEW

The focus of the paper by Kaptein and Iannuzzi was to efficiently and sequentially find continuous (with x being $\mathbb{R}$) treatment values which maximize some observable outcome of an experiment [6]. Thus, they presented the Lock-in Feedback method as a means of accomplishing this. LiF is a derivative free method to perform stochastic optimization with bandit feedback. Derivative free methods tackle the problem of optimizing an unknown function of which only its values, including possible noise, at various points can be observed [16]. They are termed derivative free methods as one has only access to the functions values rather than gradients. For a method to perform stochastic optimization means that the algorithm itself employs a probabilistic rule for improving a solution [8]. When bandit feedback is included the solution is improved by making use of the observed value of the unknown function at a single point [1].

LiF is based on the Lock-in Amplifier Technique originating from the field of physics and engineering [15]. It relies on oscillating the variable x at a fixed frequency and to look at the response of the dependent variable y at the same frequency via a lock-in amplifier. Following a scheme used in physical lock-in amplifiers helps in providing direct measurement of the value of the derivative of f at $x = x_0$. In summary: x is oscillated with time according to: $x_t = x_0 + A\cos\omega t$. Where x_0 is the central value of the oscillation and ω its angular frequency. These oscillations return values y_t for the unknown function $f(\vec{x})$. Each returned value y_t is multiplied by $\cos(\omega t)$ following the scheme used in physical lock-in amplifiers which results in y_ω. The running sum $\sum \vec{y_\omega}$ divided by the integration time T will result in y^*_ω which in turn is useful as an update strategy for x_0, namely: if $y^*_\omega < 0$, then x_0 is larger than the value of x that maximizes f. Or, if $y^*_\omega > 0$, x_0 is smaller than the value of x that maximizes f. The information obtained from calculating y^*_ω with the oscillations will help in moving x_0 closer to x_{max}. The update rule proposed in the work by Kaptein and Iannuzzi is $x_0 = x_0 + \gamma\, y^*_\omega$ where γ is the learn rate of the procedure [6].

The previous explanation for the update rule is explained in figures 1, 2, 3, 4, 5, and 6. In figure 2 a table is shown where x is oscillated and five different values for y have been observed. After multiplying each observation with $\cos(\omega t)$ the total sum is equal to 5. The sum is larger than 0 and thus the value of x is smaller than the value of x that maximizes it. Thus the value of x should be increased. This scenario is illustrated in figure 1.

Figure 3 illustrates the opposite case, where the sum is smaller than 0 and thus the value of x is larger than the value of x that maximizes it. Therefore, the value of x should be decreased. The exact values for this example are illustrated in figure 4.

The final scenario is where the value of x is near x_{max}. In this case the sum is equal to 0 and the value of x should neither be increased or decreased. Which is illustrated in figures 5 and 6.

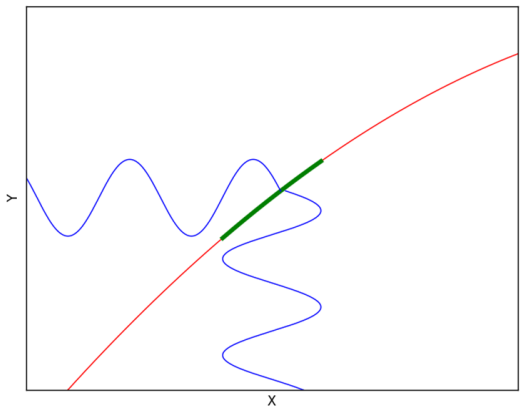

Figure 1. Illustration of the case that $x_t < x_{max}$.

x+cos(ωt)	y	y*cos(ωt)
X - 1	4	-4
X - 0.5	5	-2,5
X	6	0
X + 0.5	7	3,5
X + 1	8	8
	Sum	5

Figure 2. Table corresponding to the scenario illustrated in figure 1.

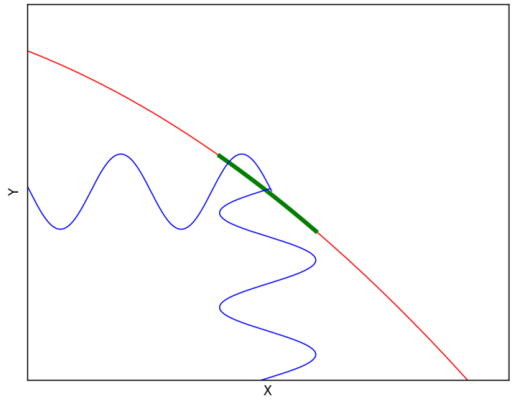

Figure 3. Illustration of the case that $x_t > x_{max}$.

x+cos(ωt)	y	y*cos(ωt)
X - 1	8	-8
X - 0.5	7	-3,5
X	6	0
X + 0.5	5	2,5
X + 1	4	4
	Sum	-5

Figure 4. Table corresponding to the scenario illustrated in figure 3.

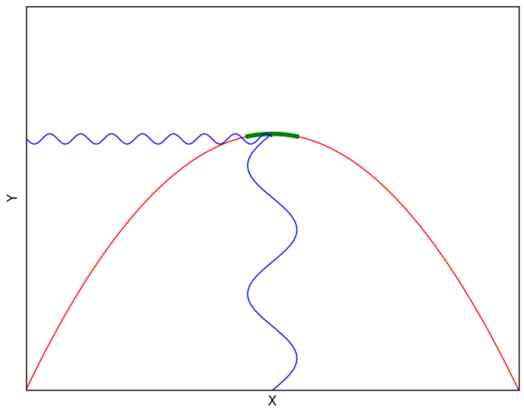

Figure 5. Illustration of the case that $x_t = x_{max}$.

x+cos(ωt)	y	y*cos(ωt)
X - 1	5	-5
X - 0.5	6	-3
X	7	0
X + 0.5	6	3
X + 1	5	5
	Sum	0

Figure 6. Table corresponding to the scenario illustrated in figure 5.

This entire process assumes x to be scalar. Kaptein and Iannuzzi suggest that, in the case x is a vector, a similar approach can be taken to find the maximum of the function $f(\vec{x})$ in more than one dimension. When x is two dimensional, LiF can deal with this problem by oscillating both elements of x at different frequencies. Where the original LiF algorithm would start to oscillate only at x_0 the new version will oscillate both elements of x at different frequencies represented as ω_1 and ω_2. Thus:

$$x_{1,t} = x_{1,0} + A_1 \cos \omega_1 t \tag{1}$$

$$x_{2,t} = x_{2,0} + A_2 \cos \omega_2 t \tag{2}$$

After having oscillated both elements of x, $y_t = f(x_{1,t}, x_{2,t})$ can be observed and information regarding the gradient can be retrieved by making use of both omegas to separately compute:

$$y_{1,\omega} = y_t \cos \omega_1 t \tag{3}$$

$$y_{2,\omega} = y_t \cos \omega_2 t \tag{4}$$

While this will allow LiF to be used for higher dimensions it should be noticed that ω_1 and ω_2 should not be equal to or multiples of each other. This should prevent that the frequencies will overlap. When the frequencies do overlap the exploration for the optimal values for each element of x will occur diagonally in the search space. When this overlap is not present, the exploration will present a more circular pattern which results in more options being explored.

Another remark made by Kaptein and Iannuzzi, relates to the second research question of this thesis. They state that the Lock in Feedback method is susceptible to finding local maxima instead of the sought global maximum. They propose that a good solution to this problem could be oscillating multiple independent $\vec{x_0}$ at different starting points. This could possibly lead to unveiling multiple maxima and this results in multiple options in which one can expand the experiment. One could for example choose to evaluate the different maxima and pick the best one or decide to ignore certain found maxima and keep updating the ones which are left.

One existing method seems to be performing the proposed solution by Kaptein and Iannuzzi for dealing with multiple maxima. This method is based on the behavior of a honey bee swarm and is therefore named the Artificial Bee Colony (ABC) algorithm. The term swarm is here used to refer to a collection of agents interacting with each other. Inside swarms exist division of labour which enables simultaneously task performance by specialized agents. The general idea behind the collective intelligence of honey bee swarms consists of three essential components according to Karaboga [7]. These components are: food sources, employed foragers and unemployed foragers. Next to these components the minimal model behind forage selection defines two modes of behaviour: recruitment of unemployed bees to a nectar source and the abandonment of a source [7]. The possible gain of each food source can be represented in a single quantity, which can be represented by a certain function $f(x)$. Bees who are at the moment busy with particular food source are employed bees as they are currently exploiting the food source. They will return to the other bees with information about this food source. The information itself is the value given by $f(x)$. Unemployed bees on the other hand are looking for a food source to react on. This gives us two more types of bees, namely scouts and onlookers. The scouts will go and search the area for unknown food sources. Onlookers wait in the nest for new information on possible profitable food sources which they could help to exploit.

This behavior relates to the strategy proposed by Kaptein and Iannuzzi to deal with local maximum as it deals with function maximization by making use of multiple starting points. When an algorithm only deals with one starting point it risks getting stuck in a local maximum as it will have no knowledge of other possible maxima. Thus by making use of multiple starting points and evaluating each observation done, this risk is reduced.

An algorithm similair to the ABC algorithm is named the Virtual Bee Algorithm and was created by Yang to solve numeric function optimization [8] [13]. This algorithm only slightly differs from the ABC algorithm. A swarm of virtual bees is created to work on functions with two parameters. The bees start to spread randomly in the space of the function. The bees will interact on some values encoded by the function as if it represents nectar. Finally, to solve the optimization problem one only needs to look at the intensity of the bee interactions on the function. Certain parameters for the function will be visited more by the swarm then others.

According to Tereshko and Loengarov [8], who established a robotic idea on the foraging behavior of bees, the swarm possesses a significant tolerance. The failure of one bee does not result in the failure of the entire swarm. Moreover, as the individual bees might have limited capabilities and limited knowledge of the environment, the entire swarm develops collective intelligence.

Taking these statements in mind, these algorithms possess the possibility to perform very smart and efficient behavior to solve complex maximization cases by making use of smaller parts of codes that perform very specific work. Where finding the global maximum of a function without getting stuck in a local maximum is an example of a complex maximization case. For this thesis it therefore makes sense that a logical extension of the current LiF-II algorithm would be to improve it with the positive aspects of the ABC algorithm. This roughly means that instead of oscillating around one x_0 the algorithm would generate multiple starting points from which to start oscillating. Each starting point could then be evaluated and from there it could be decided to abandon certain points and to focus on potentially more profitable points.

This thesis presents two extended LiF-II versions which will be applicable on two dimensional problems. One version will be extended by only making use of the suggestions made by

Kaptein and Iannuzzi by simply oscillating both elements of vector x at different frequencies. The other version will include the same extension but will also include additional changes inspired on the ABC algorithm. Next to these two versions of the LiF-II algorithm, this thesis will also be making use of the ABC algorithm itself and the ε-first algorithm. The ε-first algorithm makes use of two stages. The first stage is the exploration stage and here choices are made pure randomly. The second stage is the exploitation stage and here the alternative with the greatest estimated value rate is always chosen. Both the ABC algorithm and the epsilon-first algorithm will solely function as comparisons for the two extended versions of LiF-II. The next section will go into more detail on all proposed algorithms and will also discuss the simulation experiment setup as well as the settings and functions used.

METHOD

As is made clear in the introduction, the current thesis is interested in answering two research questions. These are:

- How can the existing Lock-in Feedback algorithm be altered to deal with higher-dimensional problems?
- How does an altered Lock-in Feedback algorithm perform when applied on functions with multiple maxima?

In order to answer these questions, two studies were performed. Each study had its own data generating function on which all implemented algorithms were tested. To accomplish this a simulator was created.

In this section the simulator setup will be discussed. The first part shall discuss the simulator and its general settings as well as the pseudocode of the different algorithms that have been implemented. The different algorithms are: LiF-II (extended version 1), LiF-II (ABC inspired), Artificial Bee Colony algorithm, ε-first, and ε-first$_2$. Following this first part will be a subsection on the functions which have been used to generate the data on which the algorithms were tested. In total two data generating functions have been used. One function only generates one global maximum, while one function generates multiple maximums. The latter generates in total 4 maximums, but with only one global maximum. The final part of this section will discuss the evaluation procedure. Thus the way the performance of each algorithm on the different data is evaluated and why. Important to know is that all code has been written in Python and in order to plot the different functions and to make the various calculations numpy.py and matplotlib.py have been imported.

Simulator and General Settings

The simulator could be operated by means of a GUI. This GUI made it easy to perform several simulation runs on different algorithms and on different data generating functions in sequence. Also, the simulator included some general settings that are used for every run no matter which algorithm or function is selected. These settings are:

- runs = 100
- horizon ($\mathbb{T}$) = 10000

Runs is set to 100, which means that every simulation test will run the selected algorithm on the selected function for 100 times. Next the horizon is set to 10000 for every simulation test, which means that every algorithm will be iterated for 10000 times. So every run exists out of iterating the selected algorithm on the selected function for 10000 times. These values have been chosen as 10000 iterations seemed sufficient for each algorithm to reach the maximum of each function. And increasing the total amount of runs would result in a large amount of time needed to collect all data. Thus, to be sure that this thesis would be finished in time, it was decided to keep the entire experiment to 100 runs.

Futhermore, every iteration is composed of three function calls to the algorithm currently being used:

- algorithm.giveNextXt
- algorithm.giveNextYt
- algorithm.evaluate

The giveNextXt call ensures that the algorithm calculates the subsequent values for each element of x_t. Then giveNextYt is used to observe the value y_t returned by the data generating function by making use of the previously calculated values for each element of x_t. Finally, the evaluate call tells the algorithm to evaluate the observation which should enable the algorithm to update each element of x_t. Thus returning to the start of the loop.

This simulator design was chosen as it led to a neatly structured implementation of the code. If needed, more algorithms could easily have been included in the experiments as for the general fashion of iterating each algorithm. Moreover, the GUI ensured that multiple runs with different algorithms and on both function could be started without having to change the code constantly. For more detail on the simulator see the Appendices.

Algorithms and Settings

LiF-II extended version 1

The pseudocode presented in Algorithm 1 represents the algorithm for the first extended version of LiF-II. In this algorithm x is no longer a scalar but a vector as this version should be used to deal with two-dimensional problems. Where the original LiF algorithm would start to oscillate only at x_0 the new version will oscillate both elements of x ($x_{1center}$ and $x_{2center}$) at different frequencies represented as ω_1 and ω_2. This can be seen in lines 5 and 6.

Lines 7, 8, and 9 show how y_t is obtained, multiplied, and stored in $y\vec{\omega}_1$ and $y\vec{\omega}_2$. Futhermore, in this implementation the summation of these multiplied observations y_t is used to update $x_{1center}$ and $x_{2center}$. Which can be seen in lines 11, 12, 13, and 14. The algorithm itself is almost identical to the algorithm used by Kaptein and Iannuzzi.

The algorithm requires several parameters which are listed below:

- $x_{1center}$ and $x_{2center}$ represent the starting point of the algorithm. Together they form $\vec{x}$ and the algorithm tries to calculate the values for both in order to obtain x_{max}. For this thesis it has been decided to randomly assign starting values to each element of $\vec{x}$. Kaptein and Iannuzzi suggested to let x_0 start as close to x_{max} as possible, which depends on the available information on $f(x)$. However, no available information is assumed making each starting point a potentially good starting point. Furthermore, as will be explained in the next subsection, LIFII2 will include multiple distinct starting points on which basically LIFII1 will be performed. This makes it even more interesting to start LIFII at random points instead of same fixed point each run.
- A_1 and A_2 denote the amplitude that affects the costs of the search procedure. This parameter will decide how wide the oscillations will be, which might result in a large number of x values with low resulting y values. Making A_1 and A_2 too large will possible lead to overshooting x_{max}. Both A_1 and A_2 are set to 1 in accordance with the simulation test performed by Kaptein and Iannuzzi. For this thesis no experiments have been run in order to find the optimal setting for the amplitude.
- Next is T which represents the integration time. Again the value given to this parameter has different effects as a larger T leads to a smoother update but also a slower convergence. For this thesis T has been set to 10. Also no additional experiments could be run to decide on the optimal setting of T.
- γ is the parameter which determines the step size at each update of x_0. In the emperical regret study performed by Kaptein and Iannuzzi they decided to set $\gamma = 0.1$, which is also the value to which γ is set during this research.
- Both $y\vec{\omega}_1$ and $y\vec{\omega}_2$ will contain a running sum of $y_t \cos \omega_i t$ over t that is used for the integration and is therefore an empty list.
- Finally, both ω_1 and ω_2 represent the frequency at which the oscilation of each x_i will be occur. ω_1 is set to $\frac{2\pi}{T}$ as this was also the setting used in the paper by Kaptein and Iannuzzi. Because ω_2 should not be equal to or a multiple of ω_1, it was decided to set ω_2 to $\frac{3\pi}{T}$. No further research has been performed on the optimal settings for both frequencies.

The algorithm makes us of continuous updates of $x_{1center}$ and $x_{2center}$ which means that after each observation $x_{1center}$ and $x_{2center}$ are updated. The previous information can also be read in the paper by Maurits and Iannuzzi [6].

A short overview of all settings for Algorithm 1:

- $x_{1center}$ and $x_{2center}$ are assigned initial values in a random fashion
- $A = 1$
- $T = 10$
- $\gamma = 0.1$
- $\omega_1 = \frac{2\pi}{T}$
- $\omega_2 = \frac{3\pi}{T}$

Algorithm 1 Extended LiFII version 1 for 2 dimensional variable maximization using continuous updates.

Require: $x_{1center}, x_{2center}, A_1, A_2, T, \gamma, y\vec{\omega}_1, y\vec{\omega}_2$

$\omega_1 = \frac{2\pi}{T}$, $\omega_2 = \frac{3\pi}{T}$
$t = 0$
while $t < \mathbb{T}$ **do**
  $t = t + 1$
  $x_{1,t} = x_{1center} + A_1 \cos \omega_1 t$
  $x_{2,t} = x_{2center} + A_2 \cos \omega_2 t$
  $y_t = \mathrm{f}(x_{1,t}, x_{2,t}) + noice_t$
  $y\vec{\omega}_1 = \mathrm{push}(y\vec{\omega}_1, y_t \cos \omega_1 t)$
  $y\vec{\omega}_2 = \mathrm{push}(y\vec{\omega}_2, y_t \cos \omega_2 t)$
  if $t > T$ **then**
    $y^*_{\omega_1} = (\sum y\vec{\omega}_1)/T$
    $y^*_{\omega_2} = (\sum y\vec{\omega}_2)/T$
    $x_{1center} = x_{1center} + \frac{\gamma}{T} y^*_{\omega_1}$
    $x_{2center} = x_{2center} + \frac{\gamma}{T} y^*_{\omega_2}$
  end if
end while

LiF-II ABC inspired

The pseudocode presented in Algorithm 2 represents the second extended version of LiF-II which is inspired on the Artificial Bee Colony algorithm by D. Karaboga and B. Basturk [8]. The algorithm will be making use of several parameters which can also be found as the parameters of either Algorithm 1 or Algorithm 3 in this thesis.

The first parameter included in Algorithm 2 is SN. It represents the popultaion size of the employed bees. The first part of the algorithm includes a for loop that ensures that all bees are initialized and the loop runs till the value of SN is reached. SN was set to 9 the entire experiment. The reason this parameter was set to 9 had to do with the initial starting point of each bee. At first each initial starting point was chosen at random. However, this could result in several bees starting at nearly the same spot which in turn lessens the benefit of starting multiple searches. Thus the current code ensures that each bee has a specific starting point with enough space between the following bee. For each function the starting positions are represented in figures 7 and 8. The other parameters required for this Algorithm to work are identical to the ones of Algorithm 1. Therefore the values assigned to each parameter are based on the same explanations as were given at Algorithm 1.

A short overview of all settings for Algorithm 2:

- SN = 9
- $A_1 = 1$
- $A_2 = 1$

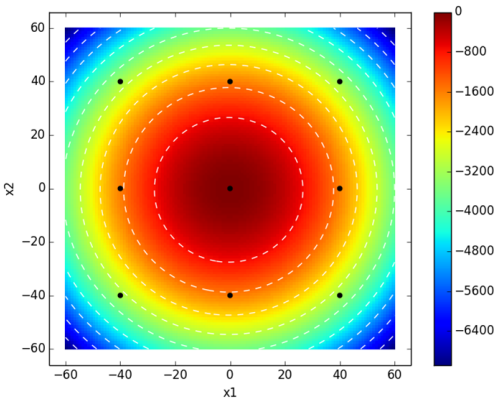

Figure 7. LIFII2 starting points for function 1

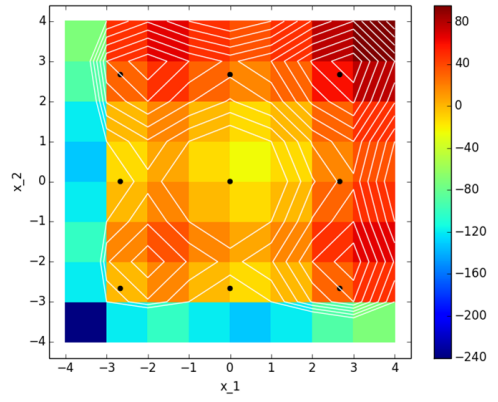

Figure 8. LIFII2 starting points for function 2

- $T = 10$
- $\gamma = 0.1$
- $\omega_1 = \frac{2\pi}{T}$
- $\omega_2 = \frac{3\pi}{T}$

Algorithm 2 Extended LiFII version 2 based on ABC algorithm for 2 dimensional variable maximization

Require: SN, A_1, A_2, T, γ, ω_1, ω_2

1: employed_bees = empty list
2: locations_visited = empty list
3: **for** $i = 1, \ldots, SN$ **do**
4: location = self.scout(location_visited)
5: employed_bees = push(LiFBee(location))
6: location_visited = push(location)
7: **end for**
8: $t = 0$
9: **while** $t < \mathbb{T}$ **do**
10: **for** bee in employed_bees **do**
11: $t = t + 1$
12: bee.$x_{1,t}$ = bee.$x_{1center} + A_1 \cos \omega_1 t$
13: bee.$x_{2,t}$ = bee.$x_{2center} + A_2 \cos \omega_2 t$
14: bee.y_t = f(bee.$x_{1,t}$,bee.$x_{2,t}$) $+ noice_t$
15: bee.$y^{\vec{}}_{\omega_1}$ = push(bee.$y^{\vec{}}_{\omega_1}$, bee.$y_t \cos \omega_1 t$)
16: bee.$y^{\vec{}}_{\omega_2}$ = push(bee.$y^{\vec{}}_{\omega_2}$, bee.$y_t \cos \omega_1 t$)
17: **if** $t > T$ **then**
18: bee.$y^*_{\omega_1} = (\sum bee.y^{\vec{}}_{\omega_1})/T$
19: bee.$y^*_{\omega_2} = (\sum bee.y^{\vec{}}_{\omega_2})/T$
20: bee.$x_{1center}$ = bee.$x_{1center} + \frac{\gamma}{T}$bee.$y^*_{\omega_1}$
21: bee.$x_{2center}$ = bee.$x_{2center} + \frac{\gamma}{T}$bee.$y^*_{\omega_2}$
22: **end if**
23: **end for**
24: **if** $t\%900 == 0$ **then**
25: (worst_bee,best_bee) = self.rank(employed_bees)
26: worst_bee.location = self.scout2(best_bee)
27: **end if**
28: **end while**

Like the Artificial Bee Colony algorithm, multiple starting points are considered in order to speed up the search in complex cases. But also to prevent getting stuck in a local maximum. In order to deal with multiple starting points, the algorithm makes use of additional functions.

The functions shown in lines 4, 5, and 6 are used to initialize the bees. The scout function requires a list of already visited locations and will then, at random, return a new location. The function also ensures that this new location has not been visited yet. Thus it should be appended to the list of visited locations, which happens at line 6. In line 5 we see that a new bee is created and appended to the list of bees named "employed_bees". We see that a LiFBee is appended, which is a class that represents a bee that will perform LiFII1 in order to maximize a two-dimensional function.

As each bee performs LiFII, the pseudocode of Algorithm 2 looks almost the same as the pseudocode of Algorithm 1. However, line 10 is included to loop over all bees. Also lines 24, 25, and 26 are included in order to evaluate all bees after each iteration. This way the algorithm can determine the current optimal values for each element of $\vec{x}$. The rank() function at line 25 will rank each bee from low to high and return a tuple with the worst bee as first element and the best bee as second element. The ranking occurs by making use of the equation:

$$Rank_{total,bee} = c_1 Rank_{observation,bee} + c_2 Rank_{update,bee} \quad (5)$$

This equation considers the value returned by the data generating function for each bee. But also the values for $y^*_{\omega_1}$ and $y^*_{\omega_2}$ calculated for each bee. These latter values determine how to update $x_{1center}$ and $x_{2center}$. As explained when $y^*_{\omega} > 0$, $\vec{x}$ is smaller than the value of $\vec{x}$ which maximizes it. But when $y^*_{\omega} < 0$, than $\vec{x}$ is larger than the value of $\vec{x}$ which maximizes it. So how closer the values $y^*_{\omega_1}$ and $y^*_{\omega_2}$ get to the value of 0, the closer the values of $\vec{x}$ get to $\vec{x}_{max}$. So when we consider the absolute value of both $y^*_{\omega_1}$ and $y^*_{\omega_2}$ we can rank bees depending on how high both values are. A bee is ranked high if it returns a high observation y_t (observation) but also still high values for both $y^*_{\omega_1}$ and $y^*_{\omega_2}$ (update). Because if both $y^*_{\omega_1}$ and $y^*_{\omega_2}$ are returning high values, this means that they still have a long way to go in reaching a value of 0. Thus the bee is still located far from the actual top, which in turn may result in even higher observations.

However, it should be noted that for this thesis it was decided to set the value of constant c_2 equal to 0 in order to ignore the values for $y^*_{\omega_1}$ and $y^*_{\omega_2}$ returned by each bee. The reason was that not sufficient time was left to explore the optimal values for both constants c_1 and c_2. Therefore c_1 was simply set to 1 and c_2 was set to 0.

Finally, after all bees have been ranked, the worst bee is reassigned a new location in the search space. This new location is near the location of the best bee, but not exactly the same spot. The precise criteria is that the new starting position should at least be 5 spots down and 5 spots to the left relatively to the spot of the best bee. But for each element of $\vec{x}$ a random value $\in \{0,\ldots,10\}$ is added.

So for example: best_bee.$x_{1,t} = 5$ and best_bee.$x_{2,t} = 5$. Now for the worst bee the new location will at least be at worst_bee.$x_{1,t} = 4.5$ and worst_bee.$x_{2,t} = 4.5$, but as for the random added value for each element of $\vec{x}$ it might as well be worst_bee.$x_{1,t} = 5.2$ and worst_bee.$x_{2,t} = 4.6$. But it is ensured that it is not the same spot as the best bee.

The entire process of evaluating all bees will only occur after every 900 iterations. This is included in line 24 to make sure that ever bee has performed 100 iterations (9 bees) before the next evaluation is performed.

Artificial Bee Colony Algorithm

Algorithm 3 represents the pseudocode for the ABC algorithm. It only requires 3 parameters which are:

- 'SN' is an interger that represents the size of the population of employed bees which the algorithm will need to initialize first. As Algorithm 2 has SN set equal to 9, this is also the case with this Algorithm.
- 'Limit' is an interger that functions as a threshold. If an employed bee has returned to the same food source for a number of iterations equeal to the limit it will need to abandon this food source and start finding a new one offered by the scout function.
- $\mathbb{T}$ represents the number of cycles the algorithm will run. Which is equal to the horizon, which in the general settings of the simulator was set to 10000.

A short overview of the used settings:

- SN = 9
- limit = 10
- $\mathbb{T} = 10000$

Algorithm 3 Artificial Bee Colony pseudocode for 2 dimensional variable maximization

```
Require: SN, limit, T
 1: employed_bees = empty list
 2: loc_visited = empty list
 3: abandoned = empty list
 4: for i = 1,...,SN do (initialize employed bees)
 5:     location = self.scout(loc_visited)
 6:     employed_bees = push(employed_bee,Bee(location))
 7:     loc_visited = push(loc_visited,location)
 8: end for
 9: t = 0
10: while t < T do
11:     for bee in employed_bees do
12:         t = t + 1
13:         bee.observe +noice_t
14:         if bee.count() > limit then
15:             abandoned = push(abandoned, bee.location)
16:             bee.location = self.scout(loc_visited)
17:             loc_visited = push(loc_visited,bee.location)
18:         else
19:             bee.location = self.scan_environment(bee)
20:             if new location found then
21:                 loc_visited = push(loc_visited,bee.location)
22:             end if
23:         end if
24:     end for
25:     best_bee = self.evaluate(employed_bees,abandoned)
26: end while
```

The ABC algorithm depends on multiple starting points which need to be chosen and assigned to a "bee". The for-loop starting at line 4 makes sure this happens. First the "scout()" function searches for a possible starting point. A point, or

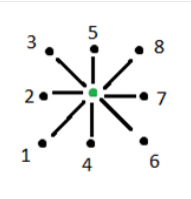

Figure 9. Neighborhood check by bees. Green represents the current position and the numbers represent the order in which the bee checks its neighbors.

location, is a combination of values for both elements of $\vec{x}$, so e.g. $x_1 = 2$ and $x_2 = 3$. The function needs a list of already visited points in order to find a new one. The scout function works entirely random by making use of the random.py library. But for every suggested random location it is also checked if it is not present in the list of already visited points.

When a location is found, it is used to assign a new bee. As can be seen in line 6, the class Bee() is called in order to create a new bee. This bee is then appended to the list of employed bees. Finally, as the new location is now engaged by a bee, the location should be appened to the list of already visited locations.

Now the iteration of the algorithm starts. Every bee has its own location, form which an observation y_t can be obtained. The first operation performed by the algorithm, directly after the observation has been memorized, is to check if the bee did not exceed the limit. A bee may only visit the same location for a certain amount of iterations. The threshold is thus the limit, which is provided as parameter.

If indeed the bee exceeded the limit, the current location is abandoned. This means that it is stored in a specific list named "abandoned", which is still used later on in the algorithm. Next a new location is searched by the scout function and the new location is also appended to the list of already visited locations.

However, if the bee did not yet exceed the limit it will perform a neighborhood scan. The bee will use the memorized observation y_t to perform the scan. Let us name it $y_{original}$ for the moment. Of every location directly adjacent to the current visited location, the observation $y_{neighbor,i}$ is retrieved and compared to $y_{original}$. When a neighbor location returns a better observation, this position is memorized until either an other location proves to be even better or all neighbors are checked. If a neighbor position did return a higher observation than the original observation, then the bee will move to this location. The entire scan process is illustrated in figure 9.

When a bee performs an environment scan it results in the execution of multiple small iterations. The reason is that every neighbor observation should count as one specific moment in time in order to still perform a sequential experiment. Thus every observation should result in the increment of t.

After all bees have been processed, an evaluation is started in order to determine the current optimal values for each element of $\vec{x}$. This evaluation is performed by comparing the observations hold by each bee. The bee holding the highest observations is ranked as the best bee. But not only the bees are taken into consideration. The abandoned locations are also considered as they might potentially hold the maximum of the function. This is not certain, but a bee may never stay at a potential maximum for longer then the determined limit. The use of a limit can also be seen in the work by Karaboga and Basturk [8].

ε-first

The ε-first pseudocode can be seen in Algorithm 4 and is composed of two different parts. The first part is the exploration part of the code. For $\varepsilon\mathbb{T}$ iterations the algorithm will perform random searches to find possible values for each element of $\vec{x}$. The values are used and y_t is observed and stored.

The function call "rand_val()" in line 4 should not be confused with known programming methods related to random number generation. In the implementation used for this thesis, "rand_val()" was a self-made funtion that would generate a random suitable value for one of the elements of $\vec{x}$. These values were suitable as the function would take into consideration the values that were already assigned to $\vec{x}$ at an earlier moment in time. And it would also take care that no values were considerded that would exceed the predefined x and y-axis of each specific data generating function. The numpy library, that includes its own "random" method, was used in order to generate these possible values for each element of $\vec{x}$.

After the first part has ended, the algorithm will start to fit a parabola using the stored observations. This can be seen in lines 8 and 9. The function "optimize.curve_fit(results)" requires the stored observations and returns the parameters for the fitted parabola. These(stored here in "info_line") are then used in order to find the top of the parabola. The values for each element of $\vec{x}$ needed to get to this maximum are then returned as solution.

Now the second part of the algorithm starts, in which the algorithm will exploit the found solution. This can be seen in lines 12 and 13. It will exploit these values for the remaining period of time, which is $\varepsilon\mathbb{T}$ till $\mathbb{T}$.

What should be noted is that this thesis uses two ε-first versions as could be seen in figure 25. This has to do with the function "optimize.curve_fit()". "curve_fit()" requires a model of a parabolic function in order to fit the parabola. It needs to know how many parameters it needs to estimate with the observations provided. It is therefore interesting to see how the algorithm works when two types of models are provided. One perfect model and one basic model.

The first version will provide the "curve_fit()" function with the best possible model parabolic function compared to the actual function being used. If a more complex function is being used, then a complex model function is provided with the exact amount of parameters present in this function. The only thing left for this version to do is to estimate the parameters of this complex function.

The second version, named ε-first$_2$, is provided with the same basic model of a parabolic function no matter how complex the actual function is. The basic model function was set to:

$$f(x_1,x_2) = -ax_1^2 + bx_1 - cx_2^2 + dx_2 + e \tag{6}$$

Thus when a more complex parabolic function is used, this version will most likely fail to approximate its maximum.

Algorithm 4 Implementation of ε-first for 2 dimensional variable maximization.

Require: $\varepsilon, \mathbb{T}$
$t = 0$
while $t < \varepsilon\mathbb{T}$ **do**
  $t = t + 1$
  $x_{1,t}$ = rand_val(), $x_{2,t}$ = rand_val()
  y_t = f($x_{1,t}, x_{2,t}$) + $noice_t$
  results = push($results, (y_t, x_{1,t}, x_{2,t})$)
end while
info_line = optimize.curve_fit(results)
solution = self.find_max(info_line)
while $t < \mathbb{T}$ **do**
  $t = t + 1$
  $x_{1,t}$ = solution[0]
  $x_{2,t}$ = solution[1]
  y_t = f($x_{1,t}, x_{2,t}$) + $noice_t$
  results = push($results, (y_t, x_{1,t}, x_{2,t})$)
end while

The following settings will be used for the parameters of Algorithm 4.

- $\varepsilon = 0.5$
- $\mathbb{T} = 10000$

Functions

The following two subsections will discuss in more detail the two data generating functions that have been used during the experiments. What should be noted is that noise is added to the values returned by the two functions. The noise is added to the returned values no matter which algorithm is currently being used. The noise is determined using the following line of code: noise = numpy.random.normal$(0.0, 0.2, 1)$. The "normal" method from the numpy library is used, which requires three paramterers. These are loc, scale, and size.

- Loc (float) is the mean of the random distribution, which is set to 0.
- Scale (float) is standard deviation for the distrubtion and is set to 0.2 for this thesis.
- Size (int or tuple of ints) is optional and stands for the number of samples that are returned. It is set to 1 for this thesis.

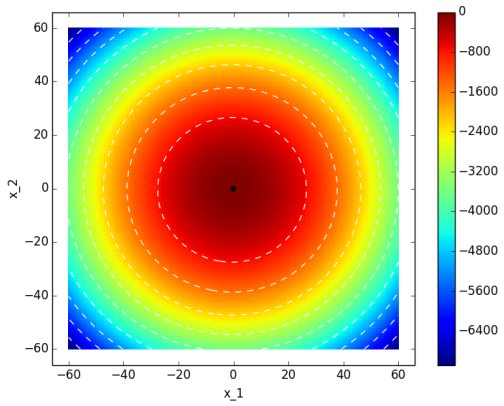

Figure 10. Plot of function 7, which represents simulated scenario 1.

Study 1

The interest of the first study was to answer the first question: How can the existing Lock-in Feedback algorithm be altered to deal with higher-dimensional problems? The data generating function corresponding to this study is illustrated in a plot in figures 10 and 11.

The function is:

$$f(x_1,x_2) = -(x_1^2 + x_1) - (x_2^2 + x_2) \tag{7}$$

It represents a 3 dimensional parabola and has only one maximum which is located at $(x_1 = 0, x_2 = 0)$. Initialization range for this function was set to $[-60, 60]$ for both elements of $\vec{x}$ during the entire experiment.

This function is included to see how the different algorithms react on two dimensional problems as they are all inspired on earlier versions used on cases of one dimension. Therefore the function is kept simple with only one maximum and symmetrically in form. Moreover, the exact coordinates of the maximum were not altered during the entire experiment.

Study 2

The second study was performed in order to answer the second research question: How does an altered Lock-in Feedback algorithm perform when applied on functions with multiple maxima? The data generating function corresponding to this study is illustrated in a plot in figures 12 and 13.

The second function is

$$f(x_1,x_2) = -(x_1+3)(x_1+1)(x_1-1)(x_1-4) - (x_2+3)(x_2+1)(x_2-1)(x_2-4) \tag{8}$$

It represents multiple 3 dimensional parabolas and has 4 maxima but with only 1 global maximum. The global maximum

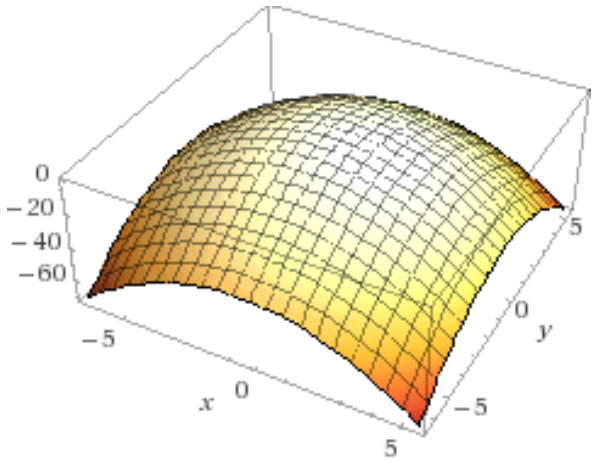

Figure 11. 3D Plot of function 7, which represents simulated scenario 1. Image source: http://www.wolframalpha.com

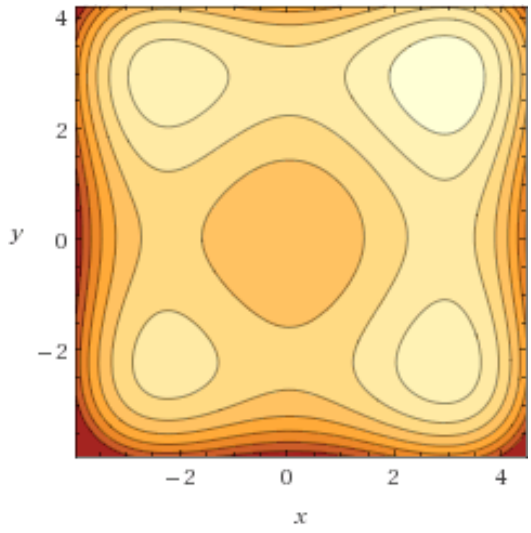

Figure 12. Plot of function 8, which represents simulated scenario 2. Image source: http://www.wolframalpha.com

is located at $(x_1 = 2.94, x_2 = 2.94)$ and the other maxima are located at $(x_1 = 2.94, x_2 = -2.22)$, $(x_1 = -2.22, x_2 = 2.94)$, and $(x_1 = -2.22, x_2 = -2.22)$. Initialization range for this function is set to $[-5, 5]$ for both elements of $\vec{x}$ during the entire experiment. This function is included to see how each algorithm handles two dimensional problems with multiple maxima. It most important role is to see if the algorithms will get stuck in local maxima. And also for this function, the exact coordinates of the maximum were not altered during the entire experiment.

Evaluation Procedure

The performance of all algorithms when applied on both functions needed to be monitored in order to be able to answer both research questions. The way these performances are monitored is by making use of the cumulative regret at each

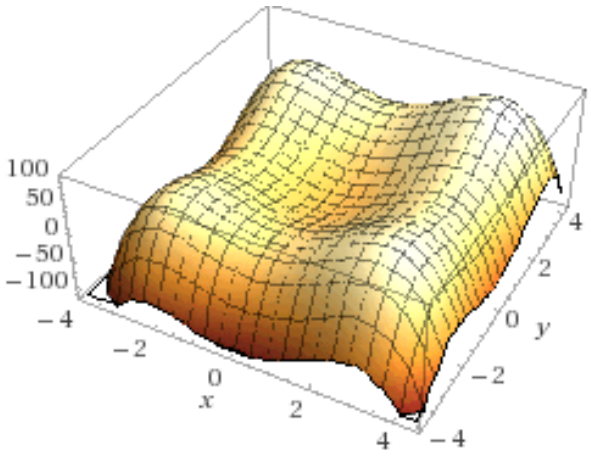

Figure 13. 3D Plot of function 8, which represents simulated scenario2. Image source: http://www.wolframalpha.com

iteration. Kaptein and Iannuzzi, when comparing the performance by their method with two existing methods, made use of emperical regret and defined it as follows [6]:

$$\mathscr{R}(t) = \sum_{t=1}^{\mathbb{T}} (f(x_{max}) - f(x_t)) \tag{9}$$

Here $f(x_{max})$ is the observed value returned by the function if x_{max} would have been picked and $f(x_t)$ is the observed value returned by the function when x_t is picked. Thus regret is the sum of the "loss" of picking a certain value for x without any knowledge about the data generating function compared to picking the optimal value for x that maximizes the function if the data generating function was known.

For this thesis it was also decided to keep track of the cumulative regret of all algorithms. As all algorithms were executed on both functions, this resulted in keeping track of two seperate values of cumulative regret for every algorithm. The cumulative regret was defined as follows:

$$\mathscr{R}(t) = \sum_{t=1}^{\mathbb{T}} (f(x_{max}) - f(x_{1,t}, x_{2,t})) \tag{10}$$

However, the way the cumulative regret was determined at each iteration for each algorithm differed.

- LiFII: For the first extended version of LiFII the cumulative regret was calculated by making use of the calculated observation $y_t + noise$ at each iteration. The noise that was added to the observation was also used in order to calculate the regret. Thus the total equation at each iteration was set to:

$$\mathscr{R}(t) = \sum_{t=1}^{\mathbb{T}} ((f(x_{max}) + noise_t) - f(x_{1,t}, x_{2,t})) \tag{11}$$

- LiFII(2): For the second extended version of LiFII the cumulative regret had to be calculated in a more complex manner.

This version includes several points that are attended. However, at each iteration only one point is observed and thus the cumulative regret should be calculated by making use of the observation of the point that is attended at each iteration t. The same equation is maintained:

$$\mathscr{R}(t) = \sum_{t=1}^{\mathbb{T}} ((f(x_{max}) + noise_t) - bee.f(x_{1,t}, x_{2,t})) \quad (12)$$

Only difference is that the observation is returned by the bee currently busy.

- ABC: The same procedure as for LiFII(2) holds. Several points are attended, but only one point at the time is truly observed at each iteration. Thus the point currently being observed should be used to calculate the cumulative regret at iteration t.

- ε-First: For both versions of the ε-First algorithm, the cumulative regret was calculated the same way. During the exploration period, the randomly chosen $x_{1,t}$ and $x_{2,t}$ were used to calculate the cumulative regret at iteration t. But during the exploitation period, the approximated x_{max} values were used for the calculation.

Moreover, the cumulative regret was also averaged. This was done as every algorithm was iterated 10000 times for 100 runs. This resulted in 100 different lists of cumulative regret for every algorithm and for every function it was applied on. As not every run returned the same data, partially because of the added noise, but also because of the design of the algorithms themselves. The logical way to evaluate the performances, was to average the data.

RESULTS

For each study performed the results were stored in seperate excel files. Every excel file held the results of one algorithm that was run on one specific function for 100 times. The excel files included the following values of interest for every algorithm for every iteration: the regret, the cumulative regret, and the noise. But the files also included the needed parameters and the calculated interim values in order to replicate the exact same test run.

Using the data collected resulted in the plots that are illustrated in figures 14, 15, 16, 17, 18, and 19. These figures depict the performance on the two data generating functions by all five algorithms used. It is expressed in the average cumulative regret over 10000 iterations.

Figure 14 shows us that for the first study, the performance by both proposed versions of the LiFII algorithm differ tremendously with the performance by ABC, ε-First, and ε-First2. When we then also look at figure 15, we notice that LiFII1 is performing even better, than LiFII2. As it does not exceed the cumulative regret of 200000, even after 10000 iterations. LiFII2, however, already exceeds the cumulative regret of 1000000 after approximately 1000 iterations.

But both LiFII1 and LiFII2 show a very steep increase in the cumulative regret in the very beginning of the iterations. This

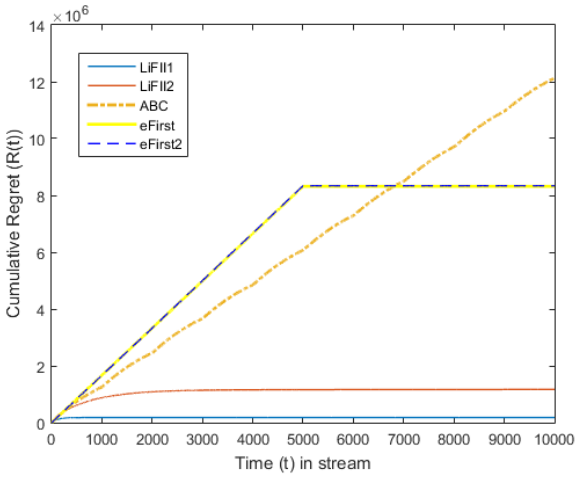

Figure 14. Overview of the mean cumulative regret of all algorithms applied on data generating function 1.

steep take-off is then also quickly followed by a more flat increase of cumulative regret. This suggests that for function 1, both algorithms tend to get near the x_{max} very soon in the iteration process. Relative to the other algorithms.

For the ABC algorithm, it is hard to say from figure 14 if it approximates x_{max} at an early or late stage. Mainly, as the results show us an almost straight line reaching high cumulative regret. Occasionally this apperent straight line, seems to show signs of flattening. This flattening occurs because of the limit set for every bee. Thus after a certain amount of time each bee tends to get close to x_{max}. If the bee then sticks with a position for more iterations than the threshold determined by the limit, it is then assigned a random new starting location. This then causes the cumulative regret to show a more steep increase again and this increase will flatten when the bees get near the x_{max} again.

Finally, for study 1 it seems that both versions of the ε-First algorithm show no additional regret after the 5000 iterations. Figure 16 was included to show otherwise. Here only the average cumulative regret of both ε-First algorithms are shown and also only from 5000 itertions till 10000 iterations. It can be seen that in fact both algorithms still show increased average cumulative regret from 5000 iterations onward. But both algorithms use a model of the data generating function which helps the algorithms approximate the x_{max}. Because for study 1 the function was not very complex, both algorithms seemed to approximate the x_{max} fairly accurate, resulting in a very flattened increase in the average cumulative regret.

Next, figure 17 includes all data on the performance of each algorithm for study 2. Again it is clearly illustrated that both versions of the LiFII algorithm show low values of cumulative regret for the duration of 10000 iterations. However, noteworthy is the fact that the two have switched in performance. Where with the results of study 1 it was clear that LiFII1 showed very low cumulative regret compared to LiFII2. Now LiFII2 shows lower values of cumulative regret over the

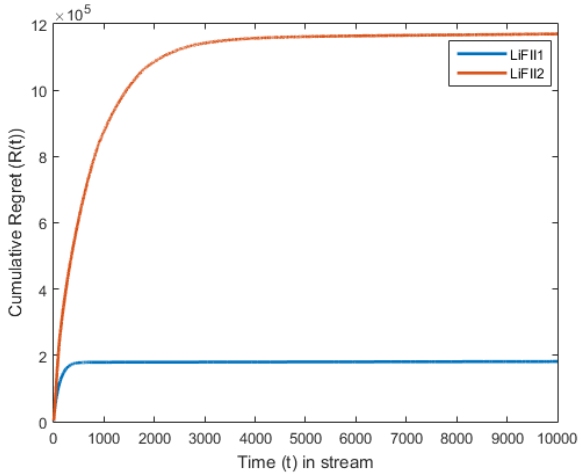

Figure 15. Overview of the mean cumulative regret of only the two proposed LiFII versions applied on data generating function 1.

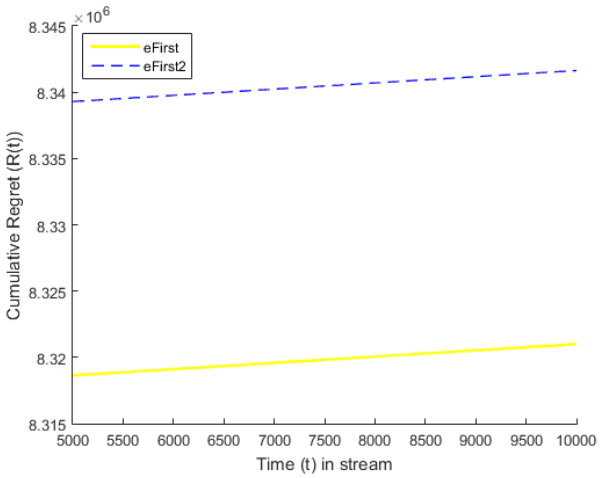

Figure 16. Overview of the mean cumulative regret of only the two ε-First versions applied on data generating function 1.

duration of the simulations for study 2. LiFII1 even seems to show straight and steep linear regret compared to the linear regret of LiFII2. The cause for this switch is partially the complexity of the data generating function for study 2 as well as the way both try to approximate x_{max}.

LiFII1 is assigend random starting positions for each run and only performs LiFII from this point onward. If this point is far from the global maximum, this algorithm may never reach it as it might get stuck in a local maximum. And because the data depicted in figure 17 shows the averaged cumulative regret, it shows straight linear regret as in a portion of the runs LiFII1 gets stuck at a local maximum. LiFII2 makes use of multiple starting positions and makes sure that after every 900 iterations the current worst position is ignored and a new position near the best current position is chosen. This ensures that eventually the algorithm will get near the global maximum.

The differences between the two algorithms is also illustrated in figure 18. Clear in this figure is that LiFII2 still shows a steep increase of cumulative regret in the first 1000 iterations. During these first iterations, LiFII1 also shows lower cumulative regret. But it quickly exceeds the cumulative regret of LiFII2 after approximately 3000 iterations.

Taking another look at figures 14 and 17, we are able to notice that the ABC algorithm seems to show the same performance for both functions. It still shows an apperent straight linear regret, but again for function 2 the cumulative regret seems to flatten for a certain amount of iterations each time. But there is a difference in the performance of the ABC algorithm on both functions. It is very clear that the ABC algorithm returns a lower cumulative regret for function 2 (approximately 650000) when compared to the cumulative regret of this algorithm for function 1 (approximately 12000000).

The ABC algorithm showed such a high cumulative regret for function 1 as it makes use of 9 points from which it tries to approximate x_{max}. But instead of ignoring the worst position after a certain amount of iterations like is the case with LiFII2, it includes a limit that prevents the algorithm from sticking to the same position for a certain threshold. Therefore it does not only take this algorithm more iterations to get to move towards the global maximum of function 1, but it has to repeatedly start from new positions and move towards the global maximum again.

This is ofcourse still the case when it is applied on function 2, but this time the function is more complex. There are more maximums and the difference between the local maximums and the global maximum is very little. Thus even if a position is visited a certain amount of iterations exceeding the limit, the algorithm now has more chance picking a new position near one of the many maximums. Resulting in a lower regret. Still, the ABC algorithm shows a higher average cumulative regret then LiFII1. While LiFII1 does not even consider multiple starting positions.

The possible reason for this difference in performance of both ABC and LiFII1 on function 2 is partially because of the complexity of the function that was also the reason for the lower

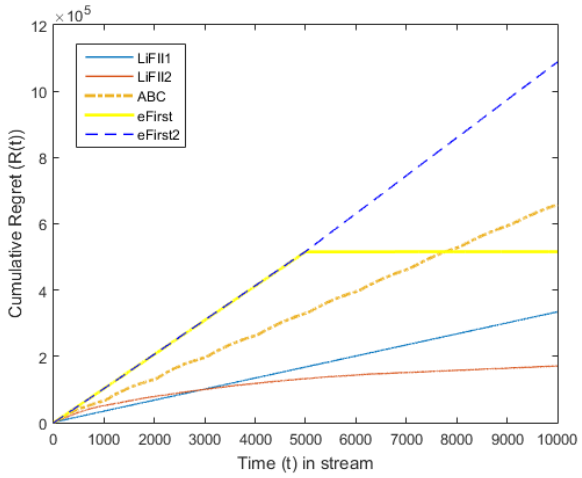

Figure 17. Overview of the mean cumulative regret of all algoritms applied on data generating function 2.

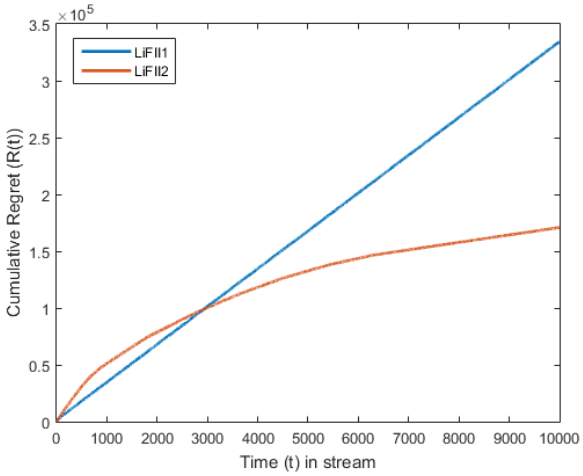

Figure 18. Overvew of the mean cumulative regret of only the two proposed LiFII versions applied on data generating function 2.

average cumulative regret for ABC on function 2 when compared with its performance on function 2. But also because of LiFII1 only considering one starting position. Again because of the complexity of the function, this starting position has now 4 possible maximums it can be near to and the differences between the maximums is not very great. Moreover, as LiFII1 only has to deal with one position in the entire simulation, it can very fast move towards one of these maximums. Therefore, LiFII1 shows lower values of average cumulative regret compared to ABC for function 2.

Finally, contrary to study 1, now only ε-First seems to show no regret after 5000 iterations. ε-First2 algorithm however, shows straight linear regret and seems to perform the worst of all included algorithms for study 2. The possible cause for the performance of ε-First2 is that it uses a simple model of the data generating function in order to approximate the x_{max} after 5000 iterations. This simple model does not fit the actual used and more complex function for study 2, but did however fit the simple function used for study 1.

Getting back at the performance of ε-First. Figure 19 was included to show that this algorithm in fact still shows an increase in average cumulative regret beyond the 5000 iterations. But compared to the cumulative regreat of the other algorithms, this increase could not be noticed in figure 17. This algorithm makes use of an accurate model of the actual used function in order to approximate the x_{max} after 5000 iterations. Which is the reason for the results showing low average cumulative regret.

The results of study 2 gave the impression that LiFII1 still performed very well when compared to the ABC algorithm and the two versions of the ε-First algorithm. But LiFII1 makes use of random starting positions each run and this means that every run on the complex function of study 2 resulted in different cumulative regret after 10000 iterations. Therefore it was decided to include boxplot 20 representing these differences in cumulative regret for every run. Here one is able to see

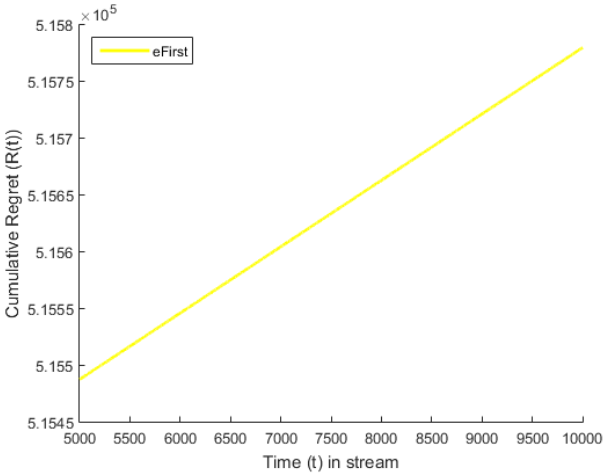

Figure 19. Overvew of the mean cumulative regret of only ε-First$_1$ applied on data generating function 2.

that while the mean cumulative regret was relatively low in comparison with the ABC and ε-first algorithms. It did return cumulative regret values for different runs that vary a lot of each other. This is suggested as for the large interquartile range of the boxplot of LiFII1. Which means that at some runs LiFII1 could have returned the cumulative regret almost equal to that of ABC and ε-First. Therefore these results suggest that LiFII1 may not be that convenient for more complex function optimization, while the cumulative regret vary tremendously for the different runs performed. Other boxplots are included as appendices.

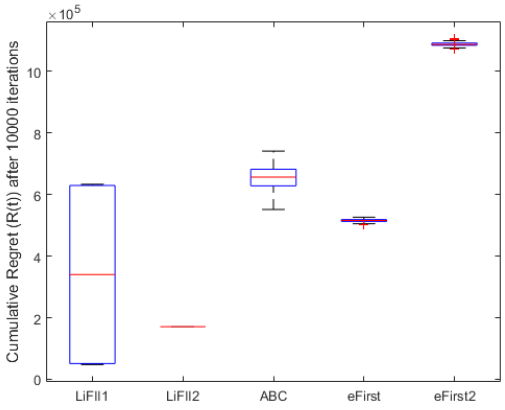

Figure 20. Boxplots for the 10000th iteration for all algorithms for study 2.

DISCUSSION AND FUTURE WORK

This section includes a discussion of the obtained results per study as well as the conclusions which can be drawn from these results. After the results of each study have been discussed, this section will consider the decisions made during the research and provide suggestions for future work.

Study 1

The results illustrated in figure 14 suggest that both extended versions of the original LiFII method perform well on a simple two-dimensional function maximization problem. Both algorithms show linear regret, but the regret flattens very fast after just a few iterations compared to the regret of the ABC, ε-First and ε-First2 algorithms. This suggest that both versions of LiFII fairly acurrately approximate x_{max} after already 1000-2000 iterations.

Meanwhile, the ABC algorithm and both ε-First algorithms still show a massive growth of cumulative regret around 1000-2000 iterations. Both versions of ε-First even show almost the same plot. As stated, the difference between the two versions of ε-First are that the first version in given an exact model of the function currently being used, while the other version gets a simple parabolic model. As the data generating function used for study 1 is a simple parabolic function, both versions of ε-First get a very accurate model that do no differ very much from each other. This results in approximately the same plot for this study.

Moreover, both versions seem to easily approximate the x_{max} of the data generating function for study 1. This can be concluded, because both versions show very little increase in cumulative regret from 5000 iteration till 10000 iterations. This is the period used by both versions to exploit the approximated x_{max}. Till this period, both versions show a massive linear growth of cumulative regret.

Likewise, the ABC algorithm shows linear regret. At first the average cumulative regret of the ABC algorithm does grow a bit slower than average cumulative regret of both versions of ε-First. But it keeps on icreasing even after the 5000th iteration. This has to do with the fact that the ABC algorithm uses bees and that each bee will explore its neighborhood in order to find a more suitable postion. The exploration of the neighborhood costs a lot of iterations and observations, resulting in a high cumulative regret. Also the fact that there are several bees observing different positions result in this fast growth of cumulative regret. Take the case where only one bee is near the top, while the other bees are not even close. They will explore a neighborhood of points that will all result in a high regret. Only the one bee near the top will return low regret.

This is also the case for LiFII2, which is inspired on the ABC algorithm. While the performance of LiFII2 in study 1 is better than the perfomance of ABC considering the average cumulative regret. It is still performing worse than LiFII1. This is depicted in figure 15. LiFII1 shows a small growth of average cumulative regret and its plot already flattens after approximately 1000 iterations.

For LiFII2 the average cumulative regret shows a higher growth and it keeps increasing until approximately 2000 iterations. The difference can be explained, because of the bees used in LiFII2. Each bee starts at a different position, with only 1 bee starting near the x_{max} as illustrated in figure 7. The other bees will return high values of regret relatively to the regret returned by the bee starting near the x_{max}. And because each bee is observed in sequence, it will result in a high increase of cumulative regret in and high amount of iterations needed for every bee to get near the top. Therefore, LiFII1 outperforms LiFII2 in terms of average cumulative regret.

But when taking a look at figures 21 and 22 we notice that the average cumulative regret returned by LiFII1 is accompanied by large varying values of cumulative regret at every 1000th iterations. Meaning that every run by LiFII1 returned vastly different results. The reason for these large values of standard devations lays in the fact that for this thesis the starting point for LiFII1 was initialized at random for each new run. As it was argued in the paper by Kaptein and Iannuzzi, the performance by LiF can be improved by picking a starting point close to the actual x_{max}. Thus, if a starting point near the x_{max} was picked and not altered for each new run, then the results should have returned cumulative regret values that would have differ a lot of each other.

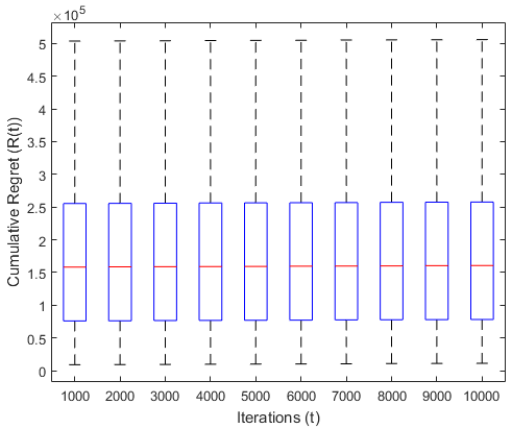

Figure 21. Boxplots for every 1000 iterations for study 1 but only of LiFII1.

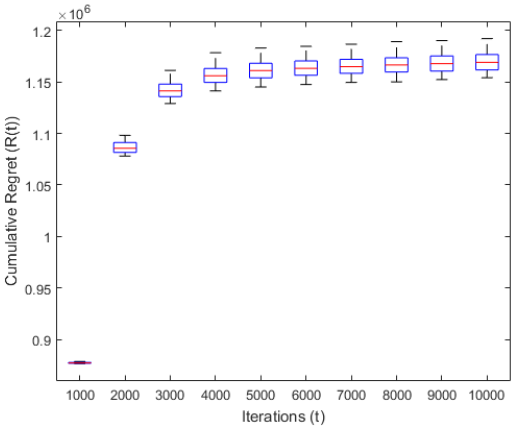

Figure 22. Boxplots for every 1000 iterations for study 1 but only of LiFII2.

Study 2

In the section Results it is already concluded that LiFII2 outperforms LiFII1 when considering the average cumulative regret over 100 runs of 10000 iterations. Also it was shown that LiFII1 showed a large interquartile range in comparison to LiFII2, ABC, and the ε-First algorithms. Meaning that LiFII1 should not be considered suitable to deal with more complex function optimization.

These differences in performances can be explained due the fact that LiFII2 uses different bees that inspect different starting points and the algorithm reassigns a bee if it is returning low observed values. The bee is then put to work near the bee currently observing the highest observed values. When taking a closer look to figure 8 it can be noticed that one of the bees is already starting near the global maximum and will thus return high values. As this bee is left alone it will move closer to the maximum each iteration that it is observing. Eventually, the other bees will be moved to a new spot near this bee and this explains the low interquartile ranges of each boxplot for LiFII2 presented in figure 23. Mainly, because the the algorithm succeeds in approximating x_{max}, which is illustrated in figures 17 and 18 by the flattening line.

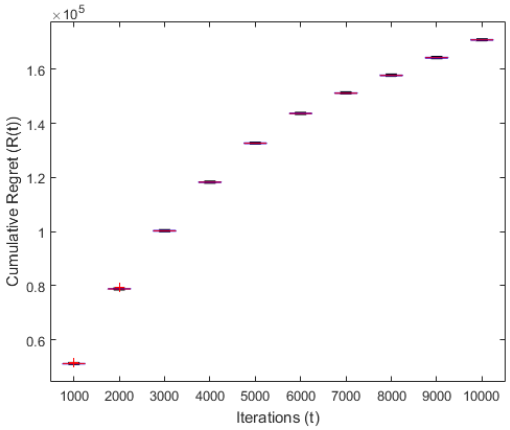

Figure 23. Boxplots for every 1000 iterations for study 2 but only of LiFII2.

The ABC algorithm, which also makes use of bees, does not show such a good performance as LiFII2. The same explanation as given for the returned results at study 1 for this algorithm holds. Each bee used by ABC needs a certain amount of iterations to check its neighborhood. This results in a lot of observations that all cause some form of regret, either low or high. But because all bees need to be attended after a certain amount of iterations and because of this neighborhood scan, this algorithm returns high values of cumulative regret for a relatively long amount of iterations when compared with LiFII2.

Also the boxplots presented in figure 24 show higher interquartile ranges for this algorithm when compared with LiFII2, mainly because the bees are initialized using random starting points. And when the limit is reached and a place needs to be abandoned, a bee is assigned a new location also at random. Which is in contrast with the technique used by LiFII2 were ther bees were assigned a new location near the currently best bee.

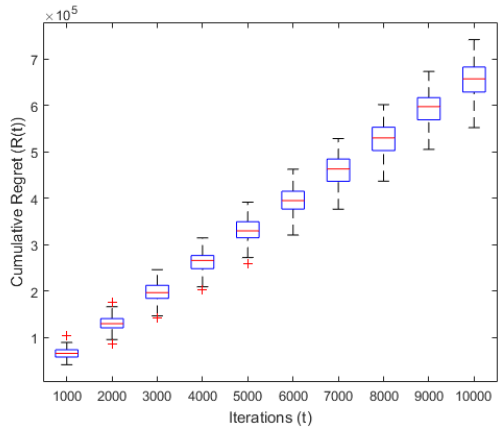

Figure 24. Boxplots for every 1000 iterations for study 2 but only of ABC.

Finally, in contrast with study 1, different results are observed between both versions of ε-First. The first version, which is given a perfect model of the function used, shows approximately the same plot of the data as was the case at study 1. After 5000 iterations the exploitation phase is entered and using this perfect model, this verson of the ε-First method is able fairly accurately approximate the x_{max}. This results in low values of additional regret beyond the 5000 iterations.

This is not the case for the second version of ε-First. The results for study 2 show linear regret for the entire period of 10000 iterations. This can be explained by the fact that this version is given a simple model of the function used and will therefore approximate a x_{max} that is still very different from the actual x_{max}. This approximation is then used beyond the 5000 iterations and results in the tremendous increase of the cumulative regret.

General Conclusions and Future Work

Several conclusions can be drawn given the results obtained by the experiments, but the main interest of this thesis was to answer the proposed research questions. When we consider the results obtained with study 1, it can be concluded that both extended versions of the original LiFII algorithm have proven to be performing well on two-dimensional function maximization problems. Thus an answer on research question 1 (How can the existing Lock-in Feedback algorithm be altered to deal with higher-dimensional problems?) is that both the suggestions made by Kaptein and Iannuzzi as well as a LiFII algorithm inspired by the ABC algorithm prove to be good alterations on the original algorithm in order to make it suitable for two-dimensional problems.

Furthermore, an answer on research question 2 (How does an altered Lock-in Feedback algorithm perform when applied on functions with multiple maxima?) is that LiFII2 knowns how to deal with a complex function consisting multiple maximums, while LiFII1 does not seem to be fit for such scenarios. If then both answers are compared, it can be concluded that LiFII2 seems to be the best possible alteration of the two proposed ideas. In both studies it showed low values of cumulative regret when compared to the other applied algorithms and it was also able to approximate the x_{max} of a function with multiple maximums.

However, several remarks can be made about these conclusions. First of all the algorithms that were included for means of comparison did not prove to be very efficient in there search for x_{max}. The ABC algorithm was constantly observing every bee it had working for it, while most of the time these bees might not even be near the actual global maximum. One suggestion for future improvement of the experiment could be to improve the way the bees scan the environment and are reassigned to a new location. This way the ABC algorithm could possible prove to be a good adversary for the LiFII2 algorithm.

Also the ε used for the ε-First algorithms during the experiment could be lowered in order to reduce the total exploration time. For this thesis the optimal value for ε has not been researched. Lowering the value could have very little effect on the approximation to be used during the exploitation phase. But it could potentially lower the total cumulative regret. As could be seen for the first version of the ε-First algorithm, it returned low values of additional regret during the exploitation phase. Therefore advancing the moment this phase starts could cause this version of ε to return a better performance in comparison to both versions of LiF.

Next, LiFII2 made use of predefined starting positions for each of the 9 bees that were used. These predefined starting positions required some sort of knowledge of the optimal search space for the data generating function that was used. Without these predefined starting positions and the required knowledge, the algorithm could have returned higher values of cumulative regret resulting in a poor performance. But the performance could then possibly be improved by experimenting with the best evaluation function for each bee. For the current experiment it was decided to neglet the information on the y^*_ω by each bee. However, as this information is used for updating the elements of $\vec{x}$ it could provide a excellent way of ranking the current positions of each bee. For future work it is therefore suggested to experiment with different evaluation functions.

Among other aspects of the experiment that provide sufficient research possibilities for future work, the settings used during the experiment could be revised. Currently each algorithm was only run for 100 times and iterated 10000 times. It was not researched what the effect of a smaller or larger amount of runs or iterations could have had on the average cumulative regret returned by each algorithm. Moreover, there was no additional research performed on using different settings and parameters for each algorithm. Most of the settings were copied directly from the work by Kaptein and Iannuzzi as well as from the work by Karaboga and Basturk. Take for example the integration time T which was set to 10 or the amplitude A which was set to 1 for the entire duration of the experiment. The adjustment of these settings or one of the other settings could also lead to totally different results. This

should therefore be researched in future work in order to find the most optimal settings for both extended versions of LiFII.

Finally, the performed research only considered two-dimensional function maximization. The proposed algorithms are thus only applicable on two-dimensional function maximization cases. If the algorithms are altered for use on cases of a dimension higher than 2, it could possibly lead to a drop in performance by both versions. Also this fact makes it difficult to give a general answer on research question 1, and instead this thesis only provides an answer for the case were the function maximization problem is of two-dimensions. For future work it is therefore suggested to alter the LiF algorithm to ensure that it can deal with a variable amount of dimensions.

REFERENCES

1. O. Dekel A. Agarwal and L. Xiao. 2010. Optimal Algorithms for Online Convex Optimization with Multi-Point Bandit Feedback. *COLT* (2010), 28–40.
2. K. Anand and R. Aron. 2003. Group Buying on the Web: A Comparison of Price-Discovery Mechanisms. *Management Science* 49, 11 (2003), 1546–1562.
3. W. Elmaghraby and P. Keskinocak. 2003. Dynamic Pricing in the Presence of Inventory Considerations: Research Overview, Current Practices, and Future Directions. *Management Science* 49, 10 (2003), 1287–1309.
4. R. Munos J. Audibert and C. Szepesvari. 2009. Exploration-exploitation Tradeoff Using Variance Estimates in Multi-Armed Bandits. *Theoretical Computer Science* 4, 10 (2009), 1876–1902. DOI: http://dx.doi.org/10.1016/j.tcs.2009.01.016
5. D. Cavagnaro J. Myung and M. Pitt. 2013. A Tutorial on Adaptive Design Optimization. *J Math Psychol.* 57, 3 (2013).
6. M. Kaptein and D. Iannuzzi. 2015. Lock in Feedback in Sequential Experiments. (2015).
7. D. Karaboga. 2005. An Idea on Honey Bee Swarm for Numerical Optimization. (2005).
8. D. Karaboga and B. Basturk. 2007. A Powerful and Efficient Algorithm for Numerical Function Optimization: Artificial Bee Colony (ABC) Algorithm. (2007). DOI: http://dx.doi.org/10.1007/s10898-007-9149-x
9. T. Lai. 1987. Adaptive Treatment Allocation and the Multi-Armed Bandit Problem. *The Annals of Statistics* 15, 3 (1987), 1091–1114.
10. M. David Lee and S. Zhang. 2010. Psychological Models of Human and Optimal Performance in Bandit Problems. (2010). DOI: http://dx.doi.org/10.1016/j.cogsys.2010.07.007
11. K. B. Monroe M. Kung and J. L. Cox. 2002. Pricing on the Internet. *Journal of Product & Brand Management* 11, 5 (2002), 274–288. DOI: http://dx.doi.org/10.1108/10610420210442201
12. J. G. March. 1991. Exploration and Exploitation in Organizational Learning. *Organization Science* 2, 1 (1991), 71–87.
13. J. Mira and J.R. Alvarez. 2005. *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*. Springer, Las Palmas, Canary Islands, Spain.
14. G. Probst S. Raisch, J. Birkinschaw and M. L. Tushman. 2009. Organizational Ambidexterity: Balancing Exploitation and Exploration for Sustained Performance. *Organization Science* 20, 4 (2009), 685–695. DOI: http://dx.doi.org/10.1287/orsc.1090.0428
15. J. H. Scofield. 1994. Frequency-domain description of a lock-in amplifier. *Am. J. Phys.* 62, 2 (1994), 129–133.
16. O. Shamir. 2013. On the Complexity of Bandit and Derivative-Free Stochastic Convex Optimization. *Workshop and Conference Proceedings* 30 (2013), 1–22.
17. J.M. White. 2013. *Bandit Algorithms for Website Optimization*. O'Reilly Media, Gravenstein Highway North, Sebastopol, USA.

APPENDICES

Simulator

The simulator used for this thesis includes a GUI which is shown in figure 25. The two leftmost columns of buttons and also the button labeled "LiFII2" have as primary function to change the current settings. The buttons in the first column may be used to change the current data generating function. For example: by clicking on the button labeled "f_1(x_1,x_2)", the current function will be changed to function 1.

The same story holds for the middle column of buttons and the button labeled "LiFII2". Only this time, by clicking on one of these buttons, the current algorithm being used will be changed to the algorithm corresponding to the button that has been clicked. For example: by clicking on the button labeled "LiFII", the current algorithm will be set to LiFII.

The rightmost column includes buttons to run a simulation test. The button labeled "start" will simply start the iteration of the chosen algorithm on the chosen function. When the start button has been pressed, it will only become available to press again after the simulation test has been finished. When a test run is finished, the button labeled "Answers" may be pressed in order for the GUI to present the results of the test in the white text field which is located just above the buttons. For now the "Answers" button will only show the name of the algorithm that has been used and the values for each element of x for which the algorithm calculated that they would provide x_{max}. However, this button was only included to be able to check for bugs and to see if the algorithms worked as they were supposed to work. Currently the button lost his functionality and was only left there to make it easy for future testing to check for bugs.

The function of the last button of this column was to reset all variables used during a run. This button was included to make sure that a new simulator test would not start with the values

returned by the previous run. But the button does not reset the settings picked by the user. All variables used are simply set to their initial values. But also this button lost his functionality when the final tests were run. Mostly because for the simulator to start several runs in sequence it should itself know when to reset the values of each variable, without an user constantly having to click the reset and start button. Thus the current code includes a loop making sure that this is the case and the reset button was only left there for future testing for bugs.

What should also be noted is that a simulator run can already be started directly after the GUI has been launched. This is enabled as the code includes initiale settings for the algorithm and function to be used during a run. The initiale algorithm is set to "LiFII" and the initial funciton is set to "function 1".

This simulator design was chosen as it led to a neatly structured implementation of the code. Moreover, the GUI ensured that multiple runs with different algorithms and on both function could be started without having to change the code constantly.

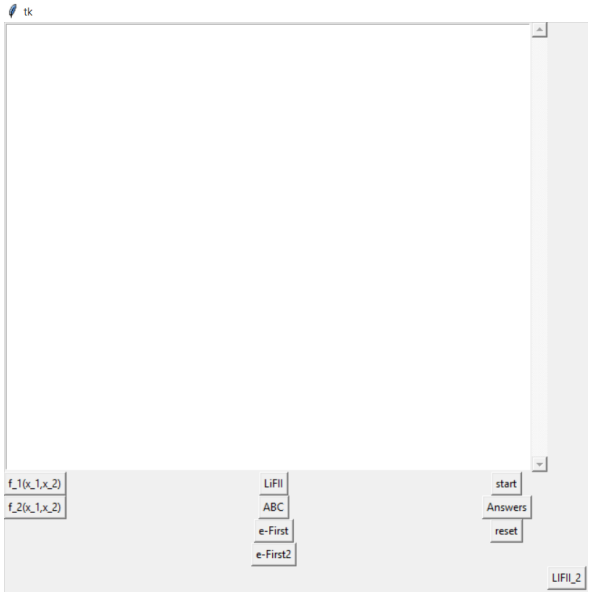

Figure 25. Simulator GUI

Boxplots Study 1

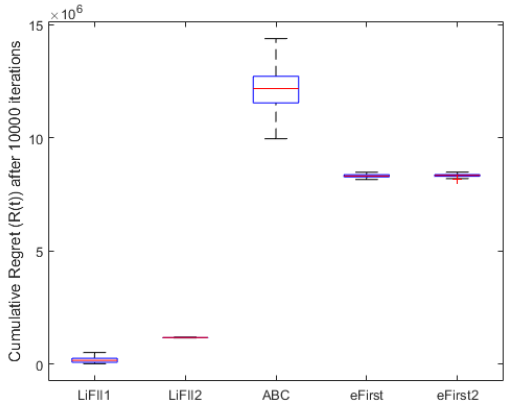

Figure 26. Boxplots for the 10000th iteration for all algorithms for study 1.

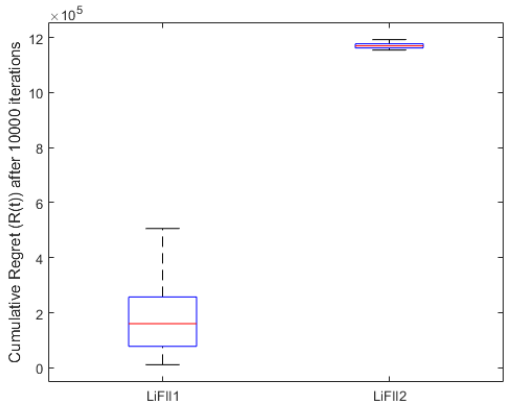

Figure 27. Boxplots for the 10000th iteration of only LiF for study 1.

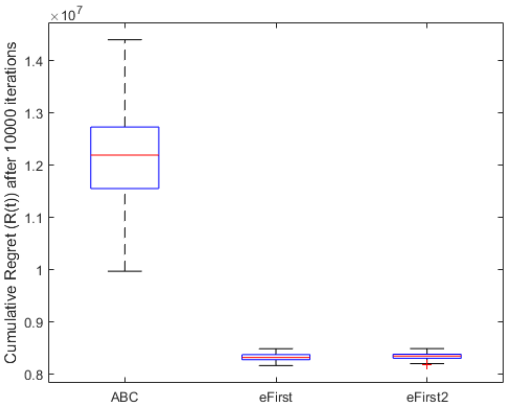

Figure 28. Boxplots for the 10000th iteration but not of LiF for study 1.

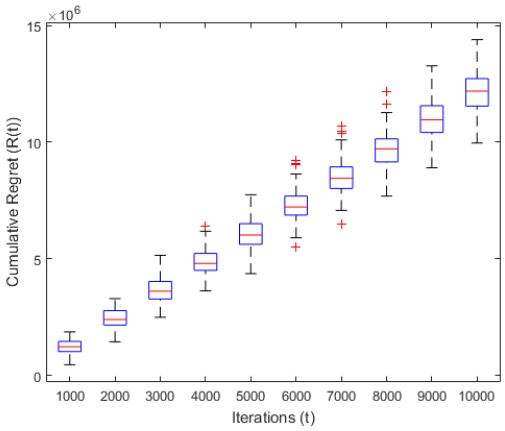

Figure 29. Boxplots for every 1000 iterations for study 1 but only of ABC.

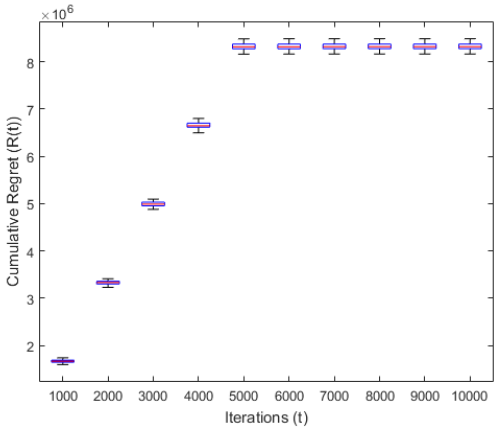

Figure 30. Boxplots for every 1000 iterations for study 1 but only of ε-First.

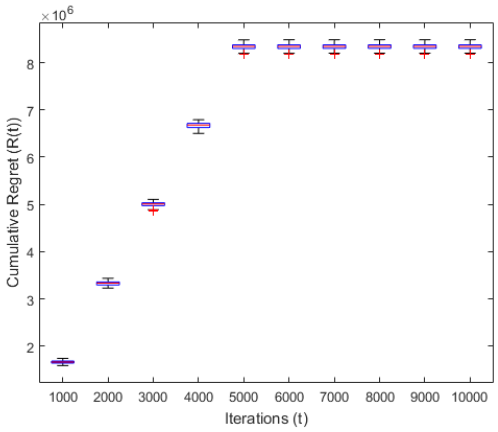

Figure 31. Boxplots for every 1000 iterations for study 1 but only of ε-First2.

Boxplots Study 2

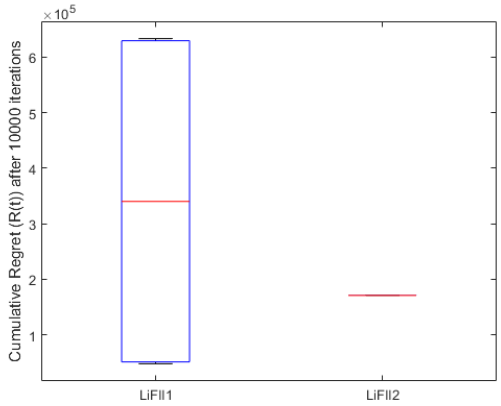

Figure 32. Boxplots for the 10000th iteration of only LiF for study 2.

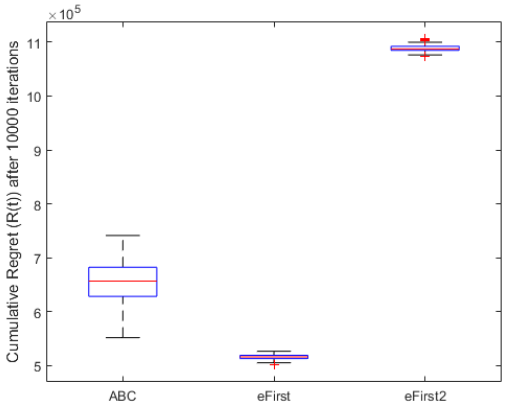

Figure 33. Boxplots for the 10000th iteration but not of LiF for study 2.

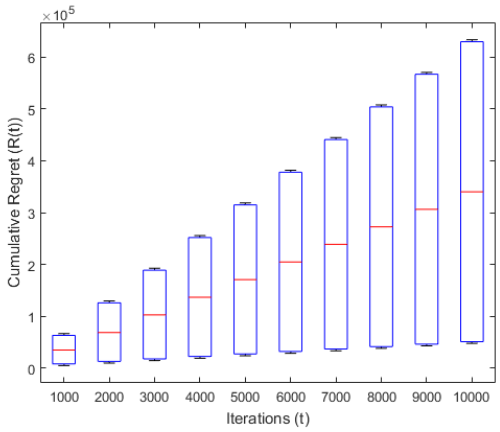

Figure 34. Boxplots for every 1000 iterations for study 2 but only of LiFII1.

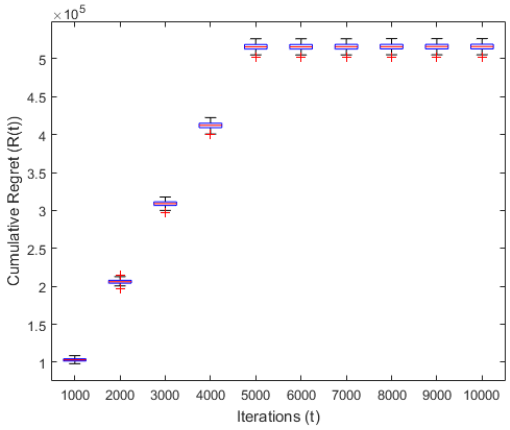

Figure 35. Boxplots for every 1000 iterations for study 2 but only of ε-First.

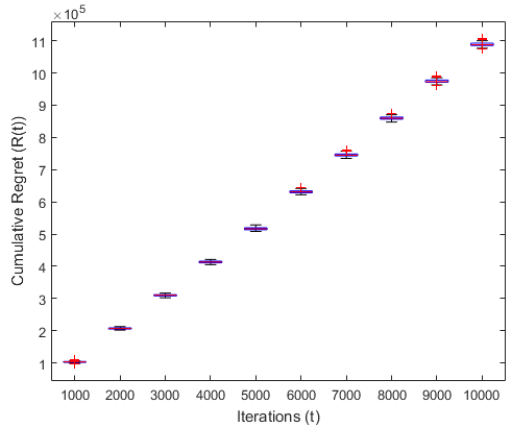

Figure 36. Boxplots for every 1000 iterations for study 2 but only of ε-First2.

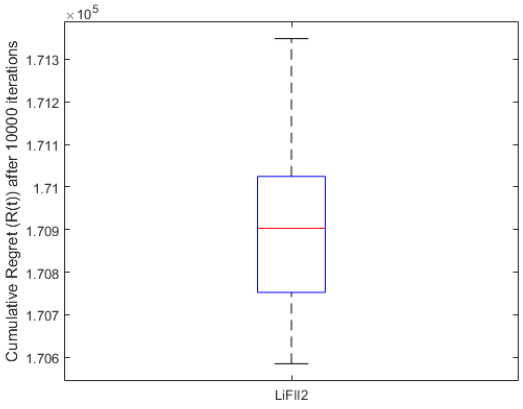

Figure 37. Boxplot for the 10000th iteration of only LiFII2 for study 2.