View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Michigan Technological University

Michigan Tech
Create the Future

Michigan Technological University
Digital Commons @ Michigan Tech

Dissertations, Master's Theses and Master's Reports - Open

Dissertations, Master's Theses and Master's Reports

2013

A COMPARATIVE STUDY OF HEURISTIC OPTIMIZATION ALGORITHMS

Rohit A. Bhatia
Michigan Technological University

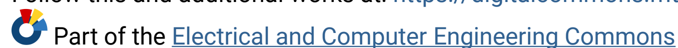
Follow this and additional works at: https://digitalcommons.mtu.edu/etds

Part of the Electrical and Computer Engineering Commons

Copyright 2013 Rohit A. Bhatia

Recommended Citation
Bhatia, Rohit A., "A COMPARATIVE STUDY OF HEURISTIC OPTIMIZATION ALGORITHMS", Master's report, Michigan Technological University, 2013.
https://digitalcommons.mtu.edu/etds/702

Follow this and additional works at: https://digitalcommons.mtu.edu/etds


Part of the Electrical and Computer Engineering Commons

A COMPARATIVE STUDY OF HEURISTIC OPTIMIZATION ALGORITHMS

By

Rohit A. Bhatia

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Electrical Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2013

© 2013 Rohit A. Bhatia

This report has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Electrical Engineering.

Department of Electrical and Computer Engineering

Report Advisor: *Dr. Ashok Goel*

Committee Member: *Dr. Zhuo Feng*

Committee Member: *Dr. Laura E. Brown*

Department Chair: *Dr. Daniel R. Fuhrmann*

Table of Contents

Acknowledgements 4

Abstract 5

1. Heuristic algorithms - Introduction 6

2. Non-adaptive simulated annealing

    2.1 Overview 9

    2.2 Working of the non-adaptive simulated annealing algorithm 11

    2.3 Pseudocode 13

    2.4 Parameterization 16

    2.5 Algorithm output 18

3. Adaptive simulated annealing

    3.1 Motivation 20

    3.2 Working of the adaptive simulated annealing algorithm 21

    3.3 Pseudocode 23

    3.4 Algorithm output 25

4. Random restart hill climbing

    4.1 Overview 27

    4.2 Working of the random restart hill climbing algorithm 28

    4.3 Pseudocode 30

    4.4 Algorithm output 33

5. Performance comparison

    5.1 Performance comparison with a cost function of two variables 35

    5.2 Performance comparison with a cost function of three variables 39

    5.3 Performance comparison with a cost function of four variables 43

6. Conclusions 47

Appendix 1 48

References 74

Acknowledgements

I am grateful to my advisor Dr. Ashok Goel for his continuous support and guidance that fostered my learning in the subject area and helped me develop a strong understanding of the matter. Without Dr. Goel's support, this would have been an enormously difficult task to implement.

Additionally, I am grateful to Dr. Zhuo Feng and Dr. Laura E. Brown for being part of my defense committee and providing me their valuable feedback so as to get the most from my efforts.

I am also glad to have had access to resources at Michigan Technological University that made my learning experience very joyous and fulfilling.

Finally, I am extremely grateful to my family and friends, and my parents in particular for their unconditional support especially in times when I needed it.

Abstract

Heuristic optimization algorithms are of great importance for reaching solutions to various real world problems. These algorithms have a wide range of applications such as cost reduction, artificial intelligence, and medicine. By the term cost, one could imply that that cost is associated with, for instance, the value of a function of several independent variables. Often, when dealing with engineering problems, we want to minimize the value of a function in order to achieve an optimum, or to maximize another parameter which increases with a decrease in the cost (the value of this function). The heuristic cost reduction algorithms work by finding the optimum values of the independent variables for which the value of the function (the "cost") is the minimum.

There is an abundance of heuristic cost reduction algorithms to choose from. We will start with a discussion of various optimization algorithms such as Memetic algorithms, force-directed placement, and evolution-based algorithms. Following this initial discussion, we will take up the working of three algorithms and implement the same in MATLAB.

The focus of this report is to provide detailed information on the working of three different heuristic optimization algorithms, and conclude with a comparative study on the performance of these algorithms when implemented in MATLAB. In this report, the three algorithms we will take in to consideration will be the non-adaptive simulated annealing algorithm, the adaptive simulated annealing algorithm, and random restart hill climbing algorithm. The algorithms are heuristic in nature, that is, the solution these achieve may not be the best of all the solutions but provide a means to reach a quick solution that may be a reasonably good solution without taking an indefinite time to implement.

1. Heuristic algorithms - Introduction

There are several algorithms at our disposal for cost reduction. We use the term "cost reduction" since we often want to minimize the value of a function of several independent variables. Such a reduction may be desired in order to either reach an optimum value, or to maximize another parameter of interest which increases with this decreasing cost. To name a few, Memetic algorithms, simulated annealing, force-directed placement, and evolution-based placement are some of the common algorithms that are used in cost reduction. The term "placement" broadly refers to the intermediate solution (or result) attained at an iterative step as the algorithm executes.

> "Placement algorithms can be divided into two major classes: constructive placement and iterative improvement. In constructive placement, a method is used to build up a placement from scratch; in iterative improvement, algorithms start with an initial placement and repeatedly modify it in search of a cost reduction. If a modification results in a reduction in cost, the modification is accepted; otherwise it is rejected." [1]

These algorithms are heuristic in nature. That is, these algorithms produce a solution that is good enough for arriving to a solution to the problem at hand. There are parameters that are used for "tuning" these algorithms to arrive at a solution in the shortest possible time. These parameters are specific to individual algorithms that are used. Often, there is a tradeoff between the speed of execution of the algorithm and the accuracy of the result obtained.

There is a class of hybrid algorithms that combine evolutionary algorithms (evolution-based placement algorithm listed above is a kind of evolutionary algorithm) and local searches and result in what we call as Memetic Algorithms.

> "Memetic Algorithms are class of stochastic global search heuristics in which Evolutionary Algorithm based approaches are combined with problem-specific solvers. Later local search heuristics techniques are implemented. This hybridisation is to either accelerate or to discover good solution from the population where the evolution alone would take long time to discover or to reach the solution. Memetic

Algorithms use heuristic local searches either approximate method or exact method to get the local refined solution from the population." [2]

Authors in [1] describe the force – directed placement with an analogy to the Hooke's law for the force exerted on stretched springs. While placing the modules, we assume that the modules are connected by a net which exerts an attractive force between them. The magnitude of this force is directly proportional to the distance between the modules. If the modules are allowed to move freely in the system, they would continue to move until they settle down at positions where there is a zero resultant force on each module and the system achieves a minimum energy state. "Hence, the force-directed placement methods are based on moving the modules in the direction of the total force exerted on them until this force is zero." [1] Authors have used the term "modules" for a specific application but it is equivalent to considering these modules as results of an optimization step.

Algorithms that fall in the genetic algorithm category have been inspired by the natural process of evolution. One such algorithm is the Simulated Evolution algorithm (SimE), which is a general search strategy for solving a variety of combinatorial optimization problems. It operates on a single solution, termed as population, and each population consists of elements. The algorithm has three basic steps in one main loop, namely, Evaluation, Selection, and Allocation. In the first step, the goodness of each element is measured as a single number between '0' and '1' which is indicative of how near the element is from its optimal solution. Following this step, Selection is carried out where unfit elements (elements that are far from their optimal solution) are selected in the current solution. It is because of Selection step that SimE does not get "trapped" at local minima since unfit elements are allowed to be a part of the intermediate solution. The last step is Allocation. The purpose here is to mutate the population by altering the current solution. This step has a high impact on the quality of the solution. [3]

For the purpose of this report, we will consider three algorithms that have been used for cost reduction optimizations. Namely, non-adaptive simulated annealing, adaptive simulated annealing, and random restart hill climbing will be evaluated in detail. The evaluation will cover the working of the algorithms, the pseudocode, MATLAB based

implementations of these algorithms, and presentation of the performance comparison results from their implementation.

2. Non-adaptive simulated annealing

2.1 Overview

Simulated annealing is a very time consuming algorithm but yields excellent results. The algorithm derives its name from metallurgy. Authors in [1] draw an interesting analogy between how the algorithm works and how metals are allowed to cool down so as to mold them in to the desired shape. A metal that is stressed has an imperfect crystal structure. How we bring about the metal to the desired form is to first heat the metal to a high temperature then cool it down gradually. In metallurgy, we refer to this as annealing.

At higher temperatures, the atoms in the metal have sufficient kinetic energy to break loose from their current incorrect positions. As the material cools down, the atoms slowly start getting trapped at the correct lattice positions. However, if we cool down the metal rather quickly, the atoms do not get a chance to get to the correct lattice positions and defects (due to the atoms being at the incorrect positions) become part of the crystal structure.

Simulated annealing algorithm does just that. In this algorithm, we start with an initial temperature and a starting configuration, or an initial guess for the solutions that would yield an optimal result. On successive iterations, we reduce the temperature and determine a configuration that results in an improvement over the current solution. We continue to reduce the temperature until we have reached a stopping temperature. The details on the working of this algorithm will be discussed in the following section.

It is imperative that the algorithm reaches the global minima of the function and does not get stuck at a local minima which may not yield the absolute minimum value, in our case, the cost. "Simulated annealing is a stochastic method to avoid getting stuck in local, non-global minima, when searching for global minima. This is done by accepting, in addition to transitions corresponding to a decrease in function value, transitions corresponding to an increase in function value. The latter is done in a limited way by means of a stochastic acceptance criterion. In the course of the minimization process, the probability of accepting deteriorations descends slowly towards zero." [4]

For the purpose of demonstrating the working of this algorithm as implemented in MATLAB, we will work with a function which we can assume determines the cost in terms of the x and y values of the intermediate solution. The aim is in reaching the global minima of this function starting with some initial guess of the x and y independent variables. The intent of doing so is to demonstrate the working of this algorithm without focusing on the function itself since that may vary depending on the application.

2.2 Working of the non-adaptive simulated annealing algorithm

We will look at the working of this algorithm over two sections. We present an abstract understanding of how the algorithm works here in this section. In the next section, we look at the pseudocode and detail how the algorithm proceeds to achieve the global minima of a given function.

To begin with, we start with an initial guess and a starting temperature for the working of this algorithm. Specific to our implementation, the initial guess is the x and y values of the intermediate solution (or the starting point). We also fix the starting temperature for the implementation of the algorithm. Once these are fixed, we set other parameters like cooling schedule, maximum consecutive rejections, stopping temperature, etc. We will discuss the relevance of each of these parameters in a following section.

The simulated annealing algorithm starts with accepting all moves but with a probability of accepting the move. At higher temperatures, the probability of accepting a move is higher. However, this probability decreases as the temperature decreases. The moves that cause a cost increase are accepted with a probability that decreases with the increase in cost.

In most implementations of this algorithm, the acceptance probability is given by $\exp(-\Delta C/T)$, where ΔC is the cost increase. Initially, the temperature is set to a very high value so most of the moves are accepted. Acceptance (or rejection) of a move is determined by comparing the acceptance probability to a random probability value between 0 and 1. At each iteration, the temperature is gradually reduced so the cost increasing moves are less likely to be accepted. Toward the end, only moves that cause a cost reduction are accepted and the algorithm converges to a low cost configuration. [1]

The fact that moves that result in a cost increase are also accepted (even though with a lower probability) ensures that the algorithm does not get “stuck” at a local minima and has a “fair” chance of covering all minima before reaching the global minima.

It must be pointed out that we use the term “non-adaptive” with our application of the simulated annealing algorithm. This simply means that the cooling schedule is fixed. We do not vary the cooling schedule on successive iterations. In case of the adaptive simulated annealing algorithm, the cooling schedule will be adaptive, that is, it will vary as the algorithm executes. The adaptive version of the simulated annealing algorithm will be covered in the next section.

Following section will give the reader a better understanding of the working of this algorithm since we take the pseudocode of the algorithm in to account.

2.3 Pseudocode

The simulated algorithm works as per the following pseudocode shown in Figure 2.3.1

```
PROCEDURE Simulated_ Annealing;
        initialize;
        generate random configuration;
        WHILE stopping_criterion (loop_count, temperature) = FALSE
                WHILE inner_loop_criterion = FALSE
                        new_configuration ← perturb(configuration);
                        ΔC ← evaluate(new_configuration, configuration);
                        IF ΔC < 0 THEN new_configuration ← configuration;
                        ELSE IF accept(ΔC, temperature) > random(0, 1)
                                THEN new_configuration ← configuration;
                                ENDIF
                        ENDIF
                ENDWHILE
                temperature ← schedule(loop_count, temperature);
                loop_ count ← loop_ count + 1;
        ENDWHILE
END.
```

Figure 2.3.1: Pseudocode for the Simulated Annealing algorithm [1]

We will now analyze the algorithm in greater detail. We first start with the initialization using “initialize.” The initialization involves setting the following parameters

1. *Schedule*: The cooling schedule. This determines the temperature decrements on successive iterations.
2. *MaxConsecutiveRejections*: Maximum number of consecutive rejections.
3. *MaxSuccessAtTemperature*: Maximum number of successful moves at a given temperature.
4. *RandomGenerator*: This generates a random configuration from an existing configuration.
5. *InitialTemperature*: The starting temperature for the simulated annealing algorithm.
6. *StoppingTemperature*: The stopping temperature for the simulated annealing algorithm. Algorithm exits implementation once this temperature has been reached.

7. *MaxTriesAtTemperature*: Maximum number of moves that are permitted at a particular temperature.
8. *StoppingValue*: The stopping value for the simulated annealing algorithm. Algorithm exits implementation once this value of the function has been reached.

Following initialization, we generate a random configuration. In case of cost estimates, this can be computed using the function that determines the cost based on the x and y values of the solution. This is the "cost function." In our case, this value equals the initial cost which we wish to minimize.

Iterative improvements are carried out next. Until the temperature doesn't reach the stopping temperature or until we haven't reached the limit for maximum successive rejections as set by the *MaxConsecutiveRejections* parameter, we perform moves at each temperature. The number of moves at each temperature depends on parameters like *MaxTriesAtTemperature* and *MaxSuccessAtTemperature*.

We now consider a static situation when we are accepting or rejecting moves at a particular temperature. This implies that we have not reached the set limits of *MaxTriesAtTemperature* and *MaxSuccessAtTemperature* parameters. This is what we call the "inner loop criterion." The new configuration is reached by perturbing the existing configuration.

Perturbing the existing configuration is done in a two-step process. First, we generate random values for the x and y variables in the neighborhood of the current values. Next, we plug in these values in to the cost function that determines the cost based on the new values. The difference in costs in evaluated to determine if the new values result in a decrease, or an increase in the cost.

There are two scenarios that arise from the perturbation. If the cost decreases, then the move is accepted without any consideration to the acceptance probability given by exp(-ΔC/T) where ΔC is the cost increase and T is the current annealing temperature. However, if the cost increases, the move is still accepted with consideration to the

acceptance probability. This is what makes the simulated annealing algorithm so special – it does not get stuck at a local minima.

Accepting (with or without consideration to the acceptance probability) or rejecting moves is done at a particular annealing temperature. Next, we reduce the temperature and follow the same procedure as in the previous step. The algorithm concludes execution if either the temperature is less than the stopping temperature or if the number of consecutive rejections has reached the preset.

.

2.4 Parameterization

This section details the parameters we have used for the implementation of the Simulated Annealing algorithm. The parameters are generally tuned depending on the application.

1. *Schedule*: Since this is the non-adaptive simulated annealing algorithm, we use a fixed value for the cooling schedule. The temperature at the next step is 0.9 times the current temperature.
2. *MaxConsecutiveRejections*: Maximum number of consecutive rejections is set to 1,000.
3. *MaxSuccessAtTemperature*: Maximum number of successful moves at a given temperature is set to 20.
4. *RandomGenerator*: This generates a random configuration from an existing configuration. This is set to the anonymous function:

   *@(param) (param+(randperm(length(param))==length(param))*randn/100)*

   Where *param* is a two-input vector with the current x and y values from the solution. The output is again a two member vector with the updated x and y values generated using randperm that generates random permutations of integers 0 and 1. This is then multiplied by *randn* that generates normally distributed random number and is divided by 100 to keep the new x and y values within the neighborhood.
5. *InitialTemperature*: The starting temperature for the simulated annealing algorithm is set to 1
6. *StoppingTemperature*: The stopping temperature for the simulated annealing algorithm is set to $1e^{-8}$. Algorithm exits implementation once this temperature has been reached.
7. *MaxTriesAtTemperature*: Maximum number of moves that are permitted at a particular temperature is set to 300.

8. *StoppingValue*: The stopping value for the simulated annealing algorithm is set to 1. Algorithm exits implementation once this value of the function has been reached. In our application, this is a reasonable value for the minimum cost.

Appendix 1 includes the complete code for the non-adaptive simulated annealing algorithm.

2.5 Algorithm output

The non-adaptive simulated algorithm is implemented using a sample cost function $2x^2 - 4xy + y^4 + 2$. A 3-Dimensional graph of function f shows that f has two local minima at (-1,-1,1) and (1,1,1) and one saddle point at (0,0,2) [5]. This is shown in Figure 2.5.1.

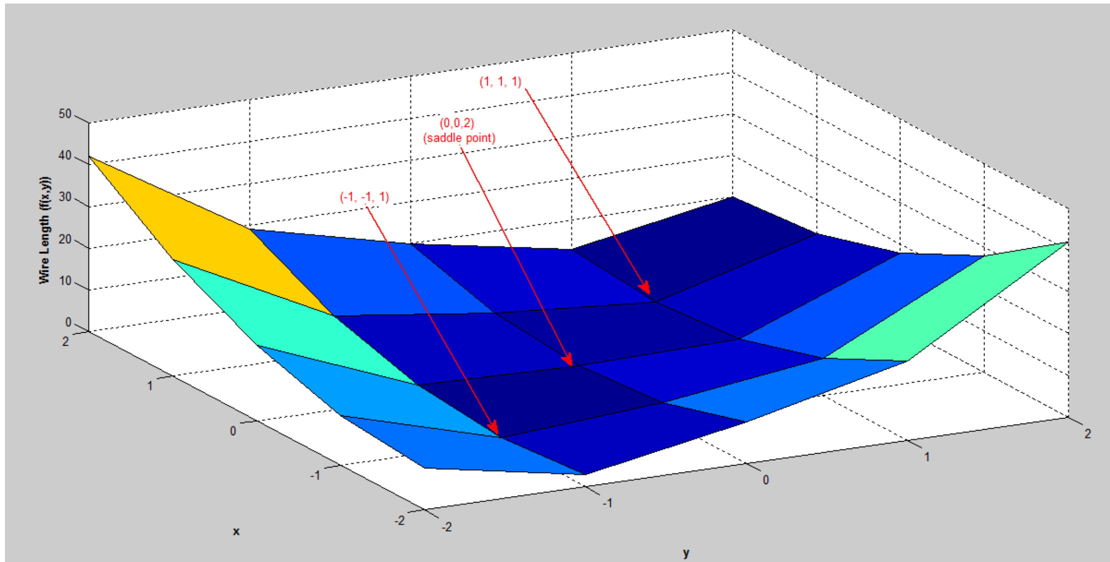


Figure 2.5.1: Cost function used for simulated annealing implementation [5]

We enter a vector as an initial guess and also a function handle so that the cost can be computed at subsequent iterations. The initial guess is a vector that contains the starting x and y values. This is shown in Figure 2.5.2

```
testFunction = @(x,y)(2*x.^2 - 4*x*y + y.^4 + 2);
ruleFunction = @(c)testFunction(c(1),c(2));          % function handle
initialGuess = [1 5];                                % initial guess for x an
tempUserOpt = [];
for performaceComp = 1:1:comparisonLimit
    [SimulatedAnnealingResults] = anneal_rev1(ruleFunction,initialGuess);
```

Figure 2.5.2: Passing the initial guess and the cost function handle in to the simulated annealing algorithm

Once the algorithm is run and it completes execution, we see that the solution converges to the minima (the minimum cost) and we get the optimal x and y values for the solution.

The annealing algorithm waveforms at successive iterations and the final output are shown in Figures 2.5.3 and 2.5.4 respectively.

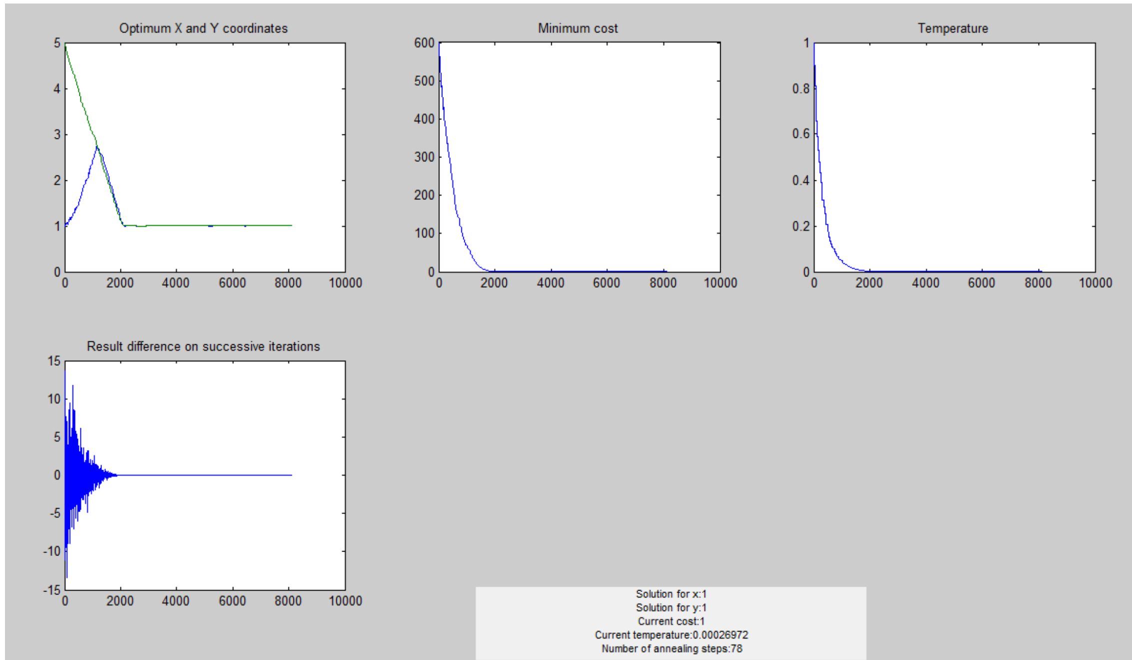


Figure 2.5.3: Simulated annealing algorithm waveforms at successive iterations

Solution for x:1
Solution for y:1
Current cost:1
Current temperature:0.00026972
Number of annealing steps:78

Figure 2.5.4: Output of the simulated annealing algorithm with the minimum cost and the x and y values for the solution

The results conform to the known minima for the given function. As is evident in Figure 2.5.4 the algorithm completes execution once the maximum number of consecutive rejections has been reached and not (in this case) because of having reached the minimum temperature.

3. Adaptive Simulated Annealing

3.1 Motivation

Iterative improvement algorithms such as Simulated Annealing produce accurate results at the cost of enormous computation time. Such time-cost considerations encourage us to seek other algorithms that are more efficient. One such enhancement is the Adaptive Simulated Annealing (ASA) algorithm. ASA reduces the computation time required to reach a solution at the cost of some loss in accuracy of the solution.

Though at times it might not seem like a significant improvement in the computation time for relatively simple computations, the computation time is significantly improved for complex computations.

Another motivation in choosing the ASA is that we have the choice of going for a parameter-free simulated annealing algorithm. Such an advantage is crucial in that we can avoid having to deal with setting parameters which ultimately determine both the computation time and the accuracy of the results. This benefits the user with results that have much less dependence on the parameters, since just a few parameters are used for this form of ASA.

What makes ASA different from SA (Simulated Annealing) is that since ASA does not require the implementer to tune any parameters, a feedback mechanism is used to adjust the annealing temperature rather than using a fixed cooling schedule as in the case of SA. The parameters that are set and adjusted are the temperature and the acceptance rate. Over the subsequent sections, we will look in to the details of how these parameters are set, and the working of the ASA.

3.2 Working of the adaptive simulated annealing algorithm

The Simulated Annealing (SA) algorithm works by reaching a state of thermal equilibrium to yield globally optimal solutions. This requires a series of annealing steps that cool down the temperature. Often, the cooling results in a prohibitively long computation time.

> "Lam and Delosme proposed an approximate thermal equilibrium they call D-equilibrium which balances the trade-off of required computation time and the quality of the solution found by the run of SA. Under certain assumptions about the forms of the distribution of the cost values and the distribution of cost value changes, they analyzed their model and determined the annealing schedule that maintains the system in D-equilibrium (i.e., the annealing schedule that optimally balances the computational cost / solution quality trade-off). This "optimal" annealing schedule adjusts the temperature based on the parameter λ which controls the cost-quality trade-off and more importantly based on the current rate of accepted moves. Analyzing their annealing schedule, Lam and Delosme determined that the temperature is reduced the quickest when the probability of accepting a move is equal to 0.44." [6]

After having made this observation that a faster cooling rate led to a shorter annealing run, the size of the neighborhood considered for the moves was allowed to fluctuate to match this target move acceptance rate of 0.44 as closely as possible. The idea behind this is to either increase the acceptance rate by decreasing the maximum distance from the current state or to decrease the acceptance rate by increasing the maximum distance from the current state.

Swartz presented a modified version of the Lam and Delosme's annealing schedule. Instead of having a monotonically decreasing temperature, Swartz proposed controlling the temperature by continuously increasing and decreasing it on the basis of the acceptance rate. Starting with an initial acceptance rate of 1.0, the rate decreases exponentially during the first 15% of the run until it reaches 0.44. Following this, it remains nearly constant for the next 50% of the run and then it exponentially decreases to 0 by the end of the run.

Boyan presented an approach similar to Swartz's where a feedback mechanism was used to adapt the temperature in order to track the theoretical "optimal" acceptance rate. Doing so had the advantage of not having to modify the neighborhood function during the search. This made the "Modified Lam" annealing schedule problem - independent. [7]

Unlike the SA algorithm, we don't use the temperature as a stopping criterion. Instead, we use the maximum number of evaluations of the cost calculations as the criteria for stopping the algorithm. Note that this value may be changed by the user.

In the following section, we present the pseudocode for the ASA algorithm and run through the steps followed therein.

3.3 Pseudocode

The simulated algorithm works as per the following pseudocode shown in Figure 3.3.1

```
SimulatedAnnealing with Modified Lam Annealing Schedule
        S <-  GenerateInitialState
        T <-  0.5
        AcceptRate <- 0.5
        for i from 1 to Evalsmax
                S' <-  PickRandomState(Neighborhood(S))
                if Cost(S') < Cost(S)
                        S <- S' {Note: accepting a move}
                        AcceptRate  <- 1/500(499.AcceptRate + 1)
                else
                        r <- Random(0; 1)
                        if r < e(Cost(S) - Cost(S'))/T
                                S <- S' {Note: accepting a move}
                                AcceptRate <- 1/500(499.AcceptRate + 1)
                        else
                                {Note: rejecting a move}
                                AcceptRate <- 1/500(499.AcceptRate)
                end
        end
        if i/Evalsmax < 0.15 then LamRate <- 0.44 + 0.56 * 560^(-i/Evalsmax/0.15)
        if 0.15 <= i / Evalsmax < 0.65 then LamRate <- 0.44
        if 0.65 <= I / Evalsmax then LamRate <- 0.44 * 440^-(((i/Evalsmax)-0.65))/0.35)
        if AcceptRate > LamRate
                T <- 0.999T
        else
                T <- T/0.999
        end
        end
```

Figure 3.3.1: Pseudocode for the Adaptive Simulated Annealing algorithm [6]

Let us take a detailed look at how the algorithm is implemented. We begin with generating an initial state based on the initial x and y values provided to the function. Next, the initial temperature is set to a value of 0.5 and initial accept rate to 0.5. We set a reasonably higher value for Evalsmax, i.e., the maximum number of evaluations that will be carried out.

Once inside the iterative loop, we carry out one iteration at a time until the number of operations has reached Evalsmax or until the algorithm reaches convergence. At each iteration, we choose a neighborhood value of S (the current values) and evaluate the

cost function, in our case the cost as a result of these neighborhood values. If the new cost is less than the current cost, accept the move with an increased acceptance rate. If the cost will increase as a result of this move, the move is accepted depending on the probability that decreases with increasing cost. However, if the cost increase is significant, the move is rejected. This is fairly similar to the SA algorithm.

What differentiates between the ASA from the SA algorithm is that we allow the current temperature to fluctuate based on the AcceptRate and the LamRate (LamRate is the "target" acceptance rate). As per the idea of the ASA, we try to stick to a probability of accepting moves to 0.44 so that the temperature reduces the fastest and results in a shorter convergence time. This is controlled by the part of the code shown in Figure 3.3.2.

```
% 7 if i/Evalsmax < 0.15 then LamRate <- 0.44 + 0.56 * 560^-i/Evalsmax/0.15
if i/EvalsMax < 0.15
    LamRate = 0.44 + 0.56 * 560^(-1*i/EvalsMax/0.15);
else
    % 8 if 0.15 <= i/Evalsmax < 0.65 then LamRate <- 0.44
    if i/EvalsMax < 1.65
        LamRate = 0.44;
    else
        % 9 if 0.65 <= i/Evalsmax then LamRate <- 0.44 *
        % 440^-(((i/Evalsmax)-0.65)/0.35)
        LamRate = 0.44 * 440^- (((i/EvalsMax)-0.65)/0.35);
    end
end
```

Figure 3.3.2: Controlling the temperature based on the LamRate and AcceptRate

Appendix 1 provides the complete MATLAB code for the Adaptive Simulated Annealing algorithm.

3.4 Algorithm output

As in the case of the non-adaptive SA algorithm, we use the sample cost function $2x^2 - 4xy + y^4 + 2$. A 3-Dimensional graph of function f shows that f has two local minima at (-1,-1,1) and (1,1,1) and one saddle point at (0,0,2) [5]. This is shown in Figure 3.4.1.

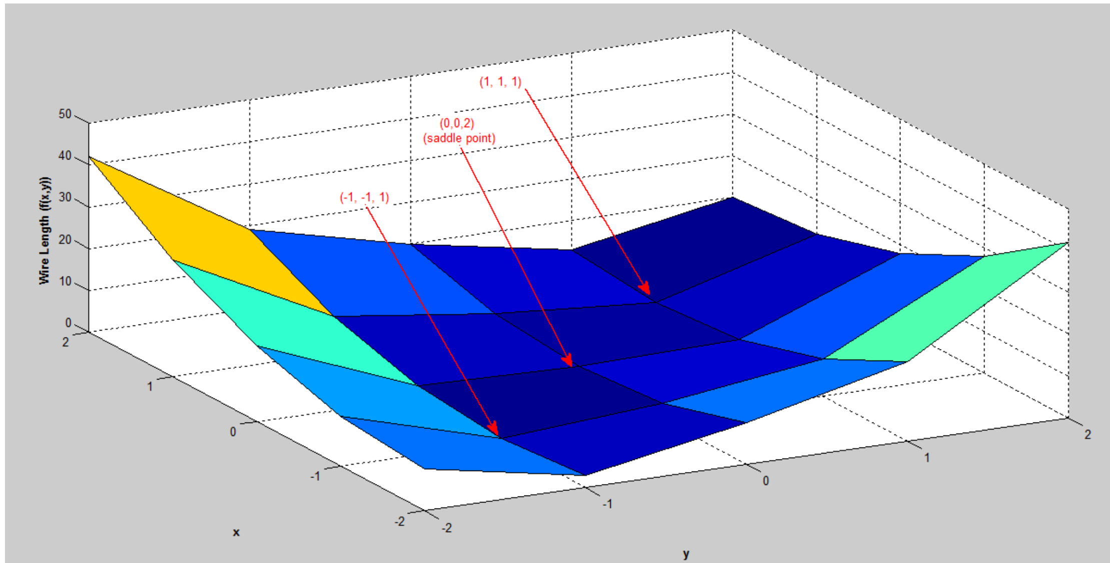


Figure 3.4.1: Cost function used for simulated annealing implementation [5]

We enter a vector as an initial guess and also a function handle so that the cost can be computed at subsequent iterations. The initial guess is a vector that contains the starting x and y values. This is shown in Figure 3.4.2

```
%%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%% EE 5991 | VLSI Cell Placement Algorithms %%%%%%%%%%%%%%%%

clearvars; %clear all variables from MATLAB workspace
GlobalResults  = [];

%sample function from http://www.analyzemath.com/calculus/multivariable/maxima_minima.html
testFunction = @(x,y)(2*x.^2 - 4*x*y + y.^4 + 2); % function that determines the wire length
                                                   % based on x and y coordinates of the module
ruleFunction = @(c)testFunction(c(1),c(2));        % function handle
initialGuess = [1 5];                              % initial guess for x and y coordinates

[SimulatedAnnealingResults] = anneal_rev0(ruleFunction,initialGuess); % Simmulated Annealing a
GlobalResults = [GlobalResults;SimulatedAnnealingResults];
message = sprintf('Starting Adaptive Simulated Annealing');
uiwait(msgbox(message, 'Note'));
[AdaptiveSimulatedAnnealingResults] = AdaptiveAnneal_rev0(ruleFunction,initialGuess);
GlobalResults = [GlobalResults;AdaptiveSimulatedAnnealingResults];
[reportGenerated] = ReportOnTable(GlobalResults);
```

Figure 3.4.2: Passing the initial guess and the cost function handle in to the adaptive simulated annealing algorithm

Once the algorithm is run and it completes execution, we see that the solution converges to the minima (the minimum cost) and we get the optimal x and y values for the solution. The annealing algorithm waveforms at successive iterations and the final output are shown in Figures 3.4.3 and 3.4.4 respectively.

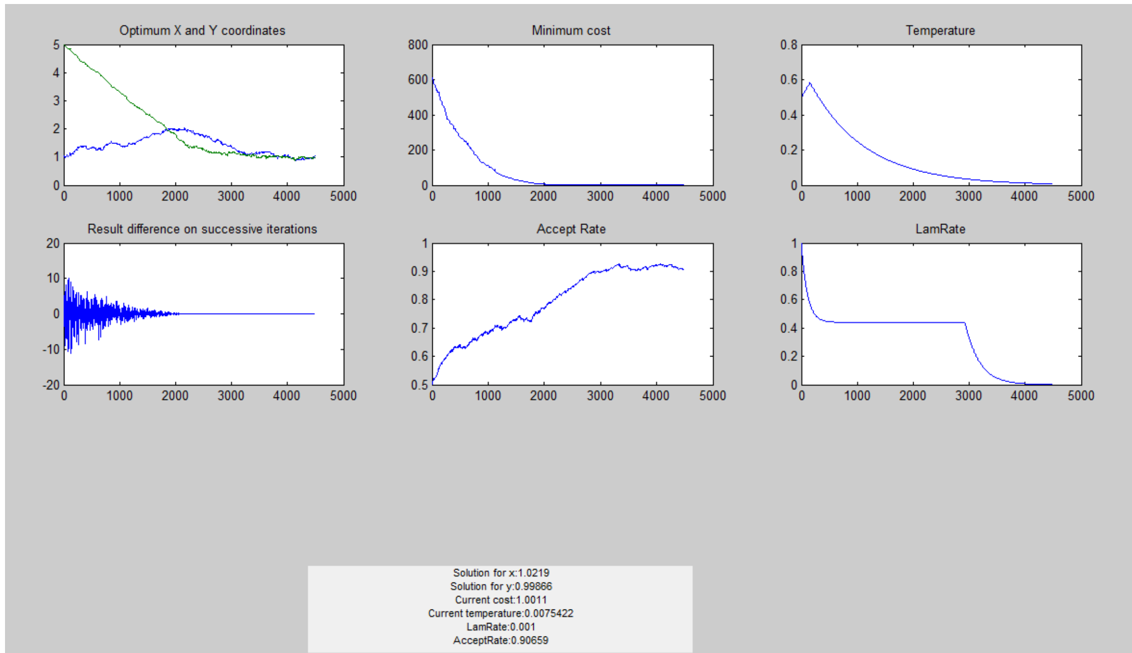


Figure 3.4.3: Adaptive Simulated Annealing algorithm waveforms at successive iterations

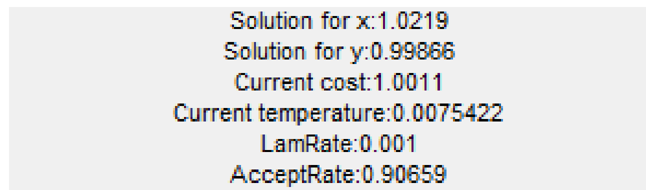


Figure 3.4.4: Output of the simulated annealing algorithm with the minimum cost and the x and y values for the solution

We notice from the results that the Adaptive Simulated Annealing algorithm converges reasonably faster than the SA algorithm at the cost of some accuracy in the results. This is the tradeoff that the user has to account for between the convergence time and the accuracy of the solution. Detailed performance comparisons are provided in a subsequent section.

4. Random restart hill climbing

4.1 Overview

The classical hill climbing algorithm (also called gradient descent algorithm) is often used in finding the optimal points of a given function. In our case, the cost function is what is of interest to us for which we want to determine the optimum x and y values that yield the minimum cost.

Though simulated annealing and adaptive simulated annealing are the usual choice in seeking the minimum value of the cost function, the hill climbing algorithm with suitable enhancement(s) can also be employed to accomplish this task.

The classical hill climbing algorithm yields the optimum points of a function. Since we are interested in the global minima, the hill climbing algorithm with suitable enhancements is used to do so. But what is the need of an enhancement to this algorithm? The downside of the classical hill climbing algorithm is that it yields only local minima while we seek the global minima. Hence, the classical hill climbing algorithm will not work for this task.

Some of the variants of the hill climbing algorithm are the stochastic hill climbing [8] and the random restart hill climbing. In this implementation, the random restart hill climbing algorithm is used to find the optimum x and y values and thereby minimize the value of the cost function. Just like simulated annealing and adaptive simulated annealing, we start with an initial guess and iteratively move toward the optimal solution. In the following section, we will review the working of the adaptation made to the hill climbing algorithm.

4.2 Working of the random restart hill climbing algorithm

Before proceeding with discussing how the random restart hill climbing algorithm works, it would be helpful to discuss how the classical hill climbing algorithm works. We start with a cost function (assume a function of two variables f(x,y), referred to as *f*) and an initial starting point x_0, y_0.

First, we compute the gradient of *f* at x_0, y_0 in terms of the partial derivatives. The gradient is given as

$$\nabla f = \left(\frac{\partial f}{dx}\right) i + \left(\frac{\partial f}{\partial y}\right) j \quad (1)$$

Where the ∂ terms are the partial derivatives of *f* at the current x and y values and *i* and *j* are unit vectors along the x and y directions respectively. This implies that we first calculate the partial derivatives of f and multiply those with *i* and *j* as in the equation above. Suppose now we fix our new points

$$x^{i+1} = x^i + h\nabla f^i$$
$$y^{i+1} = y^i + h\nabla f^i \quad (2)$$

We now have the new (neighborhood) points only in terms of one unknown, namely h. The h is what we refer to as the step size. We now express f^{i+1} in terms of only one unknown h. Next, we take the derivate of the cost (or objective) function *f* with respect to h and get a function in terms of h alone, say *r(h).* To determine what should be the step size h so as to minimize the cost function, we only need to find the roots of *r(h)* and plug the obtained value of h back in equation (2) to get the neighborhood points x^{i+1} and y^{i+1}. If the roots are not real, we start off with a random initial condition and carry out iterative searches starting from that point.

Based on the new points, we follow the same procedure again until we reach a point where the gradient of the cost function is zero. This indicates that a critical point is reached; a point beyond which further improvisation is not possible.

Having reached a critical point, we have implemented the classical form of the hill climbing algorithm. However, this may not necessarily be the global minima since there is a chance that a local minima could have been reached. This is not desirable since we seek a global minima.

Of the several alternatives available, we implement the RRHC (Random Restart Hill Climbing) algorithm. Once a local minima has been reached, do not let it terminate. Instead, we begin the search starting from a random initial condition (x_0', y_0') and iteratively seek improvement by implementing another local search with this initial condition. Whenever a local minima is reached, the x and y values are stored along with the function value.

The procedure of starting with random initial conditions up on arrival at a local minima is carried out repeatedly until the preset number of maximum number of iterations is reached. Once the limit is reached, we look through the stored results and output the x and y values that provided us the least value of the cost function. This is the result of the random restart hill climbing algorithm. Appendix 1 provides the complete MATLAB code of the Random Restart Hill Climbing Algorithm.

4.3 Pseudocode

The Random Restart Hill Climbing algorithm works as per the pseudocode shown in Figure 4.3.1

```
Random Restart Hill Climbing
Starting Point <- (x_i,y_i)
for i from 1 to Evalsmax
        derf_x = (∂f/dx) xi,yi
        derf_y = (∂f/∂y) xi,yi
        ▽f = derf_x i + derf_y j
        x_i+1 = x_i + h * derf_x (symbolic)
        y_i+1 = y_i + h * derf_x (symbolic)
        f <- f(h_i)
        solve r(h_i) = f'(h_i) = 0 for h_(xi, yi)
                if ▽ f(x_i, y_i ) ≠ 0 then
                        if real(f'(h_i)) ≠ 0 then
                                x_i+1 = x_i + h_(xi, yi) * derf_x
                                y_i+1 = y_i + h_(xi, yi) * derf_y
                        else
                                then RandomGenerator (x_i,y_i)
                else
                                results <- store (x_i, y_i, f(x_i,y_i))
                                (x_i',y_i') = RandomGenerator (x_0,y_0)
                end
        if i == Evalsmax
                result = results (min {f (x_min, y_min)})
        end
end
```

Figure 4.3.1: Pseudocode for the Random Restart Hill Climbing algorithm

Let us take a detailed look at how the algorithm is implemented. We begin with generating an initial state based on the initial x and y values provided to the function. Next, we symbolically get points x_{i+1} and y_{i+1} in terms of h, the step size and express the cost function in terms of h alone. It must be noted though that an approach other than symbolic differentiation might be required for cases where either the nature of the cost

function is not known, or to account for the possibility of the cost function not being differentiable at all or not being differentiable at certain points of interest. We take the derivative of the cost function with respect to h and solve it for h. Using this step size value, we compute the next iteration point $x_{i+1} = x_i + h\frac{\partial f}{dx}$ and $y_{i+1} = y_i + h\frac{\partial f}{\partial y}$. The same procedure is carried out until the gradient of f ∇f is not zero. If, however, ∇f is zero, the current x and y values are the critical points and we store these points, along with the value of f at these points in a results table.

Since we reach a critical point when ∇f is zero, we want to ensure that this is indeed the global minima and not a local minima. To achieve this objective, we start the same iterative improvement from a randomly chosen point and see where this point takes us in terms of reaching the critical point. This is illustrated in Figure 4.3.2. Since the algorithm Hill Climbing algorithm is restarted from randomly chosen points, we call this the Random Restart Hill Climbing algorithm.

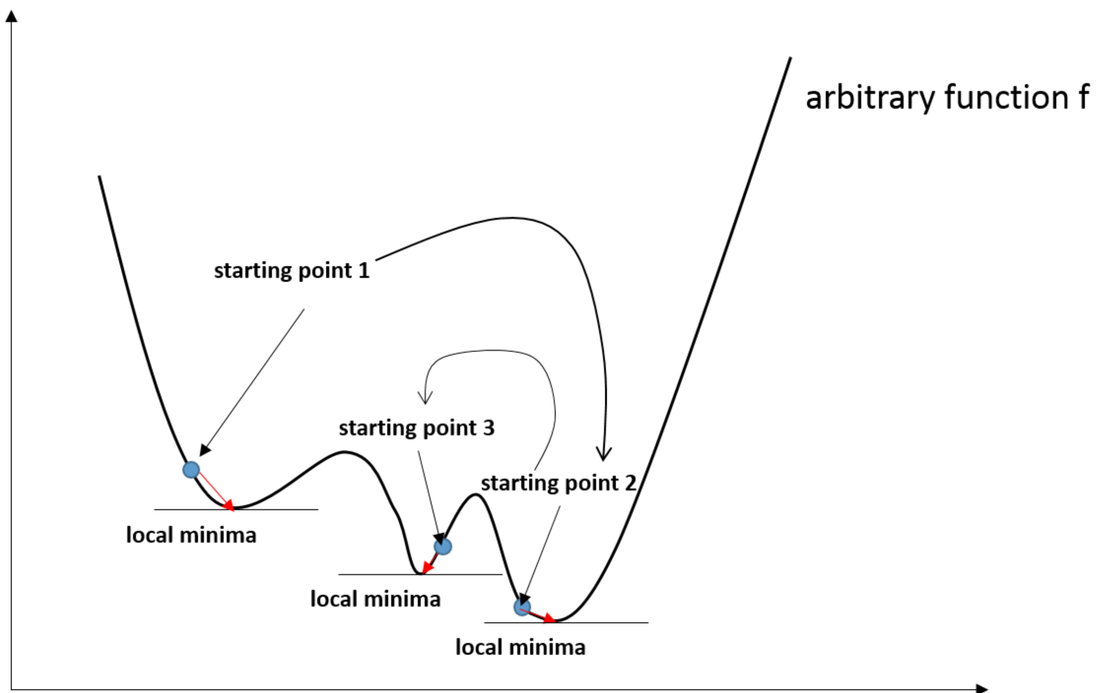


Figure 4.3.2: Random Restart of the Hill Climbing algorithm

The procedure is repeated until the maximum number of iterations is reached. Once this limit is reached, we scan through the stored results and report the x and y values that yield the minimum value of the cost function. The limit is determined empirically. This becomes the result of the implementation of this algorithm.

4.4 Algorithm output

As in the case of the previous algorithms, we use the sample cost function $2x^2 - 4xy + y^4 + 2$. A 3-Dimensional graph of function f shows that f has two local minima at (-1,-1,1) and (1,1,1) and one saddle point at (0,0,2) [5]. This is shown in Figure 4.4.1.

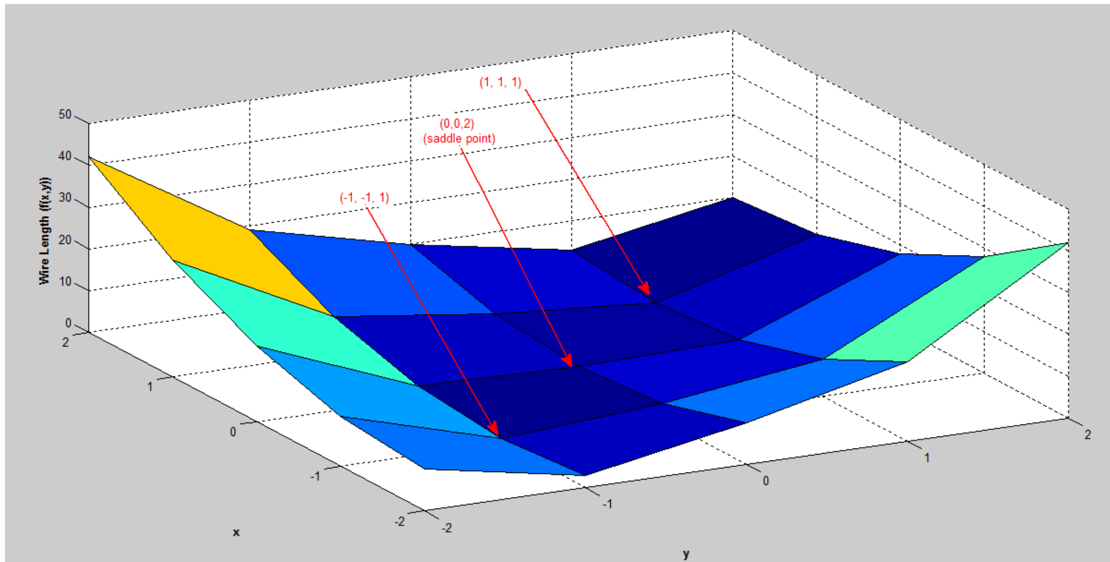


Figure 4.4.1: Cost function used for simulated annealing implementation [5]

We enter a vector as an initial guess and also the cost function so that the cost can be computed at subsequent iterations. The initial guess is a vector that contains the starting x and y values. This is shown in Figure 4.4.2

```
testFunction = @(x,y)(2*x.^2 - 4*x*y + y.^4 + 2); % function that determines the wire le
% based on x and y coordinates of the module being placed
ruleFunction = @(c)testFunction(c(1),c(2));        % function handle
initialGuess = [1 5];                              % initial guess for x and y coordinate
tempUserOpt = [];
for performaceComp = 1:1:comparisonLimit
    [SimulatedAnnealingResults] = anneal_rev0(ruleFunction,initialGuess); % Simmulated A
    completeResult = horzcat(SimulatedAnnealingResults);
    display('SA completed');
    [AdaptiveSimulatedAnnealingResults] = AdaptiveAnneal_rev0(ruleFunction,initialGuess)
    completeResult = horzcat(completeResult, AdaptiveSimulatedAnnealingResults);
    display('ASA completed');
    [RandomRestartHCResults] = randomRestartHillClimbing(testFunction,initialGuess);
    display('RRGD completed');
    completeResult = horzcat(completeResult, RandomRestartHCResults);
    GlobalResults = [GlobalResults;completeResult];
end
```

Figure 4.4.2: Passing the initial guess and the cost function in to the random restart hill climbing algorithm

Once the algorithm is run and completes execution, we see a list of critical points. These points are of interest to us. Just before the algorithm terminates, we output the result based on the minimum value of the cost function. The random restart gradient descent algorithm waveforms at successive iterations and the final output are shown in Figures 4.4.3 and 4.4.4 respectively.

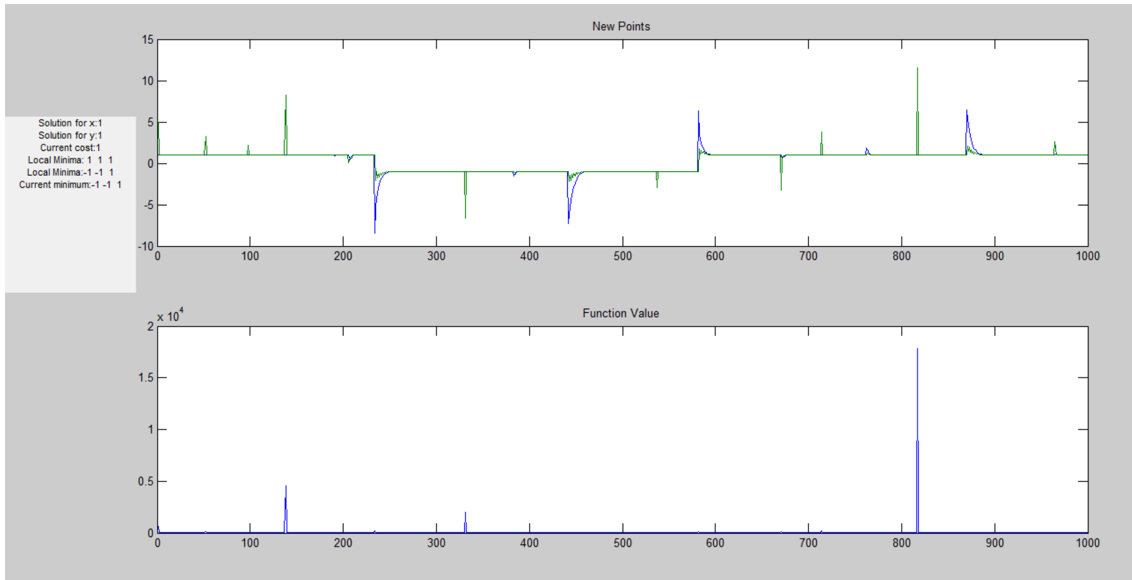


Figure 4.4.3: Random restart hill climbing algorithm waveforms at successive iterations

Solution for x:1
Solution for y:1
Current cost:1
Local Minima: 1 1 1
Local Minima:-1 -1 1
Current minimum:-1 -1 1

Figure 4.4.4: Output of the random restart hill climbing algorithm with the minimum cost and the x and y values for the solution

Since we have the set of results with local minima at (-1,-1) and (1,1) we can set additional filters so as to yield non-negative x and y values. Note that the third value in the results table indicates the value of the cost function at the given x and y values.

5. Performance comparison

Based on the settings chosen for the three algorithms, each algorithm was tested for its performance with three different functions. We present a detailed performance comparison of these algorithms followed by a concluding section.

5.1 Performance comparison with a cost function of two variables

We use the sample cost function $2x^2 – 4xy + y^4 + 2$. The solution results for all the three algorithms are given in Tables 5.1.1, 5.1.2, and 5.1.3. The error is evaluated taking the known minimum function value as 1.0.

Simulated Annealing algorithm				
	x	**y**	**minimum function value**	**Error (%)**
Trail 1	1.0000	1.0000	1.0000	0.0000
Trail 2	1.0000	1.0000	1.0000	0.0000
Trail 3	1.0000	1.0000	1.0000	0.0000
Trail 4	1.0000	1.0000	1.0000	0.0000
Trail 5	1.0000	1.0000	1.0000	0.0000
Trail 6	1.0000	1.0000	1.0000	0.0000
Trail 7	1.0000	1.0000	1.0000	0.0000
Trail 8	1.0000	1.0000	1.0000	0.0000
Trail 9	1.0000	1.0000	1.0000	0.0000
Trail 10	1.0000	1.0000	1.0000	0.0000
Average	1.0000	1.0000	1.0000	0.0000

Table 5.1.1: Solution from the SA algorithm for a cost function of two independent variables x, and y

Adaptive Simulated Annealing algorithm				
	x	**y**	**minimum function value**	**Error (%)**
Trail 1	1.0219	0.9987	1.0011	0.1100
Trail 2	1.1427	1.0316	1.0288	2.8800
Trail 3	1.0663	1.0176	1.0060	0.6000
Trail 4	1.0349	1.0619	1.0177	1.7700
Trail 5	1.1524	1.0736	1.0357	3.5700
Trail 6	1.0329	1.0047	1.0017	0.1700
Trail 7	0.9711	0.9577	1.0072	0.7200
Trail 8	1.0469	1.0101	1.0031	0.3100
Trail 9	0.9707	0.9977	1.0015	0.1500
Trail 10	0.9763	1.0202	1.0055	0.5500
Average	1.0416	1.0174	1.0108	1.0800

Table 5.1.2: Solution from the ASA algorithm for a cost function of two independent variables x, and y

Random Restart Hill Climbing algorithm				
	x	**y**	**minimum function value**	**Error (%)**
Trail 1	1.0000	1.0000	1.0000	0.0000
Trail 2	-1.0000	-1.0000	1.0000	0.0000
Trail 3	-1.0000	-1.0000	1.0000	0.0000
Trail 4	1.0000	1.0000	1.0000	0.0000
Trail 5	1.0000	1.0000	1.0000	0.0000
Trail 6	-1.0000	-1.0000	1.0000	0.0000
Trail 7	-1.0000	-1.0000	1.0000	0.0000
Trail 8	-1.0000	-1.0000	1.0000	0.0000
Trail 9	-1.0000	-1.0000	1.0000	0.0000
Trail 10	-1.0000	-1.0000	1.0000	0.0000
Average	-0.4000	-0.4000	1.0000	0.0000

Table 5.1.3: Solution from the RRHC algorithm for a cost function of two independent variables x, and y

Figure 5.1.1 shows a plot of computation time of all the algorithms for 10 trials each.

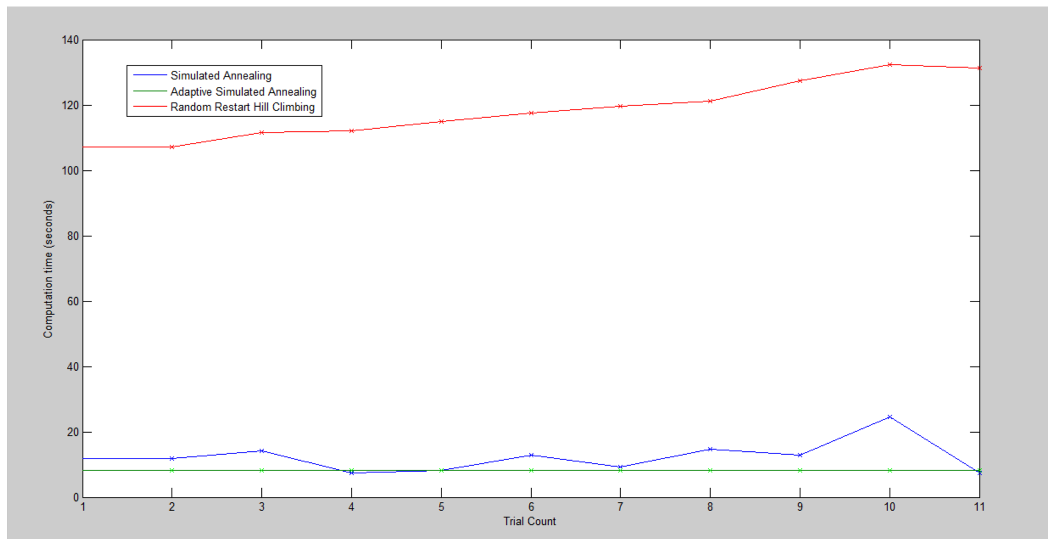


Figure 5.1.1: Computation time (sec.) for 10 tries of the three algorithms

Table 5.1.4 aims to provide information on the tradeoff between percentage error (hence accuracy) and computation time in each of the three algorithms

	Simulated Annealing		**Adaptive Simulated Annealing**		**Random Restart Hill Climbing**	
	Error (%)	time (sec.)	Error (%)	time(sec.)	Error (%)	time(sec.)
Trail 1	0.0000	11.8725	0.1100	8.3083	0.0000	107.0738
Trail 2	0.0000	14.2725	2.8800	8.2298	0.0000	111.5707
Trail 3	0.0000	7.4209	0.6000	8.1175	0.0000	111.9641
Trail 4	0.0000	8.1256	1.7700	8.1117	0.0000	114.8540
Trail 5	0.0000	12.8865	3.5700	8.0961	0.0000	117.4197
Trail 6	0.0000	9.1926	0.1700	8.1202	0.0000	119.5169
Trail 7	0.0000	14.5878	0.7200	8.0581	0.0000	121.1676
Trail 8	0.0000	12.8568	0.3100	8.2373	0.0000	127.4931
Trail 9	0.0000	24.6004	0.1500	8.0896	0.0000	132.2677
Trail 10	0.0000	7.5052	0.5500	8.2354	0.0000	119.4661
Average	0.0000	12.3321	1.0830	8.1604	0.0000	118.2794

Table 5.1.4: Computation time (sec.) vs. percentage error of the three algorithms for the given cost function of two independent variables

Figure 5.1.2 shows a scatter plot of the computation time (sec.) vs. percentage error (hence accuracy) tradeoff in each of the three algorithms. The points with a glow represent the average computation time (sec.) vs. average error (%) for each algorithm's implementation.

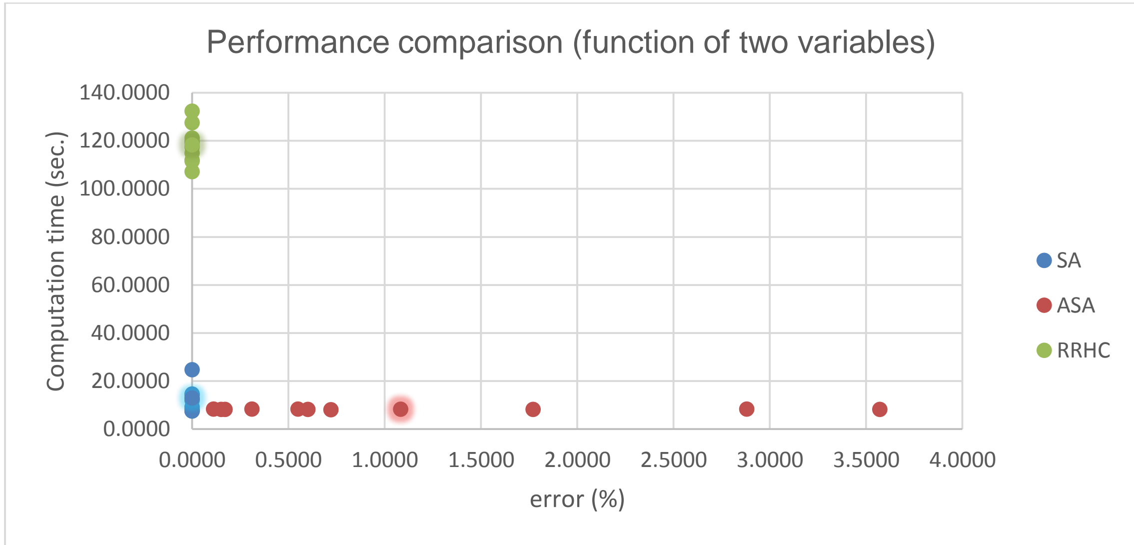


Figure 5.1.2: Computation time (sec.) vs. percentage error (hence accuracy) tradeoff in each of the three algorithms.

5.2 Performance comparison with a cost function of three variables

We use the sample cost function $x^2 + y^2 + z^2$. The solution results for all the three algorithms are given in Tables 5.2.1, 5.2.2, and 5.2.3. The error is evaluated taking the known minimum function value as 0.0 since this function represents the volume of a sphere.

Simulated Annealing algorithm					
	x	**y**	**z**	**minimum function value**	**Error (%)**
Trail 1	-4.30e-05	5.32e-06	5.48e-06	1.91e-09	0.0000
Trail 2	1.57e-07	3.37e-05	-8.28e-06	1.21e-09	0.0000
Trail 3	-1.98e-06	2.07e-05	3.58e-07	4.35e-10	0.0000
Trail 4	-4.80e-06	-5.81e-06	-3.33e-06	6.78e-11	0.0000
Trail 5	6.16e-06	3.27e-05	-4.77e-05	3.38e-09	0.0000
Trail 6	-2.26e-05	-4.74e-06	9.48e-05	9.52e-09	0.0000
Trail 7	5.79e-06	2.47e-06	-1.03e-06	4.06e-11	0.0000
Trail 8	3.85e-05	4.83e-06	4.49e-07	1.50e-09	0.0000
Trail 9	4.69e-08	5.23e-06	4.43e-05	1.99e-09	0.0000
Trail 10	2.91e-07	-5.75e-07	-5.95e-05	3.54e-09	0.0000
Average	-2.15e-06	9.39e-06	2.56e-06	2.36e-09	0.0000

Table 5.2.1: Solution from the SA algorithm for a cost function of three independent variables x, y, and z

Adaptive Simulated Annealing algorithm					
	x	**y**	**z**	**minimum function value**	**Error (%)**
Trail 1	-0.0122	0.0897	0.1514	0.0311	3.1100
Trail 2	0.0166	0.3772	0.0833	0.1495	14.9500
Trail 3	0.0927	0.2953	0.0348	0.0970	9.7000
Trail 4	0.0447	0.0673	-0.0234	0.0071	0.7100
Trail 5	-0.0142	0.2292	0.0474	0.0550	5.5000
Trail 6	-0.0395	0.2485	0.0017	0.0633	6.3300
Trail 7	-0.0046	-0.0289	-0.0005	0.0009	0.0857
Trail 8	0.0796	0.6125	0.0352	0.3827	38.2700
Trail 9	0.0269	0.1638	-0.0313	0.0285	2.8500
Trail 10	-0.0528	0.4458	0.1669	0.2293	22.9300
Average	0.0137	0.2500	0.0465	0.1044	10.4400

Table 5.2.2: Solution from the ASA algorithm for a cost function of three independent variables x, y, and z

Random Restart Hill Climbing algorithm					
	x	**y**	**z**	**minimum function value**	**Error (%)**
Trail 1	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 2	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 3	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 4	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 5	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 6	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 7	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 8	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 9	0.0000	0.0000	0.0000	0.0000	0.0000
Trail 10	0.0000	0.0000	0.0000	0.0000	0.0000
Average	0.0000	0.0000	0.0000	0.0000	0.0000

Table 5.2.3: Solution from the RRHC algorithm for a cost function of three independent variables x, y, and z

Figure 5.2.1 shows a plot of computation time of all the algorithms for 10 trials each.

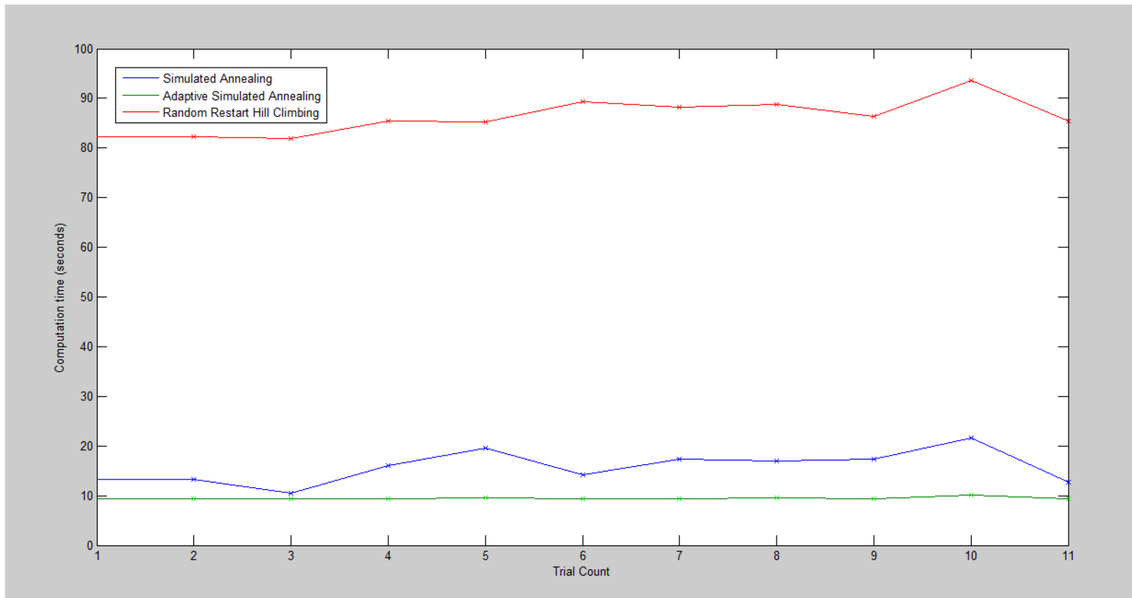


Figure 5.2.1: Computation time (sec.) for 10 tries of the three algorithms

Table 5.2.4 aims to provide information on the tradeoff between percentage error (hence accuracy) and computation time in each of the three algorithms.

	Simulated Annealing		Adaptive Simulated Annealing		Random Restart Hill Climbing	
	Error (%)	**time(sec.)**	**Error (%)**	**time(sec.)**	**Error (%)**	**time(sec.)**
Trail 1	0.0000	13.2846	3.1100	9.3280	0.0000	82.3744
Trail 2	0.0000	10.4691	14.9500	9.2858	0.0000	81.9271
Trail 3	0.0000	16.0278	9.7000	9.2807	0.0000	85.4474
Trail 4	0.0000	19.6527	0.7100	9.4716	0.0000	85.2702
Trail 5	0.0000	14.2018	5.5000	9.4105	0.0000	89.2661
Trail 6	0.0000	17.2582	6.3300	9.3741	0.0000	88.1859
Trail 7	0.0000	16.8966	0.0857	9.5158	0.0000	88.7392
Trail 8	0.0000	17.2577	38.2700	9.4201	0.0000	86.2815
Trail 9	0.0000	21.5238	2.8500	10.0450	0.0000	93.5806
Trail 10	0.0000	12.6871	22.9300	9.3865	0.0000	85.3771
Average	0.0000	15.9259	10.4436	9.4518	0.0000	86.6450

Table 5.2.4: Computation time (sec.) vs. percentage error of the three algorithms for the given cost function of three independent variables

Figure 5.2.2 shows a scatter plot of the computation time (sec.) vs. percentage error (hence accuracy) tradeoff in each of the three algorithms. The points with a glow represent the average computation time (sec.) vs. average error (%) for each algorithm's implementation.

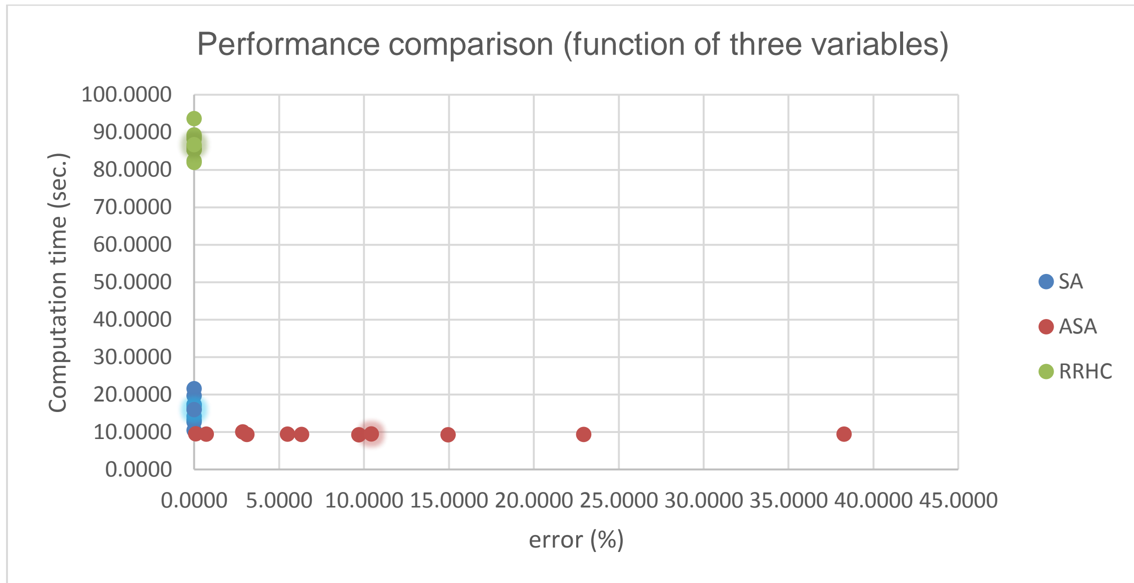


Figure 5.2.2: Computation time (sec.) vs. percentage error (hence accuracy) tradeoff in each of the three algorithms.

5.3 Performance comparison with a cost function of four variables

We use the sample cost function $(2w^2 – 4wx + x^4 + 2)*(2y^2 – 4yz + z^4 + 2)$. The solution results for all the three algorithms are given in Tables 5.3.1, 5.3.2, and 5.3.3. The error is evaluated taking the known minimum function value as 1.0.

Simulated Annealing algorithm						
	w	**x**	**y**	**z**	**minimum function value**	**Error (%)**
Trail 1	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 2	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 3	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 4	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 5	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 6	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 7	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 8	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 9	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 10	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Average	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000

Table 5.3.1: Solution from the SA algorithm for a cost function of four independent variables w, x, y, and z

Adaptive Simulated Annealing algorithm						
	w	**x**	**y**	**z**	**minimum function value**	**Error (%)**
Trail 1	1.1072	1.0052	-0.9071	-0.9738	1.0327	3.2700
Trail 2	1.0764	0.9977	-0.6883	-0.8921	1.1387	13.8700
Trail 3	1.0246	0.9841	-0.8870	-0.9478	1.0221	2.2100
Trail 4	0.9245	1.0302	-1.0296	-0.9989	1.0280	2.8000
Trail 5	0.9951	0.9799	-0.8949	-0.9394	1.0198	1.9800
Trail 6	1.0509	1.0231	-0.8316	-0.9106	1.0455	4.5500
Trail 7	0.9807	1.0058	-1.0396	-1.0193	1.0038	0.3800
Trail 8	0.9998	0.9796	-0.8535	-0.8864	1.0507	5.0700
Trail 9	0.9755	1.0065	-1.0176	-0.9904	1.0039	0.3900
Trail 10	0.9630	0.9885	-0.9836	-0.9546	1.0114	1.1400
Average	1.0098	1.0001	-0.9133	-0.9513	1.0357	3.5700

Table 5.3.2: Solution from the ASA algorithm for a cost function of four independent variables w, x, y, and z

Random Restart Hill Climbing algorithm						
	w	**x**	**y**	**z**	**minimum function value**	**Error (%)**
Trail 1	-1.0000	-1.0000	1.0000	1.0000	1.0000	0.0000
Trail 2	-1.0000	-1.0000	1.0000	1.0000	1.0000	0.0000
Trail 3	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000
Trail 4	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 5	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000
Trail 6	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 7	-1.0000	-1.0000	1.0000	1.0000	1.0000	0.0000
Trail 8	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000
Trail 9	1.0000	1.0000	-1.0000	-1.0000	1.0000	0.0000
Trail 10	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000
Average	0.4000	0.4000	0.4000	0.4000	1.0000	0.0000

Table 5.3.3: Solution from the RRHC algorithm for a cost function of four independent variables w, x, y, and z

Figure 5.3.1 shows a plot of computation time of all the algorithms for 10 trials each.

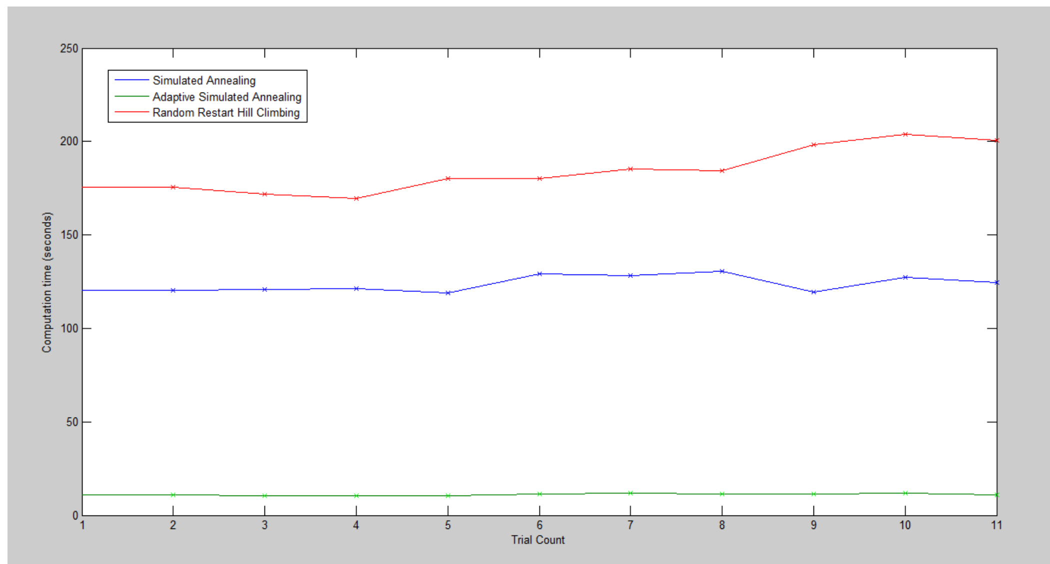


Figure 5.3.1: Computation time (sec.) for 10 tries of the three algorithms

Table 5.3.4 aims to provide information on the tradeoff between percentage error (or accuracy) and computation time in each of the three algorithms

	Simulated Annealing		**Adaptive Simulated Annealing**		**Random Restart Hill Climbing**	
	Error (%)	**time(sec.)**	**Error (%)**	**time(sec.)**	**Error (%)**	**time(sec.)**
Trail 1	0.0000	120.2330	3.2700	10.8771	0.0000	175.3429
Trail 2	0.0000	120.9094	13.8700	10.5780	0.0000	171.8625
Trail 3	0.0000	121.5211	2.2100	10.5127	0.0000	169.7304
Trail 4	0.0000	119.1562	2.8000	10.6375	0.0000	180.2140
Trail 5	0.0000	129.1668	1.9800	11.3135	0.0000	180.2062
Trail 6	0.0000	128.1730	4.5500	11.7564	0.0000	185.0980
Trail 7	0.0000	130.7918	0.3800	11.1606	0.0000	184.5443
Trail 8	0.0000	119.3853	5.0700	11.1768	0.0000	198.1524
Trail 9	0.0000	127.3835	0.3900	11.7708	0.0000	204.0424
Trail 10	0.0000	124.3711	1.1400	10.9535	0.0000	200.5978
Average	0.0000	124.1091	3.5660	11.0737	0.0000	184.9791

Table 5.3.4: Computation time (sec.) vs. percentage error of the three algorithms for the given cost function of four independent variables

Figure 5.3.2 shows a scatter plot of the computation time (sec.) vs. percentage error (hence accuracy) tradeoff in each of the three algorithms. The points with a glow represent the average computation time (sec.) vs. average error (%) for each algorithm's implementation.

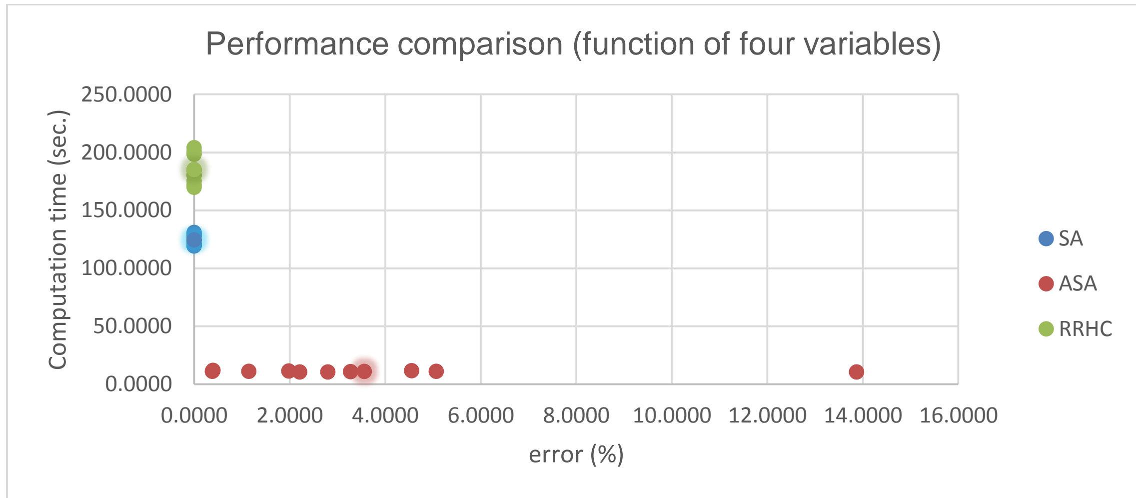


Figure 5.3.2: Computation time (sec.) vs. percentage error (hence accuracy) tradeoff in each of the three algorithms.

6. Conclusions

As is evident from the computation time plots, RRHC is the computationally most expensive algorithm among the three. The computation times for SA and ASA algorithms are comparable when working with functions of two or three independent variables. However, the difference is significant when working with a function of four independent variables.

SA and RRHC algorithms deliver higher accuracy in comparison to the ASA algorithm. This is evident from the scatter plots for these three algorithms for each of the tested functions. The tradeoff is that SA and RRHC algorithms take more computation time when compared to the ASA algorithm.

Also, it was noted that the RRHC was not able to trace out all the possible optimal solutions in the same number of iterations when implemented on the function of four variables. What this implies is that more iterations may be necessary for RRHC for relatively complex functions.

Given the performance comparison of the three algorithms and their individual accuracy vs. computation time tradeoffs, it is up to the implementer to decide which of the three algorithms would best suit the task at hand.

Appendix 1

Main file in MATLAB (for function of two variables):

```
%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EE 5991 | Heuristic Optimization Algorithms
%%%%%%%%%%%%%%%

clearvars; %clear all variables from MATLAB workspace
clc;
GlobalResults  = [];
comparisonLimit = 10;
results = [];

%sample function from
http://www.analyzemath.com/calculus/multivariable/maxima_minima.html
testFunction = @(x,y)(2*x.^2 - 4*x*y + y.^4 + 2);
ruleFunction = @(c)testFunction(c(1),c(2));       % function handle
initialGuess = [1 5];                             % initial guess for
x and y coordinates -
tempUserOpt = [];
for performaceComp = 1:1:comparisonLimit
    [SimulatedAnnealingResults] =
anneal_rev1(ruleFunction,initialGuess); % Simmulated Annealing
algorithm implementation (rev1)
    completeResult = horzcat(SimulatedAnnealingResults);
    [AdaptiveSimulatedAnnealingResults] =
AdaptiveAnneal_rev1(ruleFunction,initialGuess); % Adaptive Simmulated
Annealing algorithm implementation (rev1)
    completeResult = horzcat(completeResult,
AdaptiveSimulatedAnnealingResults);
    [RandomRestartHCResults] =
randomRestartHillClimbing_2var_rev1(testFunction,initialGuess);
    completeResult = horzcat(completeResult, RandomRestartHCResults);
    GlobalResults = [GlobalResults;completeResult];
end
[comparisonGenerated, computationTimeInformation] =
PerformaceComparision(GlobalResults,comparisonLimit);
[reportGenerated] =
ReportOnTable(GlobalResults,computationTimeInformation,comparisonLimit
);
```

Main file in MATLAB (for function of three variables):

```
%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EE 5991 | Heuristic Optimization Algorithms
%%%%%%%%%%%%%%%

clearvars; %clear all variables from MATLAB workspace
clc;
GlobalResults  = [];
comparisonLimit = 10;
results = [];
```

```
testFunction_3var = @(x,y,z)(x.^2 + y.^2 + z.^2);
ruleFunction = @(c)testFunction_3var(c(1),c(2),c(3));        % function
handle
initialGuess = [-1 3 1];                                 % initial guess
for x, y and z coordinates -
tempUserOpt = [];
for performaceComp = 1:1:comparisonLimit
    [SimulatedAnnealingResults] =
anneal_3var_rev1(ruleFunction,initialGuess); % Simmulated Annealing
algorithm implementation (rev1)
    completeResult = horzcat(SimulatedAnnealingResults);
    [AdaptiveSimulatedAnnealingResults] =
AdaptiveAnneal_3var_rev1(ruleFunction,initialGuess); % Adaptive
Simmulated Annealing algorithm implementation (rev1)
    completeResult = horzcat(completeResult,
AdaptiveSimulatedAnnealingResults);
    [RandomRestartHCResults] =
randomRestartHillClimbing_3var_rev1(testFunction_3var,initialGuess);
    completeResult = horzcat(completeResult, RandomRestartHCResults);
    GlobalResults = [GlobalResults;completeResult];
end
display('Fetch Results Now!');
[comparisonGenerated, computationTimeInformation] =
PerformaceComparision_3var(GlobalResults,comparisonLimit);
[reportGenerated] =
ReportOnTable_3var(GlobalResults,computationTimeInformation,comparison
Limit);
```

Main file in MATLAB (for function of four variables):

```
%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EE 5991 | Heuristic Optimization Algorithms
%%%%%%%%%%%%%%%

clearvars; %clear all variables from MATLAB workspace
clc;
GlobalResults  = [];
comparisonLimit = 10;
results = [];

testFunction_4var = @(w,x,y,z)((2*w.^2 - 4*w*x + x.^4 + 2)*(2*y.^2 -
4*y*z + z.^4 + 2));
% based on x and y coordinates of the module being placed
ruleFunction = @(c)testFunction_4var(c(1),c(2),c(3),c(4));        %
function handle
initialGuess = [1 1 -1 0];                                 % initial guess
for w, x, y and z coordinates -
tempUserOpt = [];
for performaceComp = 1:1:comparisonLimit
    [SimulatedAnnealingResults] =
anneal_4var_rev1(ruleFunction,initialGuess); % Simmulated Annealing
algorithm implementation (rev1)
    completeResult = horzcat(SimulatedAnnealingResults);
```

```
    [AdaptiveSimulatedAnnealingResults] =
AdaptiveAnneal_4var_rev1(ruleFunction,initialGuess); % Adaptive
Simmulated Annealing algorithm implementation (rev1)
    completeResult = horzcat(completeResult,
AdaptiveSimulatedAnnealingResults);
    [RandomRestartHCResults] =
randomRestartHillClimbing_4var_rev1(testFunction_4var,initialGuess);
    completeResult = horzcat(completeResult, RandomRestartHCResults);
    GlobalResults = [GlobalResults;completeResult];
end
[comparisonGenerated, computationTimeInformation] =
PerformaceComparision_4var(GlobalResults,comparisonLimit);
[reportGenerated] =
ReportOnTable_4var(GlobalResults,computationTimeInformation,comparison
Limit);
```

Non-adaptive simulated annealing algorithm implementation in MATLAB:

```
%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EE 5991 Algorithm - 1 %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% Simulated Annealing %%%%%%%%%%%%%%%%%%%%%%%

% >> PROCEDURE Simulated_ Annealing
% Above indicates the beginning of the algorithm implementation
% >> 1 initialize;

function [reportResults] = anneal_rev1(ruleFunction,
parentParam,userOpt)
% arrays for holding data
plot_fVal = [];
plot_minVal = [];
plot_temp = [];
tempPlot_diff = [];
solutionUpdateRow = [];
solutionMatrix = [];
reportResults = zeros(1,5);
% The following piece of code is used for initialization.
% Default parameters for Simmulated Annealing algorhtim
% length(param) returns
% randperm(length(param)) randomly returns [1 2] per the definition of
% randperm. randperm(length(param)) == length(param) randomly returns
[1 0]
% per definition of == operator the result is then multiplied with
% "randn/100" and added to the previous value of input parameter to
get the
% updated value.

defaultParam = struct(...
    'Schedule', @(T) (0.9 * T),...
    'MaxConsecutiveRejections', 1000,...
    'MaxSuccessAtTemperature', 20,...
    'RandomGenerator', @(param)
(param+(randperm(length(param))==length(param))*randn/100),...
    'InitialTemperature', 1,...
```

```
    'StoppingTemperature', 1e-8,...
    'MaxTriesAtTemperature', 300,...
    'StoppingValue', 0);

%verification of input:
if ~nargin
    minimum = defaultParam;
    return
elseif nargin < 2
    error('Please input user options.');
elseif nargin < 4
    userOpt = defaultParam;
else
    if ~isstruct (userOpt)
        error('"userOpt" is not a structure. Using default
options...');
    end
    structFormat =
{'Schedule','MaxConsecutiveRejections','MaxSuccessAtTemperature','Rand
omGenerator',...

'InitialTemperature','StoppingTemperature','MaxTriesAtTemperature','St
oppingValue'};
    for count = 1:1:length(structFormat)
        if ~isfield(userOpt,structFormat{count})
            userOpt.(structFormat{count}) =
defaultParam.(structFormat{count});
        end
    end
end

% Initialization
UpdatedCoordinates = userOpt.RandomGenerator;      % generates random
solution
InitTemp = userOpt.InitialTemperature;        % initial temperature
StoppingTemp = userOpt.StoppingTemperature;         % stopping
temperature
CoolSched = userOpt.Schedule;        % cooling schedule for annealing
MinFunc = userOpt.StoppingValue;          % minimum value of Function
MaxConsRej = userOpt.MaxConsecutiveRejections; % maximum consecutive
rejections
MaxTryAtT = userOpt.MaxTriesAtTemperature; % maximum tries at a
temperature
MaxSucAtT = userOpt.MaxSuccessAtTemperature; % maximum success at a
temperature

% Initialize counters
trialCountAtT = 0;      % iteration counter
successCountAtT = 0;    % success counter
finishedFlag = 0;    % flag to indicate that a soultion has been
reached OR that the program needs to end execution
consecRejCount = 0; % consecutive rejection count at a particular
temperature
temp = InitTemp;     % temperature initialized to Initial Temperature
preset
```

```
initialLength = ruleFunction(parentParam); % initial wire length is
computed based on the initial guess parameters
oldLength = initialLength; % initialize oldLength; will be updated on
successive iterations
funcCallCount = 0; % number of times this function was called
annealCount = 0; % number of times annealing was done
k = 1 ; % Boltzmann constant
%draw = 0;
%% Figure control parameter for graph:

fig1 = figure;
set(fig1,'name','PLOT OF SIMMULATED ANNEALING ALGORITHM (NON-
ADAPTIVE):','numbertitle','off')
Results = [];
hList1 = uicontrol(fig1,'Style','text','Position',[600 5 400 75]);

%% time / performance calculation
tic;
% >> 3 WHILE stopping. criterion (loop. count, temperature) = FALSE
%%
while ~finishedFlag
    trialCountAtT = trialCountAtT + 1;
    currentParam = parentParam;
    % >> 4 WHILE inner.loop.criterion = FALSE
    % We do the annealing if the we have reached the maximum number of
tries
    % at a particular temperature OR is the number of successful moves
is
    % greater than or equal to the preset for maximum success count at
a
    % particular temperature. This is based on the additional condition
that
    % the algorithm isn't terminated due to the temperature going
below the
    % stopping temperature OR the consecutive rejection count doesn't
exceed
    % the preset for Maximum Consecutive Rejections
    if trialCountAtT >= MaxTryAtT || successCountAtT >= MaxSucAtT
        if temp <= StoppingTemp || consecRejCount >= MaxConsRej
            finishedFlag = 1;
            funcCallCount = funcCallCount + trialCountAtT;
            break;
        else
            % >> 13 temperature <- schedule(loop_count, temperature);
%cooling is done
            temp = CoolSched(temp);
            annealCount = annealCount + 1;
            funcCallCount = funcCallCount + trialCountAtT;
            trialCountAtT = 1; %set trialCountAtT back to 1 because we
just reduced the temperature
            successCountAtT = 1; %set successCountAtT back to 1
because we just reduced the temperature
        end
    end
    % >> 5 new_configuration <-  perturb(configuration);
```

```
    % This is specific to the wire length implementation. This is
where the
    % layout equation will come in and we will optimize it.
    newParam = UpdatedCoordinates(currentParam);
    newLength = ruleFunction(newParam);
    diff = newLength - oldLength;

    % If new length is less than the minimum value of Length as set
then we
    % set the oldLength to the value of the newLength and parentParam
as
    % the value obtained for newParam
    if (newLength < MinFunc)
        parentParam = newParam;
        oldLength = newLength;
        break
    end

    % if f(x0) - f(x1) > 0 (in other words the function value
decreases,
    % then we replace initial guess with the new guess i.e.
parentParam =
    % newParam and oldLength = newLength and count that as a success
and
    % reset the consecutive rejection count
    if (oldLength - newLength > 0)
        parentParam = newParam;
        oldLength = newLength;
        successCountAtT = successCountAtT + 1;
        consecRejCount = 0;
    else
        % if f(x0) - f(x1) < 0 (in other words the function value does
NOT
        % decrease AND we still replace x0 with x1 BUT with the
probability
        % given by exp(-1 * diff)/k * temp
        if (rand < exp(-1 * diff)/k * temp)
            parentParam = newParam;
            oldLength = newLength;
            successCountAtT = successCountAtT + 1;
            % if the acceptance probability is low, then we reject the
move
        else
            consecRejCount = consecRejCount + 1;
        end
    end

    % Plotting data used only if one iteration is performed for
performance
    % comparision
    plot_fVal = [plot_fVal;oldLength;];
    plot_minVal = [plot_minVal;parentParam];
    plot_temp = [plot_temp;temp];
    tempPlot_diff = [tempPlot_diff;diff];
    solutionUpdateRow = [parentParam(1) parentParam(2) oldLength];
```

```
    solutionMatrix = [solutionMatrix;solutionUpdateRow];

    % Reporting the results on the figure
    Results {1,1} = strcat ('Solution for x:
',num2str(parentParam(1)));
    Results {1,2} = strcat ('Solution for y:
',num2str(parentParam(2)));
    Results {1,3} = strcat ('Current cost: ',num2str(oldLength));
    Results {1,4} = strcat ('Current temperature: ',num2str(temp));
    Results {1,5} = strcat ('Number of annealing steps:
',num2str(annealCount));
    set(hList1,'String',Results);  % Displays 5 lines, one result per
line
end
%% Plotting the results
set([fig1],'handlevisibility','on');
set(0,'CurrentFigure',fig1);
subplot(2,3,1); plot(plot_minVal)
title('Optimum X and Y coordinates')
subplot(2,3,2);plot(plot_fVal)
title('Minimum cost')
subplot(2,3,3);plot(plot_temp)
title('Temperature')
subplot(2,3,4);plot(tempPlot_diff)
title('Result difference on successive iterations')
drawnow;

%% Reporting the results
minimum = parentParam;
fval = oldLength;
elapsedTime = toc;
reportResults(1,1) = minimum(1);
reportResults(1,2) = minimum(2);
reportResults(1,3) = fval;
reportResults(1,4) = annealCount;
reportResults(1,5) = elapsedTime;
end
```

Adaptive simulated annealing algorithm implementation in MATLAB:

```
%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EE 5991 Algorithm - 2 %%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% Adaptive Simulated Annealing %%%%%%%%%%%%%%

% >> PROCEDURE Adaptive Simulated Annealing
% Above indicates the beginning of the algorithm implementation
% SimulatedAnnealing with Modified Lam Annealing Schedule

function [reportResults] = AdaptiveAnneal_rev1(ruleFunction,
parentParam,userOpt)
% time / performance calculation
totalFunctionTime  = tic;
% arrays for holding data
plot_fVal = [];
```

```
plot_minVal = [];
plot_temp = [];
plot_resultDiff = [];
plot_AcceptRate = [];
plot_LamRate = [];
solutionUpdateRow = [];
solutionMatrix = [];
reportResults = zeros(1,5);

% The following piece of code is used for initialization.
% Default parameters for Adaptive Simmulated Annealing algorhtim
% length(param) returns 2
% randperm(length(param)) randomly returns [1 2] per the definition of
% randperm. randperm(length(param)) == length(param) randomly returns
[1 0]
% per definition of == operator. The result is then multiplied with
% "randn/100" and added to the previous value of input parameter to
get the
% updated value.
% 2 T <-  0:5

% Set initial temperature and the random generator function
defaultParam = struct(...
    'RandomGenerator', @(param)
(param+(randperm(length(param))==length(param))*randn/100),...
    'InitialTemperature', 0.5);
%verification of input:
if ~nargin
    minimum = defaultParam;
    return
elseif nargin < 2
    error('Please input a user options.');
elseif nargin < 4
    userOpt = defaultParam;
else
    if ~isstruct (userOpt)
        error('"userOpt" is not a structure. Using default
options...');
    end
    structFormat = {'RandomGenerator','InitialTemperature'};
    for count = 1:1:length(structFormat)
        if ~isfield(userOpt,structFormat{count})
            userOpt.(structFormat{count}) =
defaultParam.(structFormat{count});
        end
    end
end

% 1 S <-  GenerateInitialState
% Initialize values / counters
UpdatedCoordinates = userOpt.RandomGenerator;      % generates random
solution
InitTemp = userOpt.InitialTemperature;        % initial temperature
temp = InitTemp;    % temperature initialized to Initial Temperature
preset
```

```
initialLength = ruleFunction(parentParam); % initial wire length is
computed based on the initial guess parameters
oldLength = initialLength; % initialize oldLength; will be updated on
successive iterations
annealCount = 0; % number of times annealing was done

% 3 AcceptRate <- 0.5
AcceptRate = 0.5; % Acceptance rate initialized to 0.5 for Adaptive
Simulated Annealing (ASA) algorithm
EvalsMax = 4500;
currentParam = parentParam;

%% Figure control parameter for graph:
fig2 = figure;
set([fig2],'handlevisibility','on');
set(0,'CurrentFigure',fig2);
set(fig2,'name','PLOT OF SIMMULATED ANNEALING ALGORITHM
(ADAPTIVE):','numbertitle','off')
Results = [];
hList2 = uicontrol(fig2,'Style','text','Position',[433 5 400 90]);

%%

% 4 for i from 1 to Evalsmax
for i=1:1:EvalsMax
    %while ~flagStop
    currentParam = parentParam;
    % 5 S' <-  PickRandomState(Neighborhood(S))
    newParam = UpdatedCoordinates(currentParam);
    newLength = ruleFunction(newParam);
    diff = newLength - oldLength;
    % 6 if Cost(S') < Cost(S)
    % S <- S' {Note: accepting a move}
    % AcceptRate  <- 1/500(499.AcceptRate + 1)
    if (diff < 0)
        parentParam = newParam;
        oldLength = newLength;
        AcceptRate = 1/500*(499*AcceptRate + 1);
        % else
        % r <- Random(0; 1)
        % if r < e(Cost(S) - Cost(S'))/T
        % S <- S' {Note: accepting a move}
        % AcceptRate <- 1/500(499.AcceptRate + 1)
    else
        if (rand < exp((-1*diff)/temp))
            parentParam = newParam;
            oldLength = newLength;
            AcceptRate = 1/500*(499*AcceptRate + 1);
        else
            % else
            % {Note: rejecting a move}
            % AcceptRate <- 1/500(499.AcceptRate)
            AcceptRate = 1/500*(499*AcceptRate);
        end
    end
```

```
    % 7 if i/Evalsmax < 0.15 then LamRate <- 0.44 + 0.56 * 560^-
i/Evalsmax/0.15
    if i/EvalsMax < 0.15
        LamRate = 0.44 + 0.56 * 560^(-1*i/EvalsMax/0.15);
    else
        % 8 if 0.15 <= i/Evalsmax < 0.65 then LamRate <- 0.44
        if i/EvalsMax < 0.65
            LamRate = 0.44;
        else
            % 9 if 0.65 <= i/Evalsmax then LamRate <- 0.44 *
            % 440^-(((i/Evalsmax)-0.65)/0.35)
            LamRate = 0.44 * 440^- (((i/EvalsMax)-0.65)/0.35);
        end
    end

    % 10 if AcceptRate > LamRate
    %       T <- 0.999T
    %    else
    %       T <- T/0.999
    %    end
    if AcceptRate > LamRate
        temp = 0.999*temp;
        annealCount = annealCount + 1;
    else
        temp = temp / 0.999;
        annealCount = annealCount + 1;
    end


    %% Update Results:
    % Plotting data used only if one iteration is performed for
performance
    % comparision
    plot_fVal = [plot_fVal;oldLength];
    plot_minVal = [plot_minVal;parentParam];
    plot_temp = [plot_temp;temp];
    plot_resultDiff = [plot_resultDiff;diff];
    plot_AcceptRate = [plot_AcceptRate;AcceptRate];
    plot_LamRate = [plot_LamRate; LamRate];
    solutionUpdateRow = [parentParam(1) parentParam(2) oldLength];
    solutionMatrix = [solutionMatrix;solutionUpdateRow];

    % Reporting the results on the figure

    Results {1,1} = strcat ('Solution for x:
',num2str(parentParam(1)));
    Results {1,2} = strcat ('Solution for y:
',num2str(parentParam(2)));
    Results {1,3} = strcat ('Current cost: ',num2str(oldLength));
    Results {1,4} = strcat ('Current temperature: ',num2str(temp));
    Results {1,5} = strcat ('LamRate: ',num2str(LamRate));
    Results {1,6} = strcat ('AcceptRate: ',num2str(AcceptRate));
    set(hList2,'String',Results);  % Displays 6 lines, one result per
line
```

```
end
%% Plotting the results
set([fig2],'handlevisibility','on');
set(0,'CurrentFigure',fig2);
subplot(3,3,1); plot(plot_minVal)
title('Optimum X and Y coordinates')
subplot(3,3,2);plot(plot_fVal)
title('Minimum cost')
subplot(3,3,3);plot(plot_temp)
title('Temperature')
subplot(3,3,4);plot(plot_resultDiff)
title('Result difference on successive iterations')
subplot(3,3,5);plot(plot_AcceptRate)
title('Accept Rate')
subplot(3,3,6);plot(plot_LamRate)
title('LamRate')

%% Reporting the results
minimum = parentParam;
elapsedTime = toc (totalFunctionTime);
reportResults(1,1) = parentParam(1);
reportResults(1,2) = parentParam(2);
reportResults(1,3) = oldLength;
reportResults(1,4) = annealCount;
reportResults(1,5) = elapsedTime;
end
```

Random Restart Hill Climbing algorithm implementation in MATLAB (for function of two variables):

```
%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EE 5991 Algorithm - 3 %%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% Random Restart Hill Climbing %%%%%%%%%%%%%%

function [reportResults] =
randomRestartHillClimbing_2var_rev1(testFunction,initialGuess)
reportResults = ones(1,5);
syms x y a b h; %symbols for calculating derivative etc.
nextIterationPoint = initialGuess;
internalFunctionUpdate = testFunction;
fValUpdate = [];
pointsUpdate = [];
resultsMemory = zeros(1,3);
intermediateResult = zeros(1,3); % For concatenating results to form
the results matrix
zeroReplaced = 0;
matchFound = 0;
iterationLimit = 1000;
Results = [];
% To generate random moves / preturb algorithm when "stuck" at a local
% minima
RandomMove =
@(param)(param+(randperm(length(param))==length(param))*randn*5);%,...
```

```
fig3 = figure;
set([fig3],'handlevisibility','on');
set(0,'CurrentFigure',fig3);
set(fig3,'name','RANDOM - RESTART HILL CLIMBING
ALGORITHM','numbertitle','off')
hList3 = uicontrol(fig3,'Style','text','Position',[5 350 150 200]);

% time / performance calculation
totalFunctionTime_2  = tic;

% RRGD code:
for i=1:1:iterationLimit
    computeFuncVal = subs(testFunction, {x,y},
{nextIterationPoint(1),nextIterationPoint(2)});
    if i == 1
        minFuncVal = computeFuncVal
    end
    fValUpdate = [fValUpdate;computeFuncVal];
    pointsUpdate = [pointsUpdate;nextIterationPoint];
    % Compute partial derivates (symbolic)
    derX_symb = diff(internalFunctionUpdate,x);
    derY_symb = diff(internalFunctionUpdate,y);

    % Substitute derivative with current coordinates to calculate
partial
    % derivatives
    derX =
subs(derX_symb,{x,y},{nextIterationPoint(1),nextIterationPoint(2)});
    derY =
subs(derY_symb,{x,y},{nextIterationPoint(1),nextIterationPoint(2)});

    % Determine next coordinates based on current coordinates and h
    % (symbolic)
    nextIterationPoint_X = sym(nextIterationPoint(1) + h*derX);
    nextIterationPoint_Y = sym(nextIterationPoint(2) + h*derY);

    % Determine function value in terms of h alone, not x and / or y
    updateFuncVal = subs(internalFunctionUpdate, {x,y},
{nextIterationPoint_X,nextIterationPoint_Y});

    % Find derivative of the function with respect to h
    derF = diff(updateFuncVal,h);

    % Find the root
    solvedH = solve(derF,h);

    [rowsResults columnsResults] = size(resultsMemory);
    if(derF ~= 0)
        for scanH= 1:1:length(solvedH)
            if double(imag(solvedH(scanH))) == 0
                hFinal = real(solvedH(scanH));
            end
        end
        nextIterationPoint(1) = nextIterationPoint(1) + hFinal*derX;
```

```
        nextIterationPoint(2) = nextIterationPoint(2) + hFinal*derY;

    else
        if i ~= 1
            % Scan through the matrix if a value has been replaced,
else
            % just store it in the matrix
            if (zeroReplaced == 1)
                for scanResults = 1:1:rowsResults
                    if (resultsMemory(scanResults,1)==
nextIterationPoint(1))
                        if (resultsMemory(scanResults,2)==
nextIterationPoint(2))
                            matchFound = 1;
                            break;
                        else
                            matchFound = 0;
                        end
                    else
                        matchFound = 0;
                    end
                    if (scanResults == rowsResults)
                        if matchFound == 0
                            intermediateResult(1,1) =
nextIterationPoint(1);
                            intermediateResult(1,2) =
nextIterationPoint(2);
                            intermediateResult(1,3) = computeFuncVal;
                            resultsMemory =
[resultsMemory;intermediateResult];
                        end
                    end
                end
            else
                resultsMemory(1,1) = nextIterationPoint(1);
                resultsMemory(1,2) = nextIterationPoint(2);
                resultsMemory(1,3) = computeFuncVal;
                zeroReplaced = 1;
            end
            nextIterationPoint = RandomMove(nextIterationPoint);
        end
    end

    if i == iterationLimit
        elapsedTime = toc (totalFunctionTime_2);
        for scanResultsMatrix = 1:1:rowsResults
            if scanResultsMatrix == 1
                reportResults(1,1) = resultsMemory(1,1);
                reportResults(1,2) = resultsMemory(1,2);
                reportResults(1,3) = computeFuncVal;
                reportResults(1,4) = iterationLimit;
            else
                if (resultsMemory(scanResultsMatrix,3) <
reportResults(1,3))
```

```
                    reportResults(1,1) =
resultsMemory(scanResultsMatrix,1);
                    reportResults(1,2) =
resultsMemory(scanResultsMatrix,2);
                    reportResults(1,3) =
resultsMemory(scanResultsMatrix,3);
                    reportResults(1,4) = iterationLimit;
                end
            end
        end
    end
end
reportResults(1,5) = elapsedTime;
%% Plotting the results
set([fig3],'handlevisibility','on');
set(0,'CurrentFigure',fig3);
subplot(2,1,1);plot(pointsUpdate)
title('New Points')
subplot(2,1,2); plot(fValUpdate)
title('Function Value')

%% Reporting the results on the figure
Results {1,1} = strcat ('Solution for x:
',num2str(nextIterationPoint(1)));
Results {1,2} = strcat ('Solution for y:
',num2str(nextIterationPoint(2)));
Results {1,3} = strcat ('Current cost: ',num2str(computeFuncVal));
Results {1,4} = strcat ('Local Minima: ',num2str(resultsMemory));
Results {1,5} = strcat ('Current minimum:
',num2str(intermediateResult));
set(hList3,'String',Results);  % Displays 5 lines, one result per line
end
```

Random Restart Hill Climbing algorithm implementation in MATLAB (for function of three variables):

```
%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EE 5991 Algorithm - 3 %%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% Random Restart Hill Climbing %%%%%%%%%%%%%%

function [reportResults] =
randomRestartHillClimbing_3var_rev1(testFunction,initialGuess)
reportResults = ones(1,6);
syms x y z a b c h; %symbols for calculating derivative etc.
nextIterationPoint = initialGuess;
internalFunctionUpdate = testFunction;
fValUpdate = [];
pointsUpdate = [];
resultsMemory = zeros(1,4);
intermediateResult = zeros(1,4); % For concatenating results to form
the results matrix
zeroReplaced = 0;
matchFound = 0;
iterationLimit = 1000;
```

```
Results = [];
% To generate random moves / preturb algorithm when "stuck" at a local
% minima
RandomMove =
@(param)(param+(randperm(length(param))==length(param))*randn*5);%,...

fig3 = figure;
set([fig3],'handlevisibility','on');
set(0,'CurrentFigure',fig3);
set(fig3,'name','RANDOM - RESTART HILL CLIMBING
ALGORITHM','numbertitle','off')
hList3 = uicontrol(fig3,'Style','text','Position',[5 350 150 200]);

% time / performance calculation
totalFunctionTime_2  = tic;

% RRGD code:
for i=1:1:iterationLimit
    computeFuncVal = subs(testFunction, {x,y,z},
{nextIterationPoint(1),nextIterationPoint(2),nextIterationPoint(3)});
    if i == 1
        minFuncVal = computeFuncVal
    end
    fValUpdate = [fValUpdate;computeFuncVal];
    pointsUpdate = [pointsUpdate;nextIterationPoint];
    % Compute partial derivates (symbolic)
    derX_symb = diff(internalFunctionUpdate,x);
    derY_symb = diff(internalFunctionUpdate,y);
    derZ_symb = diff(internalFunctionUpdate,z);

    % Substitute derivative with current coordinates to calculate
partial
    % derivatives
    derX =
subs(derX_symb,{x,y,z},{nextIterationPoint(1),nextIterationPoint(2),ne
xtIterationPoint(3)});
    derY =
subs(derY_symb,{x,y,z},{nextIterationPoint(1),nextIterationPoint(2),ne
xtIterationPoint(3)});
    derZ =
subs(derZ_symb,{x,y,z},{nextIterationPoint(1),nextIterationPoint(2),ne
xtIterationPoint(3)});

    % Determine next coordinates based on current coordinates and h
    % (symbolic)
    nextIterationPoint_X = sym(nextIterationPoint(1) + h*derX);
    nextIterationPoint_Y = sym(nextIterationPoint(2) + h*derY);
    nextIterationPoint_Z = sym(nextIterationPoint(3) + h*derZ);

    % Determine function value in terms of h alone, not x and / or y
    updateFuncVal = subs(internalFunctionUpdate, {x,y,z},
{nextIterationPoint_X,nextIterationPoint_Y,nextIterationPoint_Z});

    % Find derivative of the function with respect to h
```

```
    derF = diff(updateFuncVal,h);

    % Find the root
    solvedH = solve(derF,h);

    [rowsResults columnsResults] = size(resultsMemory);
    if(derF ~= 0)
        for scanH= 1:1:length(solvedH)
            if double(imag(solvedH(scanH))) == 0
                hFinal = real(solvedH(scanH));
            end
        end
        nextIterationPoint(1) = nextIterationPoint(1) + hFinal*derX;
        nextIterationPoint(2) = nextIterationPoint(2) + hFinal*derY;
        nextIterationPoint(3) = nextIterationPoint(3) + hFinal*derZ;
    else
        if i ~= 1

            % Scan through the matrix if a value has been replaced,
else
            % just store it in the matrix
            if (zeroReplaced == 1)
                for scanResults = 1:1:rowsResults
                    if (resultsMemory(scanResults,1)==
nextIterationPoint(1))
                        if (resultsMemory(scanResults,2)==
nextIterationPoint(2))
                            if (resultsMemory(scanResults,3)==
nextIterationPoint(3))
                                matchFound = 1;
                                break;
                            else
                                matchFound = 0;
                            end
                        else
                            matchFound = 0;
                        end
                    else
                        matchFound = 0;
                    end
                    if (scanResults == rowsResults)
                        if matchFound == 0
                            intermediateResult(1,1) =
nextIterationPoint(1);
                            intermediateResult(1,2) =
nextIterationPoint(2);
                            intermediateResult(1,3) =
nextIterationPoint(3);
                            intermediateResult(1,4) = computeFuncVal;
                            resultsMemory =
[resultsMemory;intermediateResult];
                        end
                    end
                end
            else
```

```
                resultsMemory(1,1) = nextIterationPoint(1);
                resultsMemory(1,2) = nextIterationPoint(2);
                resultsMemory(1,3) = nextIterationPoint(3);
                resultsMemory(1,4) = computeFuncVal;
                zeroReplaced = 1;
            end
            nextIterationPoint = RandomMove(nextIterationPoint);
        end
    end

    if i == iterationLimit
        elapsedTime = toc (totalFunctionTime_2);
        for scanResultsMatrix = 1:1:rowsResults
            if scanResultsMatrix == 1
                reportResults(1,1) = resultsMemory(1,1);
                reportResults(1,2) = resultsMemory(1,2);
                reportResults(1,3) = resultsMemory(1,3);
                reportResults(1,4) = computeFuncVal;
                reportResults(1,5) = iterationLimit;
            else
                if (resultsMemory(scanResultsMatrix,4) <
reportResults(1,4))
                    reportResults(1,1) =
resultsMemory(scanResultsMatrix,1);
                    reportResults(1,2) =
resultsMemory(scanResultsMatrix,2);
                    reportResults(1,3) =
resultsMemory(scanResultsMatrix,3);
                    reportResults(1,4) =
resultsMemory(scanResultsMatrix,4);
                    reportResults(1,5) = iterationLimit;

                end
            end
        end
    end
end
reportResults(1,6) = elapsedTime;
%% Plotting the results
set([fig3],'handlevisibility','on');
set(0,'CurrentFigure',fig3);
subplot(2,1,1);plot(pointsUpdate)
title('New Points')
subplot(2,1,2); plot(fValUpdate)
title('Function Value')

%% Reporting the results on the figure
Results {1,1} = strcat ('Solution for x:
',num2str(nextIterationPoint(1)));
Results {1,2} = strcat ('Solution for y:
',num2str(nextIterationPoint(2)));
Results {1,3} = strcat ('Solution for z:
',num2str(nextIterationPoint(3)));
Results {1,4} = strcat ('Current Wire Length:
',num2str(computeFuncVal));
```

```
Results {1,5} = strcat ('Local Minima: ',num2str(resultsMemory));
Results {1,6} = strcat ('Current minimum:
',num2str(intermediateResult));
set(hList3,'String',Results);  % Displays 5 lines, one result per line
end
```

Random Restart Hill Climbing algorithm implementation in MATLAB (for function of four variables):

```
%%%%%%%%%%%%%%% Rohit A. Bhatia %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EE 5991 Algorithm - 3 %%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% Random Restart Hill Climbing %%%%%%%%%%%%%%

function [reportResults] =
randomRestartHillClimbing_4var_rev1(testFunction,initialGuess)
reportResults = ones(1,7);
syms w x y z a b c d h; %symbols for calculating derivative etc.
nextIterationPoint = initialGuess;
internalFunctionUpdate = testFunction;
fValUpdate = [];
pointsUpdate = [];
resultsMemory = zeros(1,5);
intermediateResult = zeros(1,5); % For concatenating results to form
the results matrix
zeroReplaced = 0;
matchFound = 0;
iterationLimit = 1000;
Results = [];
% To generate random moves / preturb algorithm when "stuck" at a local
% minima
RandomMove =
@(param)(param+(randperm(length(param))==length(param))*randn*5);%,...

fig3 = figure;
set([fig3],'handlevisibility','on');
set(0,'CurrentFigure',fig3);
set(fig3,'name','RANDOM - RESTART HILL CLIMBING
ALGORITHM','numbertitle','off')
hList3 = uicontrol(fig3,'Style','text','Position',[5 350 150 200]);

% time / performance calculation
totalFunctionTime_2  = tic;

% RRGD code:
for i=1:1:iterationLimit
    computeFuncVal = subs(testFunction, {w, x,y,z},
{nextIterationPoint(1),nextIterationPoint(2),nextIterationPoint(3),nex
tIterationPoint(4)});
    if i == 1
        minFuncVal = computeFuncVal
    end
    fValUpdate = [fValUpdate;computeFuncVal];
    pointsUpdate = [pointsUpdate;nextIterationPoint];
    % Compute partial derivates (symbolic)
```

```
    derW_symb = diff(internalFunctionUpdate,w);
    derX_symb = diff(internalFunctionUpdate,x);
    derY_symb = diff(internalFunctionUpdate,y);
    derZ_symb = diff(internalFunctionUpdate,z);

    % Substitute derivative with current coordinates to calculate
partial
    % derivatives
    derW =
subs(derW_symb,{w,x,y,z},{nextIterationPoint(1),nextIterationPoint(2),
nextIterationPoint(3),nextIterationPoint(4)});
    derX =
subs(derX_symb,{w,x,y,z},{nextIterationPoint(1),nextIterationPoint(2),
nextIterationPoint(3),nextIterationPoint(4)});
    derY =
subs(derY_symb,{w,x,y,z},{nextIterationPoint(1),nextIterationPoint(2),
nextIterationPoint(3),nextIterationPoint(4)});
    derZ =
subs(derZ_symb,{w,x,y,z},{nextIterationPoint(1),nextIterationPoint(2),
nextIterationPoint(3),nextIterationPoint(4)});

    % Determine next coordinates based on current coordinates and h
    % (symbolic)
    nextIterationPoint_W = sym(nextIterationPoint(1) + h*derW);
    nextIterationPoint_X = sym(nextIterationPoint(2) + h*derX);
    nextIterationPoint_Y = sym(nextIterationPoint(3) + h*derY);
    nextIterationPoint_Z = sym(nextIterationPoint(4) + h*derZ);

    % Determine function value in terms of h alone, not x and / or y
    updateFuncVal = subs(internalFunctionUpdate, {w,x,y,z},
{nextIterationPoint_W,nextIterationPoint_X,nextIterationPoint_Y,nextIt
erationPoint_Z});

    % Find derivative of the function with respect to h
    derF = diff(updateFuncVal,h);

    % Find the root
    solvedH = solve(derF,h);

    [rowsResults columnsResults] = size(resultsMemory);
    if(derF ~= 0)
        for scanH= 1:1:length(solvedH)
            if double(imag(solvedH(scanH))) == 0
                hFinal = real(solvedH(scanH));
            end
        end
        nextIterationPoint(1) = nextIterationPoint(1) + hFinal*derW;
        nextIterationPoint(2) = nextIterationPoint(2) + hFinal*derX;
        nextIterationPoint(3) = nextIterationPoint(3) + hFinal*derY;
        nextIterationPoint(4) = nextIterationPoint(4) + hFinal*derZ;
    else
        if i ~= 1
            % Scan through the matrix if a value has been replaced,
else
```

```
            % just store it in the matrix
            if (zeroReplaced == 1)
                for scanResults = 1:1:rowsResults
                    if (resultsMemory(scanResults,1)==
nextIterationPoint(1))
                        if (resultsMemory(scanResults,2)==
nextIterationPoint(2))
                            if (resultsMemory(scanResults,3)==
nextIterationPoint(3))
                                if (resultsMemory(scanResults,4)==
nextIterationPoint(4))
                                    matchFound = 1;
                                    break;
                                else
                                    matchFound = 0;
                                end
                            else
                                matchFound = 0;
                            end
                        else
                            matchFound = 0;
                        end
                    else
                        matchFound = 0;
                    end
                    if (scanResults == rowsResults)
                        if matchFound == 0
                            intermediateResult(1,1) =
nextIterationPoint(1);
                            intermediateResult(1,2) =
nextIterationPoint(2);
                            intermediateResult(1,3) =
nextIterationPoint(3);
                            intermediateResult(1,4) =
nextIterationPoint(4);
                            intermediateResult(1,5) = computeFuncVal;
                            resultsMemory =
[resultsMemory;intermediateResult];
                        end
                    end
                end
            else
                resultsMemory(1,1) = nextIterationPoint(1);
                resultsMemory(1,2) = nextIterationPoint(2);
                resultsMemory(1,3) = nextIterationPoint(3);
                resultsMemory(1,4) = nextIterationPoint(4);
                resultsMemory(1,5) = computeFuncVal;
                zeroReplaced = 1;
            end
            nextIterationPoint = RandomMove(nextIterationPoint);
        end
    end

    if i == iterationLimit
        elapsedTime = toc (totalFunctionTime_2);
```

```
        for scanResultsMatrix = 1:1:rowsResults
            if scanResultsMatrix == 1
                reportResults(1,1) = resultsMemory(1,1);
                reportResults(1,2) = resultsMemory(1,2);
                reportResults(1,3) = resultsMemory(1,3);
                reportResults(1,4) = resultsMemory(1,4);
                reportResults(1,5) = computeFuncVal;
                reportResults(1,6) = iterationLimit;
            else
                if (resultsMemory(scanResultsMatrix,5) <
reportResults(1,5))
                    reportResults(1,1) =
resultsMemory(scanResultsMatrix,1);
                    reportResults(1,2) =
resultsMemory(scanResultsMatrix,2);
                    reportResults(1,3) =
resultsMemory(scanResultsMatrix,3);
                    reportResults(1,4) =
resultsMemory(scanResultsMatrix,4);
                    reportResults(1,5) =
resultsMemory(scanResultsMatrix,5);
                    reportResults(1,6) = iterationLimit;
                end
            end
        end
    end
end
reportResults(1,7) = elapsedTime;
%% Plotting the results
set([fig3],'handlevisibility','on');
set(0,'CurrentFigure',fig3);
subplot(2,1,1);plot(pointsUpdate)
title('New Points')
subplot(2,1,2); plot(fValUpdate)
title('Function Value')

%% Reporting the results on the figure
Results {1,1} = strcat ('Solution for w:
',num2str(nextIterationPoint(1)));
Results {1,2} = strcat ('Solution for x:
',num2str(nextIterationPoint(2)));
Results {1,3} = strcat ('Solution for y:
',num2str(nextIterationPoint(3)));
Results {1,4} = strcat ('Solution for z:
',num2str(nextIterationPoint(4)));
Results {1,5} = strcat ('Current Wire Length:
',num2str(computeFuncVal));
Results {1,6} = strcat ('Local Minima: ',num2str(resultsMemory));
Results {1,7} = strcat ('Current minimum:
',num2str(intermediateResult));
set(hList3,'String',Results);  % Displays 5 lines, one result per line
end
```

Performance comparison in MATLAB:

```
function [compPlotGenerated, plotTime] =
PerformaceComparision(inputTable, comparisonLimit)

fig4 = figure;
set(fig4,'name','Performance Comparison','numbertitle','off')

%% Performance Comparison
plotTime = [];
addToMatrix = [];
for iterationCount = 1:1:comparisonLimit
    set([fig4],'handlevisibility','on');
    set(0,'CurrentFigure',fig4);
    addToMatrix(1,1) = inputTable(iterationCount,5);
    addToMatrix(1,2) = inputTable(iterationCount,10);
    addToMatrix(1,3) = inputTable(iterationCount,15);
    if (iterationCount == 1)
        plotTime = addToMatrix;
        plotTime = [plotTime;plotTime];
    else
        plotTime = [plotTime;addToMatrix];
    end
end

plot (plotTime);
xlabel('Trial Count');
ylabel('Computation time (seconds)');
hold on;
for markPointCount = 1:1:comparisonLimit
    plot (markPointCount+1,plotTime(markPointCount+1,1),'Xb');
    hold on;
    plot (markPointCount+1,plotTime(markPointCount+1,2),'Xg');
    hold on;
    plot (markPointCount+1,plotTime(markPointCount+1,3),'Xr');
    hold on;
end

legend_1 = legend('Simulated Annealing','Adaptive Simulated
Annealing','Random Restart Hill Climbing');
compPlotGenerated = 1;
return
end
```

Report generation in MATLAB:

```
function [reportGenerated] =
ReportOnTable(inputGlobalTable,inputTimeTable,comparisonLimit)

fig5 = figure;
set(fig5,'name','Report','numbertitle','off','Position',[50 213 1266
300 ])
% Framework for reporting results:
```

```
specificAlgoResult = zeros(3,5);
inputTimeTableCorrected = inputTimeTable;
inputTimeTableCorrected(1,:)=[];
alg1TimeTotal = 0;
alg2TimeTotal = 0;
alg3TimeTotal = 0;
% SA performance tables
xCordTotalSA = 0;
yCordTotalSA = 0;
fValTotalSA = 0;
xCordAvgSA = 0;
yCordAvgSA = 0;
fValAvgSA = 0;
xCordTableSA = [];
yCordTableSA = [];
fValTableSA = [];
completeTableSA = [];
% ASA performance tables
xCordTotalASA = 0;
yCordTotalASA = 0;
fValTotalASA = 0;
xCordAvgASA = 0;
yCordAvgASA = 0;
fValAvgASA = 0;
xCordTableASA = [];
yCordTableASA = [];
fValTableASA = [];
% RRHC performance tables
xCordTotalHC = 0;
yCordTotalHC = 0;
fValTotalHC = 0;
xCordAvgHC = 0;
yCordAvgHC = 0;
fValAvgHC = 0;
xCordTableHC = [];
yCordTableHC = [];
fValTableHC = [];
rnamesT2_3_4 = [];


for algoScan = 1:1:15
    if algoScan <= 5
        specificAlgoResult(1,algoScan) = inputGlobalTable
(1,algoScan);
    else
        if algoScan <= 10
            specificAlgoResult(2,algoScan-5) = inputGlobalTable
(1,algoScan);
        else
            specificAlgoResult(3,algoScan-10) = inputGlobalTable
(1,algoScan);
        end
    end
end
```

```
%Table
for forrnamesT2_3_4 = 1:1:comparisonLimit
    if forrnamesT2_3_4 < 10
        rnamesT2_3_4 = [rnamesT2_3_4;strcat({'Trial '},{'
'},num2str(forrnamesT2_3_4))];
    else
        rnamesT2_3_4 = [rnamesT2_3_4;strcat({'Trial
'},num2str(forrnamesT2_3_4))];
    end
    if forrnamesT2_3_4 == comparisonLimit
        rnamesT2_3_4 = [rnamesT2_3_4;strcat('Average',{' '})];
    end
end

for averageTime = 1:1:comparisonLimit
    alg1TimeTotal = alg1TimeTotal +
inputTimeTableCorrected(averageTime,1);
    alg2TimeTotal = alg2TimeTotal +
inputTimeTableCorrected(averageTime,2);
    alg3TimeTotal = alg3TimeTotal +
inputTimeTableCorrected(averageTime,3);
end
avgTime1 = alg1TimeTotal / comparisonLimit;
avgTime2 = alg2TimeTotal / comparisonLimit;
avgTime3 = alg3TimeTotal / comparisonLimit;

addAverageRow = [avgTime1 avgTime2 avgTime3];

for positioningTable = 1:1:comparisonLimit
    % form SA table
    xCordTableSA =
[xCordTableSA;inputGlobalTable(positioningTable,1)];
    xCordTotalSA = xCordTotalSA +
inputGlobalTable(positioningTable,1);
    yCordTableSA =
[yCordTableSA;inputGlobalTable(positioningTable,2)];
    yCordTotalSA = yCordTotalSA +
inputGlobalTable(positioningTable,2);
    fValTableSA = [fValTableSA;inputGlobalTable(positioningTable,3)];
    fValTotalSA = fValTotalSA+ inputGlobalTable(positioningTable,3);
    % form ASA table
    xCordTableASA =
[xCordTableASA;inputGlobalTable(positioningTable,6)];
    xCordTotalASA = xCordTotalASA +
inputGlobalTable(positioningTable,6);
    yCordTableASA =
[yCordTableASA;inputGlobalTable(positioningTable,7)];
    yCordTotalASA = yCordTotalASA +
inputGlobalTable(positioningTable,7);
    fValTableASA =
[fValTableASA;inputGlobalTable(positioningTable,8)];
    fValTotalASA = fValTotalASA+ inputGlobalTable(positioningTable,8);
    % form RRHC table
    xCordTableHC =
[xCordTableHC;inputGlobalTable(positioningTable,11)];
```

```
    xCordTotalHC = xCordTotalHC +
inputGlobalTable(positioningTable,11);
    yCordTableHC =
[yCordTableHC;inputGlobalTable(positioningTable,12)];
    yCordTotalHC = yCordTotalHC +
inputGlobalTable(positioningTable,12);
    fValTableHC = [fValTableHC;inputGlobalTable(positioningTable,13)];
    fValTotalHC = fValTotalHC+ inputGlobalTable(positioningTable,13);

    if positioningTable == comparisonLimit
        % form complete SA results table
        xCordAvgSA = xCordTotalSA / comparisonLimit;
        yCordAvgSA = yCordTotalSA / comparisonLimit;
        fValAvgSA = fValTotalSA / comparisonLimit;
        xCordTableSA = [xCordTableSA;xCordAvgSA];
        yCordTableSA = [yCordTableSA;yCordAvgSA];
        fValTableSA = [fValTableSA ;fValAvgSA];
        completeTableSA = horzcat(xCordTableSA, yCordTableSA,
fValTableSA);
        % form complete ASA results table
        xCordAvgASA = xCordTotalASA / comparisonLimit;
        yCordAvgASA = yCordTotalASA / comparisonLimit;
        fValAvgASA = fValTotalASA / comparisonLimit;
        xCordTableASA = [xCordTableASA;xCordAvgASA];
        yCordTableASA = [yCordTableASA;yCordAvgASA];
        fValTableASA = [fValTableASA ;fValAvgASA];
        completeTableASA = horzcat(xCordTableASA, yCordTableASA,
fValTableASA);
        % form complete RRHC results table
        xCordAvgHC = xCordTotalHC / comparisonLimit;
        yCordAvgHC = yCordTotalHC / comparisonLimit;
        fValAvgHC = fValTotalHC / comparisonLimit;
        xCordTableHC = [xCordTableHC;xCordAvgHC];
        yCordTableHC = [yCordTableHC;yCordAvgHC];
        fValTableHC = [fValTableHC ;fValAvgHC];
        completeTableHC = horzcat(xCordTableHC, yCordTableHC,
fValTableHC);
    end
end
cnamesT1_2_3 = {'X-coordinate','Y-coordinate','Minimum Wire Length'};
rnamesT1 = {'Simulated Annealing','Adaptive Simulated
Annealing','Random-Restart Gradient Descent'};
cnamesT2 = {'Simulated Annealing','Adaptive Simulated
Annealing','Random-Restart Gradient Descent'};
dipslayTimeData = [inputTimeTableCorrected;addAverageRow];

height_T1_2_3 = 18*comparisonLimit +40;
t4height = 18*comparisonLimit +40;
YoffsetT1_2_3 = 768 / 2;
t4Yoffset = YoffsetT1_2_3 - (t4height + 30);


t1 =
uitable('Parent',fig5,'Data',completeTableSA,'ColumnName',cnamesT1_2_3
,...
```

```
    'RowName',rnamesT2_3_4,'Position',[131 YoffsetT1_2_3 365
height_T1_2_3]);
t2 =
uitable('Parent',fig5,'Data',completeTableASA,'ColumnName',cnamesT1_2_
3,...
    'RowName',rnamesT2_3_4,'Position',[501 YoffsetT1_2_3 365
height_T1_2_3]);
t3 =
uitable('Parent',fig5,'Data',completeTableHC,'ColumnName',cnamesT1_2_3
,...
    'RowName',rnamesT2_3_4,'Position',[871 YoffsetT1_2_3 365
height_T1_2_3]);
t4 =
uitable('Parent',fig5,'Data',dipslayTimeData,'ColumnName',cnamesT2,...
    'RowName',rnamesT2_3_4,'Position',[393 t4Yoffset 555 t4height]);

% Table Titles
headingYOffset_1_2_3 = YoffsetT1_2_3 + height_T1_2_3 + 5;
headingYOffset_4 = t4Yoffset + t4height + 5;
uicontrol('Style','text','Position',[131 headingYOffset_1_2_3 300
15],'String','X and Y coordinates and wire length for SA algorithm');
uicontrol('Style','text','Position',[501 headingYOffset_1_2_3 300
15],'String','X and Y coordinates and wire length for ASA algorithm');
uicontrol('Style','text','Position',[871 headingYOffset_1_2_3 300
15],'String','X and Y coordinates and wire length for RRHC
algorithm');
uicontrol('Style','text','Position',[393 headingYOffset_4 300
15],'String','Algorithm compuation time (seconds)');
reportGenerated = 1;
return
end
```

References

[1] K. Sahookar and P. Mazumder, "VLSI Cell Placement Techniques," *ACM Computing Surveys,* vol. 23, 1991.

[2] V. .R, P. S. .S, L. .R, and Kumaravel, "Evolutionary Algorithmical Approach for VLSI Physical Design-Placement Problem," *ACEEE Int. J. on Electrical and Power Engineering,* vol. 2, 2011.

[3] S. M. Sait, M. I. Ali, and A. M. Zaidi, "Multiobjective VLSI Cell Placement Using Distributed Simulated Evolution Algorithm," 2005.

[4] A. Dekkers and E. Aarts, "Global optimization and simulated annealing," *Mathematical Programming,* vol. 50, pp. 367-393, 1991.

[5] A. Dendane, "Maxima and Minima of Functions of Two Variables," 2013.

[6] V. A. Cicirello, "On the Design of an Adaptive Simulated Annealing Algorithm."

[7] J. A. Boyan, "Learning Evaluation Functions for Global Optimization," *PhD thesis,* 1998.

[8] A. Juels and M. Wattenbergy, "Stochastic Hillclimbing as a Baseline Method for Evaluating Genetic Algorithms," 1984.