2007

Comparison of Strategies for the Constraint Determination of Simulink Models

Charles Joseph Alex
The University of Montana

COMPARISON OF STRATEGIES FOR THE CONSTRAINT DETERMINATION OF SIMULINK MODELS

By

Charles Joseph Alex, IV

Bachelor of Science, Rensselaer Polytechnic Institute, Troy, NY, 1993

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

The University of Montana
Missoula, MT

Spring, 2007

Approved by:

Dr. David A. Strobel, Dean
Graduate School

Dr. Joel Henry, Chair
Computer Science

Dr. Alden Wright
Computer Science

Dr. George McRae
Mathematical Sciences

Alex IV, Charles, M.S., May 2007 Computer Science

Comparison of Strategies for the Constraint Determination of Simulink Models

Chairperson: Dr. Joel Henry

The SIMULINK environment allows rapid prototyping of complex software systems. Because many of these systems are mission-critical, it is of utmost importance to determine their input and output constraints. Determining input constraints is a trivial matter, but the constraint determination of a system's output values is a serious and challenging problem that historically has entailed an exhaustive exploration of the system's input states. The work presented in this thesis recounts and extends a research project supported by NASA whose focus was to develop a strategy to constrain the outputs of a SIMULINK model. SIMULINK models are quite similar to mathematical functions and therefore optimization algorithms can be applied to constrain the outputs. Optimizations of simple mathematical functions paved the way for random functions and finally led to the development of two optimization algorithms. During the exploration of potential optimization algorithms, strategies such as Monte Carlo, the simplex method, simulated annealing, and evolution strategy were explored. In the end, a combined approach utilizing both simulated annealing and the simplex method was compared with evolution strategy for relative strengths and weaknesses. It was determined that the evolution strategy algorithm was more suited to optimization of SIMULINK models due to its more effective usage of model calls and to its higher success rate.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. To my wife, Greta, who enabled this whole adventure. Without her willingness to set aside her own goals for mine, there would be no thesis. To Dr. Joel Henry, my friend and mentor, for his willingness to take a chance on a non-traditional student. To Dr. Jesse Johnson for his invaluable input, especially in the area of simulated annealing. To Dr. Alden Wright for his expertise in genetic algorithms.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Introduction

Ensuring that a software-based system performs according to its requirements involves the processes of *verification* and *validation*. Verification is the process of assessing to what degree the software meets the specified requirements through testing. Validation, on the other hand, is assessing how well the software fulfills its intended use and employs methods such as reviews and walkthroughs. While validation is critically important from the user's standpoint, it is often the case that the verification phase consumes more resources in its completion.

Because of the increasing speed and power of modern processors, many real-time and mission-critical systems are now computerized. Whether found in the fire control system of a military vehicle, the propulsion system of a communication satellite, or in the table control system of a radiation therapy machine, embedded systems have become ubiquitous. The pervasiveness of embedded systems requires even more stringent and comprehensive verification to try to minimize both financial loss and bodily harm.

In many cases, requirements documents double as a description of not only what functionality the software must implement, but also what system behaviors are acceptable and unacceptable. For these systems, verification becomes the process of

proving that the system exhibits no unacceptable behavior under any circumstances. Unfortunately, the only way to completely verify an embedded system is through a formal verification method such as exhaustive testing. In this strategy, each possible combination of the system's inputs is systematically tested for its effect on the system's behavior.

Verification through exhaustive system testing, despite its thoroughness, is not often used in practice. The reason is based on the system itself; many are quite complicated, having a large number of both inputs and outputs. To exhaustively test a system with a non-trivial number of inputs and outputs quickly explodes into a combinatorial nightmare. Even under the assumption that the number of input/output combinations could be enumerated, the amount of time required to test each value in the enumeration would be prohibitive.

Rather than attempting to exhaustively test systems, most system testers use one or more heuristics. A heuristic is a relaxed algorithm that makes no guarantees about runtime or quality of the results obtained, and are generally thought of as "rules of thumb" rather than algorithms. Despite the informal nature of heuristics, in many cases heuristic runtime and quality are both good enough for verification purposes. In fact, heuristics can often be tailored specifically to the system under review, allowing an even greater savings in terms of both runtime and accuracy.

Heuristics are usually designed to find an approximation to the optimal solution to a problem given time and space constraints. To tailor one for the purpose of system verification would involve recasting the expected system outputs into a state-space of $n+1$ dimensions, where each of the n dimensions is contributed by one of the system's input values, and the extra dimension is contributed by one of the system's output values. In this way, the approximate optimal value of each of the system outputs can be independently determined; that is, the combination of inputs that produces the

output with the most optimal value will be discovered. Comparing this optimal value to the expected system output will determine in part whether the system is verified.

Simulink is a software product that provides an environment used for the rapid creation and simulation of embedded systems. Simulink models are composed of a number of connected blocks, and each block can represent anything from a simple mathematical function to a multiple input vehicular control subsystem. The Simulink environment includes a simulator that allows the analysis of the newly created system dynamically.

1.2 Motivation

The original idea for this thesis came in part from research into constraint determination of simulated systems led by Dr. Joel Henry at the University of Montana. This research, sponsored through a grant from NASA, sought to show by proof of concept that the output constraints of a Simulink model could be determined using heuristics; that is, that the output values of a Simulink model could be determined to fall approximately within a specific range of values. A secondary goal of this research was to determine the statistical validity of such an optimization strategy, which would necessarily be heuristic in nature.

However, this thesis is concerned with a comparison of optimization strategies. It was not until after the completion of the original NASA grant that the rest of this thesis was conceived. Exposure to advanced heuristic strategies as well as the No Free Lunch Theorem (NFLT) prompted a reopening of the investigation into optimization techniques. The NFLT is described by Wolpert and Macready [1995] as the following:

> *"all algorithms that search for an extremum of a cost function perform exactly the same, when averaged over all possible cost functions."*

Thus, further investigation is prudent if only to stave off an optimization heuristic's inevitable poor performance on a SIMULINK model. More to the point, for critical embedded systems, the expectation of performance is so high and the cost of failure is so dear both in financial and human terms, that a thorough and layered approach to optimization is required.

1.3 Goal

The goal of this thesis is two-fold. The first main goal is to show that the optimization of SIMULINK models is not only possible but practical. The achievement of this goal will require the development of an optimization strategy that not only has a manageable and realistic runtime but also provides results with a reasonable amount of accuracy. The nature of heuristics, when combined with the complexity of SIMULINK models, prevents a declaration of optimality for any heuristic. However, under the guise of proof of concept, it is possible to show the validity of a combinatorial optimization heuristic.

The second major goal of this thesis is to determine which class of optimization heuristics is most appropriate for constraint determination of SIMULINK models. In this regard, relevant metrics including scalability, runtime, accuracy, and repeatability will be compared for the heuristics identified as a result of the first goal of this thesis. Scalability will be determined by examining how the number simulation calls required to optimize the model increases with an increasing number of model inputs. An examination of runtime will have a basis similar to scalability and will highlight differences in the implementations of the optimization heuristics. Accuracy will be

determined by how close the experimental results are to the actual optimal value. Finally, the accuracy of a series of optimizations will be the basis for repeatability.

1.4 Benefits

SIMULINK models are the basis for mission-critical systems that find usage in satellites, aircraft, and vehicles. Because of both the large financial investment involved and the potential impact on human life, it is imperative that these systems are adequately tested prior to deployment. Testing at the prototype stage with a SIMULINK model is faster, more controllable, and less costly than traditional prototype testing. As stated above, it is not feasible to simply enumerate all possible combinations of inputs to determine the optimal configuration. Designing and implementing an optimization tool for SIMULINK models will have far-reaching effects in both financial and safety terms.

Despite the fact that this thesis stops short of implementation of production verification software, the knowledge gained through the research of optimization heuristics is vital in advancing the process of verification and validation of embedded systems. While the extension of the heuristics presented in this thesis to a general purpose optimization application would not be trivial, the implementations created herein provide a framework that could guide future researchers in this area.

1.5 Thesis Organization

The rest of this thesis is organized as follows:

- **Chapter 2** provides a review of key concepts used in this research.
- **Chapter 3** addresses the implementations of the optimization heuristics and a review of the experimental SIMULINK model used.
- **Chapter 4** describes the experimental procedures used, the test data sets collected, and results.
- **Chapter 5** contains concluding remarks, as well as an outline for future work.

CHAPTER 2 OVERVIEW

2.1 Optimization

The process of optimization would seem to be fairly easy to define, and a naive definition would be

> *"The process of determinating a set of values for a system that gives the best results."*

However, this simple definition is quickly entangled by the possible definitions of *best results*. It is not clear exactly what will be optimized, nor what the result of the optimization will be.

One possible resolution of this problem is to treat the process of optimization as an examination of the input values for a system and how they affect the system's output. In this case, the system's inputs are being optimized, with the result being the unique combination of input values that produce the best system output.

A concrete example of this type of optimization is a software model of an anti-lock braking system, which is concerned with minimizing stopping time during a braking event. The software model would have a number of inputs, such as wheel speed, vehicle speed, and the acceptable amount of slip. Its output of stopping time would be determined through simulation of the software model. An optimizing algorithm tailored for such a problem would be mainly concerned with minimizing the stopping time, and not necessarily concerned with runtime or accuracy.

Another resolution to the ambiguity of *best results* is to include in the optimization definition the runtime of the optimizing algorithm. One way to do this is to set the maximum allowable amount of time to spend searching for an optimal value. The system is still being optimized, but only for a fixed amount of time. As a result, the value determined to be optimal may be only the best value seen during the limited optimization process rather than the true optimal value.

As an example, consider the problem of routing airplanes. Clearly, there is an optimal solution that only involves the airplanes and their relative positions. However, determining the true optimal arrangement of airplanes is less important than determining a timely arrangement that is safe. Ignoring the runtime of the optimization algorithm would likely produce overly optimal results that were too slow in arriving to be useful. While most problems are time-sensitive to one degree or another, the airplane routing problem is far more pressing in terms of urgency.

One final sticking point in the original definition centers on the meaning of *best*. Many systems are so complex that any definition of *best* degenerates into *best seen*. The traveling salesman problem highlights this difficulty. Even for a small number of cities, the number of possible routes is enormous. Without analyzing every route, it is impossible to determine whether any route is the optimal.

To combat this, statistical approaches are employed to place some qualitative value on what exactly *best* means. A *confidence interval analysis* (see Law and Kelton [1999]) of experimental data will provide both an error tolerance and a level of confidence. The error tolerance specifies a range of values inside which the optimal value is expected to reside, while the level of confidence denotes how confident the analysis is that the optimal value is within the bounds of the error tolerance. These two properties together define a confidence interval and lend statistical validity to an optimizing heuristic, with the further assumption that the data has a standard

distribution. This is an important assumption, as the underlying data may not be standard; in this case, the confidence interval is not valid and other methods would be required to qualitatively assess the results of the optimizing heuristic.

For the scope of this work, optimization refers to the process of discovering the values of each of a SIMULINK model's inputs that produce the greatest output value. Reflecting the potential negative impact of runtime, the algorithm will be further constrained to limit the runtime, thus producing the best seen output value. System states analyzed in optimizing the SIMULINK model will be stored so that a confidence interval analysis can be performed. In this way, the algorithm will clearly define what the process of optimization includes, what the results of the optimization mean, and the level of confidence that the results are truly optimal.

More formally, this work is focused on determining constraints on a SIMULINK model's output values. This overarching goal is subdivided into two optimizations for each of the outputs. One optimization would be required to determine the global maximum for the output while the other would search for the global minimum. In the case of PEAKS, a built-in MATLAB function discussed in Section 2.1.1, a constraint determination on its output might result in the following two optimizations, where:

$$\begin{aligned} O_{1,max} &= 8.0752 \quad \text{(Output 1's global maximum)} \\ O_{1,min} &= -6.5466 \quad \text{(Output 1's global minimum)} \end{aligned}$$

which then can be combined into a general result for the constraint determination:

$$-6.5466 \leq O_1 \leq 8.0752$$

2.1.1 Dimensionality

Dimensionality refers to the number of dimensions a function's geometry occupies. In simplest terms, the sum of a function's inputs and outputs is a dimension. As an example, consider the MATLAB function PEAKS, whose underlying equation is shown in Equation (2.1) and graphical representation is shown in Figure 2.1.

$$\begin{aligned} z =& 3(1-x)^2 e^{-(x^2)-(y+1)^2} - \dots \\ & 10(\frac{x}{5} - x^3 - y^5) e^{-x^2-y^2} - \dots \\ & \frac{1}{3} e^{-(x+1)^2-y^2} \end{aligned} \tag{2.1}$$

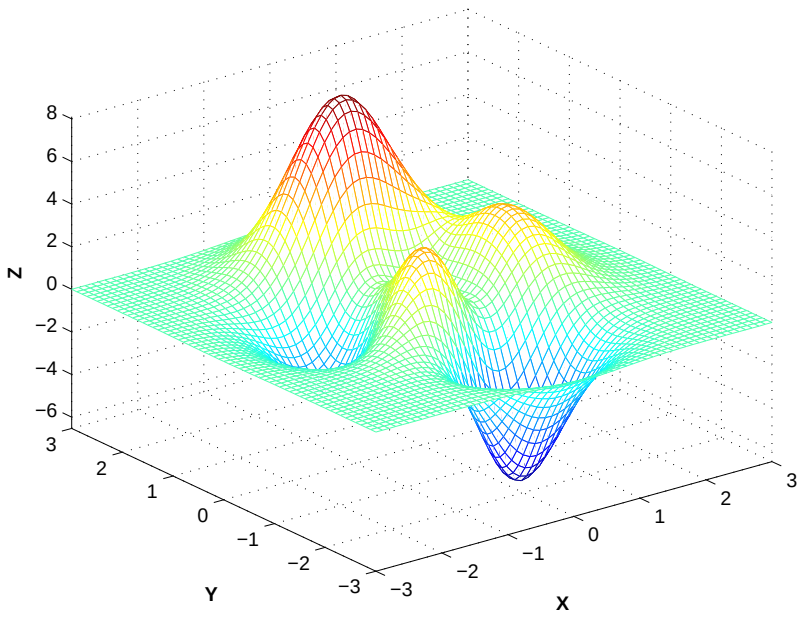


Figure 2.1 The MATLAB function PEAKS

As can be seen in Figure 2.1, PEAKS has two input dimensions, x and y, and one output dimension z. Therefore, PEAKS has a dimensionality of 3. Of course, the

dimensionality of PEAKS is apparent in Figure 2.1. Unfortunately, larger dimensionality functions are not so easily displayed visually and so the only way to determine dimensionality is to sum the number of inputs and outputs. SIMULINK models are a good example of high dimensionality systems, with the added quality of having not only multiple inputs but multiple outputs.

Another aspect of dimensionality is the division of the input or output space. A system's geometry may be comprised of discrete or continuous states. A discrete geometry would have clearly defined and separate states, while a continuous geometry would have an infinite number of states. In example, if PEAKS had a discrete geometry, then its allowable x and y dimension values might be

$$\begin{aligned} x &\in \{-3, -2, -1, 0, 1, 2, 3\} \\ y &\in \{-5/2, -3/2, -1/2, 1/2, 3/2, 5/2\} \end{aligned}$$

with resulting z values being the result of drawing one of each of the x and y values from above and applying Equation (2.1). It is not hard to see that there will be 42 possible values for z, as there are 6 allowable values for x and 7 for y.

If, on the other hand, PEAKS had a continuous geometry, then both the x and y dimensions would have an infinite number of allowable values. Machine precision would be the only limiting factor in determining whether one state was the same or different from its next nearest neighbor. Assuming a processor allowed 3 decimal digits of precision, then some allowable x and y dimension values might be

$$\begin{aligned} x &\in \{\ldots, -2.001, -2.000, -1.999, -1.998, \ldots\} \\ y &\in \{\ldots, 1.706, 1.707, 1.708, 1.709, 1.710, \ldots\} \end{aligned}$$

Continuing the assumption to the z dimension also produces an infinite number of values. In the case of continuous geometry, therefore, constraints on the inputs become important in limiting the scope of the optimization problem. Of course, the mathematics of infinity apply equally at all scales, but in software-based constraint determination, such arbitrary precision is often neither necessary nor attainable. In terms of constraints, then, the PEAKS x and y dimensions might be described as

$$-3 \leq x \leq 3$$

$$-3 \leq y \leq 3$$

Dimensionality will be managed in this thesis by optimizing a SIMULINK model for each of its outputs separately and independently. In other words, a SIMULINK model will be constrained as a series of multiple input, single output models. For each output, a full constraint determination will be performed using only that output as a guide to optimize the inputs. Thus, a SIMULINK model with inputs of $\{I_1, I_2, I_3, I_4, I_5\}$ and outputs of $\{O_1, O_2, O_3\}$ will be constrained using the following 'models'

$$M_1 = \{I_1, I_2, I_3, I_4, I_5, O_1\}$$

$$M_2 = \{I_1, I_2, I_3, I_4, I_5, O_2\}$$

$$M_3 = \{I_1, I_2, I_3, I_4, I_5, O_3\}$$

2.1.2 Combinatorics

The PEAKS function described above is a rather trivial one, and is relatively easy to optimize. However, SIMULINK models have arbitrary numbers of both inputs and outputs. Attempting to exhaustively evaluate them in search of optimal output values is futile. The reason for this is the sheer number of points that would be necessary

to examine to complete the exhaustive examination.

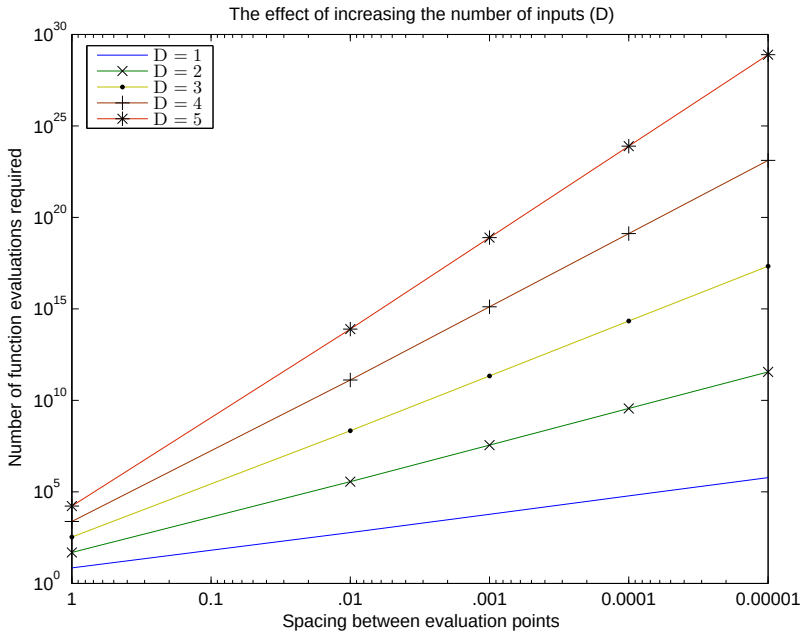


Figure 2.2 Exhaustive constraint determination evaluations required based on spacing

Figure 2.2 shows the effect of increasing the number of inputs and value spacing on an exhaustive search. To further reinforce the point, assume that the PEAKS function has a discrete geometry with spacing of $\Delta x = 0.1$ and $\Delta y = 0.1$. In this case, the x and y dimensions would both contain 61 values, leading to 3721 possible combinations. A general formula for the number of input states IS in terms of input dimensions I_n and value spacing ΔI_n, where I_{nU} and I_{nL} are the upper and lower constraints on input n, is shown in Equation (2.2).

$$IS = \left(\frac{I_{1U} - I_{1L}}{\Delta I_1} + 1\right)\left(\frac{I_{2U} - I_{2L}}{\Delta I_2} + 1\right)\cdots\left(\frac{I_{nU} - I_{nL}}{\Delta I_n} + 1\right) \tag{2.2}$$

Of note in Figure 2.2 is that as the dimension spacing values diminish toward the right of the graph, the number of required evaluations as well as the input space

more closely approximate those values found in a continuous input space. Thus, a continuous space can be thought of as a fine-grained discrete space.

In order to control the explosion of combinatorics for Simulink model optimization, one or more concessions can be made. One useful concession is to limit the precision to which the output values are optimized. Limiting precision can have a dramatic impact on the number of evaluations required and is safe as long as the model's acceptable minimum precision is known.

Another important concession is to assume that model's underlying function is continuous and differentiable. With this assumption, the requirement of exhaustive exploration for optimization is eased. For a continuous and differentiable function, a location with a certain output value is likely to have adjoining points with similar output values. This concession reduces the number of required evaluations by introducing an intelligent search that samples locations of interest rather than blindly testing every location.

One caveat for a partial search such as this is that because not all locations are evaluated, it is possible to miss the optimal value. Although a partial search will determine an approximate optimal value, it avoids the combinatorial problem and allows for multiple optimizations to be performed in far less time than one exhaustive analysis would require.

This thesis will assume that Simulink models can be represented by continuous and differentiable functions. The main reason for this assumption is that a function with numerous gross discontinuities could *only* be optimized using an exhaustive approach. This kind of behavior would not be acceptable in any embedded system, let alone a critical one, and so is ignored for the scope of this work. Indeed, an exhaustive search would likely discover gross discontinuities in the search space that are otherwise unreachable due to the system's internal logic. While it may be obvious, another very

important assumption is that a partial search using intelligent methods can produce optimal value results similar to those obtained using an exhaustive method. For without this assumption, there would be no impetus for research in this area.

2.2 Deterministic algorithms

An algorithm is comprised of a finite set of commands that will not only accomplish some task but also finish in a finite amount of time. Deterministic algorithms are one such class of algorithms, and it is these kinds of algorithm that make up the vast majority of existing algorithms. The usefulness of deterministic algorithms lies in their repeatability and predictability. Such an algorithm, when given a particular starting state and input values, will always terminate at the same end state. All the states encountered during the execution of the algorithm will be identical as well.

As an example, consider the steepest descent algorithm [Avriel, 2003]. The layout of the steepest descent algorithm is as follows:

> While *the optimal value has not been found*
>
> 1. *Determine the output value of the current location*
> 2. *Determine the output values of all adjacent locations*
>
> (a) *If none of the adjacent locations has an output value closer to the optimal value than the current location, then the current location is the optimal value*
>
> (b) *Otherwise, the adjacent location whose output value is farthest from the current location is the new current location*
>
> End While

It is not hard to see how this algorithm might fare in optimizing PEAKS. In a typical example, if the algorithm were given an initial location of $(-1.0, 1.0, 0.2289)$

and assuming the input spacing were $\Delta x = \Delta y = 0.5$, the possible moves from this position and their respective descent values are shown in Table 2.1.

x	y	z	Δz
-1.5	1.0	-0.8639	-1.0928
-1.0	1.5	2.6076	2.3787
-0.5	1.0	2.7942	2.5653
-1.0	0.0	-1.6523	-1.8812

Table 2.1 Potential moves from $(-1.0, 1.0, 0.2289)$

x	y	z	Δz
-1.0	1.0	0.2289	-2.5653
-0.5	1.5	6.1956	3.4014
0.0	1.0	3.6886	0.8944
-0.5	0.0	1.4796	-1.3146

Table 2.2 Potential moves from $(-0.5, 1.0, 2.7942)$

The next step would depend on whether the algorithm was configured to be maximum-seeking or minimum seeking. In the case of a minimum seeking configuration, the two points with negative values for Δz would be compared, resulting in a movement to $(-1.0, 0.0, -1.6523)$. For this example, however, the algorithm will be used to find the global maximum. With the assumption of a maximum-seeking algorithm, the selection of $(-0.5, 1.0, 2.7942)$ is correct. From this new locale, the possible moves and their descent values are shown in Table 2.2.

In this case, as in the previous one, there are two possible moves toward the global maximum, and the one chosen is $(-0.5, 1.5, 6.1956)$. Looking at the possible moves from this position gives values shown in Table 2.3.

This examination reveals that there is only one move that improves on the current position. Making that choice gives the values shown in Table 2.4. As can be seen, there are no legal maximum seeking moves. The algorithm then terminates, returning the global maximum value of $(0.0, 1.5, 7.9966)$.

x	y	z	Δz
-1.0	1.5	2.6076	-3.5880
-0.5	2.0	4.5569	-1.6387
0.0	1.5	7.9966	1.8010
-0.5	0.0	1.4796	-4.7160

Table 2.3 Potential moves from $(-0.5, 1.5, 6.1956)$

x	y	z	Δz
-0.5	1.5	6.1956	-1.8010
0.0	2.0	5.8591	-2.1375
0.5	1.5	6.2513	-1.7453
0.0	0.0	0.9810	-7.0156

Table 2.4 Potential moves from $(0.0, 1.5, 7.9966)$

It should be easy to see that restarting the algorithm at the original initial location would produce the same results. At no point in the algorithm are random choices or other stochastic methods employed, and so the internal logic of the algorithm will shape its execution and transition between states. Equivalent initial states will produce equivalent termination states as well as equivalent intermediary states. In other words, the steepest descent algorithm is a deterministic optimizing algorithm whose ability to optimize is dependent only on the initial conditions supplied to it.

2.2.1 The simplex method

The simplex method, also known as the Nelder-Mead method, is a deterministic algorithm based on a simplex [Nelder and Mead, 1965]. For a function having an input space having n dimensions and an output space of m dimensions, a simplex is an ordered collection of $n + m$ vertices . Each vertex is an ordered collection of values from each of the input and output dimensions. The simplex method operates by identifying the weakest vertex or vertices in the simplex and moving them to more optimal locations.

$$z = x^2 + y^2 \tag{2.3}$$

In a trivial example, an optimization of a function such as the one shown in Equation (2.3) that has $n = 2$ input dimensions and $m = 1$ output dimension would require a simplex s of size $n + m = 3$. The simplex would be a series of ordered tuples having the form (x, y, z), as shown below.

$$s = \{(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)\}$$

A simplex placed in the function space might have the vertices

$$s = \{(-1, 2, 5), (1, 1, 2), (0, 0, 0)\}$$

The graph of such a simplex is shown in Figure 2.3. Note that in technical terms a simplex is a convex hull of all its vertices, but for the simplex method the lines between the vertices are mainly to highlight the simplex's shape and their only role is in the determination of the movement of the simplex about the function space.

The vertices within the simplex are ordered in descending optimality of the output

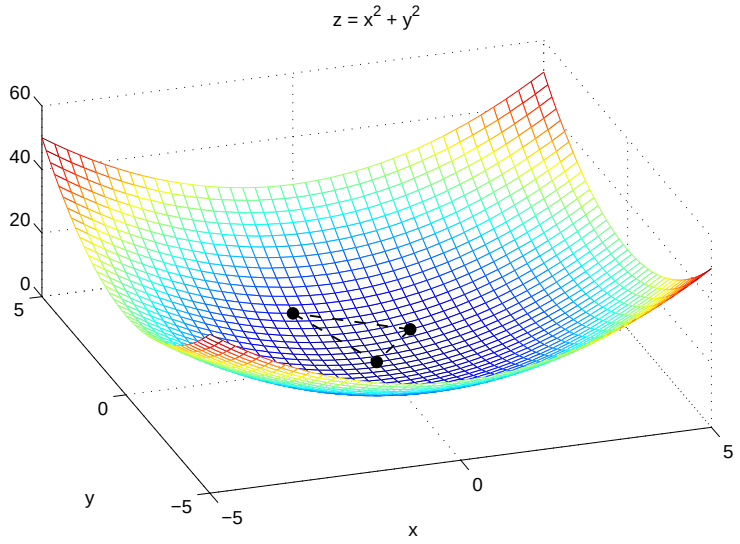


Figure 2.3 A simplex placed in the function space

values. If the object of optimization of the trivial function described above were to determine the global minimum, then the simplex would be ordered as

$$s = \{(0, 0, 0), (1, 1, 2), (-1, 2, 5)\}$$

If, on the other hand, the goal were to find the global maximum, then the simplex would be ordered as

$$s = \{(-1, 2, 5), (1, 1, 2), (0, 0, 0)\}$$

The way in which a simplex is used to optimize a function is by moving the worst of its vertices to a better location. Usually, as will be shown, only the single worst vertex is moved; however, in certain conditions all but the best vertex are moved. The result of these movements is that the overall optimality of the simplex improves with each iteration.

A general description of the simplex method is given below. Note that the following algorithm uses a simplex s where vertices are arranged from most optimal vertex to least and n refers to the number of vertices contained within the simplex:

$$s = V_1, V_2, \cdots, V_{n-1}, V_n$$

WHILE *the optimal value has not been found*

1. *Arrange the vertices in optimal-descending order*
2. *Reflect* V_n *(the worst vertex) through the other vertices' mean, producing* V_r
3. *If the new vertex* V_r *is the best of the vertices in the simplex*
   (a) *Reflect and grow the new vertex* V_r*, producing* V_{r2}
   (b) *If the new vertex* V_{r2} *is still the best*
      i. *The simplex is now ordered* $s = \{V_{r2}, V_1, V_2, \cdots, V_{n-1}\}$
   (c) *Else keep the originally reflected vertex value* (V_r)
      i. *The simplex is now ordered* $s = \{V_r, V_1, V_2, \cdots, V_{n-1}\}$
4. *Else if the new vertex's value* V_r *is better than the second-worst vertex* V_{n-1}
   (a) *Keep the reflected vertex* V_r
   (b) *The simplex is now ordered* $s = \{V_1, \cdots, V_r, \cdots, V_{n-1}\}$
5. *Else*
   (a) *Reflect and shrink the worst* V_n*, producing* V_{rs}
   (b) *If the new vertex* V_{rs} *is better than the second-worst vertex* V_{n-1}
      i. *Keep the reflected and shrunk vertex* V_{rs}
      ii. *The simplex is now ordered* $s = \{V_1, \cdots, V_{rs}, \cdots, V_{n-1}\}$
   (c) *Else shrink the worst vertex* V_n *toward the other vertices' mean, producing* V_s
      i. *If the new vertex* V_s *is better than the second-worst vertex* V_{n-1}
         A. *Keep the shrunk vertex* V_s
         B. *The simplex is now ordered* $s = \{V_1, \cdots, V_s, \cdots, V_{n-1}\}$
      ii. *Else, contract all vertices toward the best vertex, producing* $\{V_{2c}, \cdots, V_n\}$
      iii. *The simplex has the vertices* $s = \{V_1, V_{2c}, \cdots, V_{(n-1)c}, V_{nc}\}$

END WHILE

Reflection in the algorithm above entails determining the mean of all the vertices other than V_n. The mean is a temporary vertex, V_m, that allows a line v to be created from V_n to V_m. This line is then extended through V_m an equal distance to its original length. The end of this line is considered the reflection of V_n because V_m acts as a mirror, reflecting V_n to V_r.

Reflection and growing begins with the reflection described above. Rather than extending the line an equal amount, however, the line is extended twice its original length past V_m after reflecting. Reflection and shrinking has a similar mechanism, but in this case the line is extended past V_m by half its original length. Figure 2.4 shows the reflection-based vertex movements.

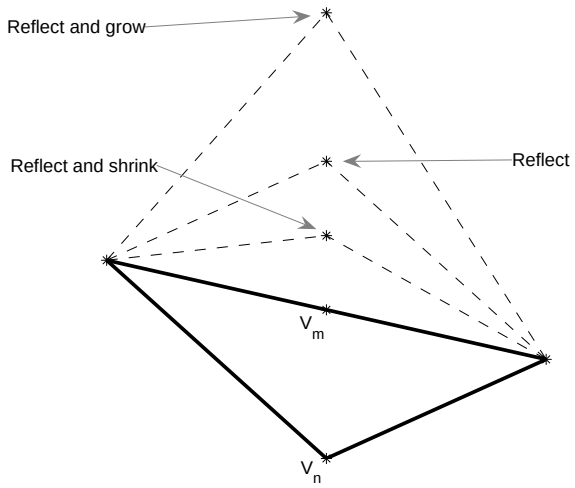
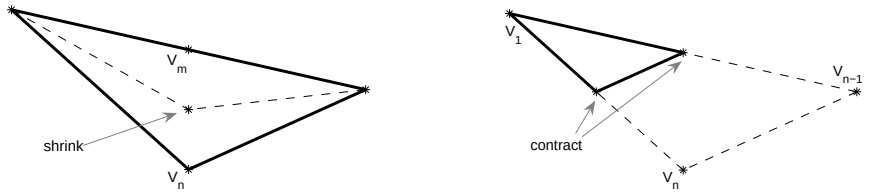


Figure 2.4 Simplex method reflection-based movements

Shrinking reduces the extension even further by setting the new vertex V_s as halfway between V_n and V_m. Lastly, contraction involves moving each vertex halfway toward the best vertex. These operations are shown in Figure 2.5(a) and Figure 2.5(b).

With the exception of the contraction movement, all of the simplex movements result in an improvement of at least one of the simplex's vertices. This feature makes



(a) Simplex method shrink movement

(b) Simplex method contract movement

Figure 2.5 Simplex method shrink and contract movements

the simplex attractive as an optimizing strategy because other strategies make no such guarantee. In addition, one of the defining characteristics of the simplex is its ability to adapt its shape to match the function space. In example, as the simplex encounters a region of relative flatness, its shape will expand to cross that region efficiently by using reflect and grow movements. Likewise in constricted regions, the simplex will use contracting movements to reduce its size enough to fit in those areas. Because of its adaptability and its constant improvement, the simplex method is useful in quickly and accurately determining optimal values.

However, there are a caveats. The simplex method is designed to find an optimal value, but whether this value is the global optimum or simply a local optimum cannot be determined. Furthermore, slight variations in the initial conditions can have a major impact on the algorithm results. Consider the PEAKS function as optimized by the simplex method with the following initial simplex s:

$$s = \{(2.125, 1.25, 0.2804), (2.5, 0.5, 0.2292), (3.0, 1.0, 0.0124)\}$$

The result of such an optimization is shown in Figure 2.6. Note that the PEAKS geometry has been rotated $90°CW$ to better display the simplex's movements. As would be expected, the simplex method has found the global maximum value, and

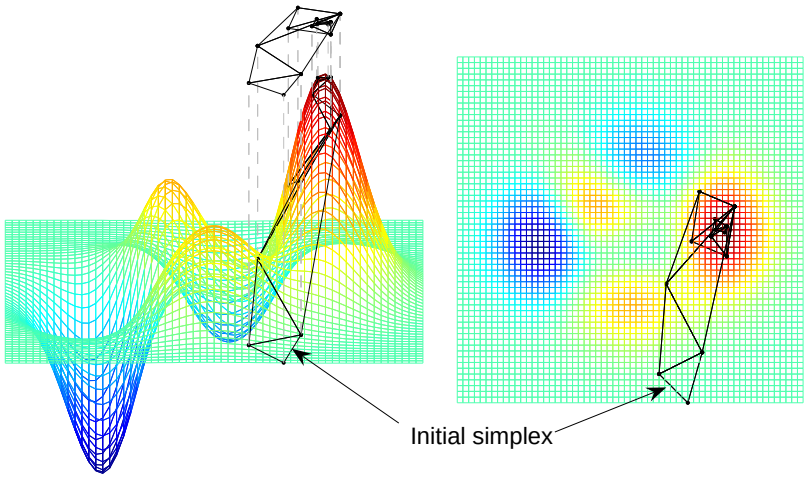


Figure 2.6 Simplex method finding the PEAKS global maximum

has done so in 11 movements. However, slightly modifying the initial simplex to

$$s = \{(2.5, 0.5, 0.2292), (2.25, 1.25, 0.1858), (3.0, 1.0, 0.0124)\}$$

produces a much different result. Referring to Figure 2.7, it is obvious that although the simplex method has finished in 6 movements, it has found a local maximum rather than the global maximum. Thus, the simplex method is sensitive to variations in initial conditions. This behavior is problematic, as it implies that successful optimization using this method would require suitable initial conditions. But knowing what initial conditions are suitable is itself an optimization problem.

Several solutions to these problems present themselves. One solution is to employ multiple simplices with the assumption that one or more of them has suitable initial conditions. While this will improve the chances of finding the global maximum, there is no guarantee that any of the simplices will do so. Furthermore, it reduces the usefulness of the algorithm by significantly increasing its runtime. Another problem is determining an appropriate number of simplices to use.

The obvious solution is to determine suitable initial conditions and then optimize using one simplex. In other words, if the simplex starts in the right area, it will consistently and accurately determine the global maximum. This approach has the benefit of maximizing the usefulness of the simplex method in terms of its quick runtime, but is complicated by the problem of determining the initial conditions.

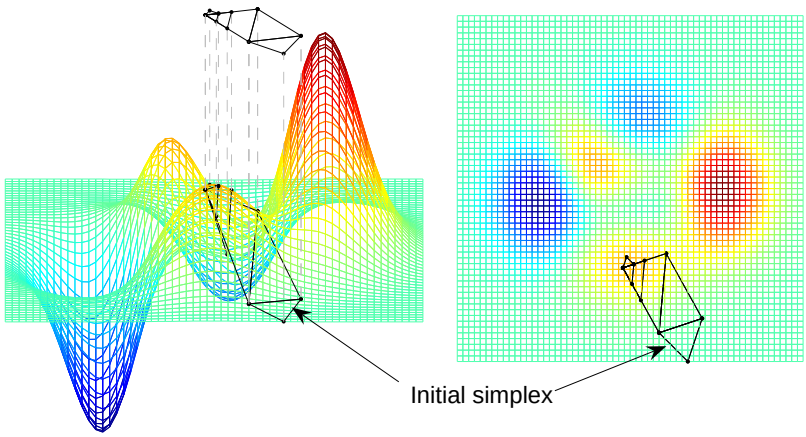


Figure 2.7 Simplex method finding the PEAKS local maximum

2.3 Stochastic algorithms

Whereas a deterministic algorithm is defined by its repeatability and consistency, a stochastic algorithm is defined by its non-repeatability. Stochastic algorithms involve an element of randomness in one or more of their decisions, which over the course of repeated executions of an algorithm can lead to many more states than would be seen by a similar algorithm that was deterministic in nature. Whereas deterministic algorithms, given specific initial conditions, will produce one answer regardless of the number of times they are executed, stochastic algorithms when given the same initial conditions have the potential to produce unique answers with each execution. This

is not necessarily a weakness of deterministic algorithms, however, because they can easily be given different initial conditions with each execution.

As an example of a stochastic algorithm, recall the Steepest Descent algorithm discussed in Section 2.2. Converting this deterministic algorithm to a stochastic one can be done in various ways, and one such way might involve the introduction of a random choice into the decision of which new location to move to. Such a modification, which can be thought of as Stochastic Descent, is shown algorithmically below. Note that the Stochastic Descent algorithm has been created for the purpose of instruction and to show the differences between deterministic and stochastic algorithms.

WHILE *the optimal value has not been found*

1. *Determine the output value of the current location*
2. *Determine the output values of all adjacent locations*
   (a) *If none of the adjacent locations has an output value closer to the optimal value than the current location, then the current location is the optimal value*
   (b) *Otherwise, randomly choose one of the adjacent locations with an output value closer to the optimal value than the current location to be the new current location*

END WHILE

The above algorithm can be used as a guide to optimize PEAKS. Using the same initial conditions as those presented in the original discussion of the Steepest Descent algorithm, which were an initial location of $(-1.0, 1.0, 0.2289)$ and input spacing of $\Delta x = \Delta y = 0.5$, the possible moves from this position and their respective descent values are shown in Table 2.5.

Assuming the algorithm is maximum seeking, then there are two possible moves.

Of the two moves, $(-0.5, 1.0, 2.7942)$ is randomly chosen as the new position. The new legal moves from this position are shown in Table 2.6.

In this case there are two legal moves toward the global maximum, and a random choice between them results in a movement to $(0.0, 1.0, 3.6886)$. From this position, the possible moves are shown in Table 2.7.

At this point there is only one legal move toward the global maximum, resulting in a move to $(0, 1.5, 7.9966)$. Looking at the possible moves from this position gives the values shown in Table 2.8.

As can be seen, there are no legal maximum seeking moves. The algorithm then terminates, returning the global maximum value of $(0.0, 1.5, 7.9966)$. Note that in this example, both Steepest Descent and Stochastic Descent found the global maximum in 3 movements. However, a subsequent optimization of PEAKS using Steepest Descent might proceed through the locations shown in Table 2.9.

x	y	z	Δz
-1.5	1.0	-0.8639	-1.0928
-1.0	1.5	2.6076	2.3787
-0.5	1.0	2.7942	2.5653
-1.0	0.0	-1.6523	-1.8812

Table 2.5 Potential moves from $(-1.0, 1.0, 0.2289)$

x	y	z	Δz
-1.0	1.0	0.2289	-2.5653
-0.5	1.5	6.1956	3.4014
0.0	1.0	3.6886	0.8944
-0.5	0.0	1.4796	-1.3146

Table 2.6 Potential moves from $(-0.5, 1.0, 2.7942)$

x	y	z	Δz
-0.5	1.0	2.7942	-0.8944
0.0	1.5	7.9966	4.3080
0.5	1.0	2.9344	-0.7542
0.0	0.0	0.9810	-2.7076

Table 2.7 Potential moves from $(0.0, 1.0, 3.6886)$

x	y	z	Δz
-0.5	1.5	6.1956	-1.8010
0.0	2.0	5.8591	-2.1375
0.5	1.5	6.2513	-1.7453
0.0	0.0	0.9810	-7.0156

Table 2.8 Potential moves from $(0, 1.5, 7.9966)$

In the preceding trivial example, the difference in the results obtained by Stochastic Descent and Steepest Descent are negligible. The true differences become apparent as both the number of dimensions increase and the input spacing decreases. Even considering the simplicity of the example, tracing the Stochastic Descent algorithm through all of its possible courses reveals that it can visit a total of 6 locations as compared to Steepest Descent's 4 locations.

Stochastic optimizing algorithms benefit from enhanced exploration abilities that come at the expense of runtime. Because a stochastic algorithm employs random chance to guide some aspect of its execution, it will encounter dead-ends, false trails, and other phenomena that delay the termination of the algorithm. Some algorithms, such as Monte Carlo, experience further delaying of termination because they do not always move from one location to another during each iteration.

Movement	x	y	z
1	-1.0	1.5	2.6076
2	-0.5	1.5	6.1956
3	0.0	1.5	7.9966

Table 2.9 An alternate route to the global maximum

2.3.1 Monte Carlo

The Monte Carlo method is a classic stochastic algorithm that is often used for relaxation techniques in system simulation [Robert and Casella, 2004]. At the heart of the Monte Carlo method is the concept of the random walk, which involves moving randomly from one location to another. The random movement is constrained to be within a certain radius of movement around the current location. The following algorithm describes the workings of the Monte Carlo method.

WHILE *the optimal value has not been found*

1. *Select a new location that is inside the radius of movement of the current location*
2. *If the new location has an output value closer to the optimal value than the current location, then the new location becomes the current location*
3. *Otherwise, the new location is discarded in favor of the current location*

END WHILE

The radius of movement and its relation to the current point is shown in Figure 2.8. When a random point is drawn during the execution of the Monte Carlo method, the radius of movement defines the area in which the new point must reside. In this

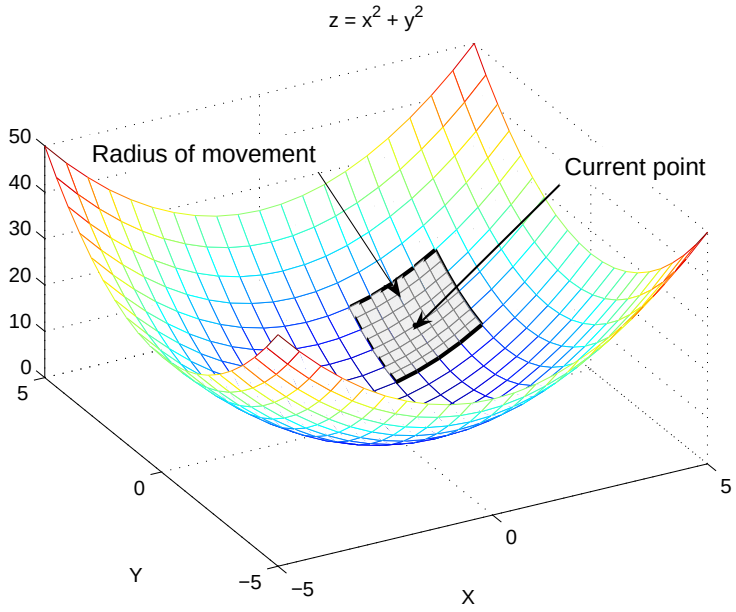


Figure 2.8 The Monte Carlo radius of movement

example, the radius is actually a square, but any closed shape will suffice. The current point need not even be in the center of the radius of movement either, as it is in the figure.

To optimize a function such as PEAKS, the Monte Carlo method requires an initial location and a radius of movement. As an example, consider the first few iterations with initial location $(-1.0, 1.0, 0.2289)$ and a square radius of movement defined by

$$\begin{aligned} |x_n - x_c| &\leq 1.0 \\ |y_n - y_c| &\leq 1.0 \end{aligned}$$

where (x_c, y_c) defines the current point's input values and (x_n, y_n) defines the new point's input values. Furthermore, assume that PEAKS is a continuous function.

Drawing a point randomly from within the radius of movement might result in $(-1.8041, 0.7395, -1.2384)$. This point is clearly inferior to the original point, so no movement occurs. Table 2.10 shows this first iteration and continues until the fifth iteration.

Iteration	Current Point			New point		
	x	y	z	x	y	z
1	-1.0000	1.0000	0.2289	-1.8041	0.7395	-1.2384
2	-1.0000	1.0000	0.2289	-0.7253	1.6291	4.6568
3	-0.7253	1.6291	4.6568	-0.2345	1.1578	5.2304
4	-0.2345	1.1578	5.2304	0.4125	0.9875	2.9479
5	-0.2345	1.1578	5.2304	0.4125	1.8125	6.1716
⋮						⋮

Table 2.10 A typical Monte Carlo optimization of PEAKS

As can be seen from Table 2.10, the efficiency of Monte Carlo can be quite low. Of the five iterations presented, only three discovered points that brought the algorithm closer to the global maximum. This problem can be further compounded by proximity to a global maximum. As the distance between the current point and the global maximum decreases, so does the probability that a randomly chosen point will improve on the current point. In fact, at the global maximum the probability of improvement is 0.

This is not so much a liability as an interesting property of the Monte Carlo method, and can be leveraged to produce a more efficiency-stable algorithm. By incorporating a strategy called the One-fifth rule, the Monte Carlo method can maintain more reasonable levels of runtime efficiency by reducing or expanding the radius of movement according to the following scheme:

The One-fifth rule

1. *Calculate the algorithm's efficiency I_e as the ratio of iterations resulting in a movement I_m to the total number of iterations I_t*
2. *If $I_e < 1/5$ then reduce the radius of movement by some factor (2 is typical)*
3. *ElseIf $I_e > 1/5$ then increase the radius of movement by some factor (2 is typical)*
4. *Else the radius of movement is appropriately scaled and does not need adjusted*

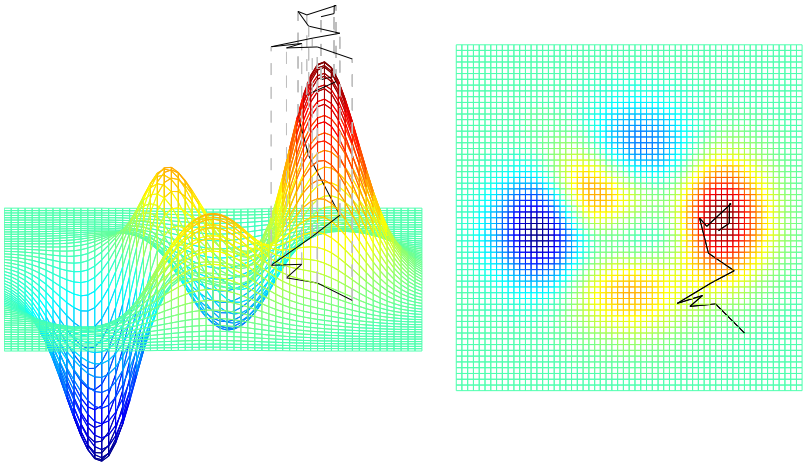


Figure 2.9 Monte Carlo finding the PEAKS global maximum

The proper use of the One-fifth rule requires the Monte Carlo algorithm to pause periodically to evaluate the current value of the radius movement. Of course, longer periods between pauses will make the One-fifth rule less effective in stabilizing the efficiency of the algorithm. In any event, the Monte Carlo method employing the One-fifth rule is limited to around one movement for every five iterations, which makes it significantly less attractive for optimization than the simplex method.

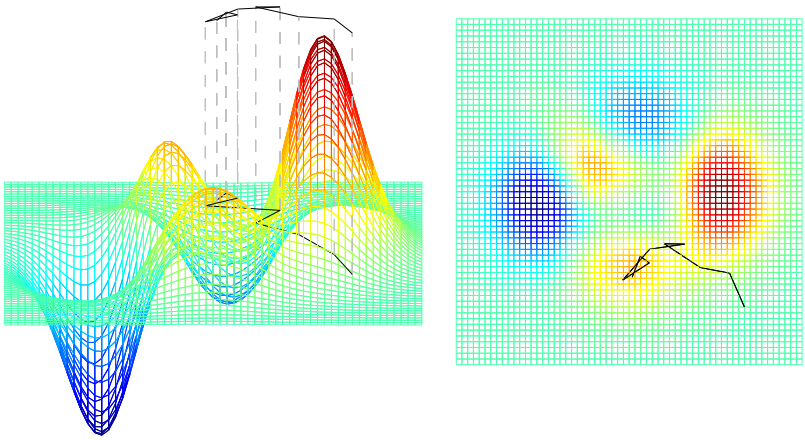


Figure 2.10 Monte Carlo finding a PEAKS local maximum

With the One-fifth rule, a simple terminating condition would be when the radius of movement shrinks below a threshold value. This works because as the algorithm approaches the global maximum, fewer iterations will result in a movement. As the efficiency drops, the One-fifth rule triggers a reduction in the radius of movement, thereby focusing the algorithm on a smaller area. The implication is that a smaller area from which to draw a new random point is more likely to contain a point that improves on the current point. If, on the other hand, the efficiency remains low, further reductions in the radius of movement will result either in algorithmic termination or enough of a focusing to raise the efficiency.

Another shortcoming of Monte Carlo is that, like the Simplex method, it will terminate at a local maximum. Monte Carlo also exhibits the classic stochastic behavior in which equivalent initial conditions will sometimes produce different results. As can be seen in Figure 2.9, the Monte Carlo method is able to find the global maximum. However, restarting the method with the same initial conditions also produces the results shown in Figure 2.10. Fortunately, there are extensions to Monte Carlo that

address all of the inadequacies previously discussed.

2.3.2 Simulated Annealing

Simulated annealing mimics the metallurgical technique of annealing [Kirkpatrick et al., 1983]. Annealing is the process of heating a material and then slowly cooling it to reduce the number and size of defects in the material. The introduction of heat causes the atoms in the material to gain thermal energy. This increase in energy allows the atoms more freedom of movement throughout the material's higher internal energy states. The slow cooling process affords the atoms more opportunity to wander to and settle in a lower energy state than they were in before the heating and cooling began. Thus, annealing has the effect of reducing the overall internal energy of the material as each atom seeks its own local minimum of internal energy.

In the case of simulated annealing, the material is the function to be optimized and the atoms are points in the function's input space. Simulated annealing is an extension of the Monte Carlo method that adds the ability to move to less optimal points. An example of the simulated annealing algorithm is shown below:

WHILE *the optimal value has not been found*

1. *Select a new location* x_n *that is inside the radius of movement of the current location* x_c
2. *If* x_n *has a value closer to the optimal value than* x_c*, then* $x_c = x_n$
3. *Otherwise, draw a uniform random number* p_r *and compare that number to the value* p_b *produced by the Boltzmann distribution for the current system temperature and change in cost function*
   (a) *If* $p_r \leq p_b$ *then* $x_c = x_n$
   (b) *Otherwise,* x_c *is kept and* x_n *is discarded*

END WHILE

There are a number of new terms contained in the above algorithm that require explanation. The first new term is system temperature. Considering that metallurgical annealing first heats and then slowly cools the material, simulated annealing must also account for the concept of temperature. Simply put, the system temperature in simulated annealing is a value that starts high and then is slowly reduced. Obviously, there is no actual temperature or heating involved in simulated annealing; it is only a mechanism whereby the real-world process of annealing can be modeled for optimization purposes.

The cost function (CF) for the purpose of optimization is the value of the output dimension being optimized. In general terms, the cost function is a mechanism that allows each point in the function's input space to be compared in some way to other points. For optimizing SIMULINK models, only the proximity to the global maximum is important, so the cost function is simply the output value corresponding to a specific set of input values. The change in the cost function (ΔCF) is calculated as the difference between the current point's output value and the newly calculated

point's output value.

The Boltzmann distribution is the formula linking the system temperature and the change in cost function, and produces a probability distribution. Equation (2.4) shows the Boltzmann distribution formula, with O_n and O_c representing the output values of the new point and current point respectively, and T representing system temperature.

$$p = e^{-(|O_n - O_c|/T)} \tag{2.4}$$

Figure 2.11 shows the shape of the Boltzmann distribution with respect to cost function change while Figure 2.12 shows the Boltzmann distribution with respect to system temperature. As can be seen in Figure 2.11, higher temperatures equate to a higher probability of acceptance with small changes in the cost function being accepted most often. Even moderate changes in the cost function are accepted more than half the time at high system temperatures. However, as the system temperature cools, the probability of acceptance for all but the smallest changes in cost function drop precipitously. At the lowest system temperatures, the probability of acceptance is so low that few non-optimal moves are accepted, resulting in behavior reminiscent of Monte Carlo.

Another term associated with simulated annealing is the cooling schedule, which governs the initial and final system temperatures as well as the rate at which the system is cooled. Determining appropriate values for initial and final system temperatures is critical to avoid excessive amounts of early stage random search and late stage loitering. One way to do so is to assume that early stages of the algorithm should accept non-optimal values at a certain average rate [Kirkpatrick, 1984]. After taking a number of samples from PEAKS as shown in Table 2.11 and assuming 80 percent of non-optimal values are accepted, then the initial system temperature can

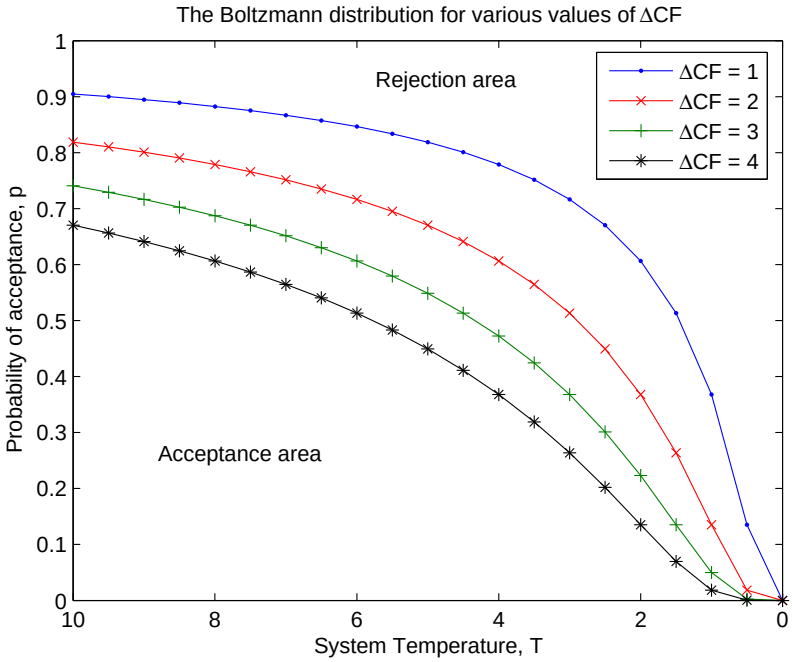


Figure 2.11 The Boltzmann distribution for various values of ΔCF

be estimated as

$$T_0 = -\frac{\overline{|\Delta CF|}}{\ln p} = -\frac{2.7268}{\ln(0.80)} = 12.2199 \tag{2.5}$$

A similar mechanism can be used to determine final system temperature. Again resorting to random sampling, this time with the assumption that 10 percent of non-optimal values are accepted, the final system temperature is estimated as

$$T_f = -\frac{\overline{|\Delta CF|}}{\ln p} = -\frac{2.7268}{\ln(0.10)} = 1.1842 \tag{2.6}$$

Now that the initial and final system temperatures are known, it remains only to determine the cooling schedule itself. Of utmost importance is allowing the system to cool slowly enough to allow it to attain a lower energy state than it started with. A

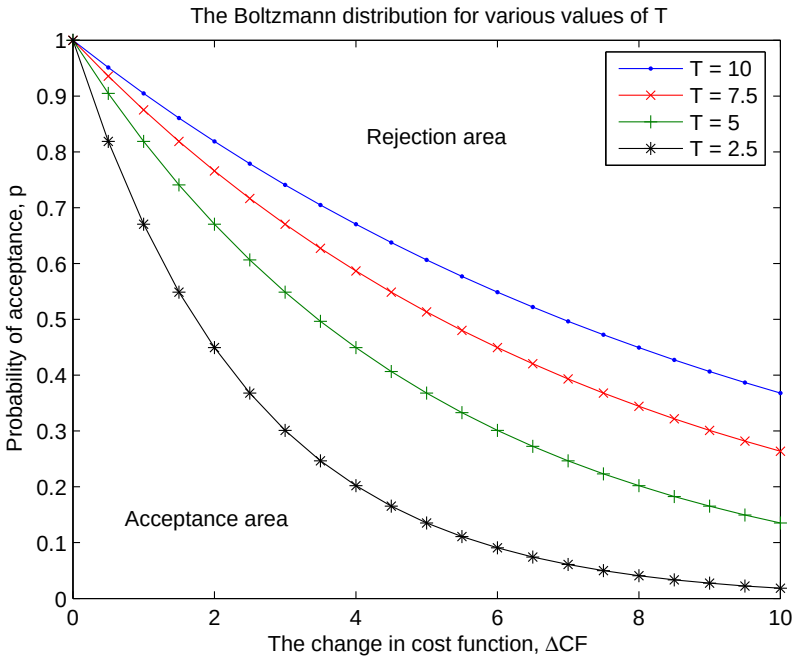


Figure 2.12 The Boltzmann distribution for various values of T

simple but effective cooling schedule is one involving a geometric progression, which of course can be described for initial system temperature T_0 and cooling rate α_c as

$$T_0, \quad T_0\alpha_c, \quad T_0{\alpha_c}^2, \quad T_0{\alpha_c}^3, \quad \ldots, \quad T_0{\alpha_c}^n$$

which can be more concisely described by Equation (3.3).

$$T_n = T_0\alpha_c^n \tag{2.7}$$

The cooling schedule can be implemented by adjusting the system temperature either after each iteration or after a set number of iterations. Both strategies are shown in Figure 2.13, and the major difference is that the latter is useful if the algorithm is already pausing for other operations such as radius of movement adjustment. Also,

Sample	Point Values			ΔCF	$\overline{\Delta CF}$
	x	y	z		
1	0.8191	-0.2945	0.0000	–	–
	-2.8080	-0.7454	-0.0390	-0.0390	-0.0390
2	-2.8080	-0.7454	-0.0390	–	–
	-1.7669	0.0308	-2.1083	-2.0692	-1.0541
3	-1.7669	0.0308	-2.1083	–	–
	0.0442	-1.4535	-5.6265	-3.5182	-1.8755
4	0.0442	-1.4535	-5.6265	–	–
	0.0974	-1.7941	-6.0885	-0.4620	-1.5221
5	0.0974	-1.7941	-6.0885	–	–
	0.1466	-1.8086	-6.0921	-0.0037	-1.2184
⋮					⋮
100	0.2636	-1.6367	-6.5360	–	–
	0.2503	-1.6039	-6.5415	-0.0055	-2.7268

Table 2.11 Sampling PEAKS to determine initial system temperature

the cooling rate values for pausing algorithms can be lower than those needed for non-pausing algorithms.

Because simulated annealing allows non-optimal movements to be made, it is capable of escaping local maxima and finding the global maximum value. To do so, the system needs to be cooled slowly so that the algorithm can accrue enough non-optimal moves to escape from the vicinity of a local maximum. An example of the ability of simulated annealing to escape a local maximum is shown in Figure 2.14. In Figure 2.14, the algorithm has been initialized very close to a local maximum and with a low system temperature. Even considering the low temperature, the algorithm quickly descends from the local maximum and begins ascending toward the global maximum.

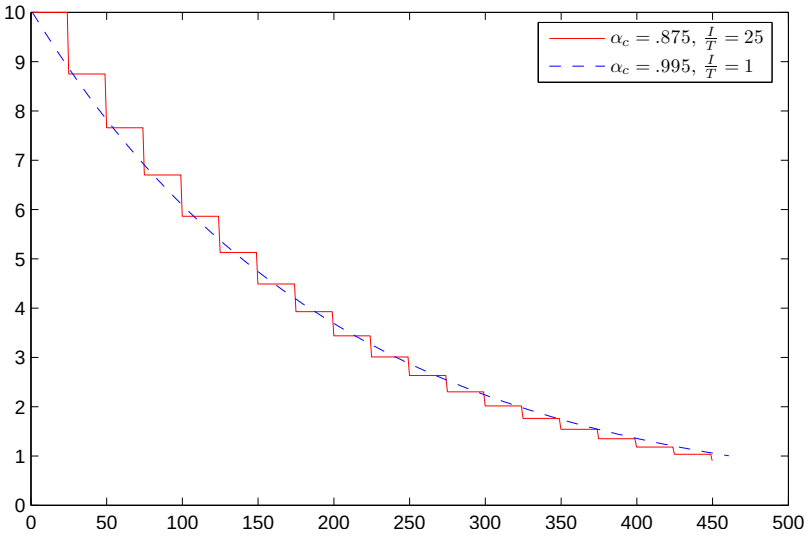


Figure 2.13 Simulated annealing cooling schedules

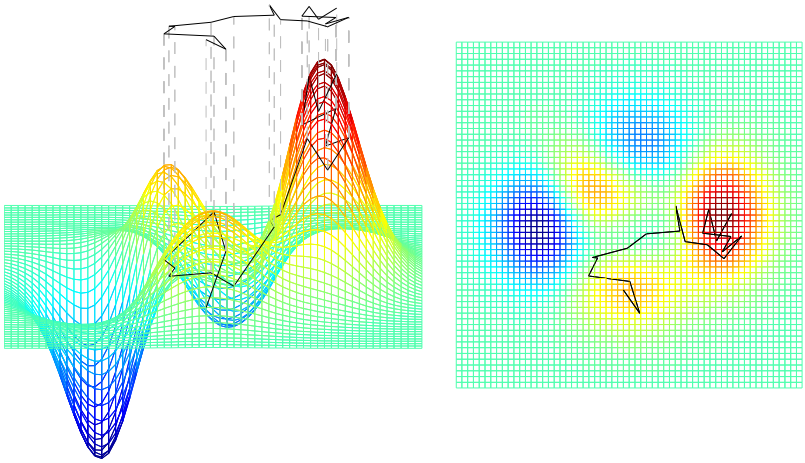


Figure 2.14 Simulated annealing escaping a PEAKS local maximum

2.3.3 Evolution Strategy

Evolution strategy simulates the biological process of evolution to search for global maxima [Beyer and Schwefel, 2002]. Rather than focusing on one random walker like Monte Carlo and simulated annealing, evolution strategy creates and maintains a population of individuals. In its simplest implementation, a population of two individuals is created that produces results similar to Monte Carlo. This is no coincidence, as evolution strategy is an extension of Monte Carlo that includes multiple walkers and mechanisms that facilitate their exploration of the input space. Indeed, the standard implementations of evolution strategy include concepts such as self-adaptation and recombination to produce better results [Eiben and Smith, 2003].

In evolution strategy it is common to use (μ, λ) or $(\mu + \lambda)$ to describe the process of survivor selection. Note that in both cases, μ refers to the number of individuals in the population while λ refers to the number of children generated during each generation. During the survivor selection phase at the end of each generation, the μ individuals are all replaced by the μ fittest individuals in the survivor pool. In the generational case of (μ, λ), the survivor pool contains only the λ children, whereas in the steady-state model of $(\mu + \lambda)$, the survivor pool contains the full population of $\mu + \lambda$ individuals.

For example, Monte Carlo can be described as a $(1 + 1)$ evolution strategy implementation. The current point would be the parent, and the new point would be the child. The comparison of the two points to determine which is more optimal is the same as placing both the parent and the child into the survivor pool to determine the more fit individual.

The difference between these two strategies lies mainly in the fact that (μ, λ) selects survivors from the children only while the latter strategy selects survivors from the

entire population of parents and children. The generational model is better able to leave local minima and to respond to a dynamic cost function than is the steady-state model. Additionally, the generational model supports self-adaptation, but the steady-state model hinders it by allowing misadapted individuals the possibility of survival from one generation to the next.

An example of a (μ, λ) evolution strategy algorithm is shown below:

WHILE *the population has not converged*

1. *Mutate each genotype in the current population of μ individuals*
2. *Select parents from the current population to participate in recombination*
3. *Create λ new individuals through recombination of the selected parents*
4. *Rank the λ children according to their fitness*
5. *Select the μ fittest children to replace the current population*

END WHILE

In biology a distinction is drawn between a genotype and a phenotype, and in evolution strategy the mechanism mapping one to the other is called representation. A genotype is defined as the blueprint for the creation of the phenotype through representation. Conversely, the phenotype is defined as the representation of the genotype. Evolution strategy aims to optimize the phenotype by selecting the most fit genotypes.

Referring again to the PEAKS function, a genotype would include one object gene for each of PEAKS 's functional input dimensions. In other words, a PEAKS -specific genotype would require two object genes because PEAKS requires two input values to calculate an output value. This would produce an object genotype $\overline{x}$ having the

form $\langle x_x, x_y \rangle$. The object genes all contribute to the representation of the phenotype. Several genotypes are shown below:

$$\overline{x}_1 = \langle -1.000, 1.000 \rangle$$

$$\overline{x}_2 = \langle -0.908, 2.983 \rangle$$

The genotype also contains strategy information in the form of real-valued genes. This information is used during the course of the algorithm to mutate population genotypes into new genotypes. In the case of uncorrelated mutation with n step sizes, one strategy gene for each of the object genes is included in the genotype. For PEAKS, the strategy genotype $\overline{\sigma}$ would have the form $\langle \sigma_x, \sigma_y \rangle$. Each strategy gene is responsible for the mutation of its respective object gene. A complete genotype would be represented as a vector with the complete object genotype followed by the complete strategy genotype, as shown in the general case below

$$\langle x_1, \ x_2, \ \ldots, \ x_n, \ \sigma_1, \ \sigma_2, \ \ldots, \ \sigma_n \rangle$$

and also for the specific case of PEAKS:

$$\langle x_x, \ x_y, \ \sigma_x, \ \sigma_y \rangle$$

The object genes of the initial population of μ individuals are randomly drawn from a uniform distribution about the input space. Each of the strategy genes is then calculated as a percentage of the size of the input space for that gene. It is important to initialize the strategy genes to large enough values so that mutation will be effective. Because the strategy genes will self-adapt, erring on the side of caution by choosing larger values for the strategy genes will only delay convergence. Table 2.12

contains a representative population for $\mu = 5$ with the initializing percentage for the strategy genes equal to 1/2 of their input dimension's range. Note also that the convention will be to label the n^th parent as $\overline{p_n}$ and the n^th child as $\overline{c_n}$. Because the children have not yet been generated, only parent individuals appear in Table 2.12.

Individual	Object genes		Strategy genes	
	x_x	x_y	σ_x	σ_y
$\overline{p_1}$	0.6926	-0.5658	3.000	3.000
$\overline{p_2}$	1.7516	2.6128	3.000	3.000
$\overline{p_3}$	2.5309	2.5014	3.000	3.000
$\overline{p_4}$	1.4292	-0.5384	3.000	3.000
$\overline{p_5}$	-1.9424	2.3619	3.000	3.000

Table 2.12 Evolution strategy initial population for PEAKS

Mutation is the first major phase in which the population participates during an evolution strategy optimization. Mutation operates on both the object genotype and the strategy genotype of each individual in the population. Uncorrelated mutation with n step sizes allows each input dimension to evolve independently of the others. The first step in the mutation phase is to mutate the strategy genotype. Equation (2.8) gives the formula for this transformation, where $\tau' \propto 1/\sqrt{2n}$ and $\tau \propto 1/\sqrt{2\sqrt{n}}$.

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)} \tag{2.8}$$

Note the two components of the exponent term. The first, $\tau' \cdot N(0,1)$, represents a common base mutation that allows for a general mutation of the genotype and involves a random number $N(0,1)$ drawn from the standard normal distribution. The second term, $\tau \cdot N_i(0,1)$, represents a gene-specific mutation that allows each input dimension to mutate and evolve independently. In the latter case, $N_i(0,1)$ is a random number drawn from the standard normal distribution for each gene.

The next step in the mutation phase is to mutate the object genotype using the newly mutated strategy genotype. Again, $N_i(0,1)$ represents a random number drawn for each gene in the object genotype, and the formula is shown in Equation (2.9). To prevent the strategy genes from approaching too close to zero, a limiting function is used, as shown in Equation (2.10).

$$x_i' = x_i + \sigma_i' \cdot N_i(0,1) \tag{2.9}$$

$$\sigma_i' < \varepsilon_0 \Rightarrow \sigma_i' = \varepsilon_0 \tag{2.10}$$

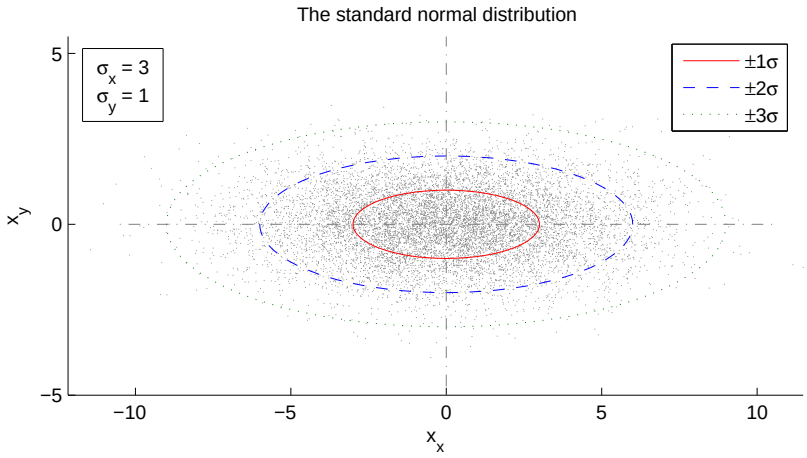


Figure 2.15 The standard normal distribution

By drawing numbers from the standard normal distribution, the mutations are statistically expected to be near the original values. However, the randomness of the drawing allows large mutations to occur with lower probability the farther from the original values they occur. Figure 2.15 demonstrates this property for 10,000 normally distributed random points and also shows the effect of having an independent strategy gene for each object gene. The result of this effect is that as the genotypes evolve and

self-adapt, the strategy genes will guide the direction and magnitude of mutation. The results of the mutation phase are detailed in Table 2.13. Again, only parents appear because the children have not yet been generated.

Individual	Object genes		Strategy genes	
	x_x	x_y	σ_x	σ_y
$\overline{p_1}$	-0.3294	-0.4881	3.6227	2.4940
$\overline{p_2}$	1.0034	2.0773	1.3791	5.0388
$\overline{p_3}$	2.5909	0.1509	2.4934	3.0830
$\overline{p_4}$	1.4807	-1.7841	6.4834	2.1462
$\overline{p_5}$	-2.9084	1.0328	1.3504	2.4796

Table 2.13 Evolution strategy population after mutation

After the mutation phase, the recombination phase begins. Evolution strategy employs a uniform random parent selection method. A uniform random selection provides each member of the μ individuals in the existing population an equal chance to pass on their genes to one or more of the λ children, regardless of fitness. In the most basic version of recombination, two parents are randomly selected from the population and recombined to produce one new child. Thus, the production of λ new children would require λ pairs of parents.

The exchange of genetic information through recombination can be performed using a discrete or an intermediary mechanism. In discrete recombination, the child's genotype is composed of genes that are randomly chosen from the parents' genotypes, while intermediary recombination passes the average of the parents' genes to the child. Thus, for parents $\overline{p_1}$ and $\overline{p_2}$, a child $\overline{c}$ would be created using

$$c_i = \begin{cases} \frac{(p_{1,i}+p_{2,i})}{2} & \text{intermediary recombination} \\ p_{1,i} \text{ or } p_{2,i} \text{ chosen randomly} & \text{discrete recombination} \end{cases} \tag{2.11}$$

for each gene $i \in \{1, \ldots, n\}$. Furthermore, recombination can act globally or locally. In global recombination, a new set of parents is selected for each gene in the new child, while local recombination uses one pair of parents per child.

Evolution strategy usually uses global recombination to produce the μ children. The actual process of global recombination is broken down into two subprocesses: global discrete recombination of the object genes and global intermediary recombination of the strategy genes. Note that during the recombination process, the children are kept separate from the parents, and so only parents are involved in the recombination process. Producing new children using local recombination and the mutated population from Table 2.13 might result in the values shown in Table 2.14. Performing global recombination on the same parent population might result in the values shown in Table 2.15. In both tables the emphasized gene values are the ones participating in recombination.

Note that in Table 2.14 that two parents recombine to produce a child. For example, the parents $\overline{p_5}$ and $\overline{p_2}$ recombine to produce the child $\overline{p_5}$. The parent $\overline{p_2}$ contributes its x_x object gene, the parent $\overline{p_5}$ contributes its x_y object gene, and both contribute to the strategy genes of the child. It is also possible for a parent to contribute both its object genes to a child during recombination, as is shown for the children $\overline{c_2}$, $\overline{c_3}$, and $\overline{c_4}$.

The next step in the process is to select the survivors and discard the rest of the population. This is the point at which the children genotypes are mapped to phenotypes for ranking purposes. Recall that evolution strategy employs either generational (μ, λ) or steady-state $(\mu + \lambda)$ survivor selection. Using the generational method, the fittest μ of the λ children are selected to survive. Note that this selection is deterministic and is based only on the relative fitnesses of the children. Table 2.16 shows the generational ranking and survivor selection of a population of children where (μ, λ)

Individual	Object genes		Strategy genes	
	x_x	x_y	σ_x	σ_y
$\overline{p_5}$	-2.9084	*1.0328*	*1.3504*	*2.4796*
$\overline{p_2}$	*1.0034*	2.0773	*1.3791*	*5.0388*
$\overline{c_1}$	1.0034	1.0328	1.3647	3.7592
$\overline{p_5}$	*-2.9084*	*1.0328*	*1.3504*	*2.4796*
$\overline{p_4}$	1.4807	-1.7841	*6.4834*	*2.1462*
$\overline{c_2}$	-2.9084	1.0328	3.9169	2.3129
$\overline{p_5}$	-2.9084	1.0328	*1.3504*	*2.4796*
$\overline{p_3}$	*2.5909*	*0.1509*	*2.4934*	*3.0830*
$\overline{c_3}$	2.5909	0.1509	1.9219	2.7813
$\overline{p_5}$	*-2.9084*	*1.0328*	*1.3504*	*2.4796*
$\overline{p_4}$	1.4807	-1.7841	*6.4834*	*2.1462*
$\overline{c_4}$	-2.9084	1.0328	3.9169	2.3129
$\overline{p_5}$	*-2.9084*	1.0328	*1.3504*	*2.4796*
$\overline{p_5}$	-2.9084	*1.0328*	*1.3504*	*2.4796*
$\overline{c_5}$	-2.9084	1.0328	1.3504	2.4796

Table 2.14 Evolution strategy population after local recombination

$= (5, 10)$. The less often used steady-state method ranks the union of the children and the parents and selects the fittest μ individuals to survive. Table 2.17 shows a steady-state ranking and survivor selection for a population of parents p and children c where $(\mu + \lambda) = (5 + 5)$. For both tables, the ranking has already been completed and only the individuals in the survivor pool are shown.

For PEAKS, the genotype to phenotype mapping mechanism is

$$z = \text{PEAKS}\ (x_x, x_y)$$

so that the child $\overline{c_3} = \langle 1.5678, 2.1278, 0.1254, 6.3985 \rangle$ in Table 2.16 would have a phenotype z, of

$$z = \text{PEAKS}\ (1.3598, -0.5283) = 2.6687$$

Individual	Object genes		Strategy genes	
	x_x	x_y	σ_x	σ_y
$\overline{p_1}$	*-0.3294*	-0.4881	3.6227	2.4940
$\overline{p_2}$	1.0034	2.0773	*1.3791*	5.0388
$\overline{p_3}$	2.5909	*0.1509*	2.4934	*3.0830*
$\overline{p_4}$	1.4807	-1.7841	*6.4834*	2.1462
$\overline{p_5}$	-2.9084	1.0328	1.3504	*2.4796*
$\overline{c_1}$	-0.3294	0.1509	3.9312	2.7813
$\overline{p_1}$	-0.3294	-0.4881	3.6227	2.4940
$\overline{p_2}$	1.0034	*2.0773*	1.3791	5.0388
$\overline{p_3}$	2.5909	0.1509	*2.4934*	*3.0830*
$\overline{p_4}$	1.4807	-1.7841	6.4834	*2.1462*
$\overline{p_5}$	*-2.9084*	1.0328	*1.3504*	2.4796
$\overline{c_2}$	-2.9084	2.0773	1.9219	2.6146

Table 2.15 Evolution strategy population after global recombination

The final facet of evolution strategy implementation to be discussed is selection pressure. Selection pressure refers to the competition among individuals to be among those chosen to survive. It is analogous to biological selection pressure, in which a population of individuals must compete for limited resources. In evolution strategy, selection pressure is high, with

$$\frac{\mu}{\lambda} = \frac{1}{7}$$

High selection pressures significantly reduce the number of generations required for a single superior individual to take over a population. Known as the takeover time, it is calculated using the formula in Equation (2.12). For a typical population size of $(15, 100)$, the takeover time would be $\tau^* \approx 2$. In other words, only two generations would be necessary for one copy of a superior individual to take over the population. The general shape of the selection pressure curve is shown for various values of μ in

Individual	Genotype		Phenotype	Rank
	x_x	x_y	z	
$\overline{c_3}$	1.3598	-0.5283	2.6687	1
$\overline{c_9}$	0.6432	0.7793	1.5291	2
$\overline{c_5}$	-0.3605	2.6003	1.2084	3
$\overline{c_7}$	2.0354	0.7727	0.7276	4
$\overline{c_2}$	-1.0798	2.7606	0.2431	5
Survival threshold				
$\overline{c_8}$	-2.1974	-1.7572	0.0364	6
$\overline{c_4}$	1.4674	-1.3923	-0.3302	7
$\overline{c_{10}}$	-0.7771	0.4509	-0.9450	8
$\overline{c_1}$	-1.7162	0.8610	-1.1207	9
$\overline{c_6}$	1.1000	-1.7246	-2.1497	10

Table 2.16 Evolution strategy population showing generational survivor selection

Figure 2.16

$$\tau^* = \frac{\ln \lambda}{\ln(\lambda/\mu)} \tag{2.12}$$

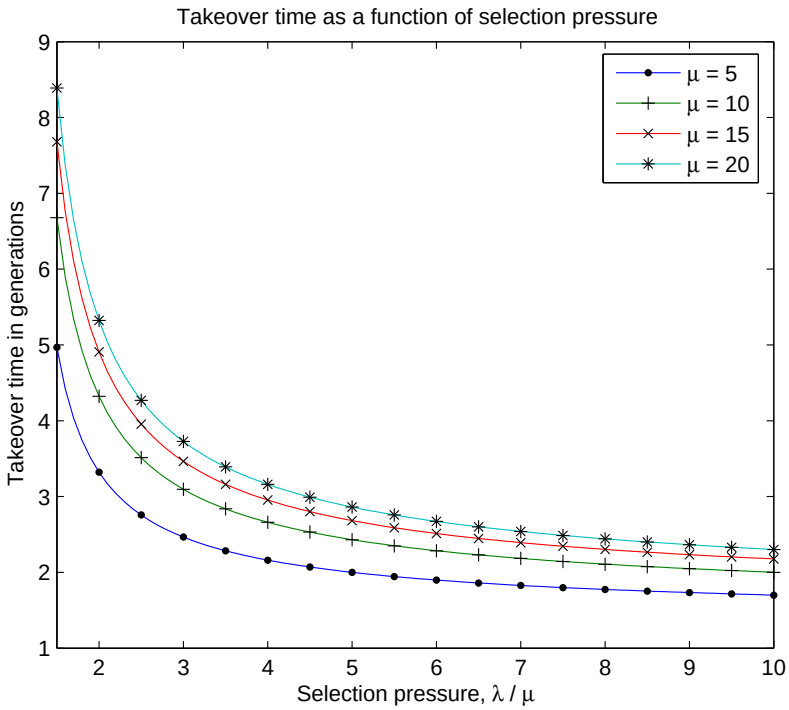


Figure 2.16 Takeover time as a function of selection pressure

Individual	Genotype		Phenotype	Rank
	x_x	x_y		
$\overline{p_1}$	-0.3294	-0.4881	3.5099	1
$\overline{c_3}$	1.3598	-0.5283	2.6687	2
$\overline{p_2}$	1.0034	2.0773	1.9283	3
$\overline{c_5}$	-0.3605	2.6003	1.2084	4
$\overline{c_2}$	-1.0798	2.7606	0.2431	5
Survival threshold				
$\overline{p_3}$	2.5909	0.1509	0.2029	6
$\overline{p_5}$	-2.9084	1.0328	-0.0195	7
$\overline{c_4}$	1.4674	-1.3923	-0.3302	8
$\overline{p_4}$	1.4807	-1.7841	-0.6583	9
$\overline{c_1}$	-0.3294	0.1580	-1.1207	10

Table 2.17 Evolution strategy population showing steady-state survivor selection

CHAPTER 3 METHODS

This chapter details the implementations that were written throughout the course of the research discussed in this paper. Because the objective was to create a proof-of-concept constraint determination tool for use with SIMULINK models, all of the implementations were written in MATLAB. Any interested readers are encouraged to email the author (chap.alex@hotmail.com) or Dr. Joel Henry (joel.henry@mso.umt.edu) for any source code written for this research.

3.1 The MATLAB function peaks

Much of the early research into useful optimization strategies relied on the PEAKS function, and while it was discussed in Section 2.1.1, it is worthwhile to reiterate and expand here. As was stated before, PEAKS is a MATLAB built-in function whose internal function is given in Equation (3.1). Its geometry is reprinted in Figure 3.1.

$$\begin{aligned} z =& 3(1-x)^2 e^{-(x^2)-(y+1)^2} - \dots \\ & 10(\frac{x}{5} - x^3 - y^5) e^{-x^2-y^2} - \dots \\ & \frac{1}{3} e^{-(x+1)^2-y^2} \end{aligned} \tag{3.1}$$

Recall that PEAKS has two input dimensions and one output dimension, producing a function whose inputs and outputs are continuous. Its dimensionality makes

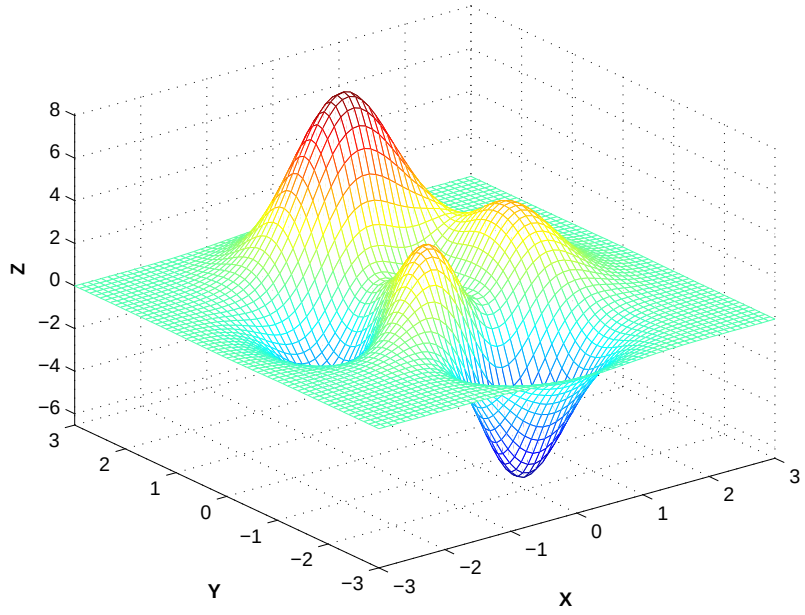


Figure 3.1 The MATLAB function PEAKS

visualization of any optimizing algorithms easy and informative. In addition, it is a function in both mathematical and computer science terms. It is a mathematical function because each set of inputs produces one and only one output value. It is a function by computer science standards because it can be called with a parameter list, and upon completion it returns a value. This feature is important also because SIMULINK models have a similar form, in that they also are called with a parameter list and return a set of values.

One of the most interesting features of PEAKS is that it has multiple local maxima and minima, making it attractive as a test bed for potential optimization strategies. Any strategy that can successfully and repeatedly optimize PEAKS is obviously better suited to SIMULINK model optimization than a strategy that gets stuck in local maxima. Although the full range of real values can be passed into PEAKS, the input

space is effectively limited by the construction of the internal function to values between ±3. The cleverly designed function produces a rather bland, featureless output geometry for any input values outside ±3. This conveniently constrains the input space and allows the development of strategies without having to account for large input spaces prematurely.

One final note about PEAKS is that rather than calling it directly from the optimizing algorithm, MATLAB allows the use of function pointers. This convenience allows the algorithms to be developed using a generic function pointer that can point to any function desired. In example, the following code shows a function call to PEAKS

```
function z = eval_peaks(point)

    z = peaks(point(:,1), point(:,2));

end
```

The consequence of directly calling a function from within the optimizing algorithm is the difficulty in maintaining the source code as the function to be optimized is changed. Delving into the code to make all the necessary changes is both time-consuming and fraught with dangers. At the heart of the problem is the need to change the algorithm's treatment of the function to be optimized. The need for code modification cannot be eliminated, but its impact can be minimized.

Using a function pointer as the function to be optimized allows the full range of code modification without the issues normally involved. By encapsulating all the function-specific code into its own function and pointing the optimizing algorithm to that function, the optimizing algorithm code can remain unchanged no matter what function is being optimized. The following code snippet shows this concept:

```
function z = eval_func(func, point)

    z = func(point);

end
```

Note that the first parameter to the function is the function pointer to the code that handles the actual function call. The second parameter is a data structure containing all the input values necessary to call the function.

This approach introduces more complexity than the previous approach because the optimizing algorithm must be written with as much generality as possible. Every variable must be an array, a struct, or other generic data structure that can store arbitrary amounts of information, and the custom function code must be able to retrieve this information reliably. However, this generality allows the optimizing algorithm's usefulness to extend well beyond its initial scope. In addition, such generality in design mimics the object-oriented model of code reuse.

Much of the work involving optimization of PEAKS was later found to be insufficient in the optimization of SIMULINK models. This early work is included here for completeness and to show the process of research and experimentation that was conducted in this area.

3.1.1 Monte Carlo

The first method explored was Monte Carlo because its simplicity made it a natural first choice for a potential optimization strategy. The initial implementation of this optimization strategy involved one walker (see Figure 3.2), but was quickly modified to include multiple walkers (see Figure 3.3). For this limited implementation, the One-Fifth rule was ignored and the radius of movement was set at an arbitrary value of 0.1. As can be clearly seen from Figure 3.3, using multiple walkers is prudent

when implementing Monte Carlo because of its tendency to find local maxima, as was mentioned in Section 2.3.1.

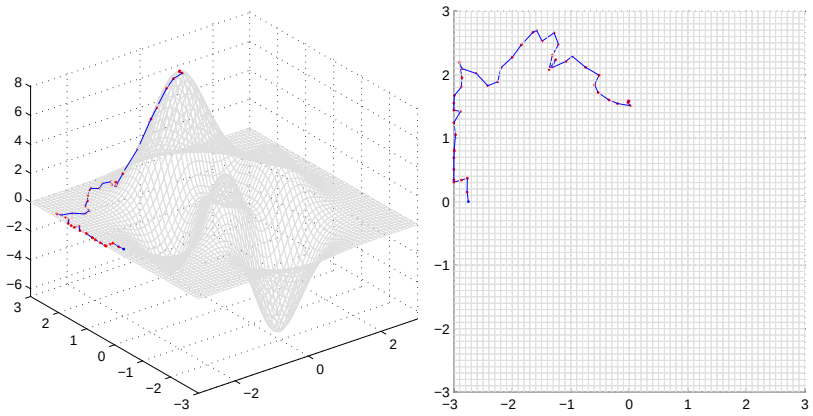


Figure 3.2 Monte Carlo with one walker

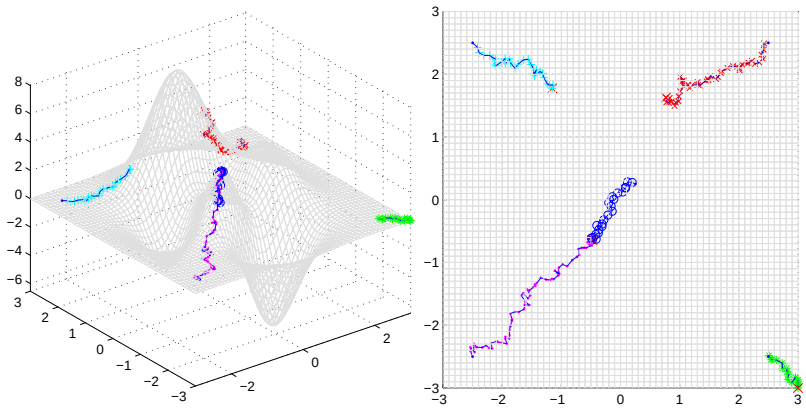


Figure 3.3 Monte Carlo with multiple walkers

Another issue that was resolved in this early work was to determine an appropriate stopping condition. The method used was a percent difference between the most recent move's output and the previous location's output. In example, for two points p_p and p_c, which are the previous and current locations respectively, then the percent change is 854.18 percent, as shown below.

$$\left.\begin{array}{l} p_p = (-1.4532, 2.7897, 0.0838) \\ p_c = (-.9875, 2.4568, 0.7996) \end{array}\right\} \quad \frac{|0.7996 - 0.0838|}{0.0838} = 8.5418$$

The stopping condition would compare this value to some preset minimal value, such as 0.01 for 1 percent or 0.001 for .1 percent. This naive approach was found later to be wanting, and a more appropriate stopping condition is presented in a later section. As a preview of the reasoning, any output geometry that is flat will cause premature stopping even if the walkers are still moving. This is because the relative difference in output value extrema may only differ by a few percent. In addition, care must also be taken to check for a stopping condition only if the walker has moved. For obvious reasons, the percent difference will be zero if the walker has not moved during an iteration.

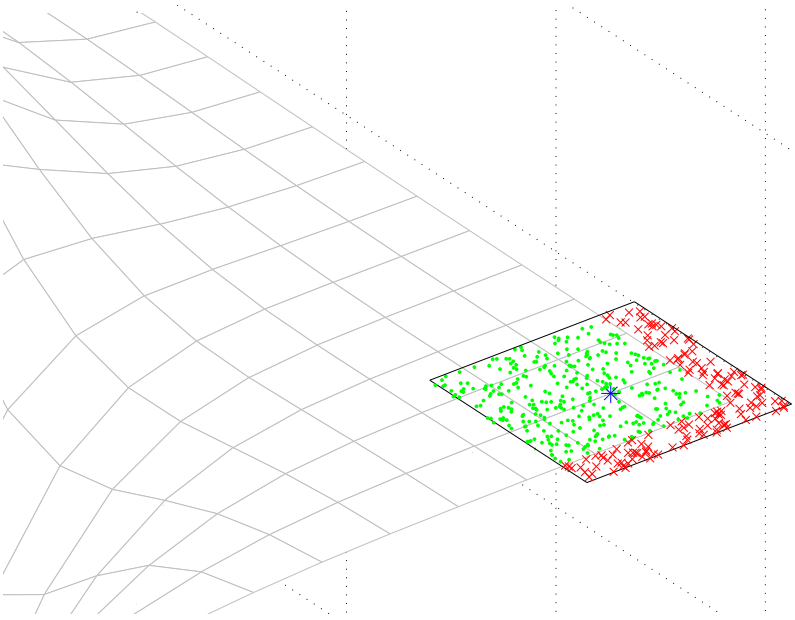


Figure 3.4 Boundary conditions for a uniform random movement

One final issue that arose in this early work was the handling of walker movement

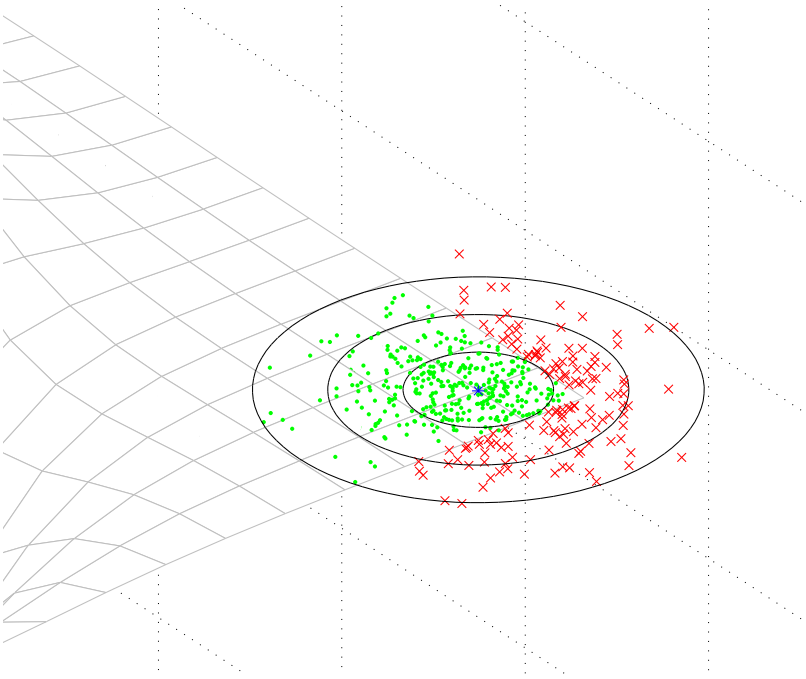


Figure 3.5 Boundary conditions for a normal random movement

near boundary conditions. The behavior of a walker near one or more boundaries is much different than a movement in open space. Figure 3.4 shows potential uniform random movements from $(2.75, -2.75, 0)$ with a radius of movement of

$$\begin{aligned} |x_n - x_c| &\leq 0.5 \\ |y_n - y_c| &\leq 0.5 \end{aligned}$$

The obvious question is what to do with the newly generated points that are clearly outside input boundaries. One potential solution is to consider any move beyond the boundary as a non-optimizing move and reject it. Thus, as a walker approaches a boundary, a larger percentage of potential moves will be rejected and less movement will occur. Another solution is to project the movement onto the boundary itself, as shown in Figure 3.6. Using the well-known similar triangles relations, the location

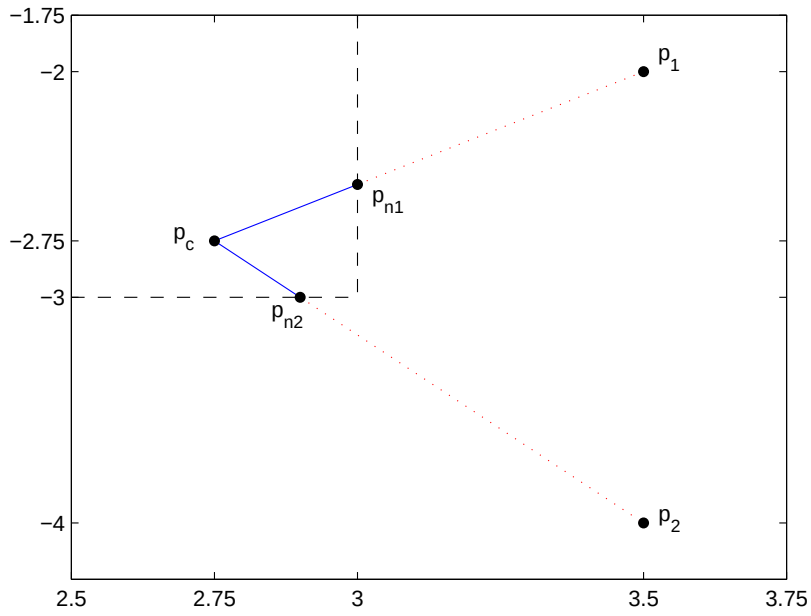


Figure 3.6 Resolving boundary condition issues using similar triangles

p_{n1} can be calculated from

$$p_{n1} = p_c + \min\left(\frac{p_B - p_c}{p_1 - p_c}\right)(p_1 - p_c) \tag{3.2}$$

where all the points are in vector form, p_B is the value at the crossed boundary, and the *min* term is calculated for each input dimension and the smallest resulting value is used. For p_1, which only crosses one boundary, $p_B = 3.00$. In this case the min function reduces to a trivial fraction. The location of p_{n1} is

$$\begin{aligned} p_{n1,x} = &\quad 2.75 + \left(\tfrac{3.00-2.75}{3.50-2.75}\right)(3.5 - 2.75) &= &\quad 3.00 \\ p_{n1,y} = &-2.75 + \left(\tfrac{3.00-2.75}{3.50-2.75}\right)(-2.00 - (-2.75)) &= &-2.50 \end{aligned}$$

resulting in $p_{n1} = (3.00, -2.50)$. The next potential location, p_2, crosses two bound-

aries. The min function is no longer trivial, and both dimensions must be checked.

$$\begin{aligned}\left(\frac{p_{B,x}-p_{c,x}}{p_{2,x}-p_{c,x}}\right) &= \left(\frac{3.00-2.75}{3.50-2.75}\right) &&= 1/3\\ \left(\frac{p_{B,y}-p_{c,y}}{p_{2,y}-p_{c,y}}\right) &= \left(\frac{-3.00-(-2.75)}{-4.00-(-2.75)}\right) &&= 1/5\end{aligned}$$

The minimum border crossing produces $1/5$ as the limiting ratio, and so the new location p_{n2} is

$$\begin{aligned}p_{n2,x} &= 2.75 + (1/5)\,(3.5 - 2.75) &&= 2.90\\ p_{n2,y} &= -2.75 + (1/5)\,(-4.00 - (-2.75)) &&= -3.00\end{aligned}$$

resulting in $p_{n2} = (2.90, -3.00)$.

Other techniques involve discarding the current point in favor of a newly chosen random point or treating the boundaries as mirrors against which new locations reflect back into the search space. However, these techniques favor interior locations over boundary locations. Considering the optimization of SIMULINK models is a black-box endeavor, favoring interior points over boundary points is not necessarily a successful strategy. By allowing the search to include the boundaries through limiting ratios, a more complete search can take place.

3.1.2 Simplex method

The next major implementation was of the simplex method, again attempting to optimize PEAKS. In this case, the stopping condition was more involved than the percent difference calculation used in the original Monte Carlo implementation. The centroid of a simplex is the mean of all the vertices in the simplex, and is similar to the center of mass for the simplex. Using the centroid as a reference point, the distances to each vertex in the simplex were summed to get an overall length for

the simplex. When the overall length dropped below a set value, the simplex was considered to have converged. The value of the convergence threshold was set to 0.01. For an example, see Table 3.1.

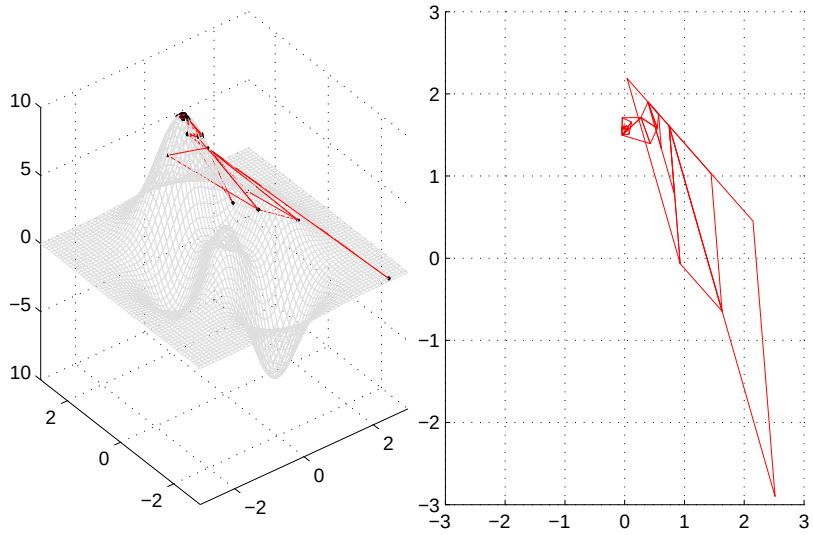


Figure 3.7 Simplex method with one simplex

Vertex	Vertex values			centroid C			$\|\|V-C\|\|$	$\sum \|\|V-C\|\|$
	V_x	V_y	V_z	C_x	C_y	C_z		
V_1	2.50	-1.75	0.0063	0.500	-1.75	-1.535	2.5250	8.0662
V_2	-1.00	-1.5	0.1483				2.2685	
V_3	0.00	-2.00	-4.7596				3.2727	

Table 3.1 Determining simplex size for convergence

Figure 3.7 shows a typical result from the first attempt at optimization of PEAKS using the simplex method. It quickly became apparent that one simplex would not be sufficient to optimize PEAKS, so the implementation was modified to include multiple simplices. Figure 3.8 shows typical results from a multiple simplex optimization. Disappointingly, even multiple simplices were not enough to optimize PEAKS.

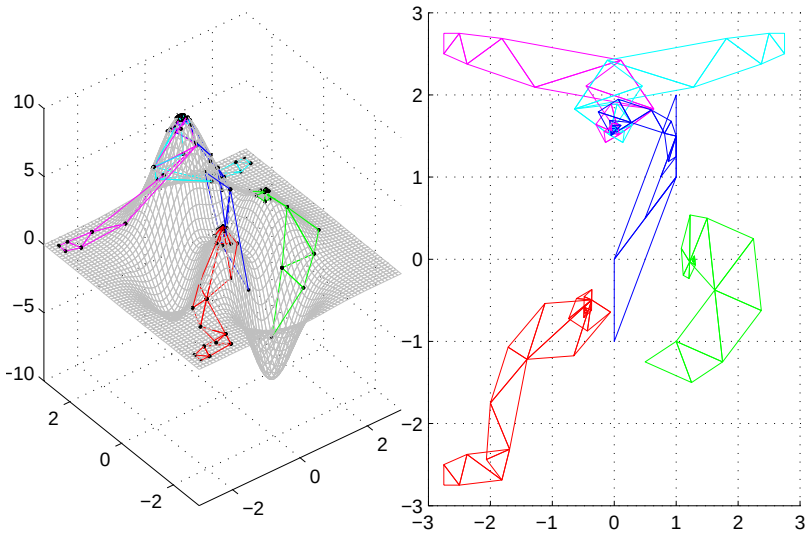


Figure 3.8 Simplex method with five simplices

3.2 The random function

The main contributor to the creation of the random function generator was the deceptively simple geometry of PEAKS. While at first glance PEAKS is a good candidate for optimization, the global maximum is rather large and so controls a significant portion of the input space. Because the global maximum has a powerful influence over the function's geometry, it leads to overly positive optimization results. Also, the geometry of PEAKS is well-known and relatively uncomplicated. As a consequence, otherwise general optimization strategies could be inadvertently modified to optimize PEAKS. In essence, the familiarity of PEAKS could drive the construction of an optimization strategy and therefore destroy its generality.

To combat these two serious issues, a random function generator was implemented. Its purpose was to randomly generate a two-input, one-output function similar to PEAKS. The similarity to PEAKS allowed visualization of optimization attempts, which provided valuable feedback regarding strengths and weaknesses of the optimiz-

ing algorithms.

At the heart of the random function generator was a matrix of random numbers. The matrix was generated as the basis for the random function's output values. The granularity of the matrix was defined as the value spacing for each input dimension. As an example, consider a random function for which the input constraints are equal to those for PEAKS and a granularity of $\Delta X = \Delta Y = 1$. The matrix of random numbers would be a 7-by-7 matrix of values, as shown below. Note that the first entry, (.5162), would be the output value corresponding to an input of $(-3.00, -3.00)$, while the second entry, (.3693), would correspond to an input of $(-2.00, -3.00)$.

$$z(x,y) = \begin{bmatrix} .5162 & .3693 & \dots & .3342 & .8259 \\ .2252 & .0295 & \dots & .4341 & .3353 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ .9706 & .4331 & \dots & .0641 & .6237 \\ .8215 & .6111 & \dots & .5217 & .2679 \end{bmatrix}$$

Of course, MATLAB 's random number generator produced pseudo-random numbers. That meant that there was some underlying order to the sequence of numbers produced by the random number generator. Because of the underlying order, it was possible to force MATLAB to reproduce a sequence of random numbers. This process is known as setting the random seed, and was used to be able to reproduce the same random function at different times and locations.

Once the random output matrix was generated, it was passed through a series of discrete Laplacians. The discrete Laplacian had the effect of aging the output geometry in much the same fashion as wind and rain age the landscape. Finally, the output matrix was scaled to produce an arbitrary output geometry. Thus, a random

$Range_x$	$-3 \leq x \leq 3$
$Range_y$	$-3 \leq y \leq 3$
$Range_z$	$-10 \leq z \leq 10$
Granularity	3
Age	0

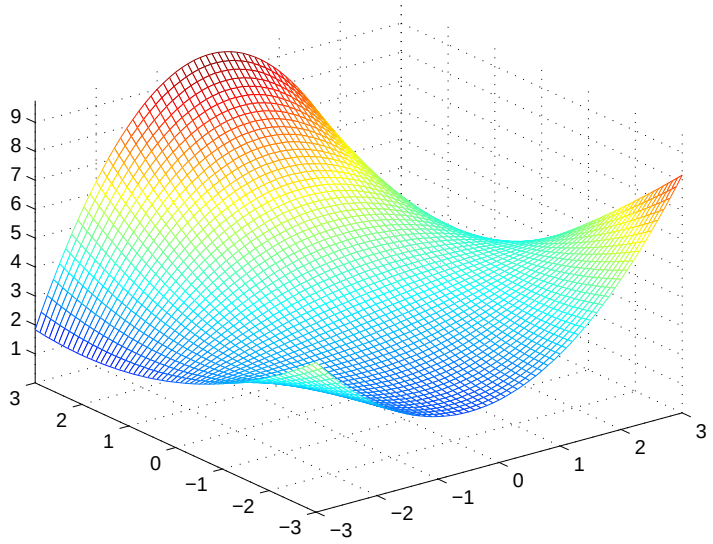


Figure 3.9 Random function generation example I

function could be generated with arbitrary input and output geometries. Two such random functions are shown in Figure 3.9 and Figure 3.10.

3.2.1 Simplex method

The simplex method implementation received an overhaul in an attempt to improve on its optimizing ability. The largest impediment to its use as an optimizing algorithm is its inability to escape local maxima. With this in mind, momentum was incorporated into the simplex method. The idea of simplex momentum draws from the real-world concepts of inertia and momentum and is displayed graphically in Figure 3.11.

Starting from the original location P_0, a movement to P_1 creates a momentum vector in that direction. A subsequent movement to P_2 is further displaced by the momentum vector generated in the original move from P_0 to P_1. The amount of displacement due to momentum depends only on the percentage of the previous move-

$Range_x$	$-10 \leq x \leq 10$
$Range_y$	$-10 \leq y \leq 10$
$Range_z$	$-20 \leq z \leq 20$
Granularity	2.5
Age	1

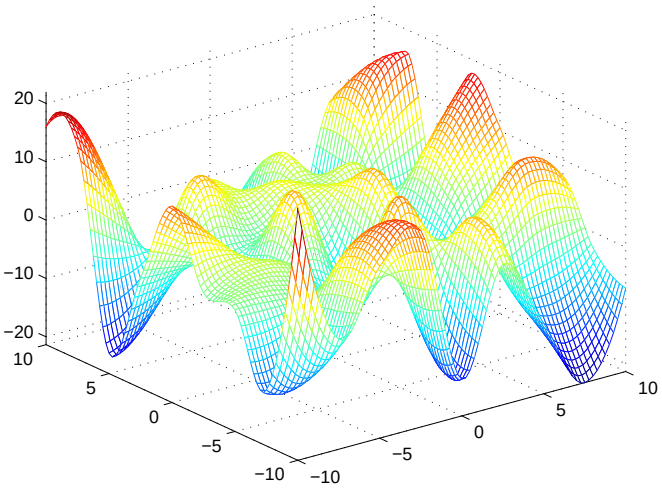


Figure 3.10 Random function generation example II

ment that contributes to momentum, and is typically a percentage between 0 percent and 100 percent. A momentum contribution of 50 percent would result in a movement from P_1 to P_{2a}, while a contribution of 100 percent would result in a movement from P_1 to P_{2b}.

For a simplex, momentum is based on the centroid rather than any single vertex. The difference between two successive simplices' centroids determines the magnitude and direction of momentum. A sequence of simplex movements in the presence of 50 percent momentum contribution is shown in Figure 3.12. The initial simplex $S_1 = \{P_2, P_3, P_1\}$ has a momentum vector of $\overline{V_1} = \langle 0, 0 \rangle$. It moves to simplex $S_2 = \{P_4, P_2, P_3\}$, which produces a momentum vector of $\overline{V_2} = \langle 0.5, 0.5 \rangle$. The next movement would be to $S_3 = \{P_{5a}, P_4, P_3\}$, but the momentum vector contributes an additional displacement from P_{5a} to P_{5b}. The new simplex $S_3 = \{P_{5b}, P_4, P_3\}$ has a momentum vector $\overline{V_3} = \langle -0.0833, 0.9167 \rangle$. The final movement shown is to $S_4 = \{P_{6b}, P_{5b}, P_3\}$ due to the contribution of V_3.

With the additional displacements due to momentum, walkers can escape local

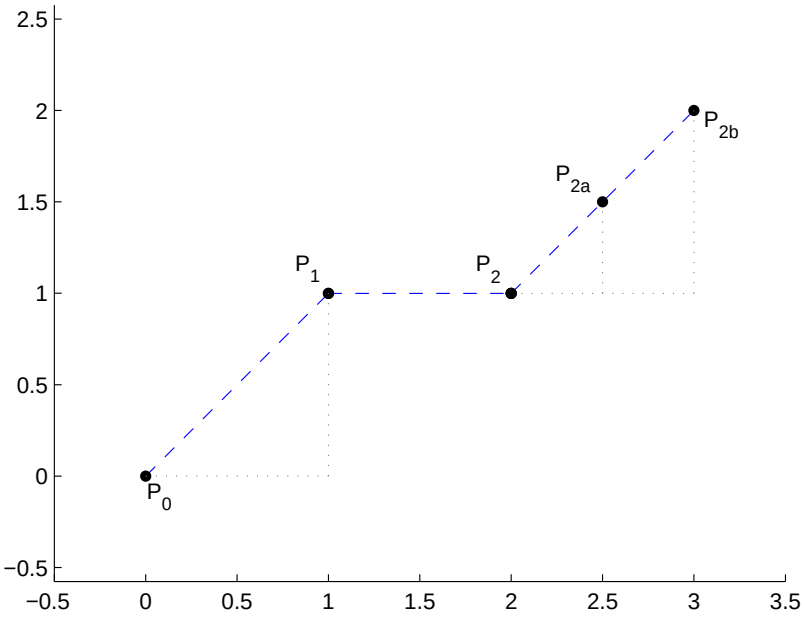


Figure 3.11 Momentum's effect on point movement

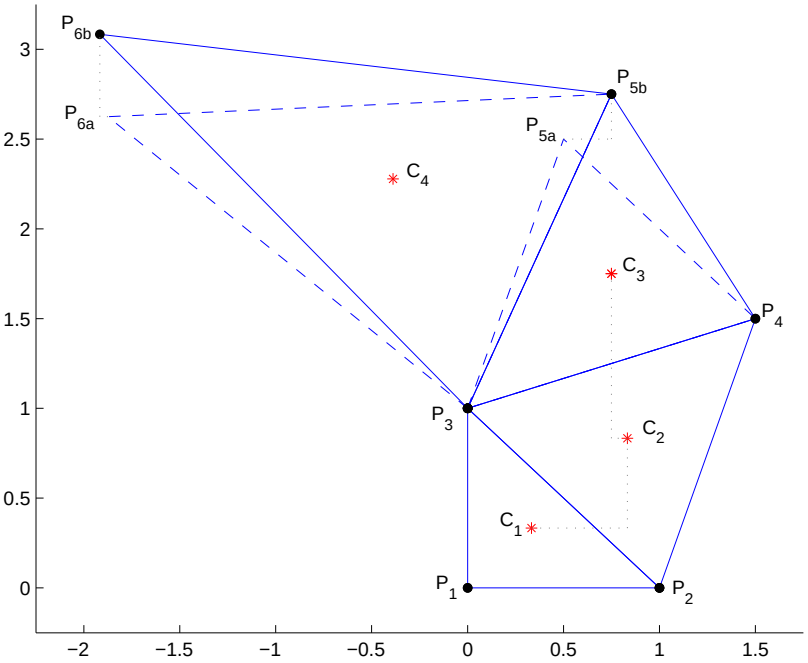


Figure 3.12 Momentum's effect on simplex movement

maxima. Naturally, the contribution of momentum needs to be large enough to escape a local maximum and its area of influence. However, too large a value for momentum will cause the simplex to precess around a potential global maximum rather than converge. This sensitivity and the marginal gains due to its use ultimately led to the rejection of momentum as an extension to the simplex method.

3.2.2 Simulated annealing

Simulated annealing was the next optimizing algorithm implemented. Nearly all of the techniques and strategies learned from the previous attempts were incorporated into the simulated annealing implementation. Because simulated annealing is an extension to the Monte Carlo method, much of the original code was reused. That allowed the easy incorporation of a static radius of movement, and a dynamic radius of movement was also implemented whose basis was the One-Fifth rule.

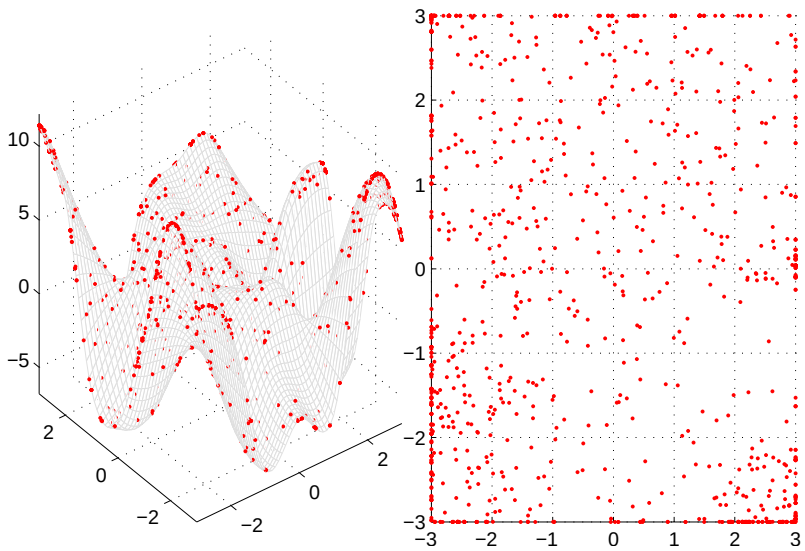


Figure 3.13 Simulated annealing with 1 walker

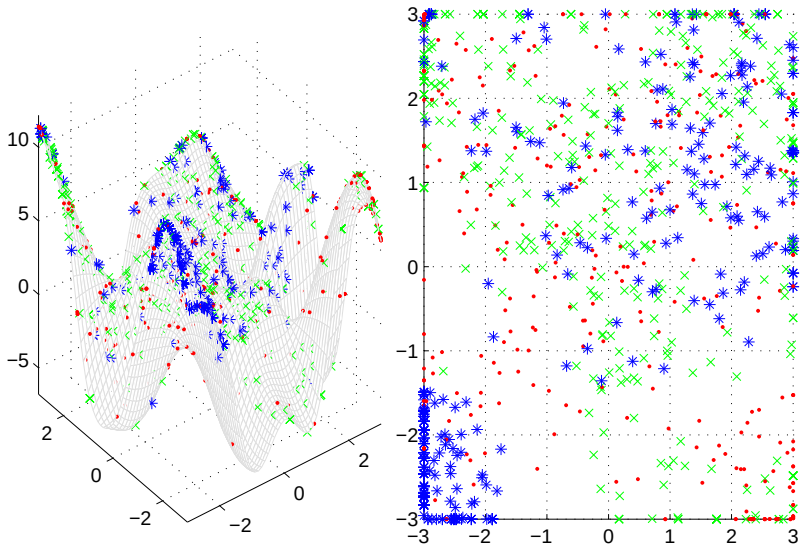


Figure 3.14 Simulated annealing with 3 walkers

Simulated annealing follows a different route to completion than Monte Carlo. Because it is possible for simulated annealing to move to less optimal locations, it is the system temperature that determines when the algorithm will finish. If a static radius of movement is implemented, then the system temperature is the only determining factor for algorithm termination. However, if a dynamic radius of movement is implemented, then a second terminating condition can be enforced for walkers whose radius of movement has become smaller than some threshold value.

For this work, simulated annealing was implemented to allow an arbitrary number of walkers. Figure 3.13 shows simulated annealing with one walker, while Figure 3.14 employs three walkers. For this work, both static and dynamic radii of movement were implemented and tested for their usefulness in optimizing the random function. The cooling schedule, shown as the stepped line in Figure 2.13, was a geometrically decreasing progression of the form

$$T_{i+1} = \begin{cases} \alpha_c * T_i & \mod\ (i, {}^{I}\!/_{T}) = 0 \\ T_i & \mod\ (i, {}^{I}\!/_{T}) \neq 0 \end{cases} \tag{3.3}$$

where $\alpha_c = 0.95$ and the initial and final system temperatures were calculated by a taking a number of random samples and applying Equation (2.5) and Equation (2.6). Furthermore, the maximum number of iterations until termination, I_{max}, was set to 1500, and the number of iterations between temperature changes, ${}^{I}\!/_{T}$, was calculated using the formula shown in Equation (3.4).

$${}^{I}\!/_{T} = \left\lfloor I_{max} \cdot \left(\frac{\log \alpha_c}{\log \left({}^{T_f}\!/_{T_0} \right)} \right) \right\rfloor \tag{3.4}$$

One side effect of using the floor function in Equation (3.4) is that it introduces another terminating condition to simulated annealing. For example, consider the data in Table 3.2. Recalling that the temperature is changed after ${}^{I}\!/_{T} = 32$ iterations and knowing from Equation (2.7) that $n = 45$ temperature changes occur between T_0 and T_f, then $32 * 45 = 1440$ iterations are required to cool from the initial system temperature to the final system temperature. Thus, the maximum number of allowable iterations, the final system temperature, or the radius of movement can all cause the algorithm to terminate.

T_0	13.25
T_f	1.27
α_c	0.95
I_{max}	1500
${}^{I}\!/_{T}$	32

Table 3.2 Example data for determination of cooling schedule

To determine an adequate number of random samples to use for these calculations, an exponential growth formula was used. This formula was based on the observation,

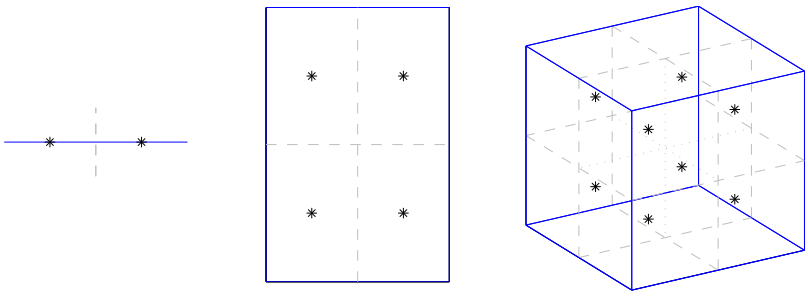


Figure 3.15 Number of samples required to determine system temperature

shown graphically in Figure 3.15, that if s regularly spaced samples were taken for each input dimension, then the number of samples N needed for d dimensions would be

$$N = s^d \tag{3.5}$$

Sample spacing, s	Number of dimensions, d				
	2	3	4	$\cdots$	10
2	4	8	16	$\cdots$	2^{10}
3	9	27	81	$\cdots$	3^{10}
4	16	64	256	$\cdots$	4^{10}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
10	100	1000	10000	$\cdots$	10^{10}

Table 3.3 Number of samples needed to determine system temperatures

This relationship is shown in Table 3.3. For the optimization of the random function, $s = 10$ was used, which required at least 100 random samples to determine system temperatures. One important note is that only sample pairs that produced a non-optimal move were included in N.

3.2.3 Combined approach

The combined approach is a novel approach to function optimization. It is a multiple stage algorithm that starts with simulated annealing and finishes with the simplex method. The main reason behind the implementation of the combined method was simulated annealing's poor performance with respect to runtime. In the worst case simulated annealing has an exponential runtime, which makes it infeasible as an optimizing strategy for functions of high dimensionality.

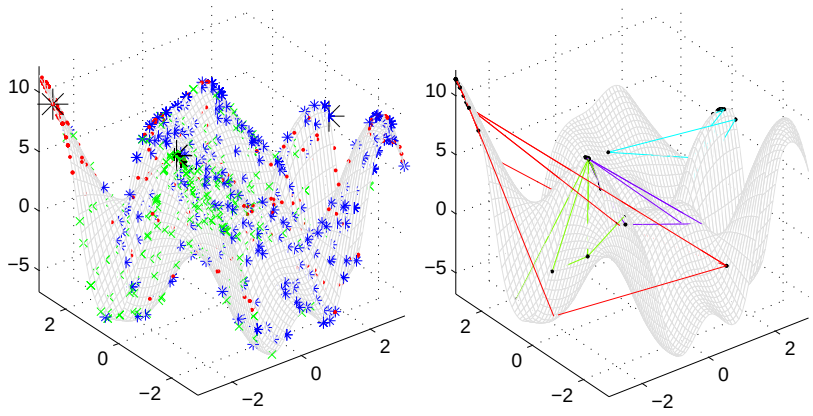


Figure 3.16 The combined approach

Simulated annealing's performance is very tightly linked with system temperature. High system temperature performance is essentially random due to the large number of non-optimal moves that are accepted. Little progress toward the global maximum is made during this phase. Likewise, little progress is made during the lower system temperatures. Low system temperatures will still allow slightly non-optimal moves, producing a precession around the global maximum as shown in Figure 3.17. It is the midrange system temperatures that make the most progress toward the global maximum.

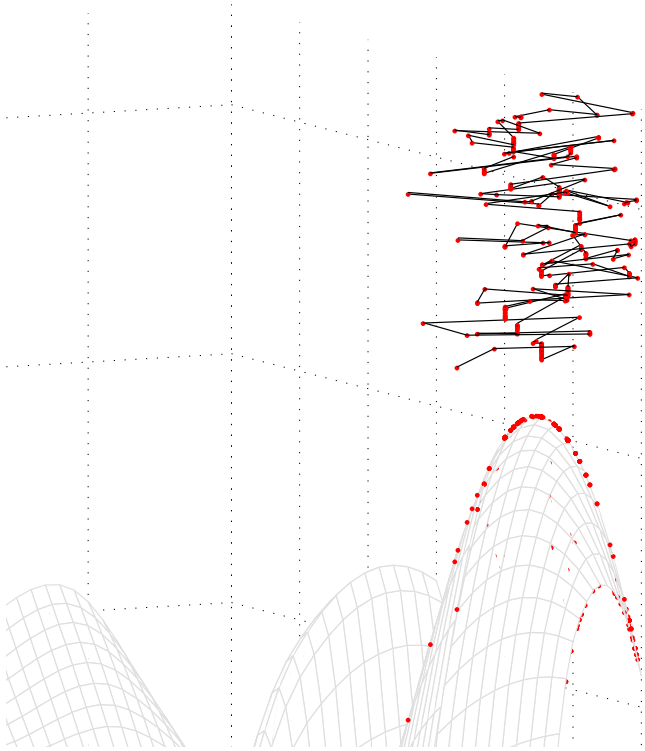


Figure 3.17 Low temperature simulated annealing precession

The high temperature range was eliminated altogether and because the low temperature performance was similar to Monte Carlo, it was replaced by the simplex method. This modification produced a general performance curve that is shown in Figure 3.18. Note that the simulated annealing phase begins with T_h, the high system temperature, and ends at T_c, the low system temperature. Upon the completion of the simulated annealing phase, the simplex method initializes and terminates in the usual way.

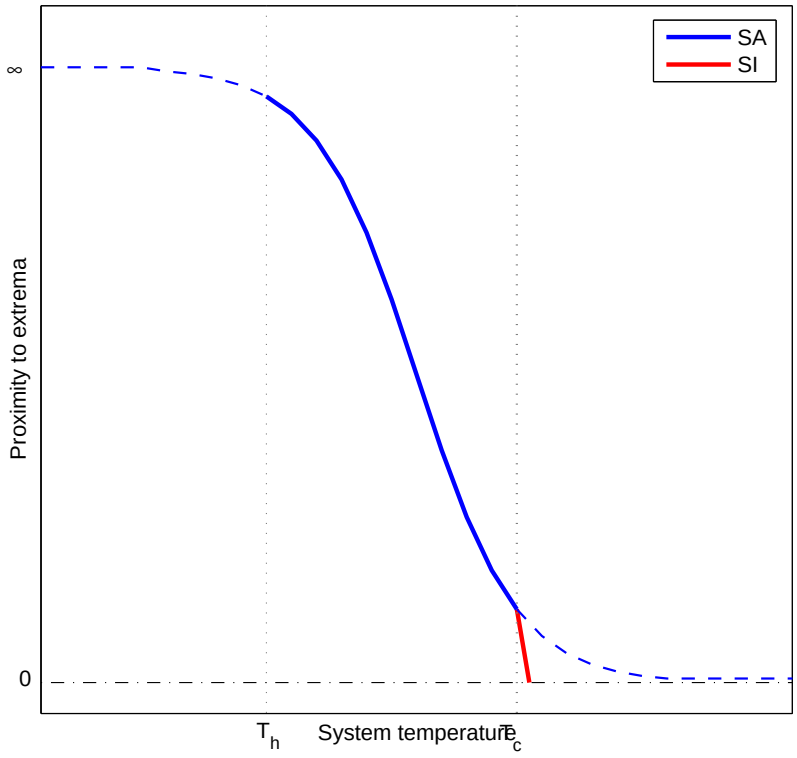


Figure 3.18 The combined approach performance curve

The idea behind the combined method is to contain the inefficiency of simulated annealing at its temperature extremes. It is not so important to finish at the global maxima using simulated annealing as it is to get finish at a location under its influence. If simulated annealing can terminate at an appropriate location, then the simplex

method can reach the global maximum from that location much more quickly and accurately.

Obviously, care must be taken when choosing T_h and T_c. The choice of T_h will mainly influence the runtime because it controls the amount of random searching performed. The choice of T_c is more important, as it controls the transition between simulated annealing and the simplex method. A T_c value that is too high may cause simulated annealing to terminate prematurely, resulting in a lower probability of the simplex method reaching the global maximum. For this work, the values of T_h and T_C were chosen using Equation (3.6) and Equation (3.7) to maximize the efficiency of simulated annealing without sacrificing any of its ability to explore the function's geometry.

$$T_0 = -\frac{\overline{|\Delta CF|}}{\ln p} = -\frac{2.7268}{\ln(0.80)} = 12.2199 \tag{3.6}$$

$$T_f = -\frac{\overline{|\Delta CF|}}{\ln p} = -\frac{2.7268}{\ln(0.10)} = 1.1842 \tag{3.7}$$

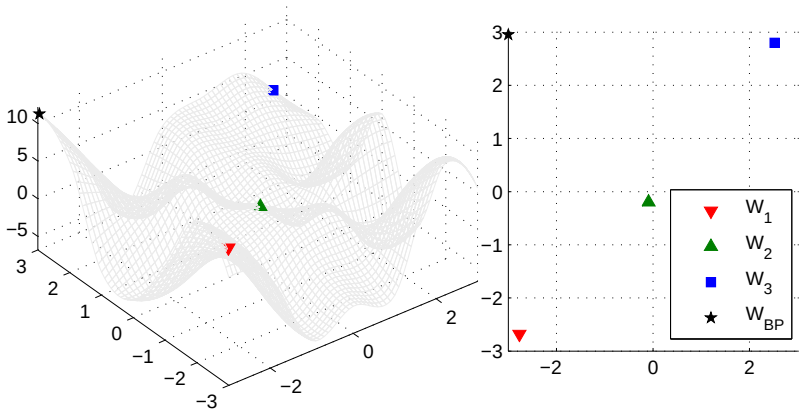


Figure 3.19 Walker positions at the end of the simulated annealing phase

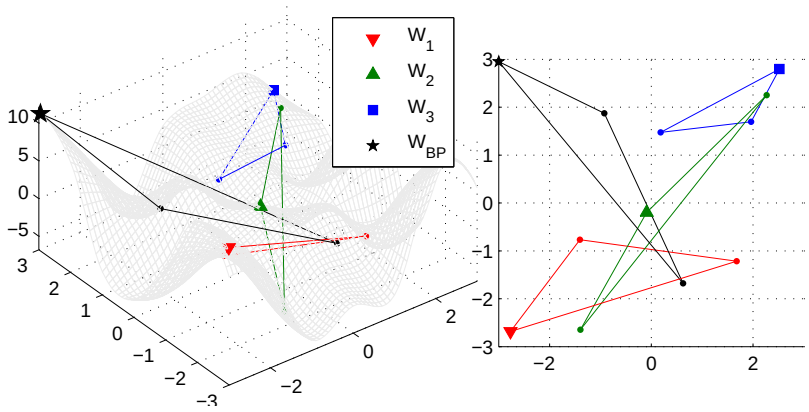


Figure 3.20 Simplex positions at the start of the simplex method phase

This implementation of the combined approach employed multiple walkers in the simulated annealing phase. Because the random function had a dimensionality of three, three walkers were used. In addition, the most optimal point seen during the simulated annealing phase was tracked (see Figure 3.19). At the beginning of the simplex method phase, four simplices were created (see Figure 3.20). Each of the simplices had one of the final points from the simulated annealing phase as a vertex and the other vertices were randomly chosen. This multiple walker/simplex approach was used to further increase the probability of finding the global maximum.

The reason for multiple simplices was to minimize the impact of the premature completion of the simulated annealing phase. Because the simulated annealing phase was not run to convergence, it was possible that one or more final walker locations was at or near a local maximum at the algorihtm's completion. By creating multiple simplices, the independence of each walker could be preserved during the simplex method phase. In addition, the amount of agreement or disagreement between the simplices at the end of the simplex phase could uncover more information about the effectiveness of the combined approach.

3.3 The Simulink ABS brake model

The ABS brake model is a built-in MATLAB SIMULINK model. It models the anti-lock braking system in a vehicle and is shown in Figure 3.21. Essentially a SIMULINK model is a series of blocks that are wired together, not unlike a circuit diagram. Whereas the components of a circuit diagram strictly model electrical structures such as resistors and inductors, the components of a SIMULINK model can represent anything from constants to functions to entire flight control subsystems. Each block can contain one or more variables as well as an internal state. Some models are completely self-contained with their own data stores, but the ABS model relies on variables stored in the MATLAB workspace.

The ABS model is attractive for testing optimization algorithms because it models a familiar system and also because it has modest number of inputs. Furthermore, its output, *stop_time*, is the amount of time required for the vehicle to come to rest from its initial velocity. Thus, the goal of optimizing the ABS model would be to determine the combination of input values that minimize the stopping time.

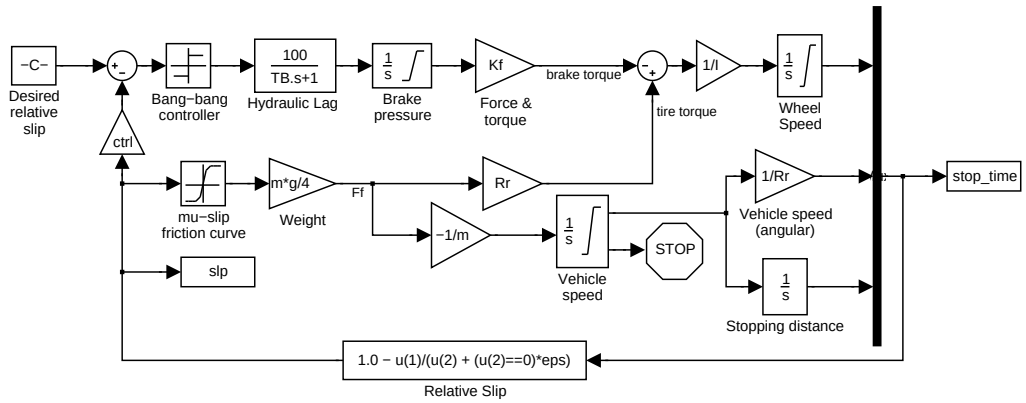


Figure 3.21 The ABS braking SIMULINK model

Because the ABS model relies on MATLAB workspace variables, any of the variables

that it relies on can be considered an input dimension. This allows the ABS model to attain a range of dimensionalities limited only by the number of variables required to simulate the model. The variables used by the ABS model can be partitioned into vehicle-specific variables and ABS-specific variables. It makes little sense to optimize the ABS model using any of the vehicle-specific variables as input dimensions, as they are outside the control of the ABS model. For this reason and also to limit the scope of the experimentation, only the variables shown in Table 3.4 were considered for inclusion in the input space.

Block Name	Variable
$1/I$	I
Desired relative slip	*slip_const*
Brake Pressure	$PBmax$
Force & Torque	Kf
ctrl	*ctrl*
Hydraulic Lag	TB

Table 3.4 Possible ABS model input dimensions

The ABS model was simulated for dimensionalities ranging between 3 and 7. For the dimensionality of 3, only the I and *slip_const* variables were included as input dimensions and the rest were given constant values. For the dimensionality of 8, all of the variables in Table 3.4 were included in the input space. The input dimensions included in each of the ABS model configurations are shown in Table 3.5. Note that in each case, the single output dimension was *stop_time*.

The ABS braking model was originally configured to set its variables, simulate a braking event, and graphically display the results. To modify the model to allow optimization activities, the normal behaviors were removed and constraints were created for each of the input dimensions. The constraints, shown in Table 3.6, were designed such that the original values for each variable fell roughly in the center of

Dimensionality	Input Dimensions
3	$\{I, slip\_const\}$
4	$\{I, slip\_const, PBmax\}$
5	$\{I, slip\_const, PBmax, Kf\}$
6	$\{I, slip\_const, PBmax, Kf, ctrl\}$
7	$\{I, slip\_const, PBmax, Kf, ctrl, TB\}$

Table 3.5 ABS model input dimension configurations

its respective input dimension's range. In example, the original ABS braking model value for *slip_const* was 0.2, and so the range for its input dimension was set to $0 \leq slip\_const \leq 0.5$.

Dimension	Lower constraint	Upper constraint
I	0.5	10
$slip\_const$	0	0.5
$PBmax$	100	2000
Kf	0.5	2
$ctrl$	0.5	2
TB	0.01	1

Table 3.6 ABS model input dimension constraints

The input dimension constraints were set to the values shown in Table 3.6 to mimic what an inexperienced tester might do under similar circumstances. In fact, the value for TB was deliberately chosen so that the original model value of $TB = 0.01$ became the lower bound of its range. This skewed choice of range values should also illuminate an optimization algorithm's performance at the input boundary.

3.3.1 Combined approach

The combined approach required very little modification to optimize the ABS braking model, because of the decision to use function pointers rather than actual functions in the algorithm. Nevertheless, the experimentation with the random function gen-

erator illuminated some deficiencies in the combined approach that were addressed in ABS braking model implementation. The first major modification involved the determination of the cooling schedule.

Recall that for the random function the determination of the upper and lower system temperatures involved approximating a regular division of the two input dimensions by randomly sampling points from the input space. For example, subdividing each input dimension into two sub-dimensions, as in Figure 3.15, produces four points. By randomly sampling points until four non-optimal moves are recorded, the average magnitude of a non-optimal move could then be calculated. This value was then used to determine T_h and T_c using Equation (3.6) and Equation (3.7).

While this approach may be appropriate for low dimensionality SIMULINK models, complex models will spend more time determining system temperatures than is feasible. In fact, even for the six input dimensions of the ABS braking model, this approach quickly becomes burdensome. To combat this exponential problem, a short-circuit mechanism was implemented that allowed the sampling to terminate early. The stopping condition was again based on a percent difference, but was not checked until the actual number of random samples had surpassed $\sqrt{N}$. Specifically, the number of divisions per input dimension was set to $s = 10$, and the percent difference threshold was set to 0.1 percent.

This approach allowed the input space to be adequately sampled so that appropriate system temperatures could be determined. It also kept the amount of over-sampling to a minimum by allowing the temperature determination algorithm to terminate early if no significant temperature changes have been made. While this strategy is by no means ideal, it was a good heuristic that produced system temperatures that allowed the simulated annealing phase to remain competitive in terms of number of model calls.

Another departure from the random function implementation of the combined approach concerned the number of walkers employed by the simulated annealing phase. In light of the simplex method's requirement of $n+m$ starting vertices, where n and m refer to the number of input and output dimensions respectively, $n+m-1$ walkers were employed in the simulated annealing phase. By treating a multiple-output SIMULINK model as a series of single output models, the value of m could be set to 1. This produced a requirement of n walkers during the simulated annealing phase and $n+1$ vertices for the simplex method phase. The remaining vertex was created from the best point seen during the simulated annealing phase.

The decision to reduce the number of simplices was based on the observation that the simplex method phase of the original combined approach implementation was wasting model calls. Had the goal been to discover all of the global and local maxima, then the use of multiple simplices would have been warranted. Considering the simulated annealing phase was also returning the best point seen, then only one simplex was needed in any event and so the others were removed from the implementation.

3.3.2 Evolution Strategy

The implementation of an evolution strategy algorithm for the optimization of the ABS braking model followed the outline given in Section 2.3.3. It employed a $(\mu, \lambda) = (105, 15)$ generational model for children creation and survivor selection. The object parameters for the initial population were all randomly chosen from the input space, and the strategy parameters were chosen using the formula given in Equation (3.8), where I_n is the n^{th} input dimension with upper and lower range constraints $I_{n,U}$ and $I_{n,L}$ and σ_n is its corresponding strategy parameter.

$$\sigma_n = \frac{I_{n,U} - I_{n,L}}{2} \tag{3.8}$$

Considering that the input ranges will likely have dissimilar ranges and scales, separate values for the minimum strategy parameter value ε_0 were kept. This resulted in an array of values of the form $(\varepsilon_{0,1}, \varepsilon_{0,2}, \ldots, \varepsilon_{0,n})$. Equation (3.9) shows the formula used to create the array.

$$\varepsilon_{0,i} = .001(I_{i,U} - I_{i,L}) \tag{3.9}$$

As a further constraint on the strategy parameters, any changes that resulted in more than a five-fold reduction of a strategy parameter was limited to that value. This additional requirement was designed to ease boundary condition behavior in which an individual would be crippled by a severe but otherwise acceptable reduction in one or more of its strategy parameters. In essence, the $\varepsilon_{0,n}$ value was designed to limit the long-term value of the strategy parameters, while the second constraint worked in the short-term to moderate the value of strategy parameters.

Mutation was performed first on the strategy parameters and then on the object parameters for each individual in the population. The values for τ' and τ are given in Equation (3.10), where n was the number of strategy parameters.

$$\tau' = 1/\sqrt{2n} \tag{3.10}$$

$$\tau = 1/\sqrt{2\sqrt{n}} \tag{3.11}$$

Any boundary crossings as a result of mutation were handled using Equation (3.2), with both the object and strategy parameters receiving adjustments. The object parameters are scaled according to the similar triangles method shown in Figure 3.6, while the strategy parameters become the product of the previous strategy parameter

values and the minimum boundary crossing ratio. Table 3.7 shows the adjustments made for an individual that crosses both the I and *slip_const* upper boundaries of the two-input ABS braking model. In this case, the minimum boundary crossing ratio is $r_{min} = 0.2$.

State	Object genes		Strategy genes	
	I	*slip_const*	σ_I	σ_{slip_const}
before mutation	9.000	0.400	0.200	1.250
after mutation	12.000	0.900	0.375	0.875
after adjustment	9.600	0.500	0.040	0.250

Table 3.7 Evolution strategy boundary crossing example

Recombination of the object parameters was accomplished using global discrete recombination. For the strategy parameters, global intermediary recombination was used. In the case of the strategy parameters, two parents were selected to participate in each global intermediary recombination, resulting in the child having the average of the two parents' parameters. Finally, survivor selection was generational with a selection pressure of $\mu/\lambda = 7$.

This implementation of the evolution strategy algorithm was limited to completing 50 generations before termination. To determine whether the population had converged, first the maximum and minimum values for each input dimension were determined. Next, the mean value for each input dimension was calculated. Finally, the normalized spread $NS_n$was calculated for each input dimension I_n using the formula given in Equation (3.12), where $max(I_n)$ and $min(I_n)$ represent the maximum and minimum input dimension values, respectively, currently held by individuals in the population. If the normalized spread value was not within 1 percent for all input dimensions, then the population had not converged and the algorithm would continue.

Table 3.8 shows a representative population of individuals being considered for convergence and Table 3.9 shows the calculations made to determine the convergence of each input dimension. Note that for convergence, only the object parameters are important and the strategy parameters are ignored.

$$NS_n = \left| \frac{max(I_n) - min(I_n)}{mean(I_n)} \right| \tag{3.12}$$

As can be seen, it is clear that the input dimension I has converged to within 0.96 percent, while the input dimension *slip_const* has converged to within 2.172 percent. Using the minimum convergence value of 1 percent as a guide, the population has not converged due to the *slip_const* input dimension.

Individual	Object genes	
	I	*slip_const*
1	0.520	2.93
2	0.524	2.61
3	0.522	2.50
4	0.523	2.72
5	0.519	2.36

Table 3.8 Evolution strategy population data

Metric	Input dimension	
	I	*slip_const*
max	0.5240	2.93
min	0.5190	2.36
$mean$	0.5216	2.624
NS	0.0096	0.2172

Table 3.9 Evolution strategy convergence data

3.3.3 Experimental setup

This section describes the experimental setup for comparing the effectiveness of the algorithms described in Section 3.3.1 and Section 3.3.2. Five experiments were run for each algorithm, and the specifics of each experiment are given in Table 3.10 and Table 3.11. For example, the first experiment run required the ABS braking model to be configured to have two input dimensions and then each algorithm ran 100 complete optimizations of the model.

Experiment	Input Dimensions	Optimizations
1	$\{I, slip\_const\}$	100
2	$\{I, slip\_const, PBmax\}$	100
3	$\{I, slip\_const, PBmax, Kf\}$	100
4	$\{I, slip\_const, PBmax, Kf, ctrl\}$	100
5	$\{I, slip\_const, PBmax, Kf, ctrl, TB\}$	100

Table 3.10 ABS braking model experiment specifics for the combined approach

Experiment	Input Dimensions	Optimizations
1	$\{I, slip\_const\}$	100
2	$\{I, slip\_const, PBmax\}$	100
3	$\{I, slip\_const, PBmax, Kf\}$	100
4	$\{I, slip\_const, PBmax, Kf, ctrl\}$	100
5	$\{I, slip\_const, PBmax, Kf, ctrl, TB\}$	100

Table 3.11 ABS braking model experiment specifics for evolution strategy

In each experiment, the total running time of the algorithm, the total number of model calls, and both the best point seen and all final points were recorded. From the combined approach's perspective, the final point was the centroid of the simplex at the

conclusion of the simplex method. In the case of evolution strategy, the most optimal point in the final generation was recorded as the final point. In addition, the number of iterations completed were recorded for each algorithm. For the combined approach, the number of iterations and model calls were recorded separately for the simulated annealing and simplex algorithms, while for the evolution strategy algorithm, the number of generations were recorded. Table 3.12 shows the struct returned at the end of the evolution strategy optimization algorithm and its members, while Table 3.13 shows the data structure returned at the end of the combined approach optimization algorithm.

Data structure	Member	Data description
stats	es_time	Algorithm runtime
	nfc	Number of model calls
	max_vertex	Best point seen (not in final generation)
	num_generations	Number of generations
	best_individual	Most optimal point (in final generation)

Table 3.12 Evolution strategy optimization result data structure

Data structure	Member	Data description
stats	sa_time	Simulated annealing runtime
	si_time	Simplex method runtime
	sa_n	Simulated annealing model calls
	si_n	Simplex method model calls
	sa_max_vertex	Best point seen during the simulated annealing phase
	sa_iterations	Number of iterations during the simulated annealing phase
	sa_final_points	Final locations of the simulated annealing walkers
	si_final_points	Centroid of the final simplex from the simplex method

Table 3.13 Combined approach optimization result data structure

CHAPTER 4 RESULTS

In this chapter, the results of all the work described in the previous chapters will be shown. Section 4.1 presents the results of the early research, in which algorithms useful for the optimization of PEAKS were explored. Section 4.2 builds on these results and presents the results of optimizing a random function with the combined approach algorithm. Finally, Section 4.3 presents a comparison between the combined approach and evolution strategy algorithms in their effectiveness in the optimization of the ABS braking model.

4.1 peaks results

The results of the work done with the PEAKS function are relevant mainly for their effect on the subsequent research. For this reason, a rigorous defense of the merits of the methods employed when attempting to optimize PEAKS will not presented. Rather, this section will contain a high-level description of the experiments and corresponding data.

The first experiment compared the relative abilities of Monte Carlo and the simplex method in optimizing PEAKS. For the Monte Carlo part of the experiment, 20 independent walkers were employed, and for the simplex method 20 independent simplices were employed. The two methods were each run 100 times, and data was collected for average deviation from the global maximum, the number of simplices or walkers

that arrived at the global maximum, the runtime, and the number of function calls made during each run.

In order to make a true comparison, the two methods were normalized with respect to average deviation from the global maximum. In other words, calibrating runs were used to determine appropriate terminating thresholds for both methods. First the simplex method's terminating condition of minimum simplex size was set to 0.01. The Monte Carlo method was then calibrated to produce equivalent average deviations, resulting in a minimum percent difference of 0.000001.

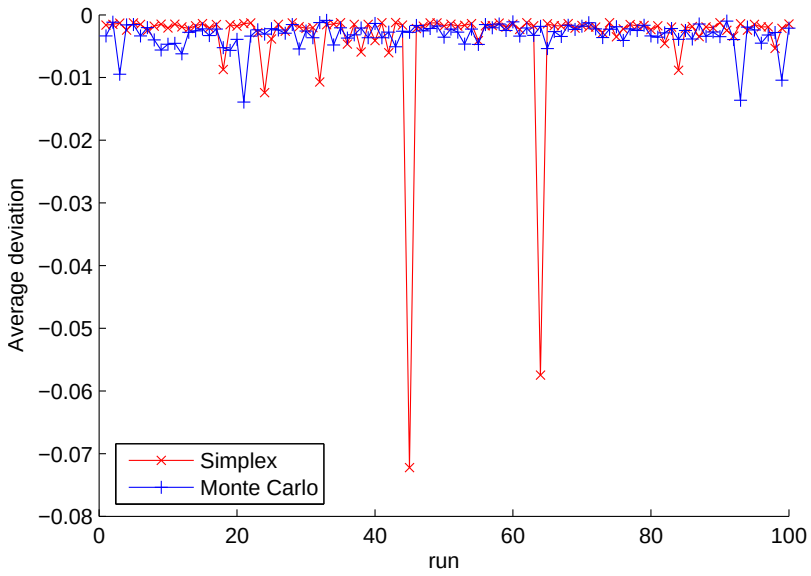


Figure 4.1 Algorithmic calibration between Monte Carlo and the simplex method

The result of this effort is shown graphically in Figure 4.1. Due to the stochastic nature of the Monte Carlo algorithm, complete agreement between it and the simplex method was all but impossible. Nonetheless, after calibration Monte Carlo produced an average deviation of 0.325 percent to the simplex method's 0.365 percent, agreeing to within 0.020 percent over the 100 runs executed.

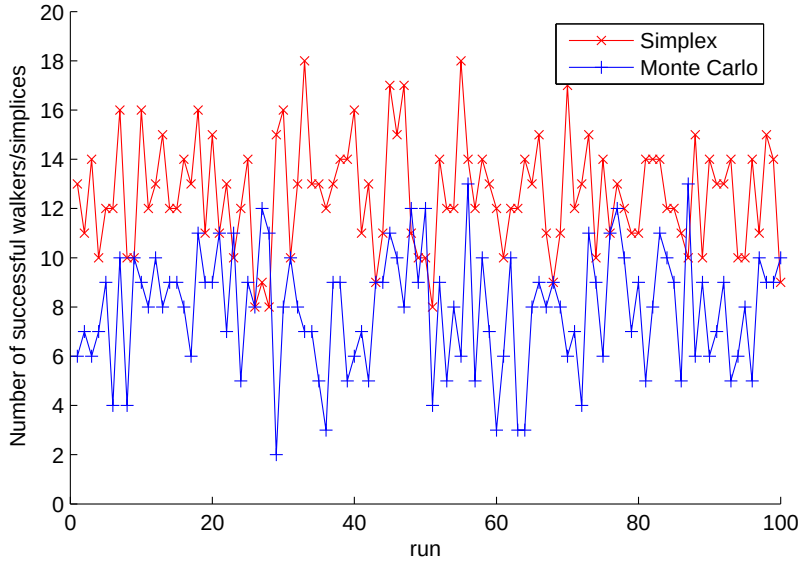


Figure 4.2 Comparison with respect to number of successful walkers/simplices

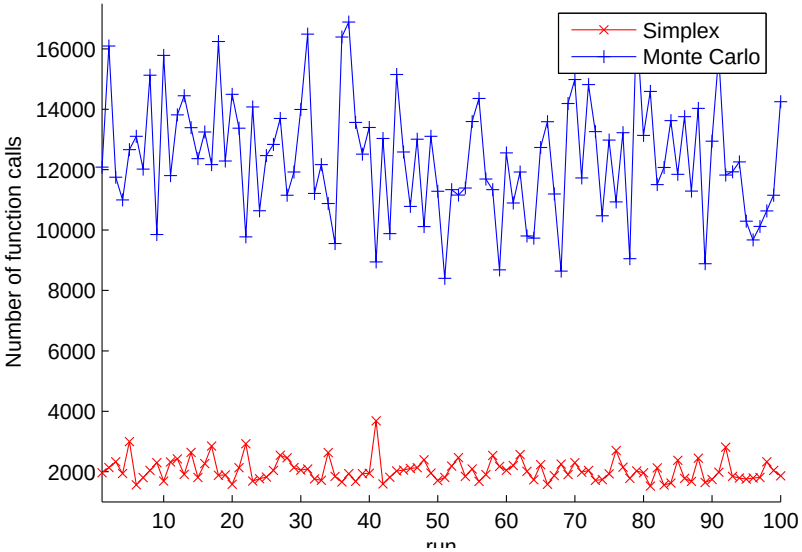


Figure 4.3 Comparison with respect to number of function calls

Figure 4.2 shows the number of walkers or simplices that arrived at the PEAKS global maximum for each algorithm. This was somewhat surprising, as the assumption was that a stochastic algorithm would outperform a deterministic one. However, recall that a simplex with two input dimensions has three points from which it can potentially move in comparison to Monte Carlo's one point. Furthermore, the simplex method is based on constant, incremental improvement. These two facts facilitated the simplex method's 12.57 hits on average to Monte Carlo's 7.9 hits, and certainly pointed to the simplex method as having the better ability to optimize PEAKS.

The next metric evaluated was the number of function calls. As can be seen in Figure 4.3, the simplex method again dominated Monte Carlo. Whereas Monte Carlo took 12398 function calls on average to finish an optimization, the simplex method took only 2045 function calls. Even more striking, the simplex method had a much lower standard deviation than the Monte Carlo data (see Table 4.1). This all means that the simplex method not only required less function calls than Monte Carlo, but also that the simplex method was much more consistent in the number of function calls it required.

	Monte Carlo	Simplex method
mean (μ)	12398	2045
std dev (σ)	1974	367

Table 4.1 Number of function calls data when optimizing PEAKS

Despite the observed superiority of the simplex method to Monte Carlo when optimizing PEAKS, the simplex method was still susceptible to initial conditions. In fact, the simplex method had an average success rate of only 62.85 percent. To combat this problem, momentum was added to the simplex method. Separate experimental setups were devised to test both centroid and worst-point momentum. The momen-

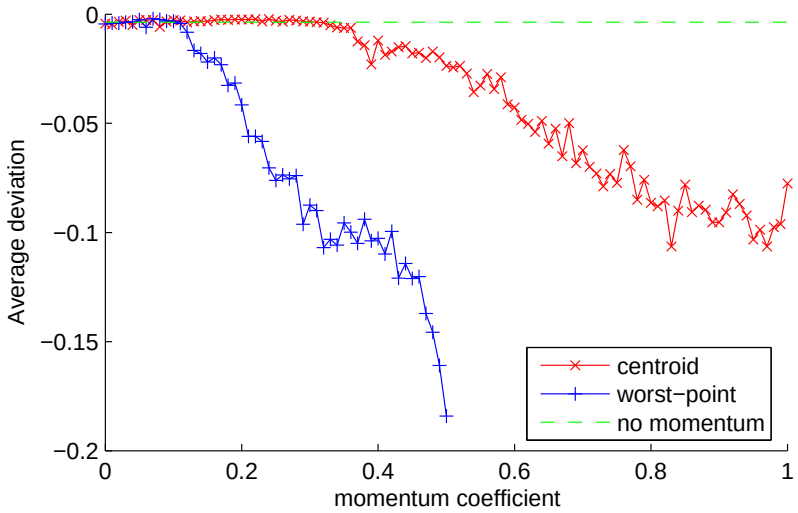


Figure 4.4 The effect of momentum on average deviation in the optimization of PEAKS

tum coefficient, which is the percent value of momentum's contribution to the simplex movement, was varied between 0 and 1. For each value of the momentum coefficient, 100 optimizing runs were executed and the results recorded.

As is readily apparent in Figure 4.4, both of the momentum schemes initially produced results comparable to the original simplex method algorithm. However, as the effect of momentum was increased, the momentum-based implementations began to falter and the original algorithm dominated. Even more alarming, the worst-point momentum algorithm degenerated so quickly and produced such poor results that its momentum coefficient range was reduced to between 0 percent and 50 percent.

This alarming trend can be seen more clearly in Figure 4.5, in which the worst-point algorithm appears to take an exponential shape. Also of interest in Figure 4.5 is that again neither of the momentum-based algorithms are comparable to the base simplex algorithm. Despite these dismal results, momentum could have still been a viable addition to the simplex method if the number of successful optimizations were

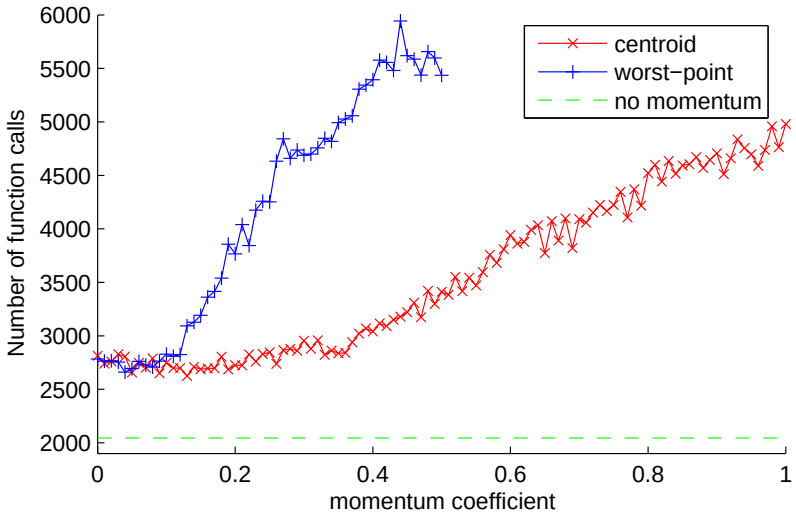


Figure 4.5 The effect of momentum on the number of function calls

significantly better than the basic algorithm. Unfortunately, as Figure 4.6 shows, this was not the case. In what was by now a familiar result, the centroid momentum outperformed the worst-case implementation, but the original simplex algorithm dominated.

4.2 Random function generator results

The results discovered through the optimization of PEAKS were deceptively optimistic. Because of the large area controlled by the PEAKS global maximum, the number of successful simplices and the effectiveness of the algorithm were both overly positive. When using the same algorithm to optimize a typical random function, such as the one shown in Figure 4.7, the results were much less impressive.

The results of 100 optimization runs using the random function shown in Figure 4.7 are shown in Figure 4.8. Clearly the more convoluted function geometry is wreaking havoc on the simplex method, producing an average success rate of only 18.95 percent.

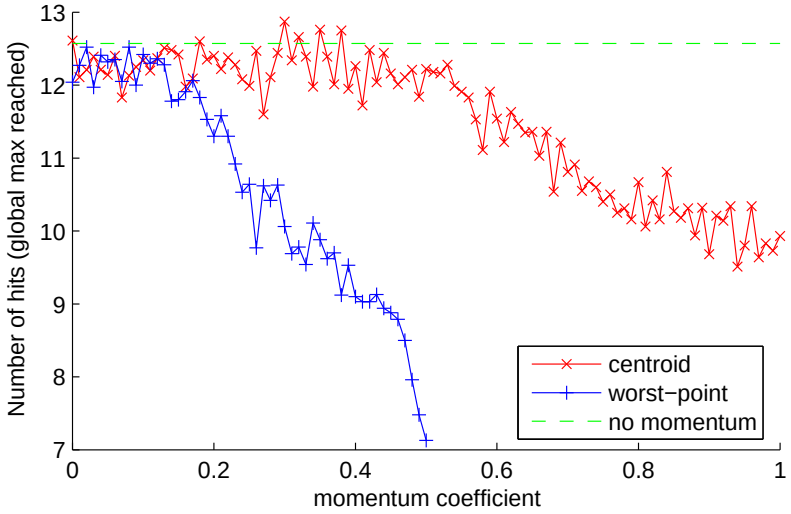


Figure 4.6 The effect of momentum on the number of successful simplices

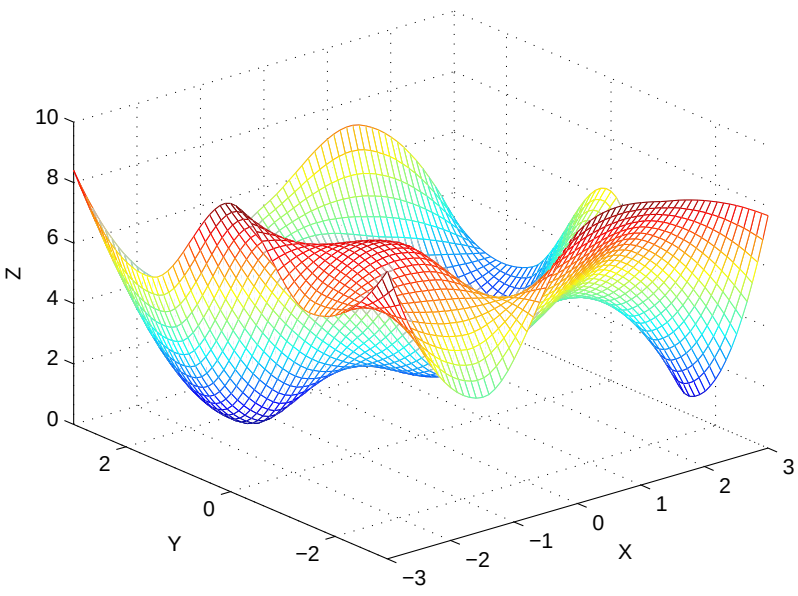


Figure 4.7 A typical random function optimized using the simplex method

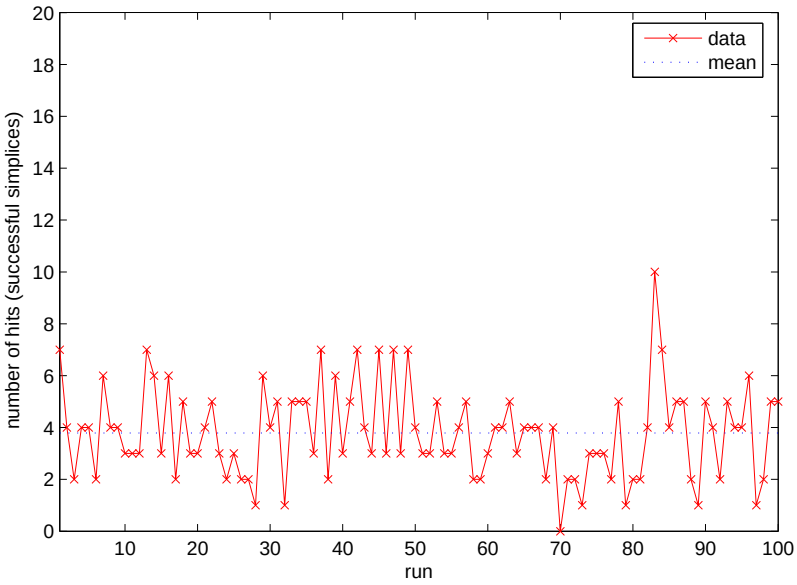


Figure 4.8 The number of successful simplices when optimizing a typical random function

It was for this reason that alternate algorithms and in particular simulated annealing were explored.

Repeating the above experiment with simulated annealing instead of the simplex method improved the average success rate to 42 percent. The results, shown graphically in Figure 4.9, at first appeared to reveal a large number of successful walkers. However, for the random function used there was a local maximum that was near to the global maximum with respect to output value. In fact, the vertex of the local maximum was $(0.60, -3.00, 9.4258)$ while the vertex of the global maximum was $(-3.00, 0.00, 9.5388)$.

The usefulness of simulated annealing was highly dependent on the ability of the walker to reach the global maximum value. By increasing the number of walkers involved in the algorithm, the usefulness of the algorithm could be extended. With multiple walkers, it was less important that every walker reach the global maximum;

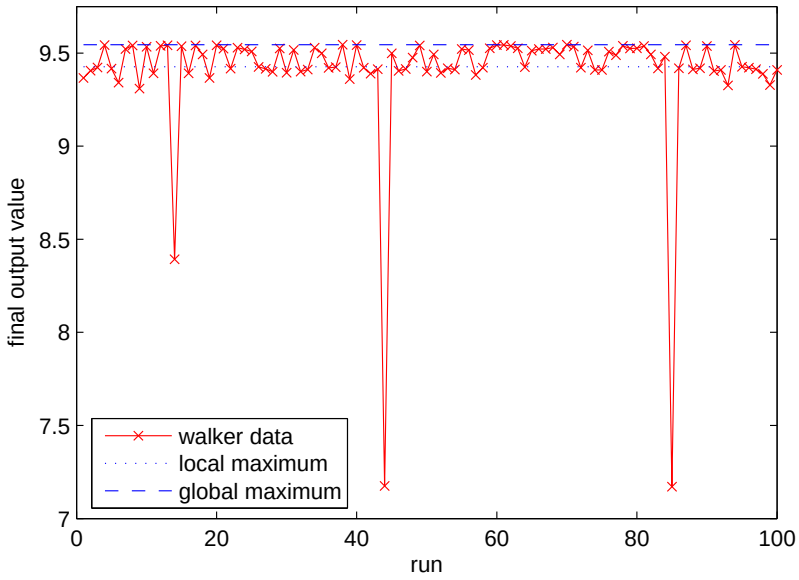


Figure 4.9 The most optimal output value using simulated annealing with one walker

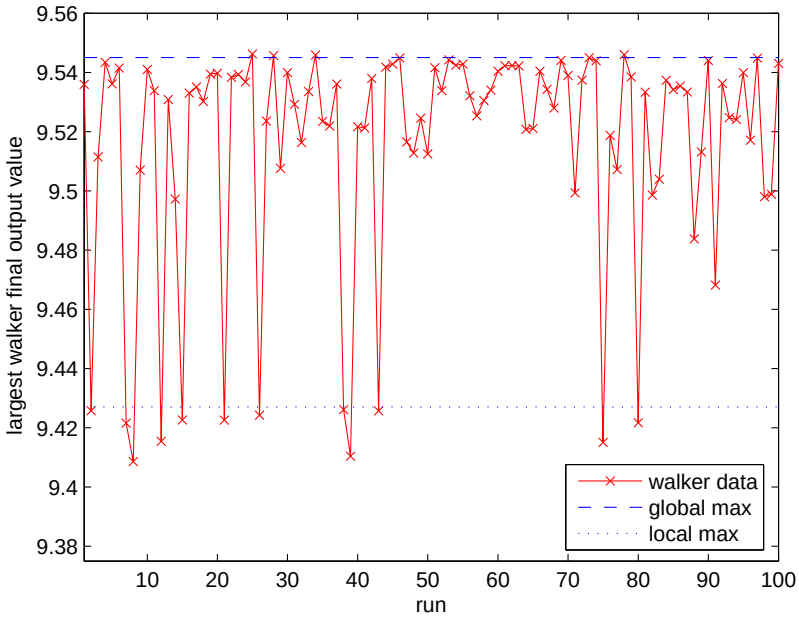


Figure 4.10 The most optimal output value using simulated annealing with three walkers

in fact, it was only important that one of the walkers do so.

Figure 4.10 shows the results of optimizing a random function using simulated annealing with three walkers over 100 runs. Note that only the walker whose output value is closest to the global maximum is shown for each run. As can be seen from the figure, using just three walkers when optimizing a typical random function yielded an average success rate of 82 percent. Also of note is that all of the runs resulted in either a local maximum value or a global maximum value, as opposed to Figure 4.9, in which three of the runs produced non-optimal results.

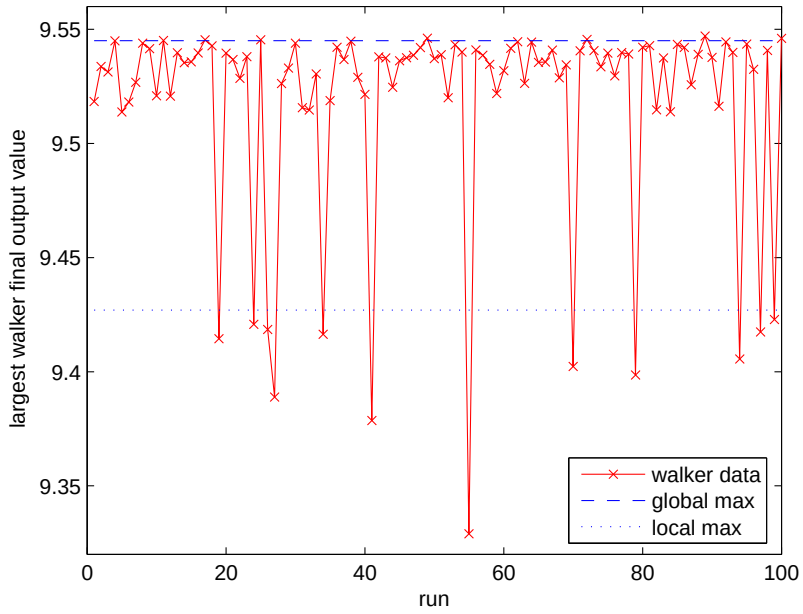


Figure 4.11 The maximum final walker output value using simulated annealing with three walkers and compressed cooling schedule

Reducing the number of iterations had little effect on the results. For the simulated annealing results of Figure 4.9 and Figure 4.10, 3000 iterations were used. The results obtained in Figure 4.11 required only 1500 iterations, and yielded an average success rate of 88 percent using the best of the three walkers. Of course, reducing the number

of iterations also required the number of iterations per system temperature to drop from 25 to 12 to maintain equivalent cooling schedules. The important insight was the ability to compress the cooling schedule to improve the algorithm's efficiency with respect to runtime.

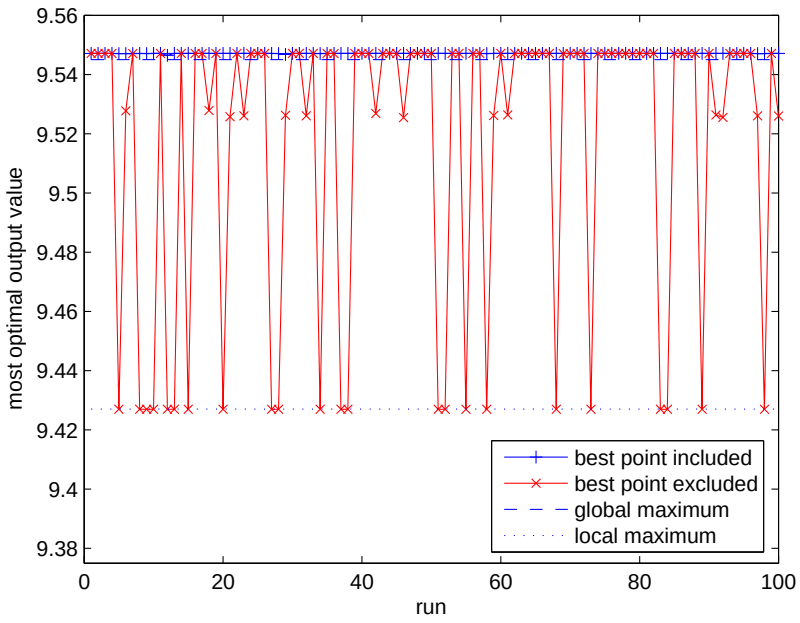


Figure 4.12 The maximum final output value using the combined approach

When optimizing using the combined approach, the results were even more impressive. So impressive, in fact, that the average success rate was 100 percent! This result was tempered by the fact that the combined approach, during the simulated annealing phase, was also tracking the best point seen. Thus, the best point seen was always being returned by simulated annealing and therefore counting toward the success rate. Figure 4.12 shows this phenomenon. In the case of inclusion of the best point seen, the average success rate was 100 percent, while for the exclusion of the best point seen, the average success rate dropped to 77 percent.

Because the motivation for this part of the research was the development of a proof-

of-concept tool, runtime and algorithmic efficiency were explored only minimally. The focus was on creating a prototype tool for the optimization of SIMULINK models, and so only the ability to optimize was explored, although a basic examination of runtime resulted in the combined approach. In subsequent research, and in particular the next section, the combined approach is compared to evolution strategies in a formal manner.

4.3 ABS braking model results

The ABS braking model proved more problematic to optimize than was anticipated. To gain some insight into possible causes and to determine the true optimal value for each configuration of the ABS braking model, an exhaustive method was employed. This method regularly sampled the input space for each configuration of the SIMULINK model and tracked the best point seen.

Input	Number of input dimensions				
	2	3	4	5	6
I	0.50	0.50	0.50	0.50	0.50
slip_const	0.18	0.18	0.175	0.3667	0.30
$PBmax$	(1500)	556	290	353.3	290
Kf	(1)	(1)	2	2	2
ctrl	(1)	(1)	(1)	1.90	1.55
TB	(0.01)	(0.01)	(0.01)	(0.01)	0.01
divisions	25	25	20	15	10
Stop time	13.5899	13.5899	12.3707	12.2711	12.2715
Model calls	676	17576	194481	1048576	1771561

Table 4.2 Semi-exhaustive results for all ABS model configurations

The results of these semi-exhaustive explorations are shown in Table 4.2. Note that values shown in parenthesis were held constant. For example, during the semi-exhaustive search of the two input ABS model, *PBmax*, *Kf*, *ctrl*, and *TB* were held

constant. The reason for the term semi-exhaustive is that a truly exhaustive search would take an unreasonably long time to complete, as was discussed in Section 2.1.2. Table 4.2 also shows the details of each semi-exhaustive search for the ABS model's true optimal value. Note that the *divisions* term represents the number of equal divisions made on each input dimension. For an input dimension r whose range is $1 \leq r \leq 10$, subdividing this range into 5 equal divisions would result in the following six sample points:

$$(1, \quad 2.8, \quad 4.6, \quad 6.4, \quad 8.2, \quad 10)$$

For example, the two-input model configuration required an exploration of 26^2 input states while the six-input model required 11^6. The values discovered during the semi-exhaustive search stand in sharp contrast to the values determined as a result of optimization. Table 4.3 shows the optimization results obtained using the combined approach, while Table 4.4 shows the results obtained using the evolution strategy algorithm. Note that in both cases, the number of model calls presented was the number of model calls required during the run in which the most optimal stopping time was discovered.

Input	Number of input dimensions				
	2	3	4	5	6
I	0.500	0.5008	0.5877	0.5007	0.5927
$slip\_const$	0.1966	0.3655	0.2336	0.3447	0.1709
$PBmax$	(1500)	504.7888	252.6435	252.6811	256.8028
Kf	(1)	(1)	2.000	2.000	1.9698
$ctrl$	(1)	(1)	(1)	1.6855	0.8451
TB	(0.01)	(0.01)	(0.01)	(0.01)	0.01
stop time	13.5108	13.4994	12.2400	12.2267	12.2494
Model calls	3154	4670	6317	6138	9178

Table 4.3 Combined approach results for all ABS model configurations

In all configurations except the two-input model, both stochastic algorithms re-

quired far fewer model calls in determining the most optimal output value. In addition, the output values returned using the two algorithms were superior to those values determined using the semi-exhaustive search.

Input	Number of input dimensions				
	2	3	4	5	6
I	0.500	0.500	0.500	0.500	0.500
$slip\_const$	0.1965	0.1966	0.2015	0.3662	0.3192
$PBmax$	(1500)	508.4308	252.6701	1989.4	1924.00
Kf	(1)	(1)	2.000	2.000	2.000
$ctrl$	(1)	(1)	(1)	1.8967	1.6529
TB	(0.01)	(0.01)	(0.01)	(0.01)	0.01
stop time	13.5107	13.5062	12.2267	12.271	12.271
Model calls	960	1275	1695	1485	2430

Table 4.4 Evolution strategy results for all ABS model configurations

Another complicating factor was the shape of the ABS model's output geometry, an example of which is shown in Figure 4.13. The ABS model's output geometry often contained features such as troughs, ridges, and plateaus, but rarely contained a single optimal value. A notable exception that does contain a single optimal value is shown in Figure 4.14. These results were different enough from the assumptions made about the random function generator that they had a major impact on the effectivenesses of the two algorithms implemented. Another interesting phenomenon observed was a region of output geometry that was featureless, as shown in Figure 4.15.

The first comparison metric was the number of model calls required to optimize the ABS braking model. As shown in Figure 4.16 for the two-input ABS braking model configuration, evolution strategy performed better than the combined approach. Recalling that for the two-input configuration the combined approach performed a maximum of 3000 model calls during the simulated annealing phase (2 walkers for 1500 iterations), then it is clear that the simplex method contributed very little to the

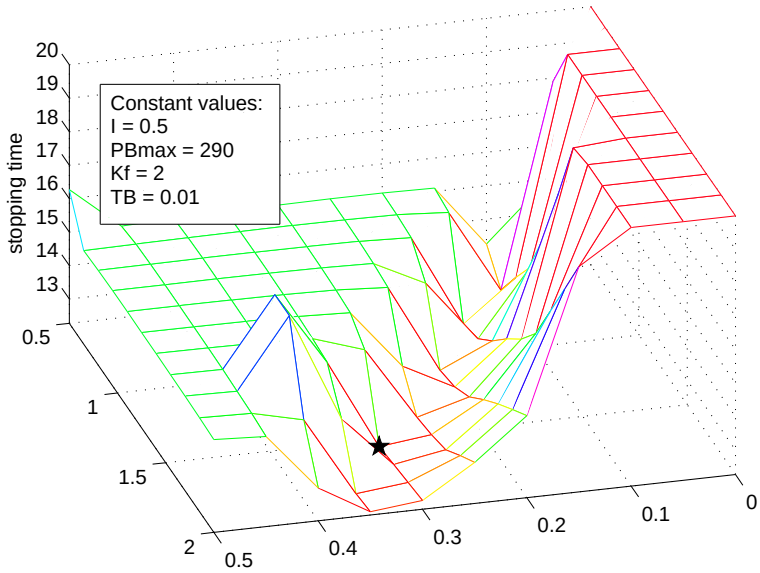


Figure 4.13 Six-input ABS model output geometry showing plateaus and a trough

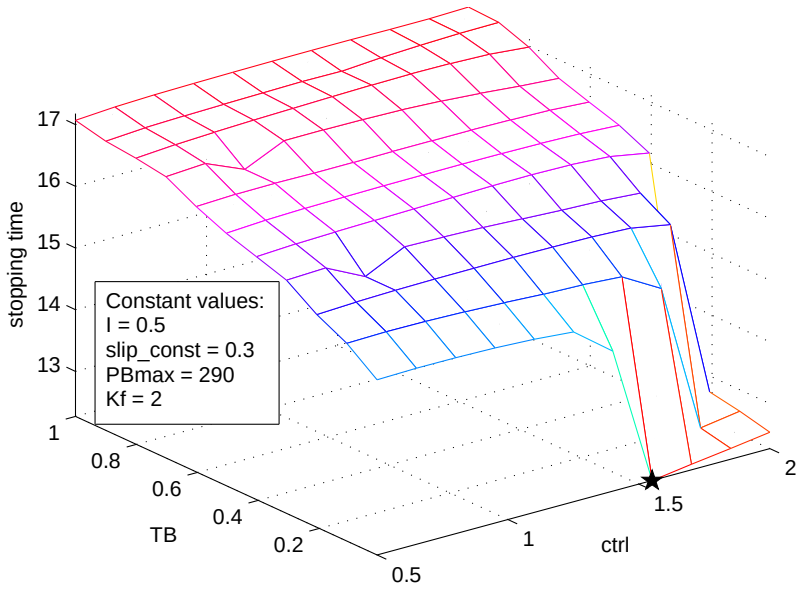


Figure 4.14 Six-input ABS model output geometry showing a single optimal point

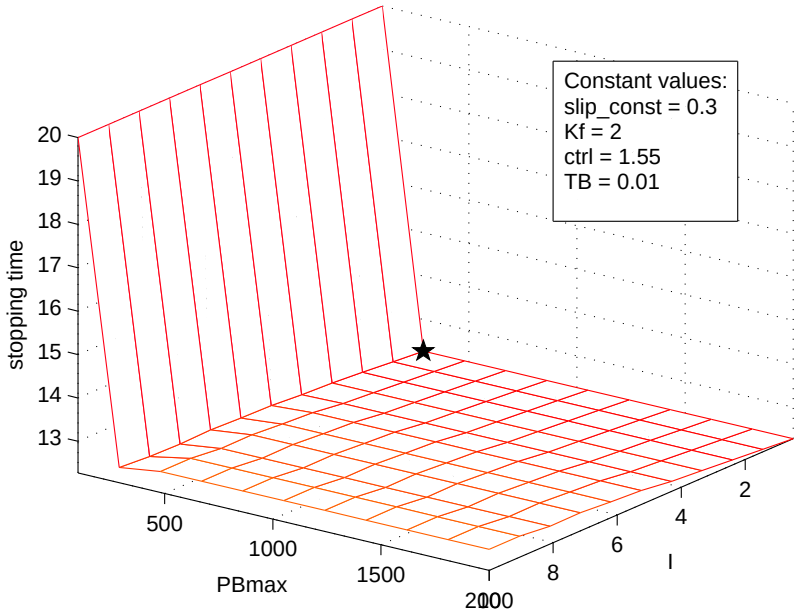


Figure 4.15 Featureless six-input ABS model output geometry

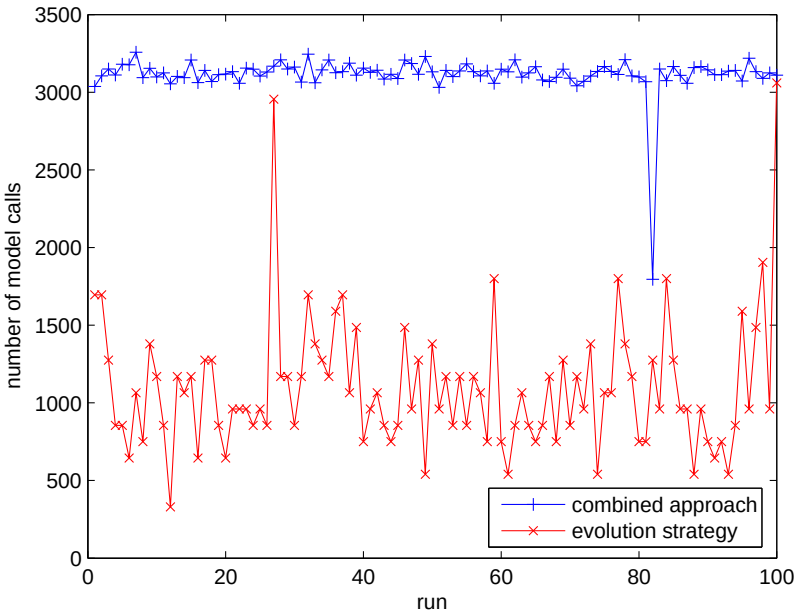


Figure 4.16 Number of model calls to optimize the ABS model with two input dimensions

overall number of function calls. Also of note is the high variability of the evolution strategy results; however, considering the upper limit for number of model calls when using evolution strategy is 5250 (105 children for 50 generations), then the results for the two-input model are encouraging for evolution strategy.

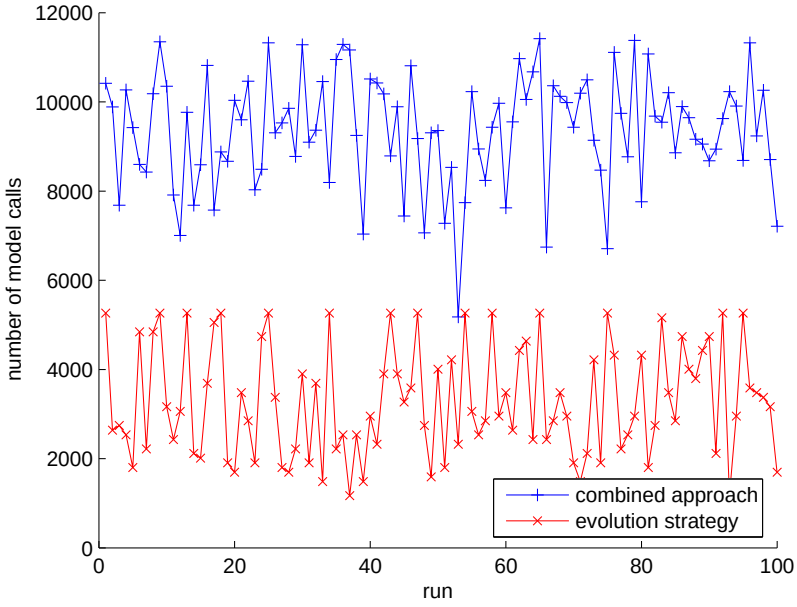


Figure 4.17 Number of model calls to optimize the ABS model with six input dimensions

The results of optimizing the six-input ABS braking model configuration are shown in Figure 4.17. In this case, the combined approach performed a maximum of 9000 model calls during the simulated annealing phase (6 walkers for 1500 iterations) while again the evolution strategy required a maximum of 5250 model calls. As can be seen from Figure 4.17, much more variability can be seen in the combined approach results.

In addition, many of the runs required less than 9000 model calls. The reason for this surprising result was early walker completion due to the use of dynamic radii of movement. Recall that the walkers' radii of movement were governed by the One-

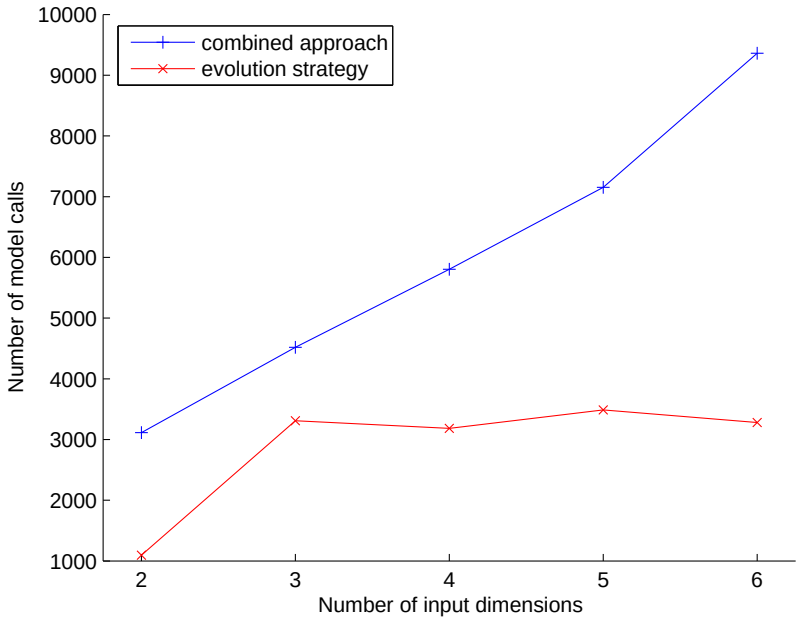


Figure 4.18 Average number of model calls to optimize all ABS model configurations

Fifth rule. Thus, as a walker's radius of movement shrank below a threshold value, the algorithm terminated the walker early. As for the evolution strategy results in Figure 4.17, a larger percentage of runs completed all 50 generations without converging. Despite this increase, the evolution strategy algorithm still managed to outperform the combined approach with respect to number of model calls required.

The results of the 100 runs for each configuration were averaged and are shown in Figure 4.19. Note that the average number of model calls for the combined approach has the beginnings of an exponential shape. This is not surprising, as the main contributing factors to the number of function calls when using the combined approach are the system temperature determination and the simulated annealing phase.

Certainly the simulated annealing phase's contribution is linear, because the number of model calls due to simulated annealing is simply the product of the maximum

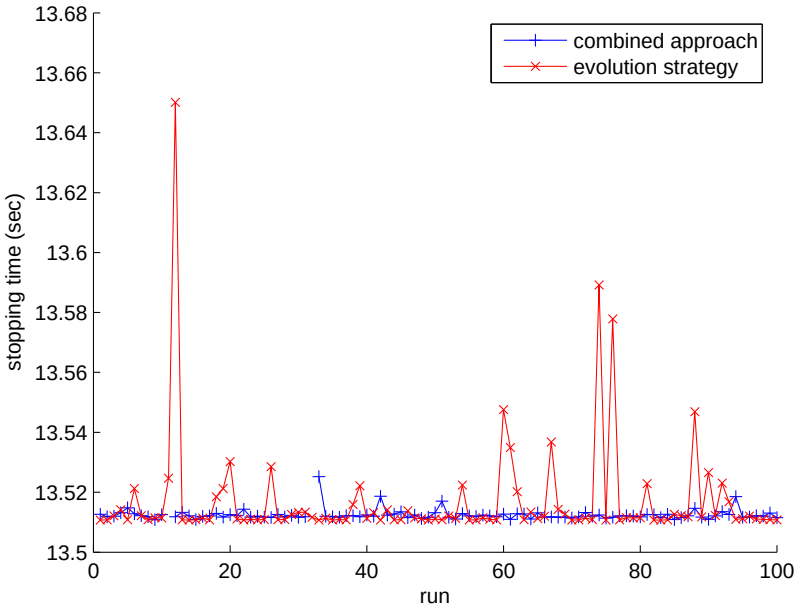


Figure 4.19 Optimal stopping times for the ABS model with two input dimensions

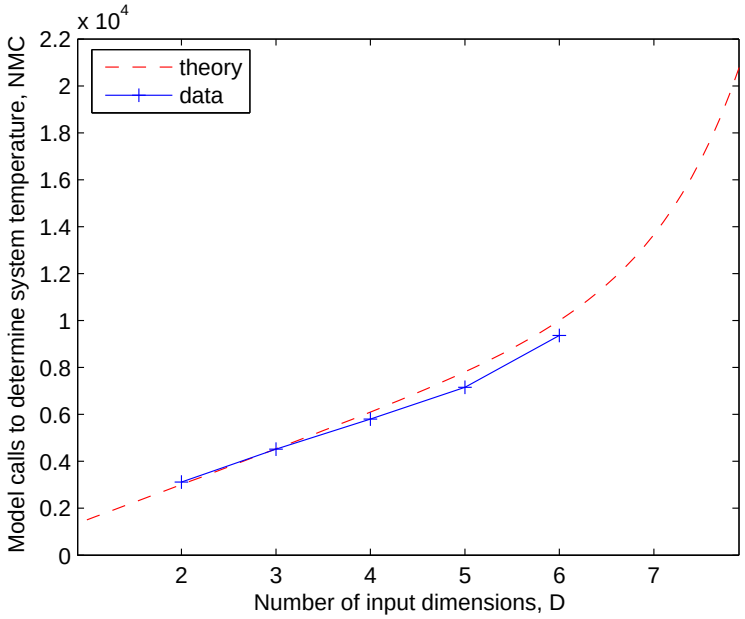


Figure 4.20 The emergent exponential effect of system temperature determination on the combined approach

number of iterations and the number of input dimensions. However, for a system with D input dimensions where each dimension has s regularly spaced samples, the number of model calls NMC required to determine system temperature is exponential in nature (see Equation (4.1)). Because the ABS braking model has a low dimensionality, the exponential effect is minimal, but as Figure 4.20 shows, the exponential nature of the temperature determination algorithm is emergent and will eventually dominate.

$$NMC = s^d \tag{4.1}$$

In Figure 4.20, the recorded data is plotted along with a curve generated using the formula shown in Equation (4.2), where $s = 10$. Equation (4.2) represents the total number of model calls for the simulated annealing phase and includes two terms. The first term is the contribution of the walkers over 1500 iterations and the second term is the contribution due to the system temperature determination algorithm.

$$NMC = 1500D + \sqrt{10^D} \tag{4.2}$$

In comparison, the average number of model calls for evolution strategy has a roughly asymptotic shape. Of course, as the number of input dimensions increases, it is logical to assume that the average number of model calls will approach the maximum number of model calls allowable using evolution strategy. Furthermore, the evolution strategy's data point to the fact that it is superior to the combined strategy in terms of model calls required to optimize the ABS braking model.

The next metric by which the two algorithms were compared was the accuracy of the optimal value produced during each of the 100 runs. Because the output value was real-valued and also because of floating-point concerns, the determining factor for a successful optimization was agreement with the true optimal value to within

0.05 percent. For example, the two-input ABS braking model optimal stopping time was 13.5107, so any runs whose stopping time was 13.5175 or less was considered to have successfully optimized the model.

As can be seen for the two-input results shown in Figure 4.19, both algorithms produced a large number of successful runs. Surprisingly, the combined approach outperformed the evolution strategy algorithm. The combined approach succeeded 95 percent of the time while the evolution strategy algorithm produced a success rate of 81 percent. Also of note was the fact that evolution strategy produced the most optimal stopping time of 13.5107 seconds while the combined approach was slighly behind with a stopping time of 13.5108 seconds.

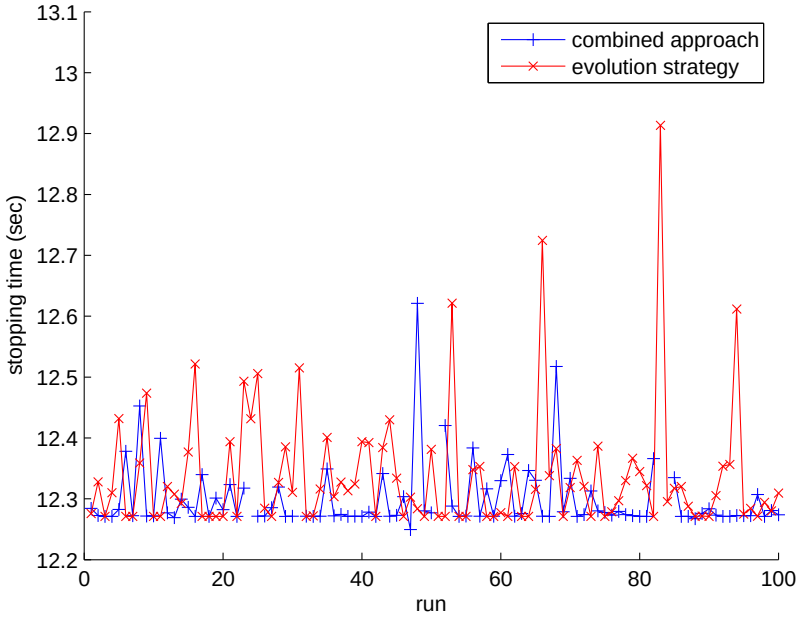


Figure 4.21 Optimal stopping times for the ABS model with six input dimensions

The six-input results shown in Figure 4.21 were even more striking. According to the data, the combined approach succeeded in 55 percent of the runs while the

evolution strategy algorithm produced a meager 35 percent success rate. Also notable for this dataset is the fact that the combined approach produced an unexpectedly optimal stopping time of 12.2494 seconds. This appeared to be needle-in-the-haystack behavior, as none of the other optimization runs for either algorithm produced such a short stopping time.

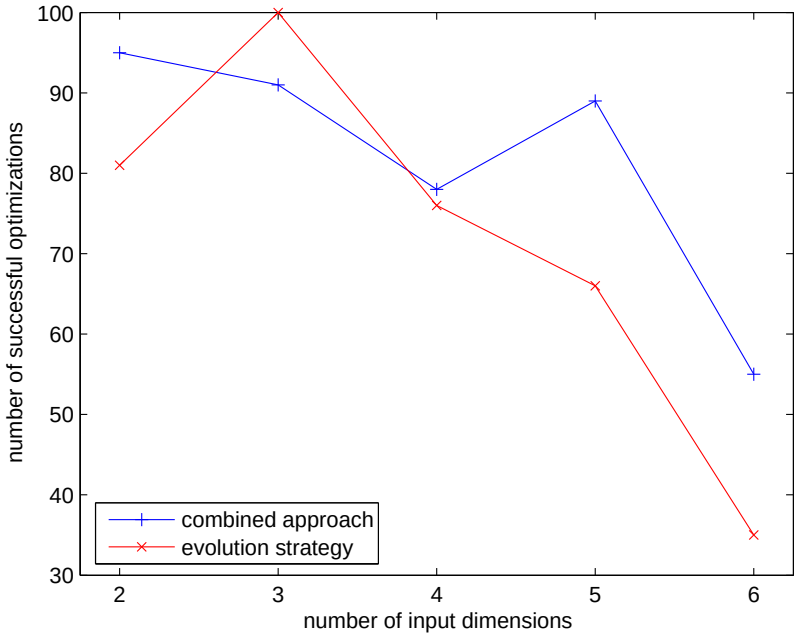


Figure 4.22 Number of successful optimizations for all ABS model configurations

Overall, the combined approach appeared to be more effective in discovering the optimal stopping time than the evolution strategy algorithm (see Figure 4.22). However, the disparity in the number of model calls between the two optimization strategies required another metric to be examined. Figure 4.23 shows the ratio of the average number of model calls to the number of successful optimizations for both strategies. For this metric lower values are more desirable, because an algorithm that produces a lower value implies that it makes more efficient use of each model call than does

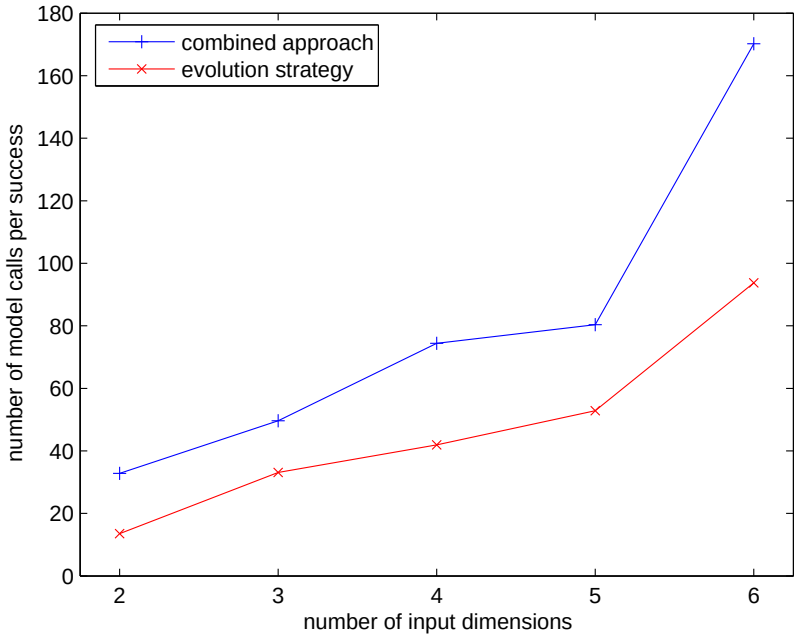


Figure 4.23 Ratio of the average number of model calls to successful optimizations for all ABS braking model configurations

an algorithm with a higher ratio. According to Figure 4.23, the evolution strategy algorithm is more effective than the combined approach.

Number of input dimensions	2	3	4	5	6
Maximum number of model calls	1096	3309	3185	3486	3281

Table 4.5 Limitations placed on the simulated annealing phase's number of model calls

To further explore this idea, a second set of runs was taken for the combined approach. For this new dataset, the combined approach had its simulated annealing phase limited to calling the model only as many times as the evolution strategy algorithm did on average. The maximum numbers of model calls for each configuration are shown in Table 4.5. Note that although the simulated annealing phase was constrained, the cooling schedule and system temperatures were adjusted so that in every case a full optimization was performed.

Once the algorithms were balanced by equalizing the number of model calls, a strikingly different picture emerged. The average number of model calls shown in Figure 4.24 reflects the balance between the algorithms. The disparity between the two algorithms for the six-input ABS braking model was due to the emerging exponential behavior of the system temperature determination algorithm.

Because the simulated annealing phase went through fewer iterations, less exploration of the input geometry was possible. As the system temperature cooled and the exploration transformed into exploitation, the incomplete exploration had a negative impact. As can be seen in Figure 4.25, the evolution strategy algorithm has become the dominant algorithm. Thus, when the algorithms are balanced, the evolution strategy algorithm will be more likely than the combined approach to produce the optimal value.

Returning finally to the ratio of number of model calls to the number of successful

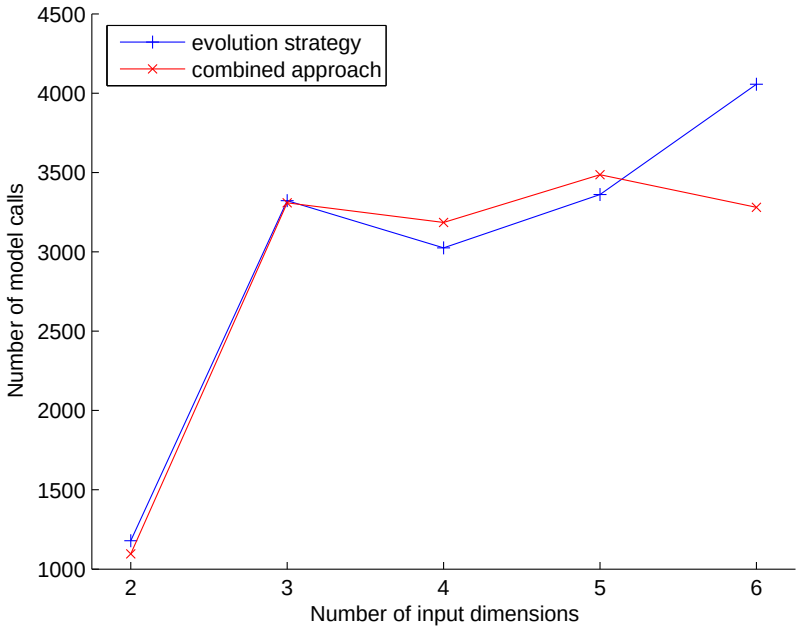


Figure 4.24 Average number of model calls to optimize all ABS model configurations, balanced algorithms

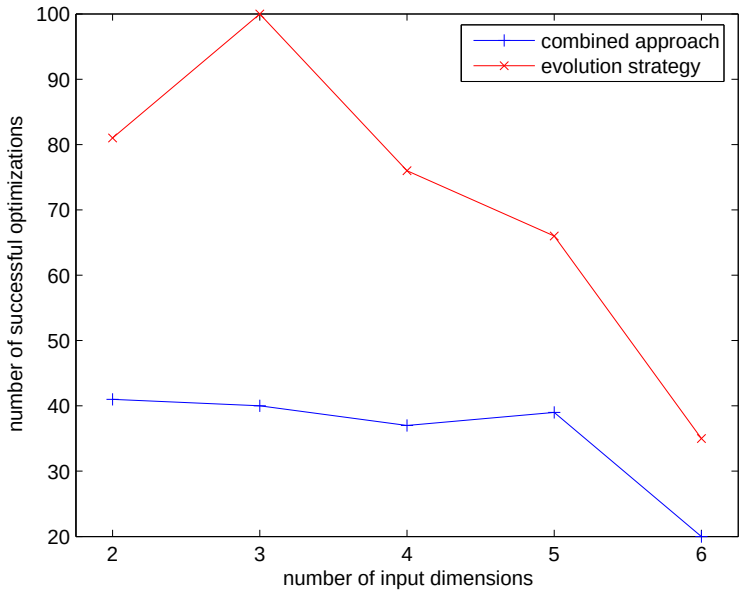


Figure 4.25 Number of successful optimizations for all ABS model configurations, balanced algorithms

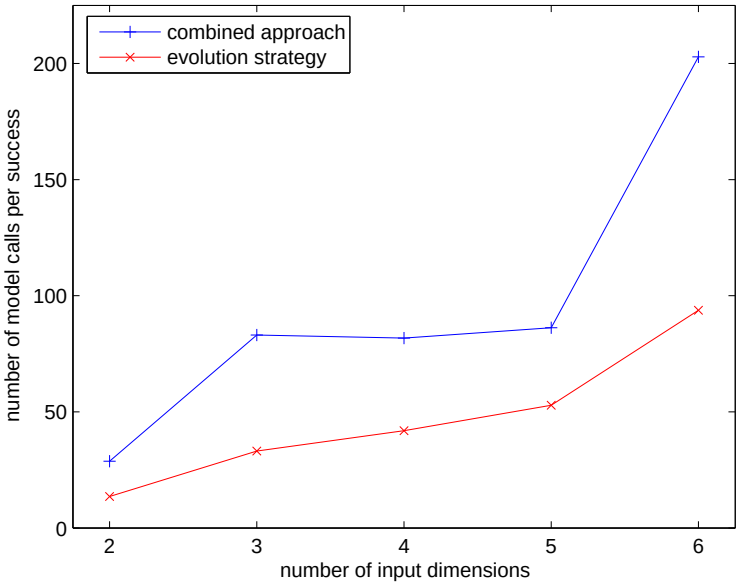


Figure 4.26 Ratio of the average number of model calls to successful optimizations for all ABS braking model configurations, balanced algorithms

runs, Figure 4.26 shows that the evolution strategy algorithm is again superior to the combined approach. Interestingly, despite the severe handicap thrust upon the combined approach, it produced results quite similar to those found in Figure 4.23.

Because each experiment involved repeated optimization runs, the results of the 100 optimizations could be used to calculate confidence intervals for the algorithms. The optimized output values from each run were used as the sample population. For each confidence interval analysis, the level of confidence was allowed to take values ranging between 95 percent and 100 percent and the corresponding error tolerance was calculated using Equation (4.3), where s was the standard deviation of the sample population, n as the size of the sample population, and $z_{\alpha/2}$ was determined using a table of normal probabilities found in [Billingsley and Huntsberger, 1986].

$$ET = \frac{z_{\alpha/2} \cdot s}{\sqrt{n}} \tag{4.3}$$

Figure 4.27 compares the confidence interval analysis results for the optimization of the two-input model. The figure shows that the two algorithms produced similar confidence intervals, with the combined approach having a smaller interval. In fact, for two input dimensions, the combined approach's confidence interval was 49.28 percent smaller than the evolution strategy's confidence interval. The algorithms traded positions for the six-input model, as shown in Figure 4.28. One difference for the six-input model was that the evolution strategy's confidence interval was only 29.87 percent smaller than the combined approach's confidence interval.

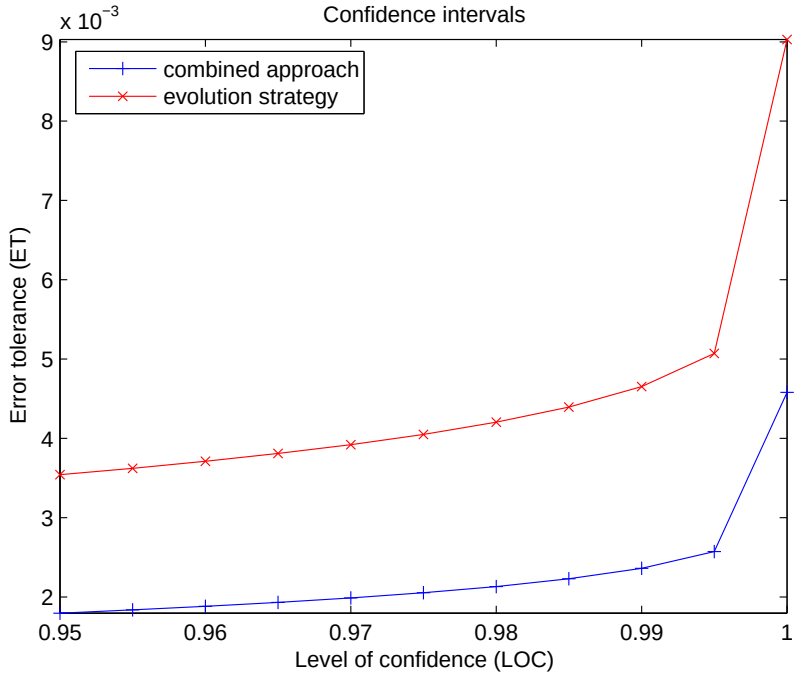


Figure 4.27 Comparing confidence interval data for the two-input ABS model

In terms of error tolerance, neither of the algorithms was the clear winner (see

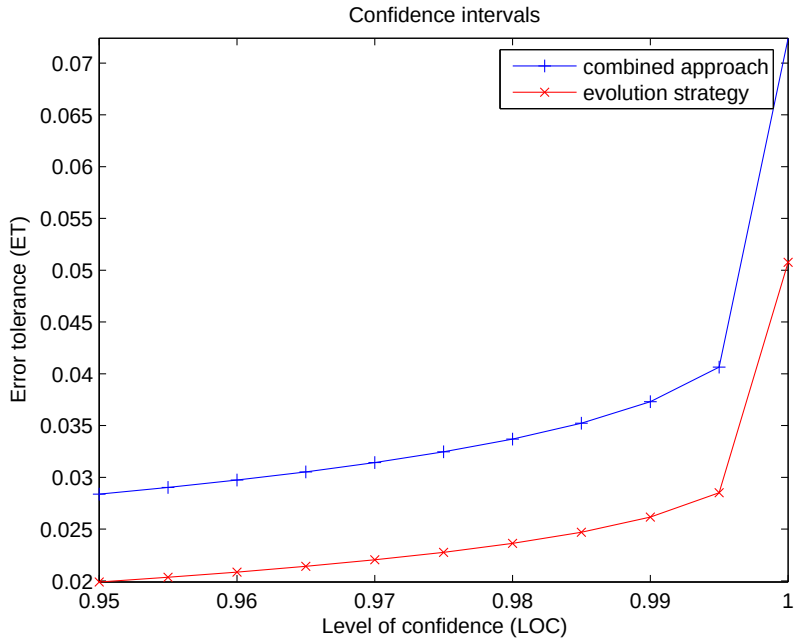


Figure 4.28 Comparing confidence interval data for the six-input ABS model

Number of input dims	Combined Approach	Evolution Strategy
2	0.0018	0.0035
3	0.0207	0.0001
4	0.0036	0.0073
5	0.0034	0.0080
6	0.0283	0.0199
mean (μ)	0.0116	0.0078
std dev (σ)	0.0121	0.0075

Table 4.6 Comparison of error tolerances at 95 percent level of confidence

Table 4.6). In three out of the five model configurations, the combined approach outperformed the evolution strategy algorithm. However, the evolution strategy algorithm had a lower mean and standard deviation over all the model configurations than the combined approach. Furthermore, the combined approach had the largest overall error tolerance (0.0283 for the six-input model), while the evolution strategy had the lowest overall error tolerance (0.0001 for the three-input model).

One last comparison between the two algorithms involved their execution times. Figure 4.29 shows the result of dividing the average algorithm runtime by the average number of model calls for the two algorithms. For example, the two-input combined approach experiment produced an average execution time of 132.8329 seconds and an average of 1178.5 model calls. Dividing these two numbers yields an average of 0.1127 seconds per model call.

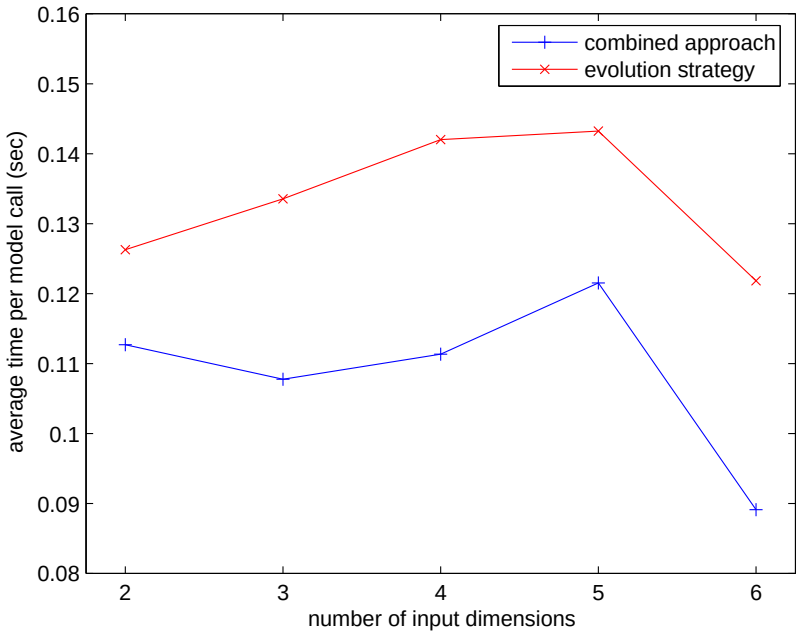


Figure 4.29 Comparing average execution time per model call for all model configurations

As can be seen from Figure 4.29, the evolution strategy algorithm was slightly slower than the combined approach. This can be explained by the fact that the evolution strategy algorithm required more complex processing of the population than did the combined approach. At the end of each the evolution strategy's generations, the children were sorted in order of their fitnesses to determine the survivors. There was no such sorting in the combined approach, thus allowing the combined approach to run slightly faster than the evolution strategy algorithm.

CHAPTER 5 CONCLUSIONS AND FUTURE DIRECTIONS

5.1 Conclusions

It is clear from the analysis of the experimental data that in the case of optimization of a low dimensionality SIMULINK model the evolution strategy algorithm was superior to the combined approach. Not only did the evolution strategy algorithm require less model calls on average than the combined approach, it also made more effective use of each model call. In addition, although initial data pointed to the combined approach's superiority in optimal value determination, after balancing the algorithms it was seen that it was the evolution strategy algorithm that had the better success rate. The confidence interval data also pointed to the evolution strategy's superiority.

In fact, the only metric that favored the combined approach was the average execution time per model call data. However, it should be noted that neither of the algorithms presented were code-optimized. In other words, both algorithms were basic implementations that contained no specialized code to reduce runtime or processor load. Such an optimization of the source code would likely have benefited the evolution strategy algorithm greatly, especially for the sorting routines of the survivor pool creation.

There are other qualitative considerations that further bolster case for the evolution

strategy's superiority. Whereas the simulated annealing algorithm is rather abstract and difficult to grasp, the idea behind evolution strategy is quite simple. Assuming an expert were given a choice between the two algorithms, it is likely they will choose evolution strategy simply on the basis of their understanding of the algorithms.

In further support of this conclusion, consider the number of tunable parameters contained in an evolution strategy algorithm. Selection pressure, population size, number of generations, and convergence threshold are the parameters that have the largest effect on the algorithm, and all are comprehensible with little explanation. Contrast that with the many abstract tunable parameters in simulated annealing, such as the cooling schedule, upper and lower system temperature thresholds, and initial and minimum radius of movement values.

Clearly, then, evolution strategy is quantitatively and qualitatively superior to the combined approach. Unfortunately, its success rate was disappointingly low. Further tuning of the evolution strategy algorithm would be required to increase the success rate, but this would defeat its general-purpose construction. Future work will need to address this shortcoming if the evolution strategy algorithm is ever to see commercial-grade implementation.

5.2 Future Directions

Certainly a valid avenue of future research would be the extension of the evolution strategy algorithm to higher dimensionality SIMULINK models. A necessary part of such research would be the discovery of a mechanism to overcome the algorithm's observed lackluster performance for higher dimensionality ABS braking model configurations. Possible mechanisms might be a higher selection pressure, a larger population, or even more generations before termination. In addition, a more

stable interface between the algorithm and the SIMULINK model would need to be constructed. This would not be a trivial implementation, as many SIMULINK models initialize their blocks and variables with internal scripts.

The algorithm presented in this work employed uncorrelated mutation, but evolution strategy is capable of incorporating correlated mutation. Correlated mutation would further enhance the population members' ability to explore the input space by allowing the standard normal distribution to rotate off-axis. This rotation would have an effect similar to the simplex's ability to adapt to suit the input space geometry.

Considering the reason behind the creation of the combined approach, another interesting area of research would be the independent optimization of a SIMULINK model using several different algorithms. The independent optimization results would then be combined and analyzed to produce more accurate results. Many different models could be optimized using this approach, because although no single algorithm would be efficient for every model, combining the strengths of various algorithms would overcome their weaknesses. In example, a SIMULINK model could be independently optimized using both simulated annealing and evolution strategy. By comparing and contrasting the results of each optimization, a more confident conclusion might be reached.

BIBLIOGRAPHY

Mordecai Avriel. *Nonlinear Programming.* Dover Publications, 2003.

H.-G. Beyer and H.-P. Schwefel. Evolution strategies: A comprehensive introduction. *Natural Computing*, 1(1):3–52, May 2002.

Patrick Billingsley and David V. Huntsberger. *Statistical Inference for Management and Economics*, pages 400–401. Allyn & Bacon, Boston, second edition, 1986.

Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing (Natural Computing Series)*, chapter 4, pages 71–87. Springer, Berlin, 2003.

Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5–6):975–986, March 1984.

Scott Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

Averill M. Law and David M. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill Higher Education, 3rd edition, 1999.

John A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, January 1965.

Christian Robert and George Casella. *Monte Carlo Statistical Methods.* Springer, Berlin, second edition, 2004.

D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, The Santa Fe Institute, Santa Fe, N.M., 1995.