William & Mary
CHARTERED 1693

W&M ScholarWorks

Undergraduate Honors Theses

Theses, Dissertations, & Master Projects

2000

Simplex Search Behavior in Nonlinear Optimization

Adam P. Gurson
College of William and Mary

Follow this and additional works at: https://scholarworks.wm.edu/honorstheses

Recommended Citation

Gurson, Adam P., "Simplex Search Behavior in Nonlinear Optimization" (2000). *Undergraduate Honors Theses.* Paper 449.
https://scholarworks.wm.edu/honorstheses/449

This Honors Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Simplex Search Behavior in Nonlinear Optimization

A thesis submitted in partial fulfillment of the requirements for a
Bachelor of Science with Honors in Computer Science
from the College of William & Mary in Virginia,

by
Adam P. Gurson

Accepted for Highest Honors

Thesis Advisor: Virginia J. Torczon

Weizhen Mao

Michael W. Trosset

Abstract

It is often a desire in many fields such as mathematics, physics, and engineering to solve bound constrained minimization problems. Non-derivative based direct search methods each use a specific method of function sampling in an attempt to home in on the minimizers of a given function. Here we focus on three simplex based direct search methods: the original simplex search provided by the description of Spendley, Hext and Himsworth; a variation on its theme provided by Nelder and Mead; and a sequential version of Torczon's multidirectional search. It is a common assumption that these searches are easily implemented and used. This research addresses this claim and suggests that these searches are more intricate and should be implemented and used in a careful fashion.

We first provide a formal description of each algorithm using a common notation, providing a means of direct comparison between algorithms. We then discuss the importance of resolving seemingly small ambiguities in these algorithms before using them for optimization. For this, we provide an anomaly specific to the Nelder-Mead algorithm in which the search fails to converge to a constrained stationary point. We conclude with some preliminary results of execution of these algorithms on a specific set of objective functions.

Acknowledgments

This research would not have been possible without the many generous contributions I received from various organizations[1] and other special individuals. First, I have to greatly acknowledge my advisory panel: Dr. Virginia Torczon, Dr. Weizhen Mao, and Dr. Michael Trosset for all of their guidance. I would like to thank Dr. Trosset for providing me with a clear focus for my research. He helped a great deal to give my thesis some well-defined direction. I would also like to thank Dr. Mao, especially for her thoughtful suggestions and all of the time she dedicated to me during her year on sabbatical. I would like to acknowledge all of my friends who kept me focused and never let me drop the ball. Chris Siefert, I can never express how much I appreciate all of your time and effort in guiding me through more LaTeX, shell scripting, Splus, and plot generation than I ever thought possible. Anne Shepard, thank you for agreeing to proof read my thesis on numerous occasions. The oatmeal-raisin cookies were wonderful, and I am glad you liked the pictures. Liz Dolan, thank you for providing me with the inspiration to attempt research of this kind. I need a special thank you to Karen Phillips and Alex Burke, two very special people whose company was often superseded by my work. Thank you for understanding. Finally, I have to thank Dr. Torczon for being not only an amazing thesis advisor, but also an amazing friend. Dr. Torczon, I have enjoyed this year more than I can express in words. Your support has carried me from beginning to end, and I never would have accomplished so much without you.

[1]This research was supported by NSF Grant CCR-9734044.

Contents

1 Introduction **10**

2 The Search Algorithms **13**

2.1 The Simplex Search of Spendley, Hext and Himsworth 13

2.1.1 General Description 14

2.1.2 Formal Description 15

2.1.3 Ambiguities 16

2.2 The Simplex Search of Nelder and Mead 23

2.2.1 General Description 23

2.2.2 Formal Description 25

2.2.3 Subtleties 26

2.3 Sequential Multi-Directional Search 26

2.3.1 General Description 26

2.3.2 Formal Description 28

2.3.3 From Multi-Processor to Single-Processor 29

3 The Importance of Resolving Ambiguities in the Nelder-Mead Search Algorithm **31**

3.1 The Discovery of the Problem 31

3.2 The Cause 33

3.3 The Solution 34

3.4 Discussion 37

4 Testing and Results **41**

4.1 The Testing Setup 41

4.1.1 The Objective Functions 41

4.1.2 Description of the Tests 43

4.1.3 The Function Minima Density Plots 44

4.2 General Comparison of the Search Methods 46

4.3 Visual Results of Nelder-Mead Improvement 51

5 Conclusion **53**

A The C++ Code **55**

A.1 C++ Code for the Simplex Search of Spendley, Hext, and Himsworth 57

A.1.1 SHH - Header File 57

A.1.2 SHH - ExploratoryMoves() 62
A.1.3 SHH - Constructors and Destructor 62
A.1.4 SHH - Simplex Initialization Routines 64
A.1.5 SHH - Other Unique Functions 66
A.2 C++ Code for the Simplex Search of Nelder and Mead 70
A.2.1 NM - Header File 70
A.2.2 NM - ExploratoryMoves() 75
A.2.3 NM - Constructors and Destructor 77
A.2.4 NM - Simplex Initialization Routines 79
A.2.5 NM - Other Unique Functions 81
A.3 C++ Code for the Sequential Multi-Directional Search 86
A.3.1 SMDS - Header File 86
A.3.2 SMDS - ExploratoryMoves() 91
A.3.3 SMDS - Constructors and Destructor 92
A.3.4 SMDS - Simplex Initialization Routines 94
A.3.5 SMDS - Other Unique Functions 96
A.4 Functions Common to All Three Searches 101

List of Figures

2.1 A labeling example for $\mathbb{R}^2$ such that $f(x_0) \leq f(x_1) \leq f(x_2)$. 14

2.2 In this reflection step, vertex x_2 has the highest function value (left), so we reflect it through $\bar{x}$, the centroid of x_0 and x_1, to a new vertex x_r (center). The result is a new nondegenerate simplex consisting of x_0, x_1 and x_r (right). 15

2.3 If the true minimizer of the function, x_{min}, lies with the current simplex (left), shrinking the current simplex around x_0 (right) results in a smaller simplex, which refines the search. 17

2.4 A sequence of reflections $\{x_r^1, x_r^2, x_r^3, x_r^4, x_r^5\}$, each of which fails to replace the best vertex x_0, which brings the search back to the simplex from which this sequence started. 18

2.5 a) The initial simplex vertices and their respective ages, initialized to 1. b) If the reflection point is *not* a new minimum (i.e. $f(x_r) \geq f(x_0)$), simply increment the ages of the remaining simplex vertices. c) If the reflection point *is* a new minimum (i.e. $f(x_r) < f(x_0)$), all simplex vertices begin the next iteration with their ages set to 1. 19

2.6 a) In this example, reflecting x_2^k to x_r^k makes x_r^k the new maximum. b) On the next iteration, reflecting x_2^{k+1} through $\bar{x}^{k+1}$ yields the original simplex. 20

2.7 In an evolution from left to right, this example shows how oscillation without correction could permit the simplex to contract to a point that is not the true minimizer. 21

2.8 This is an example of label swapping to prevent simplex oscillation. This will resolve the problem encountered in Figure 2.7. a) On the kth iteration, the reflection step is taken to yield x_r^k. b) The $(k+1)$th iteration begins by labeling the points such that $f(x_0^{k+1}) < f(x_1^{k+1}) < f(x_2^{k+1})$. c) Because $x_2^{k+1} \equiv x_r^k$, we swap the labels on x_1^{k+1} and x_2^{k+1}. d) Having made this change, the reflection step in the $(k+1)$th iteration yields a new point $x_r^{k+1} \not\equiv x_2^k$, and oscillation is no longer an issue. The simplex will continue to move as in Figure 2.4, allowing it to search in as many directions as possible, preventing collapse to a false minimizer. 22

2.9 a) The *reflection step* of the Nelder-Mead algorithm is similar to that of Spendley, Hext and Himsworth. b) The *expansion step* further reflects x_2 through $\bar{x}$ to yield the expansion point, x_e. c) If $f(x_2) \leq f(x_r)$, perform an *inside contraction* from $\bar{x}$ toward x_2 to find x_{ic}. d) If $f(x_2) > f(x_r)$, perform an *outside contraction* from $\bar{x}$ toward x_r to find x_{oc}. 24
2.10 Begin by labeling the basepoint x_0 and arbitrarily labeling the other points. Calculate only $f(x_0)$. 27
2.11 A reflection of the primary simplex through x_0 yields the reflection simplex. 27
2.12 If all points in both simplices have been evaluated without improvement, shrink the primary simplex around x_0. 28
2.13 a) $f(x^k_{r_2}) < f(x^k_0)$, therefore $x^k_{r_2}$ will be the new basepoint for the next iteration (i.e. $x^k_{r_2} \equiv x^{k+1}_0$). b) We begin the $(k+1)$th iteration by evaluating $f(x^{k+1}_{r_2})$ first because the step from x^k_0 (which is equivalent to x^{k+1}_2) to x^{k+1}_0 yielded improvement, so we continue in the same direction. 30

3.1 Our 2-dimensional objective function. For a given point (x, y), x runs down along the lower left axis and y runs up along the lower right axis. 32
3.2 This figure shows one iteration of the algorithm which leads to a collapse of the simplex. a) The simplex begins with only one feasible point, x_0. b) The reflection step is taken, however it is such that x_r is also infeasible. c) Having a failed reflection, the algorithm makes an outside contraction to find x_{oc}, which is also infeasible, yet accepted, which terminates the iteration. This process is continued to yield a series of infeasible outside contraction vertices, which is shown in Figure 3.3. 35
3.3 The collapse of the simplex {1., 2., 3., 4., 5., 6.} toward the line defined by x_0 and x_1. Note how as the process continues, x_2 gets closer to the centroid of x_0 and x_1. 36

4.1 Two krigifier objective functions with the similar input parameters. Above: A function with a constant trend. Below: A function with a quadratic trend. 42
4.2 Three examples of density plots. 45
4.3 The lower two density plots show the advantage of the $(n+1)$ shrinking criterion over the $2(n+1)$ criterion for the Spendley, Hext and Himsworth algorithm. 46
4.4 An example similar to Figure 4.3, but for a different objective function. 47
4.5 These density plots representing each algorithm suggest that the pattern searches show more satisfactory results than the simplex searches. 48
4.6 Density plots representing each algorithm on a 3-dimensional function given a budget of 20. 49

4.7 Density plot representing the same objective function as Figure 4.6, but with a budget of 50. 50
4.8 These plots show the clear advantage of the Nelder-Mead implementation based on Lagarias, Reeds, Wright, and Wright. 51
4.9 Another example of the clear advantage of the Nelder-Mead implementation based on Lagarias, Reeds, Wright, and Wright, over that described by Avriel. 52

List of Tables

3.1 The twelve unusually high putative minimizers returned by the Nelder Mead search. 33
3.2 The starting and finishing simplices and the final vertex function values for the twelve unique test cases. 39
3.3 The twelve minimizers returned by the Lagarias, Reeds, Wright, and Wright algorithm. 40

Chapter 1

Introduction

It is often a desire in many fields such as mathematics, physics, and engineering to solve the bound constrained minimization problem:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & l \leq x \leq u \end{array} \tag{1.1}$$

where $f\colon \mathbb{R}^n \to \mathbb{R}$, $l, x, u, \in \mathbb{R}^n$, and $l < u$. It is also possible to allow for the possibility of certain variables to be unconstrained by setting $l, u = \pm\infty$. For instance, during the development of a new type of machinery, there can be many variables, such as angles, rotational speeds, and temperature which can be combined to create a function f representing the tendency for the given machinery to shake in an unstable fashion. In an effort to minimize this shaking tendency, the designers will attempt to look for a combination of those variables which minimize f as much as possible.

When dealing with these types of problems, the most likely first inclination is to attempt minimization via calculus. However, it is often the case in the physical world that the functions represented by f are quite complicated, with many variables. A simple function evaluation at one point is often an expensive process. With this being the case, derivatives are most likely difficult, if not impossible, to calculate in a form suitable for calculus. It may therefore be preferable to use one of a series of non-derivative-based search methods.

These non-derivative-based search methods each use a specific method of function sampling in an attempt to hone in on the minimizers of a given function. For a physical representation of the task at hand, consider the following two dimensional analogy due to J. E. Dennis [13]. A person is placed in a boat on a lake and given only a very long string with a weight tied to the end of it. The person cannot see the bottom of the lake, and there are no currents in the water. The boat is free to move about the surface of the lake. Using only this equipment, the person must attempt to find the point on the surface of the lake at which the lake is deepest. This is to be done with a series of depth samples made by dropping the string into the lake.

There are many ways in which this person can move the boat around. Each of these separate methods defines a specific non-derivative-based search. Many of these searches are heuristic based methods whose descriptions were first published in the 1960's. A more recent need to solve these types of problems using derivative-free

methods has brought about renewed interest in the subject. While there may be analysis available for individual search algorithms, there has been little in the way of direct comparison of these algorithms, the theory behind them, and their individual behavior and success for a given set of problems.

In this research, we focus on a specific subset of the direct search methods, which we will refer to as simplex-based methods. Our three chosen searches are the original simplex-based search described by Spendley, Hext and Himsworth [17]; a variation on its theme provided by Nelder and Mead [8]; and a sequential version of Torczon's Multi Directional search [18]. This research is complementary to that of Dolan [4], who focused primarily on pattern search methods, another subset of direct search methods. We have chosen to examine the simplex methods here because at least one of them [8] is widely used, yet their behavior is still not well understood.

We would like to use this paper as a resource for the clarification, implementation and documentation of these simplex algorithms. Because much of the literature leaves these methods open for interpretation, we start by resolving the ambiguities in these algorithms to produce clearly defined search techniques. This will allow for implementation of these algorithms into C++ software that permits easy general use and availability. Also, we will directly compare and contrast the heuristics underlying the simplex search methods as well as their general performance statistics.

While the simplex search algorithms are quite popular, we are not aware of a single location in which computer implementations can be obtained for all of them. Therefore, one of our primary goals is to implement three of the simplex-based searches using C++ classes. With these and the work of Dolan, we can then create a software repository suitable for their distribution. Our complete implementation can be found in Appendix A.

For this to be achieved, we must first modernize and disambiguate the Spendley, Hext and Himsworth description, which was first introduced in 1962 and is not entirely suitable for numerical optimization techniques. Also, thought must be given to determine a good and efficient way to transform Torczon's Multi Directional Search, an algorithm originally intended solely for use on parallel processor machines, into a sequential search algorithm suitable for single processor machines. Our discussion can be found in Chapter 2.

Once this has been achieved, we will analyze and compare the behavior of the three simplex-based algorithms for problems of the form given in (1.1). As was mentioned above, evaluating a given objective function f at one point can often be a rather expensive and time-consuming process. It is therefore desirable to find a search algorithm which not only achieves sufficient resolution on the location of the minimizer, but also locates the minimizer in as few function calls as possible. We will therefore focus primarily on the algorithms' overall efficiency and their success near constraints of a bounded objective function f. By efficiency, we mean the ability of the algorithms to quickly locate the region of a true function minimizer. During the course of our testing, a unique anomaly associated with the Nelder-Mead algorithm was discovered. This anomaly is illustrated and is discussed in Chapter 3. A general discussion of the results from our testing can be found in Chapter 4.

Thus our goal is to evaluate and better understand the behavior of the simplex-

based algorithms and make it easier for others to experiment with each of these alternatives to discover which is most effective in practice.

Chapter 2

The Search Algorithms

The algorithms used for this research all come from a class of nonlinear optimization techniques known as simplex searches. Each of the algorithms uses a nondegenerate simplex as its design for function sampling. By a nondegenerate simplex, we mean a set of $n+1$ vertices in $\mathbb{R}^n$ that has the property that the set of simplex edges adjacent to any given vertex spans $\mathbb{R}^n$. A simplex is a line in $\mathbb{R}^1$, a triangle in $\mathbb{R}^2$, a tetrahedron in $\mathbb{R}^3$, and so on.

The three simplex search algorithms addressed herein are those of Spendley, Hext and Himsworth [17]; Nelder and Mead [8]; and a sequential version of Torczon's multi-directional search [18]. While each has its own specific rules for simplex manipulation, they all roughly follow the same basic heuristic. Each begins with a nondegenerate simplex in $\mathbb{R}^n$. The function is then evaluated at the vertices of the simplex. As the algorithms proceed, they continually label a point x_0 whose function value $f(x_0)$ is better than any of the other vertices evaluated thus far. For each iteration of the algorithm, they attempt to replace simplex vertices that yield high function values with new vertices whose new function values are lower than $f(x_0)$.

After each iteration, the vertices that replaced those in the starting simplex leave us with a new simplex. If we label a given iteration with the letter k, and let x_0^k be the best point currently found by iteration k, we can state the following: It is the goal of these algorithms to yield a sequence $\{x_0^k\}$ such that

$$f(x_0^{k+1}) \leq f(x_0^k). \tag{2.1}$$

In other words, once we have a given x_0, we do *not* replace it until we find another point whose function value is lower than $f(x_0)$.

The goal of all three algorithms is to move the simplex, by replacing vertices, into the neighborhood of a minimizer.

2.1 The Simplex Search of Spendley, Hext and Himsworth

To the best of our knowledge, the simplex search method of Spendley, Hext and Himsworth is the original nonlinear simplex search. Aside from being the earliest

of the three search algorithms considered here, it has a minimal number of rules for simplex manipulations between iterations, which make it a good algorithm to describe first.

2.1.1 General Description

In this section we will outline the algorithm as originally stated. It is important to note that the original specification was concerned with evolutionary operation, not numerical optimization. It is therefore incomplete by modern standards for use as an optimization method; there are ambiguities, which we will address in more detail in section 2.1.3. While Spendley, Hext, and Himsworth noted these ambiguities in their original paper [17] and hinted at resolutions similar to those we suggest in 2.1.3, we present solutions that are designed to be more rigorous for the purposes of optimization.

First, begin with a nondegenerate simplex of $n+1$ vertices for $\mathbb{R}^n$. Each new iteration of the algorithm begins by labeling the vertices of the simplex

$$x_0, x_1, ..., x_n \tag{2.2}$$

such that

$$f(x_0) \leq f(x_1) \leq ... \leq f(x_n). \tag{2.3}$$

See Figure 2.1.

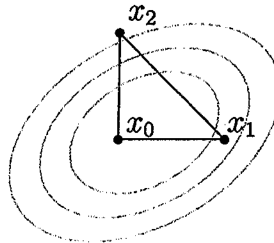


Figure 2.1: A labeling example for $\mathbb{R}^2$ such that $f(x_0) \leq f(x_1) \leq f(x_2)$.

We now know that x_n has the highest function value $f(x_n)$ of all simplex vertices. It is therefore desirable to replace x_n with a new point whose function value is lower than $f(x_n)$. This is done by reflecting x_n through the centroid of the remaining n vertices, effectively flipping the simplex away from the area of highest function value. This reflection step always has the beneficial result of providing yet another nondegenerate simplex, consisting of n of the original vertices and x_n's reflection point (Fig. 2.2). This step of the algorithm conveys the underlying assumption that flipping the simplex away from its highest function value will most likely flip the simplex along the gradient of the function. If we flip the simplex along the gradient, we are likely to uncover new points whose function values are lower than $f(x_n)$ and even possibly lower than $f(x_0)$.

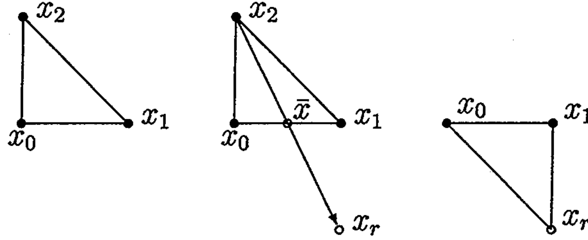


Figure 2.2: In this reflection step, vertex x_2 has the highest function value (left), so we reflect it through $\bar{x}$, the centroid of x_0 and x_1, to a new vertex x_r (center). The result is a new nondegenerate simplex consisting of x_0, x_1 and x_r (right).

The centroid $\bar{x}$ of all vertices *excluding* x_n is calculated using the standard formula:

$$\bar{x} = \frac{1}{n} \sum_{i=0}^{n-1} x_i. \tag{2.4}$$

The final step is the actual reflection of x_n through $\bar{x}$ to obtain a new point x_r. This is simply

$$x_r = 2\bar{x} - x_n. \tag{2.5}$$

From here, x_r replaces x_n in the simplex, and a new iteration is begun. As the algorithm cycles, the function values of the simplex vertices should decrease overall as each consecutive simplex flips away from areas of high function values. It is the hope, although not a guarantee, that this will eventually flip the simplex toward a basin in the function, thereby locating a valid local minimizer.

2.1.2 Formal Description

This section contains our formal description of the Spendley, Hext and Himsworth algorithm. Please note that our version introduces some notation and terminology which has not yet been discussed. These have been included to help resolve certain ambiguities and are discussed in greater detail in section 2.1.3. These include the notion of point age, shrinking criteria, unnecessary simplex oscillation, and criteria for terminating a given execution of the algorithm. The age of a simplex vertex x is designated by the notation $A(x)$, and we will again be using the notation x_i^k to designate a specific instance of a vertex x_i for a given iteration k, where $k \geq 0$, and $i \in \{0, ..., n\}$.

0. **Initialize.** Start with a nondegenerate simplex for $\mathbb{R}^n$ and calculate the function values at all of the vertices. Then at each iteration $k, k \geq 0$:

1. **Order.** Order the vertices $x_0^k, x_1^k, ..., x_{n-1}^k, x_n^k$ such that

$$f(x_0^k) \leq f(x_1^k) \leq ... \leq f(x_{n-1}^k) \leq f(x_n^k). \tag{2.6}$$

If $k > 0$ and $x_n^k \equiv x_r^{k-1}$, swap the labels of x_n^k and x_{n-1}^k to prevent simplex oscillation (described in section 2.1.3).

2. **Check Ages for Shrinking Criterion.** If the current best simplex vertex x_0^k has an age $A(x_0^k)$ such that

$$A(x_0^k) > n + 1, \tag{2.7}$$

then shrink the simplex around x_0^k by replacing x_i^k with

$$\hat{x}_i^k = x_i^k + \frac{1}{2}(x_0^k - x_i^k), \ i = 1, ..., n. \tag{2.8}$$

Replace x_i^k with $\hat{x}_i^k$, for $i = 1, ..., n$. Reset the ages so that $A(x_i^k) = 1$ for all $i = 0, ..., n$. Set $k = k + 1$ and return to Step 1.

3. **Reflect.** If the maximum age has not been exceeded (i.e., $A(x_0^k) \leq n+1$), find the centroid

$$\bar{x}^k = \frac{1}{n} \sum_{i=0}^{n-1} x_i^k, \tag{2.9}$$

and reflect x_n^k through $\bar{x}^k$ to obtain the vertex

$$x_r^k = \bar{x}^k + (\bar{x}^k - x_n^k). \tag{2.10}$$

Replace x_n^k with x_r^k.

4. **Update Ages.** The ages for all simplex vertices are then as follows: First, since x_n^k has been replaced by x_r^k,

$$A(x_n^k) = 1. \tag{2.11}$$

Next, if $f(x_n^k) \geq f(x_0^k)$ (i.e., we do *not* have a replacement for x_0^k), then

$$A(x_i^k) = A(x_i^k) + 1, \ i = 0, ..., n - 1; \tag{2.12}$$

otherwise, $f(x_n^k) < f(x_0^k)$ (i.e., we *do* have a replacement for x_0^k), so

$$A(x_i^k) = 1, \ i = 0, ..., n - 1. \tag{2.13}$$

5. **Check Termination.** Terminate if any of the possible stopping criteria in force are satisfied. Otherwise, set $k = k + 1$ and return to Step 1.

Our complete C++ implementation of this algorithm is available in Appendix A. The heart of the algorithm explained above can be found in the ExploratoryMoves() function, located in section A.1.2. Also, there are some aspects of the algorithm, such as the notion of point age, unnecessary oscillation, and algorithm termination, that were not addressed in section 2.1.1. These will be addressed fully in the next section.

2.1.3 Ambiguities

The general algorithm of section 2.1.1 has some ambiguities which were resolved by methods alluded to in the formal description in section 2.1.2. As mentioned above, Spendley, Hext and Himsworth were aware of these ambiguities and hinted at resolutions. We have expanded upon their initial suggestions to provide the solutions addressed below.

Shrinking and the Concept of Point Age

While the algorithm as described in section 2.1.1 is sufficient for finding the general vicinity of a function minimizer, as stated it does not do well when it comes to narrowing in on the exact location of the minimizer. For example, consider a two-dimensional example for an iteration k in which a true function minimizer, x_{min} lies somewhere inside the triangle formed by the simplex (Fig 2.3 left). If the size of the simplex is not changed for further iterations, at best, the algorithm could only approximate the location of x_{min} with its current value of x_0^k. In this instance, to better refine the search, the algorithm should halve each side of the simplex around x_0 in the following manner: Each simplex vertex x_i^k should be replaced with a new point x_i^{k+1} where

$$x_i^{k+1} = x_i^k + \frac{1}{2}(x_0^k - x_i^k),\ i = 0, ..., n. \tag{2.14}$$

See Figure 2.3 (right).

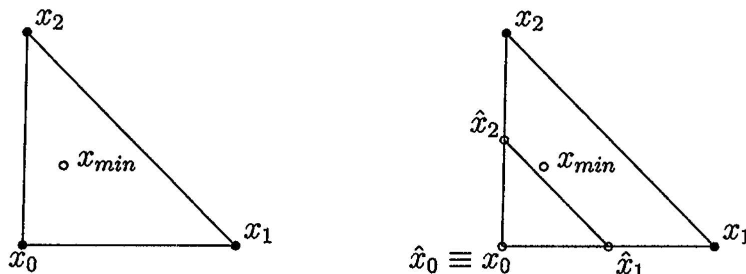


Figure 2.3: If the true minimizer of the function, x_{min}, lies with the current simplex (left), shrinking the current simplex around x_0 (right) results in a smaller simplex, which refines the search.

This concept of shrinking for refinement purposes is quite common and is used in most of the non-derivative-based optimization methods. The real question is how to determine when the simplex actually *is* in the area of a minimizer and should therefore shrink.

To answer the question we return to the two-dimensional example given above (Figure 2.3 left). If the algorithm were to be carried out further without allowing the simplex to shrink, the simplex would begin to revolve around x_0. After six consecutive flips (not allowing the simplex to oscillate between two consecutive simplices), it would be back where it started (Fig. 2.4) Clearly, this is now a case for shrinking.

It might be tempting at this point to assume that a solution would be to keep track of the simplices and shrink whenever the current simplex equals a simplex that was previously encountered. This is not plausible for two reasons. First, while there are only six simplices in this example, going to higher dimensions would quickly increase this number. From a mere performance standpoint, a coded implementation of this algorithm would require unnecessarily large amounts of memory to store enough simplices. Second, even if memory size is not a consideration, we run into another problem in cases where $n > 2$: although in $\mathbb{R}^2$ all possible simplices for this algorithm lie on a well-defined lattice and therefore can loop back on themselves, this is *not* true in higher dimensions. We therefore need to find another shrinking criterion.

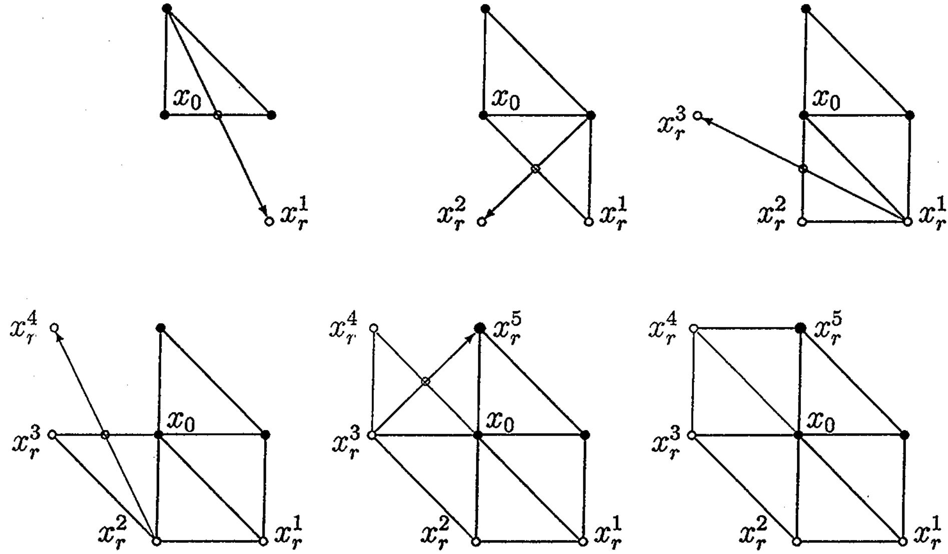


Figure 2.4: A sequence of reflections $\{x_r^1, x_r^2, x_r^3, x_r^4, x_r^5\}$, each of which fails to replace the best vertex x_0, which brings the search back to the simplex from which this sequence started.

Looking again at the example, we note that another sign of a need for shrinking is that one vertex, more specifically x_0, remains in the simplex for a significantly long time. This is a general feature of simplex searches: the simplex should *not* reflect the vertex labeled x_0, even after exhausting all possible search directions. Therefore, if we can spot a simplex vertex that has not been replaced for a while, we can define an absolute shrinking criterion. This requirement naturally introduces the need for point age.

The rough definition of point age is how many times a given point has been a vertex of any simplex thus far. The rules for determining age are straightforward. Before the first iteration, all simplex vertices have age 1. During each iteration, x_n is replaced with a new point, whose age is set to 1. All other points have their age incremented by 1. If the new point has a function value less than x_0 (i.e. it is the new best point), then reset the age of all simplex vertices to 1 (This has the value of waiting for the simplex to establish a true "orbit" around a minimizer. If new points with smaller function values are still being found, the simplex is probably not yet near a function minimizer). Finally, after shrinking a simplex, reset the age of all simplex vertices to 1.

For example, consider a simplex in $\mathbb{R}^2$ with vertices x_0, x_1, x_2 such that $f(x_0) < f(x_1) < f(x_2)$. Let $A(x_i)$ represent the age of a vertex x_i. Initially,

$$A(x_0) = A(x_1) = A(x_2) = 1. \tag{2.15}$$

See Figure 2.5 a. Now because $f(x_2)$ is the highest function value so far, we reflect x_2 and replace it with a new point x_r. $A(x_r)$ is set to 1, and $A(x_0)$ and $A(x_1)$ are set as follows (Fig. 2.5 b):

$$A(x_0) = A(x_1) = 2 \text{ if } f(x_r) \geq f(x_0), \;\; (x_r \text{ is } \textit{not} \text{ a new minimum}) \tag{2.16}$$

or (Fig. 2.5 c)

$$A(x_0) = A(x_1) = 1 \text{ if } f(x_r) < f(x_0). \text{ } (x_r \textit{ is} \text{ a new minimum}) \tag{2.17}$$

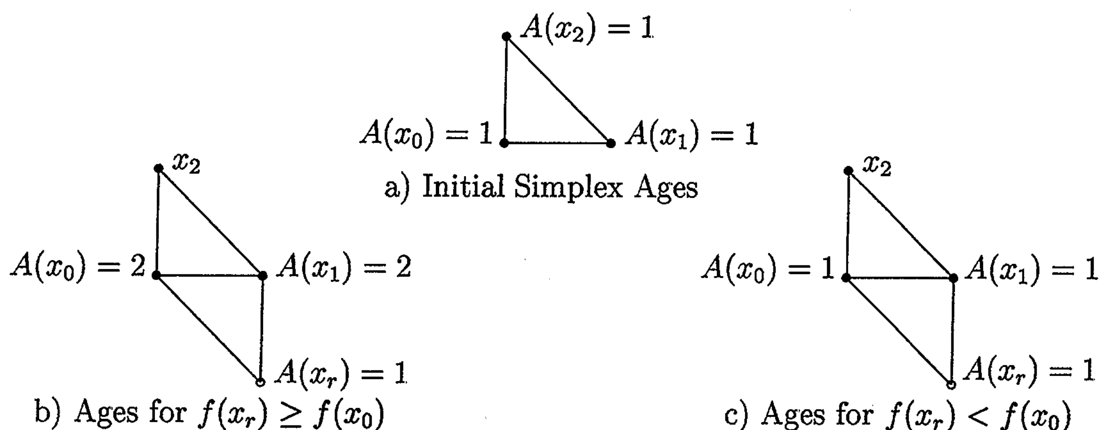


Figure 2.5: a) The initial simplex vertices and their respective ages, initialized to 1. b) If the reflection point is *not* a new minimum (i.e. $f(x_r) \geq f(x_0)$), simply increment the ages of the remaining simplex vertices. c) If the reflection point *is* a new minimum (i.e. $f(x_r) < f(x_0)$), all simplex vertices begin the next iteration with their ages set to 1.

Now that we have a means for determining how long a point has been a vertex of various simplices, all that remains is to determine the age that should trigger a shrink. Going back to the two-dimensional example, there are six possible different simplices around x_0 before the simplices begin to repeat themselves. On the first repetition, x_0 has reached an age of 7. We can generalize this to multiple dimensions and state that for a search in $\mathbb{R}^n$, if at any time the age of one of the simplex vertices reaches a value greater than $2(n+1)$, a value representing the maximum positive basis for dimension n, a shrink should occur.

Some initial testing on this subject found $2(n+1)$ an acceptable criterion; however, we believe it to be too conservative. The $2(n+1)$ criterion is closely related to the concept of a maximal positive basis, which suggests that the search will have looked in a sufficient number of directions to identify a direction of descent if the current iterate is not a stationary point. It is therefore expected that a criterion related to a minimal positive basis, $n+1$, should be sufficient. A further discussion of this choice of criterion is available in section 4.2. Practical testing appeared to favor the $n+1$ case. It is therefore the choice used in the formal description of the algorithm stated above.

It is important to note that the concept of point age was actually addressed by Spendley, Hext and Himsworth in their original paper. They believed that point age could be used to determine when the simplex neared the area of a minimizer. However, they were not interested in shrinking the simplex. While they granted shrinking as a plausible solution to refinement, they were more concerned with stopping the algorithm at a specific point and then using a quadratic fitting method to approximate the true location of the minimizer.

This, however, was flawed in that it was based on the assumption that the size of the simplex was in perfect relative proportion to the function being minimized. If, however, the simplex was quite large and entered the area of two distinct minimizers, a quadratic fitting method would not pick out either individual minimizer but would most likely approximate the minimizer to some incorrect median value. Also, because they were using a quadratic fitting method, their critical values of age were on the order of n^2. This is unnecessarily conservative for our purposes.

Simplex Oscillation

The formal description of the algorithm states that when labeling simplex vertices, the labels on x_n^k and x_{n-1}^k may need to be swapped. This section addresses the reasoning behind this requirement.

Consider the following scenario for the algorithm: The iteration begins, as usual, by ordering the vertices and continues until the worst vertex x_n^k, the centroid $\bar{x}^k$, and the reflection point x_r^k are found. However, let the simplex be aligned with the function such that

$$f(x_r^k) > f(x_{n-1}^k). \tag{2.18}$$

In other words, x_r^k is now the simplex vertex with the greatest function value. (Fig. 2.6 a) For now, assume that the label swapping has *not* been included in the algorithm. This means that on the next iteration, x_r^k is guaranteed to be replaced by x_r^{k+1}. However, because x_0^{k+1} ... x_{n-1}^{k+1} have not changed since the last iteration and are therefore equal to x_0^k ... x_{n-1}^k, $\bar{x}^{k+1}$ must be equal to $\bar{x}^k$. In this case, when the simplex flips to replace x_r^k, it will necessarily be replacing it with x_r^{k+1}, which is x_n^k, and we will have the original simplex back again (Fig. 2.6 b). The above process will repeat itself until the age of x_0 reaches a critical limit and a shrink occurs.

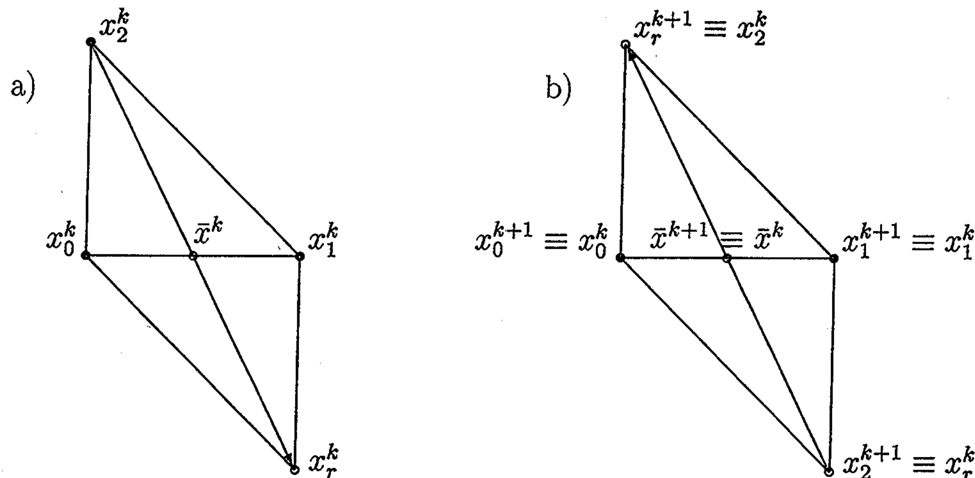


Figure 2.6: a) In this example, reflecting x_2^k to x_r^k makes x_r^k the new maximum. b) On the next iteration, reflecting x_2^{k+1} through $\bar{x}^{k+1}$ yields the original simplex.

Clearly, it is not sufficient on each iteration to blindly replace the vertex which yields the highest function value, for unnecessary oscillation between two adjacent

simplices could occur. This wastes extra function calls and could potentially affect the success of the algorithm by causing the simplex to collapse prematurely to a false minimizer (Figure 2.7). In this example, oscillation only permits the simplex to

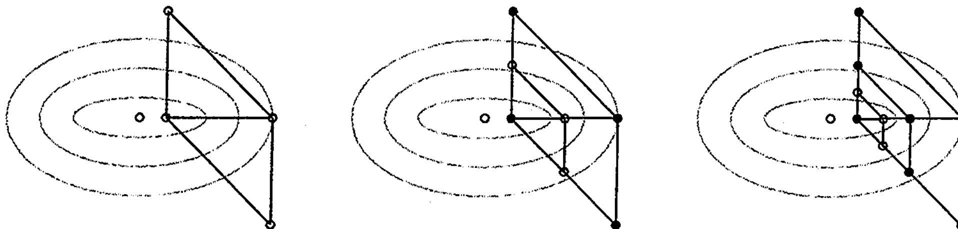


Figure 2.7: In an evolution from left to right, this example shows how oscillation without correction could permit the simplex to contract to a point that is not the true minimizer.

sample the function in a specific area. The range of the sampling is limited and the simplex never looks in the true direction of descent. What we truly desire is for the simplex to sample the function in as many directions as possible and to continually minimize $f(x_0)$. This can be accomplished if two criteria are upheld on each iteration:

(1) Flip the simplex such that it does *not* return immediately to the previous simplex, and

(2) Whatever manipulations are made to the simplex, do *not* replace x_0.

Basically, we want the algorithm to keep the simplex moving in new directions and to look for new points with smaller function values, but we don't want to let go of the best point we've found so far.

The following solution to this problem was suggested by Spendley, Hext, and Himsworth: Execute the first iteration as stated above. On the second iteration and all iterations thereafter, when looking for a vertex to replace, do *not* consider the vertex that was newly added from the previous iteration. We can state this more formally: At the beginning of iteration k, order the vertices as to conform with equations (2.2) and (2.3). Now if this is at least the second iteration ($k \geq 2$) and $x_n^k = x_r^{k-1}$, swap the labels on the two worst vertices (i.e. the vertex labeled x_n^k gets labeled x_{n-1}^k and vice-versa) and proceed as usual (Figure 2.8).

There are two minor points about this addendum that should now be mentioned. First, if a label swap between x_n and x_{n-1} does occur, (2.2) and (2.3) no longer necessarily hold. This is irrelevant to the algorithm, however, for it only cares about x_n as far as it is the vertex to be replaced by a reflection through the centroid of the other n vertices. Second, this will not work for $\mathbb{R}^1$, for in this case x_{n-1} is, by definition, x_0, and a swap would force the algorithm to replace the only vertex we were adamant about keeping. If this algorithm is to be used for one-dimensional optimization, the algorithm as stated in 2.1.1 is sufficient, for although oscillation is still possible, in one dimension an instance of simplex oscillation is necessarily a sign

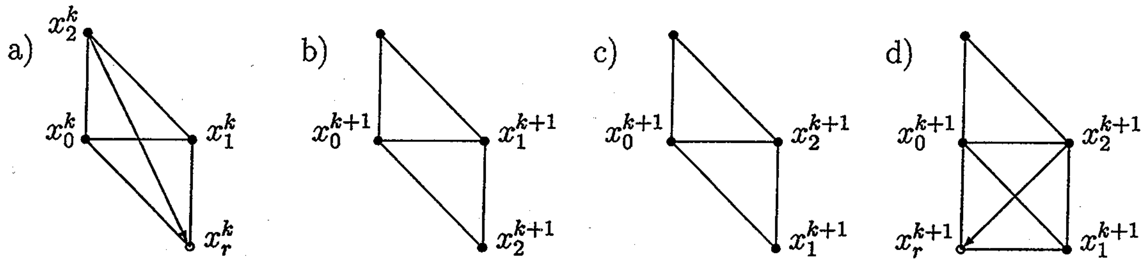


Figure 2.8: This is an example of label swapping to prevent simplex oscillation. This will resolve the problem encountered in Figure 2.7. a) On the kth iteration, the reflection step is taken to yield x_r^k. b) The $(k+1)$th iteration begins by labeling the points such that $f(x_0^{k+1}) < f(x_1^{k+1}) < f(x_2^{k+1})$. c) Because $x_2^{k+1} \equiv x_r^k$, we swap the labels on x_1^{k+1} and x_2^{k+1}. d) Having made this change, the reflection step in the $(k+1)$th iteration yields a new point $x_r^{k+1} \not\equiv x_2^k$, and oscillation is no longer an issue. The simplex will continue to move as in Figure 2.4, allowing it to search in as many directions as possible, preventing collapse to a false minimizer.

that the vicinity of a minimizer has been located, and therefore shrinking the simplex is the proper next move.

The method by which we have addressed the issue of oscillation here is crucial to the success of the age-based shrinking criterion described in the previous section. While Spendley, Hext and Himsworth provided a similar solution, theirs was not as concrete. It was therefore necessary for us to refine the solution as discussed above.

This change, although subtle, allows the algorithm to satisfy the two criteria stated above. First, it prevents the simplex from immediately returning whence it came and entering an instance of indefinite oscillation, which fulfills criterion (1). Also, (aside from the $\mathbb{R}^1$ case) we are still holding on to our best vertex and simply looking all around it, which fulfills criterion (2).

Termination

All that is left for the Spendley, Hext, and Himsworth algorithm is to define the stopping criteria. While stopping criteria should be defined as to best suit the specific purposes for which the algorithm is being used, there are three common criteria.

Many searches of this kind are designed to approximate the minimum of the function by using the smallest possible number of function calls to achieve that task. Therefore, a reasonable stopping criterion is to simply count the number of function calls made by the algorithm and terminate it when the number of function calls reaches a predefined maximum number.

Another option is to define a value Δ which represents a length inherent to the current simplex. This length can vary slightly in definition, but it is usually defined to be the length of the longest simplex edge. As the algorithm executes and the simplex continues to shrink, Δ decreases along with it. When Δ becomes less than some small predefined tolerance constant ϵ, it is assumed that the simplex has effectively contracted to a single point, and the algorithm should terminate. The constant ϵ is

usually on the order of 10^{-8}.

A final method is based on the stopping criteria given by Avriel [1] for suggested use with the simplex search of Nelder and Mead. Find the mean, $\mu = \frac{1}{n}\sum_{i=1}^{n} f(x_i)$, of the function values for all vertices excluding that of the best point. Terminate the algorithm if

$$\left[\frac{1}{n+1}\sum_{i=0}^{n}\left[f(x_i)-\mu\right]^2\right]^{\frac{1}{2}} < \epsilon, \tag{2.19}$$

where ϵ is a small tolerance constant, usually on the order of 10^{-8}.

2.2 The Simplex Search of Nelder and Mead

While the simplex search method of 2.1 is straightforward and sufficient for optimization, the simplex in use never changes its shape and therefore is effectively unaware of the true curvature of the function which it is sampling. Nelder and Mead realized this fact and believed the simplex search method could be made more efficient if the simplex were allowed to adjust its shape with each iteration to better suit the curvature of the function. Also, the original Nelder-Mead paper [8] was the first paper dedicated solely to the optimization of nonlinear functions using a simplex method. (Recall that evolutionary operation was the primary concern of Spendley, Hext, and Himsworth.) It is also often assumed to be very easy to implement (although the discussion of Chapter 3 suggests that one must be careful with this assumption). It is probably for these reasons that Nelder-Mead is one of the most widely used non-derivative-based search algorithms.

2.2.1 General Description

The Nelder-Mead algorithm is similar to that of Spendley, Hext, and Himsworth, except that here the simplex can alter its shape. While with the Spendley, Hext, and Himsworth method, the simplex will only reflect or shrink with each iteration, Nelder and Mead allow for a reflection, an expansion, a contraction, and a shrink. These four steps are labeled respectively by the coefficients: ρ (reflection), χ (expansion), γ (contraction), and σ (shrink), governed by the rules:

$$\rho > 0,\ \chi > 1,\ \chi > \rho,\ 0 < \gamma < 1,\ \text{and } 0 < \sigma < 1. \tag{2.20}$$

While these are general rules, they almost always are seen by convention yielding the following values:

$$\rho = 1,\ \chi = 2,\ \gamma = \frac{1}{2},\ \sigma = \frac{1}{2}. \tag{2.21}$$

Please note that while γ and σ are conventionally given the same value of $\frac{1}{2}$, their roles in the algorithm are quite different. The constant γ is used during the contraction step to determine how the location of just *one* of the simplex vertices is adjusted. The constant σ, however, is used during a shrink step to determine the adjustment of n of the vertices.

Each of the four types of simplex alterations is designed to help the simplex better follow the gradient of the function it is sampling. It is the intention of the Nelder-Mead algorithm to provide a means whereby the simplex can expand itself along directions of improvement and contract itself opposite directions where improvement is not found. The reflection is calculated the same as that of Spendley, Hext, and Himsworth (Figure 2.9); however, Nelder-Mead recognizes that if the new reflection point has a lower function value than $f(x_0)$, it could be advantageous to immediately continue sampling the function in the same direction. This is the purpose of the expansion step (Figure 2.9). The simplex is expanding itself outward from $\bar{x}$ in the direction of plausible function decrease. If, however, the reflection and expansion steps do not improve upon the current lowest function value, the assumption is that the simplex is straddling a minimizer. The simplex then contracts itself in one of two ways: If x_n has a function value less than the reflection point x_r, then x_n is more likely to be closer to a minimizer. In this case, the simplex contracts from x_n toward the centroid $\bar{x}$ (Figure 2.9). Otherwise, if x_r has a function value less than x_n, the simplex contracts from x_r toward $\bar{x}$ (Figure 2.9). Finally, if no improvement has been found, the simplex shrinks toward x_0 as with the Spendley, Hext and Himsworth algorithm (Figure 2.3).

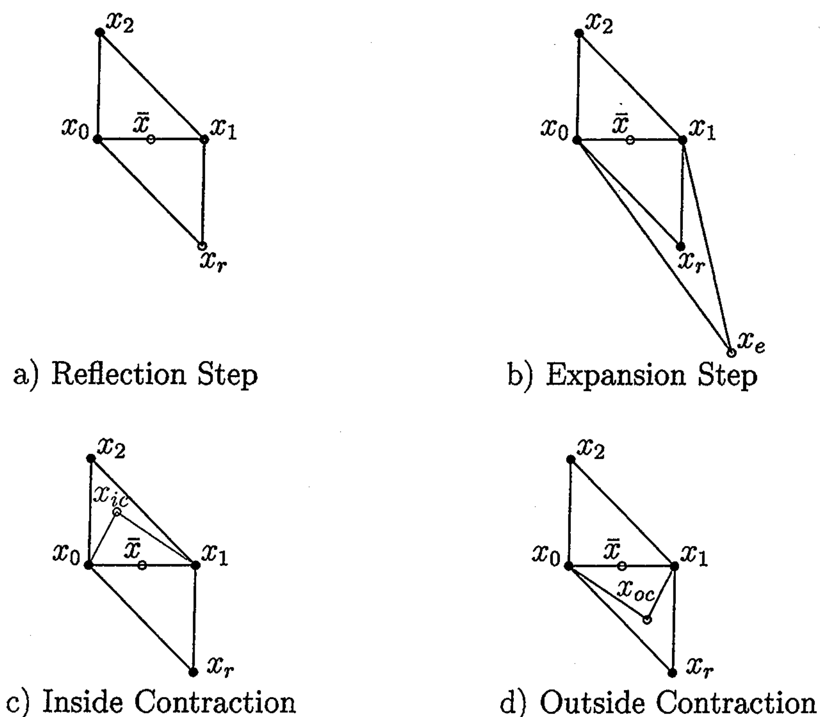


Figure 2.9: a) The *reflection step* of the Nelder-Mead algorithm is similar to that of Spendley, Hext and Himsworth. b) The *expansion step* further reflects x_2 through $\bar{x}$ to yield the expansion point, x_e. c) If $f(x_2) \leq f(x_r)$, perform an *inside contraction* from $\bar{x}$ toward x_2 to find x_{ic}. d) If $f(x_2) > f(x_r)$, perform an *outside contraction* from $\bar{x}$ toward x_r to find x_{oc}.

2.2.2 Formal Description

The formal description of the algorithm used in this research and described below is from Lagarias, Reeds, Wright, and Wright [6]; however, some notation has been changed here to better coincide with the description of 2.1.2.

0. **Initialize.** Start with a nondegenerate simplex for $\mathbb{R}^n$ and calculate the function values at all of the vertices. Then at each iteration $k, k \geq 0$:

1. **Order.** Order the vertices $x_0^k, x_1^k, ..., x_{n-1}^k, x_n^k$ such that

$$f(x_0^k) \leq f(x_1^k) \leq ... \leq f(x_{n-1}^k) \leq f(x_n^k). \tag{2.22}$$

2. **Reflect.** After computing the centroid $\bar{x}^k = \frac{1}{n}\sum_{i=0}^{n-1} x_i^k$, compute the *reflection point* x_r^k from

$$x_r^k = \bar{x}^k + \rho(\bar{x}^k - x_n^k). \tag{2.23}$$

If $f(x_0^k) \leq f(x_r^k) < f(x_{n-1}^k)$, replace x_n^k with x_r^k and go to Step 6.

3. **Expand.** If $f(x_r^k) < f(x_0^k)$, calculate the *expansion point* x_e^k from

$$x_e^k = \bar{x}^k + \chi(x_r^k - \bar{x}^k). \tag{2.24}$$

If $f(x_e^k) < f(x_r^k)$ replace x_n^k with x_e^k and go to Step 6; otherwise ($f(x_e^k) \geq f(x_r^k)$), replace x_n^k with x_r^k and go to Step 6.

4. **Contract.** If $f(x_r^k) \geq f(x_{n-1}^k)$, perform a *contraction* between $\bar{x}^k$ and whichever of x_r^k and x_n^k has the lower function value.

a. Outside. If $f(x_{n-1}^k) \leq f(x_r^k) < f(x_n^k)$, perform an *outside contraction*: calculate

$$x_{oc}^k = \bar{x}^k + \gamma(x_r^k - \bar{x}^k). \tag{2.25}$$

If $f(x_{oc}^k) \leq f(x_r^k)$, replace x_n^k with x_{oc}^k and go to Step 6; otherwise perform a shrink (Step 5).

b. Inside. If $f(x_r^k) \geq f(x_n^k)$ perform an *inside contraction*: calculate

$$x_{ic}^k = \bar{x}^k + \gamma(x_n^k - \bar{x}^k). \tag{2.26}$$

If $f(x_{ic}^k) < f(x_n^k)$, replace x_n^k with x_{ic}^k and go to Step 6; otherwise perform a shrink (Step 5).

5. **Shrink.** Shrink the simplex around x_0^k by replacing x_i^k with

$$\hat{x}_i^k = x_i^k + \frac{1}{2}(x_0^k - x_i^k),\ i = 1, ..., n. \tag{2.27}$$

6. **Check Termination.** Terminate if any of the possible stopping criteria in force are satisfied. Otherwise, set $k = k + 1$ and return to Step 1.

Our complete C++ implementation of this algorithm is available in Appendix A. The heart of the algorithm explained above can be found in the ExploratoryMoves() function, located in section A.2.2.

2.2.3 Subtleties

At this time, it is important to point out some subtle aspects of the Nelder Mead algorithm. The algorithm above is very specific about putting conditions of equality in certain places. The original Nelder-Mead paper left these choices ambiguous, while Lagarias, Reeds, Wright, and Wright needed to specify the algorithm in a very particular unambiguous fashion for analysis purposes.

For instance, there are two small, yet important differences between this algorithm and that of Avriel's interpretation which deal specifically with the contraction step. 2.2.2 states that an outside contraction is to be taken if $f(x_r)$ is *strictly less than* $f(x_n)$. Otherwise, $f(x_r) \geq f(x_n)$, which is the condition for an inside contraction. Avriel, however, leaves the case $f(x_r) = f(x_n)$ open, not defining any choice of contraction for this case. Second, in 2.2.2, shrinking after an outside contraction occurs if $f(x_{oc}) > f(x_r)$, while shrinking after an inside contraction occurs if $f(x_{ic}) \geq f(x_n)$. Note the addition of the $f(x_{ic}) = f(x_n)$ case for the inside contraction but *not* for the outside contraction. Avriel, however, does not shrink after an inside contraction if $f(x_{ic}) = f(x_n)$, while leaving the criteria for a shrink after an outside contraction the same as 2.2.2.

While these may appear to be minor and unimportant discrepancies in the interpretation of the Nelder-Mead algorithm, we've found they actually play a large role in the algorithm's success near the boundaries of a bounded objective function. This role and its effects are further described in Chapter 4.

2.3 Sequential Multi-Directional Search

In 1989, Torczon published a paper introducing the Multi-Directional Search Algorithm [18]. Intended for execution on a multi-processor machine, its most attractive feature was a design constructed to utilize the ability to make multiple function calls simultaneously, without additional real-time cost. Here we introduce an adapted sequential version of the algorithm, intended for single-processor machines.

2.3.1 General Description

The Multi-Directional Search, like the other simplex searches, is based on a simplex of $n+1$ vertices in $\mathbb{R}^n$. However, its algorithm contains one main difference from the other two algorithms previously discussed. During a reflection step, rather than simply reflecting the vertex with greatest function value, x_n, through the centroid of the other vertices, $\bar{x}$, the Multi-Directional Search instead reflects all vertices, x_1 ... x_n, through the best *vertex*, x_0. The following general description of the algorithm will elaborate on this procedure.

Each iteration of the Sequential Multi-Directional Search really consists of two simplices: a primary simplex, and its reflection simplex through the current best minimizer x_0. Because both simplices share x_0, this gives a total of $2n+1$ simplex vertices. Unlike the simplex algorithms encountered so far, not all of the function values of these vertices are known. Because it is a greedy algorithm, function values

are only calculated until a new minimizer is found. Therefore, at any given time during an iteration, the algorithm may know anywhere from only 1 to all $2n + 1$ of these values. To exemplify this concept, in the figures accompanying this section, a simplex vertex whose function value is known will be represented by a filled-in dot, while those vertices whose function values have not yet been calculated will be denoted by an empty circle.

Begin with a nondegenerate simplex of $n + 1$ vertices in $\mathbb{R}^n$, but do *not* evaluate any of the vertices' function values. This is the primary simplex. Choose one of these vertices to be the basepoint, x_0, and calculate only $f(x_0)$. The other n points can be arbitrarily labeled x_1 ... x_n for reference purposes (Figure 2.10). This algorithm is only concerned with the current best point having the label x_0. Because not all vertex function values are known, the algorithm does not (and cannot) require the ordering of all $n + 1$ points based on function value.

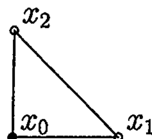


Figure 2.10: Begin by labeling the basepoint x_0 and arbitrarily labeling the other points. Calculate only $f(x_0)$.

The next step is to reflect each point x_1 ... x_n through x_0 to create a reflection simplex. The reflection simplex and the primary simplex will, by construction, share x_0, and the reflection simplex will contain n new points, x_{r_1} ... x_{r_n} (Figure 2.11). A reflection in this manner guarantees that if the primary simplex is nondegenerate, the reflection simplex will also be nondegenerate.

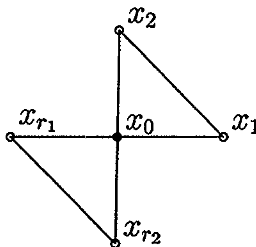


Figure 2.11: A reflection of the primary simplex through x_0 yields the reflection simplex.

From here, the function value of each vertex in the reflection simplex is calculated until a vertex, x_{r_i} is identified whose function value is less than $f(x_0)$. If x_{r_i} is found, the reflection simplex becomes the new primary simplex with x_{r_i} as its basepoint, and the algorithm returns to the first step. If improvement cannot be found in the reflection simplex, the primary simplex is searched one vertex at a time for a vertex x_i is less than $f(x_0)$. If such an x_i is found, the primary simplex makes x_i its basepoint, and the algorithm returns to the first step. If all $2n+1$ points in both the primary and

reflection simplices have been evaluated and improvement over x_0 has not been found, the primary simplex shrinks around x_0, and a new iteration is begun (Figure 2.12).

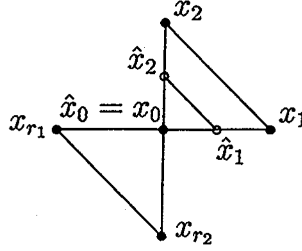


Figure 2.12: If all points in both simplices have been evaluated without improvement, shrink the primary simplex around x_0.

2.3.2 Formal Description

The following is a formal description of the Sequential Multi-Directional Search algorithm.

0. **Initialize.** Given x_0^0, construct a simplex of $n+1$ points for $\mathbb{R}^n$ that includes x_0^0, but calculate *only* $f(x_0^0)$. Then, for each iteration $k, k \geq 0$:

1. **Label.** Label x_0^k as the basepoint. Arbitrarily label the other vertices x_1^k ... x_n^k.

2. **Create Reflection.** Reflect each of x_1^k ... x_n^k through x_0^k to obtain the vertices

$$x_{r_i}^k = x_i^k + (x_i^k - x_0^k), \; i = 1, ..., n. \tag{2.28}$$

3. **Look through Reflection for Minimizer.** Evaluate (sequentially) $f(x_{r_i}^k)$, for $i = 1, ..., n$ until either

$$f(x_{r_j}^k) < f(x_0^k) \text{ for some } j \in \{1, ..., n\} \tag{2.29}$$

or it is determined that

$$f(x_{r_i}^k) \geq f(x_0^k) \text{ for all } i = 1, ..., n. \tag{2.30}$$

If, and only if, equation 2.29 is satisfied, then

$$x_i^k = x_{r_i}^k \text{ for all } i = 1, ..., n, \tag{2.31}$$

swap the labels on x_j^k and x_0^k, and go to Step 6. Otherwise (i.e., equation 2.30 is satisfied), continue to Step 4.

4. **Look through Primary for Minimizer.** Evaluate (sequentially) $f(x_i^k)$, for $i = 1, ..., n$ until either

$$f(x_j^k) < f(x_0^k) \text{ for some } j \in \{1, ..., n\} \tag{2.32}$$

or it is determined that

$$f(x_i^k) \geq f(x_0^k) \text{ for all } i = 1, ..., n. \tag{2.33}$$

If, and only if, equation 2.32 is satisfied, then swap the labels on x_j^k and x_0^k, and go to Step 6. Otherwise (i.e., equation 2.33 is satisfied), continue to Step 5.

5. **Shrink.** Since neither (2.29) nor (2.32) have been satisfied, shrink the (primary) simplex around x_0^k by replacing x_i^k with

$$\hat{x}_i^k = x_i^k + \frac{1}{2}(x_0^k - x_i^k), \ i = 1, ..., n. \tag{2.34}$$

Replace x_i^k with $\hat{x}_i^k$ for $i = 1, ..., n$.

6. **Check Termination.** Terminate if any of the possible stopping criteria in force are satisfied. Otherwise, set $k = k + 1$ and return to Step 1.

Our complete C++ implementation of this algorithm is available in Appendix A. The heart of the algorithm explained above can be found in the ExploratoryMoves() function, located in section A.3.2.

2.3.3 From Multi-Processor to Single-Processor

This algorithm is different from the two previous simplex algorithms in that a reflection occurs about a vertex of the simplex rather than about a centroid. This has the side-effect of n new simplex vertices for each iteration, rather than one. The advantage is a greater overall sampling of the function; however, finding their function values requires n new function calls on each iteration. If run on a machine with at least n processors, as was Torczon's intention, this was not a significant problem, for n function calls could be made simultaneously and therefore effectively cost as much as only one call. This is the true power of the original algorithm. Here, we do not have n processors to work with, and therefore an algorithm that evaluated all vertex function values would be too expensive to execute effectively.

It was therefore necessary to modify the multi-processor algorithm so that we could keep the advantage of reflecting n vertices simultaneously, but at the same time make as few function calls per iteration as possible. The purpose of evaluating all function values of the n new vertices was to locate a vertex whose function value was not only less than $f(x_0)$, but also less than the function values of the other $n-1$ new vertices. In a choice to compromise absolute minimization for speed, the solution was to invoke the style of a greedy algorithm: Once a reflection is made through x_0, keep track of the location of the new n vertices, but only evaluate enough function values, one at a time, to single out one vertex whose function value is less than $f(x_0)$. While this criterion is clearly less strict that the multi-processor algorithm, it was best suited for the problem and most consistent with other existing search methods.

The question which now needs to be addressed is "Once the n new vertices have been identified, in what order should their function values be evaluated?" To answer this question, we pulled from the ideology of Nelder and Mead's expansion step: If a step is taken from a point of low function value toward a new point whose function value is even lower, continue to look for the next point by taking a further step in the same direction.

This concept is extended here and can best be understood through an example. Let x_i^k denote a specific point x_i for a given iteration k. Consider a scenario where

we have just reflected the simplex through our current minimizer x_0^k and have n new simplex vertices. Now say we have located a vertex of the new simplex $x_{r_2}^k$ such that $f(x_{r_2}^k) < f(x_0^k)$ (Figure 2.13 a). This would then be the basepoint for the next iteration ($x_{r_2}^k \equiv x_0^{k+1}$). To start the $(k+1)$th iteration, we reflect the simplex, this time through x_0^{k+1}. Now we again have n new simplex vertices and we need to begin looking for a new minimizer. This is where the rule comes in. We had a minimizer x_0^k that was replaced by a better minimizer x_0^{k+1}. Therefore, necessarily x_0^k, which is equivalent to x_2^{k+1}, has been reflected through x_0^{k+1} to a new vertex $x_{r_2}^{k+1}$. We therefore begin looking in the direction of descent by evaluating $f(x_{r_2}^{k+1})$ first (Figure 2.13 b). If $f(x_{r_2}^{k+1})$ is evaluated and improvement is not found (i.e. $f(x_{r_2}^{k+1}) \geq f(x_0^{k+1})$), the function values of the remaining $n-1$ vertices can be evaluated in any order.

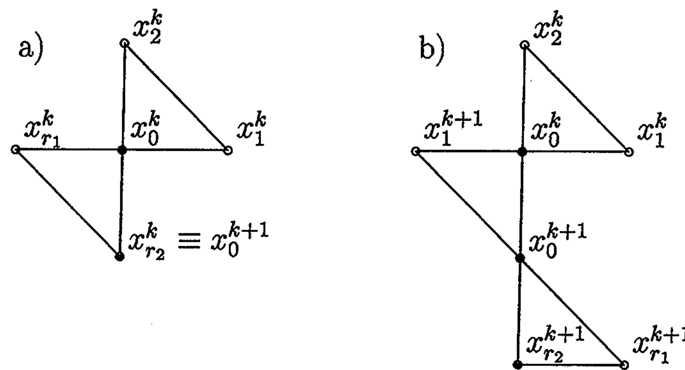


Figure 2.13: a) $f(x_{r_2}^k) < f(x_0^k)$, therefore $x_{r_2}^k$ will be the new basepoint for the next iteration (i.e. $x_{r_2}^k \equiv x_0^{k+1}$). b) We begin the $(k+1)$th iteration by evaluating $f(x_{r_2}^{k+1})$ first because the step from x_0^k (which is equivalent to x_2^{k+1}) to x_0^{k+1} yielded improvement, so we continue in the same direction.

This is the primary adjustment that has been made to go from many processors to one. Finally, it should be mentioned here that Torczon originally included an expansion step in the Multi-Directional Search algorithm which we have decided not to transcribe to the sequential version. Otherwise, the progression of each algorithm is, in essence, the same.

Chapter 3

The Importance of Resolving Ambiguities in the Nelder-Mead Search Algorithm

It was mentioned in section 2.2.3 that the Nelder and Mead simplex algorithm contains some important subtleties which, if not treated appropriately, can have an adverse effect on how the algorithm behaves. We discovered an instance of a bound constrained optimization problem where the Nelder and Mead simplex algorithm converged to a nonstationary point near the boundary. We thus are interested in what happens when at least one of the components of the current iterate is at or near one of the bounds (i.e., $x_i \cong l_i$ or $x_i \cong u_i$ for at least one $i \in \{1, ..., n\}$).

Here we discuss our specific encounter with these subtleties, which occurred during testing of the Nelder-Mead algorithm on numerous bound constrained problems.

3.1 The Discovery of the Problem

Testing was done using the Nelder-Mead algorithm on a set of smooth bounded objective functions, each of whose independent variables was restricted to the range 0 through 10. Every test function had multiple local extrema. Also, each was continuous and had continuous first and second derivatives.

During our testing, each of the simplex searches was run on each function 1000 times, each time with a different starting simplex which was randomly placed as follows. A random number generator would choose one floating point number between 0 and 10 for each dimension. These random numbers made up the coordinates of the basepoint v_0 for the starting simplex, which is guaranteed to be feasible . The basepoint was designated a vertex of the simplex. To generate the remaining n vertices of the starting simplex, the program then took a step of a distance of 2.0 from the basepoint along each of the positive coordinate directions; i.e. $v_i = v_0 + 2e_i, i = \{1, ..., n\}$, where e_i denotes the standard unit coordinate vector. This process had the effect of creating what is often referred to as a fixed-length right-angled simplex to begin each test.

Finally, because we were dealing with bounded problems, it was necessary to deal

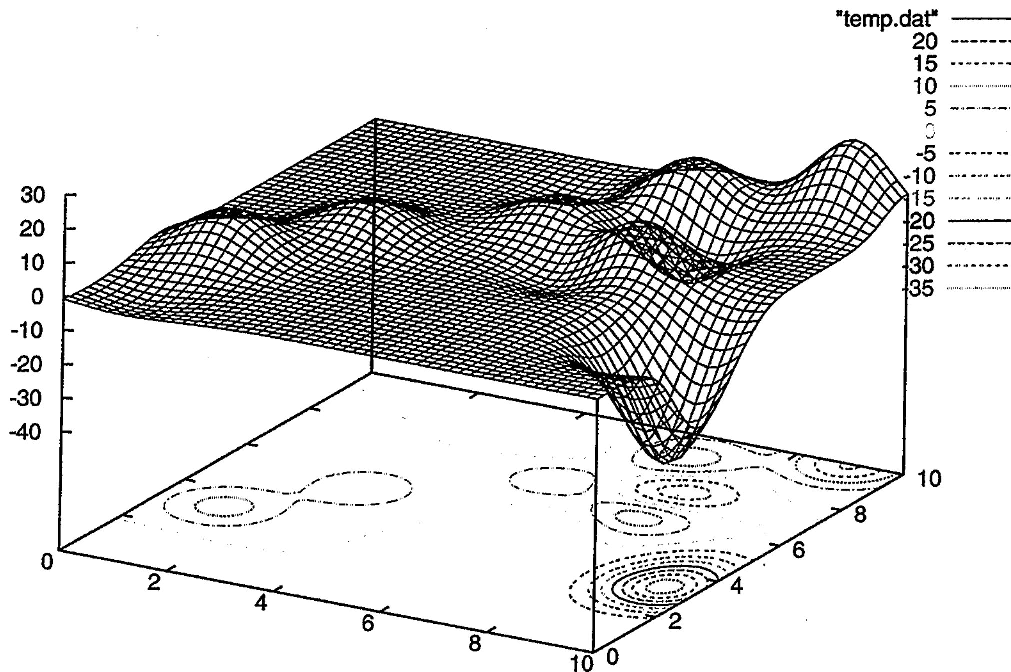


Figure 3.1: Our 2-dimensional objective function. For a given point (x, y), x runs down along the lower left axis and y runs up along the lower right axis.

with the possibility that a simplex might flip into an infeasible region. To account for this, we used a method common to non-derivative-based searches and suggested by Nelder and Mead [8]: whenever a simplex vertex is infeasible, the function value assigned to the vertex should be a large number, in our case infinity. This will cause the search to instantly reject any infeasible vertex, thus ensuring that the sequence of best vertices remains feasible.

After extensive testing of all three simplex searches, we noted that the Nelder-Mead method sometimes gave anomalous results. Occasionally, a point returned by the search as a function minimizer would actually not be anywhere close to a true minimizer—or *any* stationary point at all. We decided to examine the situation more closely; specifically, we chose for analysis the 2-dimensional objective function shown in Figure 3.1. It should be noted that this particular objective function has no minimizers with a positive function value.

While scanning through the 1000 test cases on this function, we found that twelve cases had returned false minimizers with positive function values. The twelve false minimizers and their values are shown in Table 3.1.

Comparing these values with the objective function shown in Figure 3.1 yields some surprising results. First, all of the function values returned are positive. However, the objective function has *no* local minimizers with positive function values.

Ex.	Min. Location	Min. $f(x)$ Value
1.	(9.54547, 9.22147)	9.18047
2.	(9.94115, 9.26812)	5.50756
3.	(8.34786, 9.65801)	12.5530
4.	(8.54405, 9.79115)	16.3964
5.	(9.96021, 9.25700)	5.23315
6.	(9.48996, 9.64920)	15.1434
7.	(9.74767, 9.98758)	12.1590
8.	(9.16491, 9.47893)	16.2598
9.	(9.92376, 9.96469)	9.11214
10.	(8.33212, 9.53514)	11.3507
11.	(8.67829, 9.63636)	16.9166
12.	(8.56104, 9.94855)	17.0677

Table 3.1: The twelve unusually high putative minimizers returned by the Nelder Mead search.

Second, and perhaps more interesting, is that all of the points chosen as minimizers fall relatively close to (10,10), the upper limit of the domain variables. Looking at the objective function in Figure 3.1, it is clear that this area is near the *absolute maximum* of the function. Nelder-Mead was actually returning some of the *highest* possible function values. Also interesting was that even though the other two simplex algorithms, that of Spendley, Hext, and Himsworth and the Sequential Multi-Directional Search, were run on the exact same function, with the exact same starting points and initial simplices, neither of them were returning such solutions. Nelder-Mead alone was falling into some trap that was causing its simplex to collapse to a nonstationary point.

3.2 The Cause

The only way to figure out why Nelder-Mead was returning false results was to look more closely at the twelve specific runs. At this point, we only had information regarding the putative location of the function minimizer. We thought that it would be advantageous to know the exact nature of the entire simplex.

Recreating the same twelve test cases, this time we logged the exact starting and finishing positions of all the vertices in the simplices. These data, as well as the final function values of each of the final simplex vertices, are shown in Table 3.2.

Even though this information is limited, there are definite trends among the twelve cases. First, every simplex begins with all but one of its vertices infeasible. The fact that this is a valid starting position is not surprising, for we generate the simplices by adding 2.0 to each of the basepoint components. Therefore, any time the random number generator creates a basepoint whose components are all above 8.0, this starting scenario will occur.

Looking at the ending locations of the twelve simplices, the problem becomes clear.

Upon termination, two of the three simplex vertices are still in the infeasible region. In fact, in every case, x_0 and x_1 do not move at all during the search, and x_2 has become equivalent to the centroid of x_0 and x_1. At this point, the three vertices are necessarily collinear, the simplex has lost full dimensionality, and the simplex remains with only one feasible vertex. It is this one feasible vertex which (of course) represents the lowest function value found thus far and is therefore the vertex returned by the algorithm as the likely function minimizer. A comparison of Tables 3.1 and 3.2 show this to be the case.

3.3 The Solution

At this point, it was clear that in each of the twelve test cases, the simplex, which should be a triangle for this two-dimensional problem, was collapsing to a line. All that was left was to determine why this was happening. For this, it was necessary to track through one instance of the algorithm step-by-step to determine the cause of the behavior.

It is important to note that for these tests, we were *not* yet using the algorithm of Lagarias, Reeds, Wright, and Wright [6] which was described in section 2.2.2. At the time, our implementation of the algorithm was based on the description found in Avriel [1]. We note here that either of the two descriptions is a perfectly plausible interpretation of the original specification found in [8].

In our initial testing, for these twelve cases, we started with one feasible vertex; the remaining two vertices were infeasible. Our implementation of the algorithm would then label the simplex vertices x_0, x_1, x_2 such that $f(x_0) \leq f(x_1) \leq f(x_2)$. Since only one vertex was feasible, the choice for x_0 was easy; because the remaining two vertices were given function values of infinity, the algorithm arbitrarily chose which of the two remaining vertices would be x_1 and which would be x_2. (Note that this arbitrary labeling is not the cause of the problem.)

Next x_2 was reflected through the centroid of x_0 and x_1 to obtain x_r. However, the simplex was positioned such that x_r was also infeasible. This made $f(x_r) = f(x_2) = \infty$, making the expansion step unnecessary. The next step, therefore, was to take a contraction step. Normally, if $f(x_r) < f(x_n)$, an outside contraction is made toward x_r, and if $f(x_n) < f(x_r)$, an inside contraction is made toward x_n. (Of course, for this particular example, $n = 2$, so $x_2 \equiv x_n$.) Avriel, however, states that a contraction step should be taken toward $\min\{f(x_n), f(x_r)\}$ and does not address our case where $f(x_r) = f(x_n)$. This decision seemed inconsequential during the coding of the algorithm, so it was simply determined that if $f(x_r) = f(x_n)$, an *outside* contraction toward x_r would be taken.

So, an outside contraction was taken to find x_{oc}. This is the crucial step that allows the simplex to collapse: Avriel states that if $f(x_{oc}) > f(x_r)$ (or $f(x_{ic}) > f(x_2)$—Avriel makes no distinction between inside and outside contractions), a shrink of the simplex should occur. But here, x_{oc} is still infeasible, making $f(x_{oc}) = f(x_r) = \infty$, so the contraction point is *accepted* and a new iteration begins. On the next iteration, what was x_{oc} in the last iteration is now x_n. The same thing happens again. The simplex contracts to a point still out of the defined region, and the cycle continues.

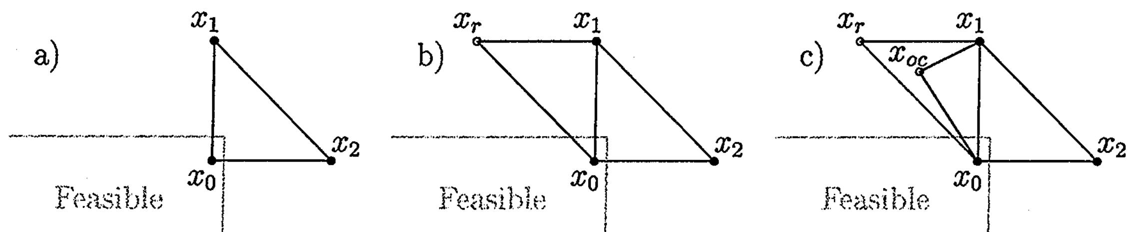


Figure 3.2: This figure shows one iteration of the algorithm which leads to a collapse of the simplex. a) The simplex begins with only one feasible point, x_0. b) The reflection step is taken, however it is such that x_r is also infeasible. c) Having a failed reflection, the algorithm makes an outside contraction to find x_{oc}, which is also infeasible, yet accepted, which terminates the iteration. This process is continued to yield a series of infeasible outside contraction vertices, which is shown in Figure 3.3.

On each iteration, the simplex contracts and accepts x_{oc}. Also, since each iteration is necessarily a contraction, x_{oc} continues to approach the centroid of x_0 and x_1, collapsing the once nondegenerate simplex to a straight line. The evolution of this process is shown in Figure 3.3. Each iteration makes the simplex contract closer to the line defined by x_0 and x_1.

It may appear that a crucial part of this infinite contraction relies on the consistent labeling of the vertices. The vertex labeled x_2 had to be labeled as such throughout the entire execution of the algorithm. While it is true that the initial choice of x_1 and x_2 was arbitrary, it should be clear that this choice only changes the line to which the simplex collapses. The inherent problem is still present. It may also be possible to fix this problem by alternating the points labeled x_1 and x_2. This would, after a while, have the result of slowly shrinking the simplex to a degree where all three vertices were again defined. While possible, a robust algorithm should not have to rely on labeling to prevent this problem. In fact, as will be discussed below, the description of the Nelder-Mead algorithm given by Lagarias, Reeds, Wright, and Wright [6] *does* prevent this problem, and all it takes is some cleverly placed equal signs.

The problem is that the simplex cannot adequately sample the function, for the majority of its vertices are infeasible. Our method of resolving the ambiguities in Avriel's description leads to an algorithm which accepts an infeasible contraction point, when we would rather the simplex perform a shrink around x_0. If the simplex continues to shrink, eventually it will permit all of the vertices to become feasible, and further progress toward a true minimizer can then be made.

If the same scenario is run through the Lagarias, Reeds, Wright, and Wright algorithm (described in section 2.2.2), we find a different outcome for our twelve examples. Here, if $f(x_r) = f(x_n)$ the algorithm chooses to take an *inside* contraction toward x_n. This, however, is not enough to solve our problem, for if we had made the same choice, the simplex would still have collapsed, it just would not have oscillated about the centroid. The key difference is that Lagarias, Reeds, Wright, and Wright make a clear distinction between the shrinking criterion of a inside contraction verses

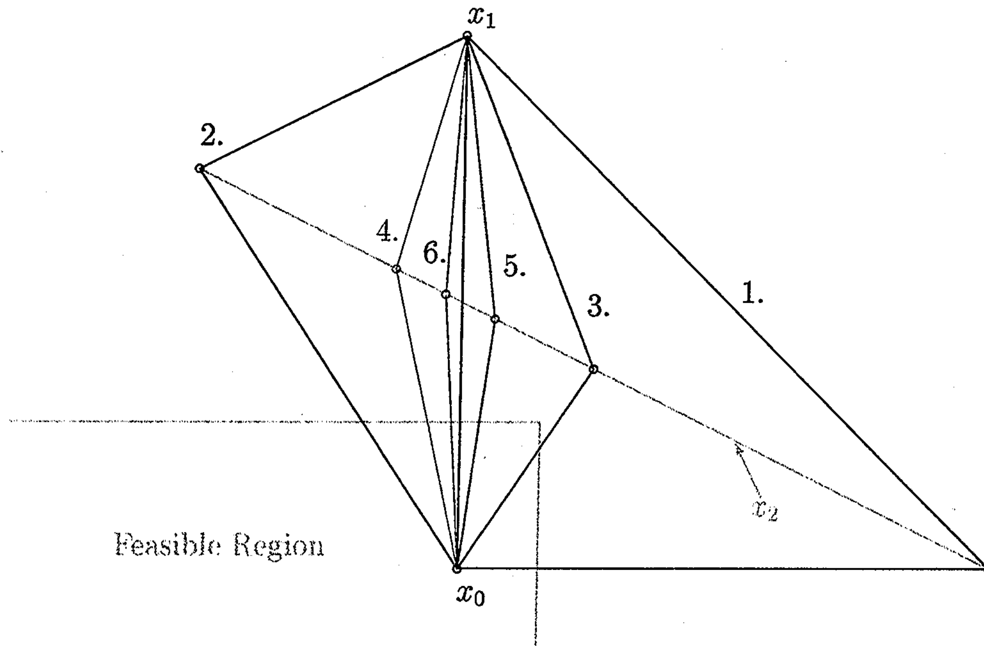


Figure 3.3: The collapse of the simplex {1., 2., 3., 4., 5., 6.} toward the line defined by x_0 and x_1. Note how as the process continues, x_2 gets closer to the centroid of x_0 and x_1.

that of an outside contraction. For an *outside* contraction, the case $f(x_{oc}) = f(x_r)$ results in an acceptance of x_{oc}, just as with Avriel. However, if $f(x_{ic}) = f(x_n)$ during an *inside* contraction, the simplex shrinks around x_0, which is exactly what we hope to accomplish.

Once again for clarity, the two important features of the description given by Lagarias, Reeds, Wright, and Wright are:

(1) If it is necessary to take a contraction step and $f(x_r) = f(x_n)$, take an *inside* contraction step toward x_n.

(2) If during an *outside* contraction, $f(x_{oc}) = f(x_r)$, accept x_{oc} and begin a new iteration. However, if during an *inside* contraction, $f(x_{ic}) = f(x_n)$, shrink the simplex around x_0.

We reran the twelve test cases with the modified algorithm. For each case, the simplex was able to shrink, thus permitting all vertices to become feasible after about two or three shrinks. Afterwards, it could move toward true function minimizers. Table 3.3 shows the new minima found by our twelve test cases using the Lagarias, Reeds, Wright, and Wright variant. Comparing the results with Figure 3.1, we are pleasantly surprised to find that all of the test cases now each converge to a true function minimizer, and nine of the twelve cases actually converge to the global minimizer.

While the analysis developed in this section was strictly for a specific 2-dimensional case, this same scenario originally occurred for problems in higher dimensions as well.

The use of the Lagarias, Reeds, Wright, and Wright algorithm also corrected the results for these higher dimensional cases. The manifestations of these changes can be clearly seen in the full testing results (Chapter 4). It is also important to note that this particular ambiguity is resolved once all of the simplex vertices are within the feasible region. Because this algorithm guarantees a shrinking of the simplex in this scenario, eventually all of the simplex points will necessarily move within the feasible region. This algorithm is therefore guaranteed to prevent the anomaly.

3.4 Discussion

While the anomaly described here may seem rather artificial, it is important because there are many implementations of Nelder-Mead used today in mathematical software packages that would also fail for the scenario presented. For bound constrained problems, it is common practice to simply set the function value to infinity for any infeasible vertex when using a simplex search algorithm. Any user blindly using these packages could fall into this trap, so it is important to recognize its significance.

Note that for this scenario to occur, the initial simplex must *start* in the given configuration. Once all of the vertices are feasible, at most one vertex at a time could wander into an infeasible region, only to be quickly rejected and replaced. This particular instance could then be avoided if the simplex is guaranteed to start completely within a feasible region. We could have achieved this by only allowing the random number generator to generate basepoint components that were between 0 and 8.0. Another suggestion is to allow a range of 0 through 10.0 for the generator, but to take a step of -2.0 from the basepoint for any coordinate direction whose basepoint component was 8.0 or greater. Although this does prevent the problem discussed, it is too restrictive, for a completely general search algorithm should not have to be aware of the limitations of the objective function on which it is working. While the user could be held responsible for guaranteeing that the simplex begin entirely within the feasible region, it is preferable to specify an algorithm which is not based on this assumption. Therefore, for purposes of robustness, it is better to focus on a specification of the algorithm that is less prone to the possible vagaries introduced by a user's choice of starting criteria.

It is interesting to look at different implementations of the Nelder-Mead algorithm which are currently being used. Some of the specifications we examined are immune to the potential pitfall outlined above, while some will fall into the same trap as our original version did. We will compare each of the specifications against criteria (1) and (2) from the last section.

First, we consider the original source, Nelder and Mead [8]. They first defined their search algorithm using an algorithmic flowchart. While this method of explanation made for an easy grasp of the algorithm, it was not at all concerned with the specifics of equality, which is probably the source of ambiguity that has been open to interpretation by so many others. In fact, according to their flowchart, their original statement of the algorithm does not meet either criterion (1) or (2). Like our interpretation drawn from Avriel, they suggest an outside contraction if $f(x_r) = f(x_n)$, and they accept x_c for either an inside or outside contraction if $f(x_c) = f(x_n)$. To

be fair, they did not claim that their algorithm could handle the above scenario at all, for they clearly stated that for their algorithm to work for constraints, the entire simplex would have to start completely within the feasible region.

Looking at some others, we find that a Fortran implementation (`minim.f`) provided by Shaw, with amendments by Wedderburn and Miller [15] fits Nelder and Mead's flowchart perfectly, failing to meet both criteria (1) and (2).

The Fortran implementation of Nelder Mead (`nelmin.f`) [9] provided by O'Neill [10], with subsequent comments by Chambers and Ertel [3], Benyon [2], and Hill [5] is more interesting. Like our original implementation, which fails to meet criterion (1), if $f(x_r) = f(x_n)$, it takes an outside contraction toward x_r. The shrinking criterion for the *inside* contraction is like Avriel's: shrink the simplex only if $f(x_{ic}) > f(x_n)$, otherwise accept x_{ic} and continue. The *outside* contraction, however, does not even have a shrinking criterion. In this case, if $f(x_{oc}) \leq f(x_r)$, we accept x_{oc} and continue. Otherwise ($f(x_{oc}) > f(x_r)$), we accept x_r and continue. Clearly, this would also fail for our scenario.

The *Numerical Recipes in C* implementation provided by Press, Teukolsky, Vetterling, and Flannery [14] does not have an outside contraction at all. If a contraction step is necessary, it will always be an *inside* contraction. However, if $f(x_{ic}) \geq f(x_n)$, they *do* shrink, so this implementation would not fall into our trap.

Finally, we have Torczon's Fortran implementation of `nelder.f` [18]. It does meet criterion (1) above. *Both* the inside and outside contraction steps have the same shrinking criterion: if $f(x_c) \geq f(x_n)$, shrink. Otherwise ($f(x_c) < f(x_h)$), accept x_c and continue. This implementation would also not fail in our test cases.

We have shown here how simple changes in the interpretation of the Nelder-Mead algorithm can either promote or prevent the simplex from collapsing and losing dimensionality. While these changes make the algorithm more robust, they cannot prevent collapsing in every scenario. In 1998, McKinnon [7] published a specific example where Nelder-Mead converged to nonstationary points, yet all simplex vertices were always feasible. McKinnon actually uses an algorithm identical to that of Lagarias, Reeds, Wright, and Wright for the purposes of contraction and shrinking, so his argument is still valid and cannot be contradicted through any method mentioned here. Nonetheless, it is still clear that simple choices made when dealing with the ambiguity of a few equal signs can have significant repercussions.

Ex.	Point	Initial Location	Final Location	Final $f(x)$ Value
1.	x_2	(11.5455, 9.22147)	(9.54547, 10.2215)	∞
	x_1	(9.54547, 11.2215)	(9.54547, 11.2215)	∞
	x_0	(9.54547, 9.22147)	(9.54547, 9.22147)	9.18047
2.	x_2	(11.9412, 9.26812)	(9.94115, 10.2681)	∞
	x_1	(9.94115, 11.2681)	(9.94115, 11.2681)	∞
	x_0	(9.94115, 9.26812)	(9.94115, 9.26812)	5.50756
3.	x_2	(10.3479, 9.65801)	(8.34786, 10.6580)	∞
	x_1	(8.34786, 11.6580)	(8.34786, 11.6580)	∞
	x_0	(8.34786, 9.65801)	(8.34786, 9.65801)	12.5530
4.	x_2	(10.5440, 9.79115)	(8.54405, 10.7912)	∞
	x_1	(8.54405, 11.7912)	(8.54405, 11.7912)	∞
	x_0	(8.54405, 9.79115)	(8.54405, 9.79115)	16.3964
5.	x_2	(11.9602, 9.25700)	(9.96021, 10.2570)	∞
	x_1	(9.96021, 11.2570)	(9.96021, 11.2570)	∞
	x_0	(9.96021, 9.25700)	(9.96021, 9.25700)	5.23315
6.	x_2	(11.4900, 9.64920)	(9.48996, 10.6492)	∞
	x_1	(9.48996, 11.6492)	(9.48996, 11.6492)	∞
	x_0	(9.48996, 9.64920)	(9.48996, 9.64920)	15.1434
7.	x_2	(11.7477, 9.98758)	(9.74767, 10.9876)	∞
	x_1	(9.74767, 11.9876)	(9.74767, 11.9876)	∞
	x_0	(9.74767, 9.98758)	(9.74767, 9.98758)	12.1590
8.	x_2	(11.1649, 9.47893)	(9.16491, 10.4789)	∞
	x_1	(9.16491, 11.4789)	(9.16491, 11.4789)	∞
	x_0	(9.16491, 9.47893)	(9.16491, 9.47893)	16.2598
9.	x_2	(11.9238, 9.96469)	(9.92376, 10.9647)	∞
	x_1	(9.92376, 11.9647)	(9.92376, 11.9647)	∞
	x_0	(9.92376, 9.96469)	(9.92376, 9.96469)	9.11214
10.	x_2	(10.3321, 9.53514)	(8.33212, 10.5351)	∞
	x_1	(8.33212, 11.5351)	(8.33212, 11.5351)	∞
	x_0	(8.33212, 9.53514)	(8.33212, 9.53514)	11.3507
11.	x_2	(10.6783, 9.63636)	(8.67829, 10.6364)	∞
	x_1	(8.67829, 11.6364)	(8.67829, 11.6364)	∞
	x_0	(8.67829, 9.63636)	(8.67829, 9.63636)	16.9166
12.	x_2	(10.5610, 9.94855)	(8.56104, 10.9485)	∞
	x_1	(8.56104, 11.9485)	(8.56104, 11.9485)	∞
	x_0	(8.56104, 9.94855)	(8.56104, 9.94855)	17.0677

Table 3.2: The starting and finishing simplices and the final vertex function values for the twelve unique test cases.

Ex.	Min. Location	Min. $f(x)$ Value
1.	(9.37702, 3.34053)	−38.3280
2.	(9.37702, 3.34053)	−38.3280
3.	(7.32816, 7.46766)	−12.9065
4.	(9.37702, 3.34053)	−38.3280
5.	(9.37702, 3.34053)	−38.3280
6.	(9.37702, 3.34053)	−38.3280
7.	(9.37702, 3.34053)	−38.3280
8.	(7.32816, 7.46766)	−12.9065
9.	(9.37702, 3.34053)	−38.3280
10.	(7.32816, 7.46766)	−12.9065
11.	(9.37702, 3.34053)	−38.3280
12.	(9.37702, 3.34053)	−38.3280

Table 3.3: The twelve minimizers returned by the Lagarias, Reeds, Wright, and Wright algorithm.

Chapter 4

Testing and Results

Having given a full description of the simplex-based algorithms, we suggest that they are now well-defined for the purposes of optimization. This chapter discusses our testing of the search methods and provides a general comparison of performance between them for a specified group of objective functions.

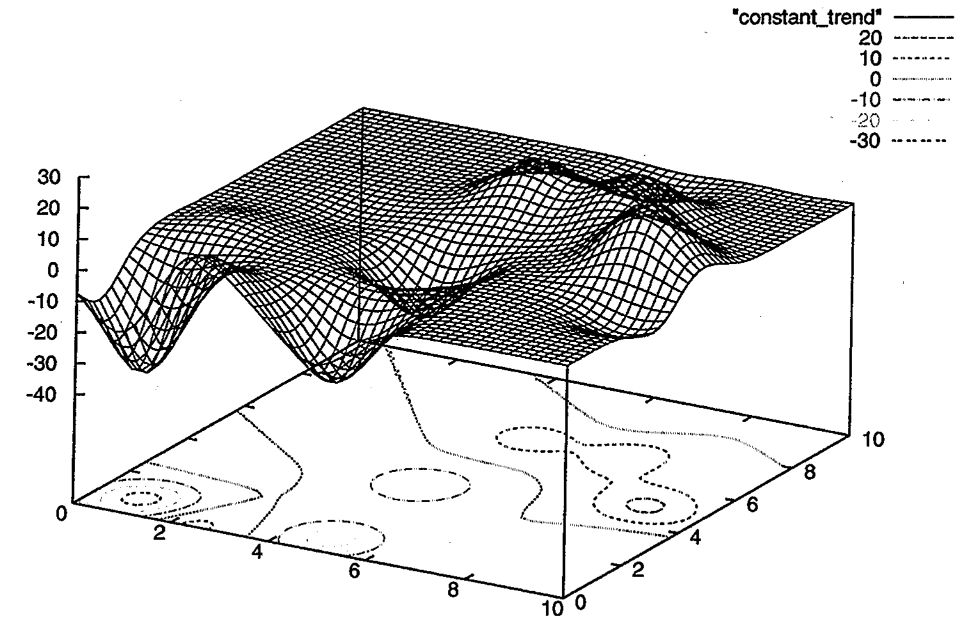
4.1 The Testing Setup

We begin with a description of our testing of the simplex search methods. First, we describe the objective functions used, as well as their creation and incorporation into our optimization procedures. We then describe the type of tests run employing the simplex search methods to these functions. Finally, we conclude with a general description of the function minima density plots, our medium for comparison of the data.

4.1.1 The Objective Functions

All the objective functions used in this research were created using a function generator called the krigifier [11, 19]. The krigifier is a function generator which accepts general parameters describing a function, such as overall trend, as well as a number of other parameters that determine how much curvature the function contains. The krigifier can then be used to generate continuous, differentiable, nonlinear functions with characteristics that vary based on the input given.

In this research, we consider two types of trends for our objective functions: constant and quadratic. To generate a function with a constant trend, the krigifier begins with a constant function surface (for a function of two dimensions, this would represent a plane parallel to the x-y plane). The krigifier then uses its input parameters to add smooth bumps to this surface, creating ranges of local extrema. An objective function with quadratic trend begins with a quadratic surface (a paraboloid in two dimensions), to which smooth bumps are then added based on the krigifier parameters. Figure 4.1 shows two objective functions generated by the krigifier, one with a constant trend (above) and one with a quadratic trend (below), both based on similar input parameters.



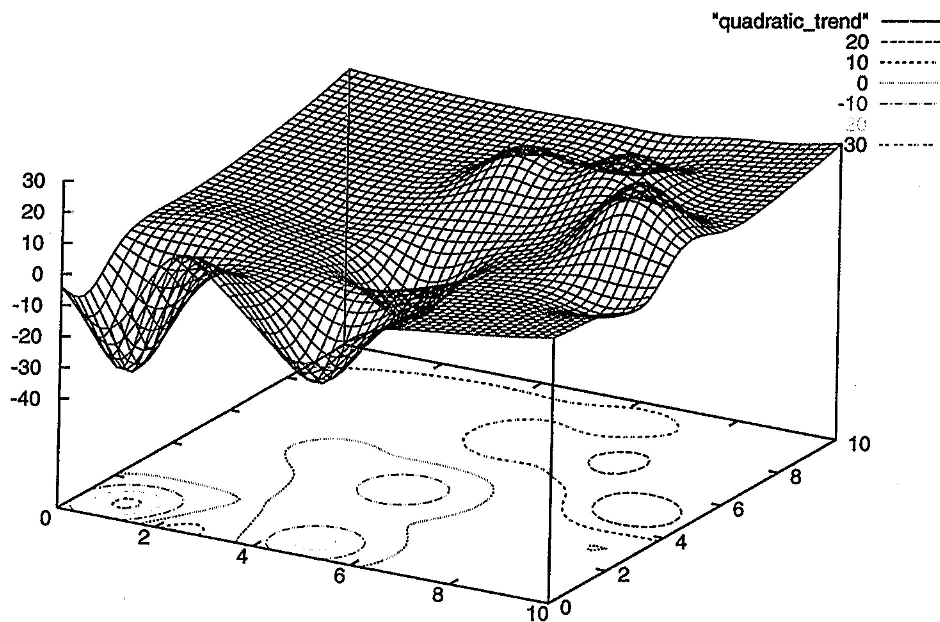


Figure 4.1: Two krigifier objective functions with the similar input parameters. Above: A function with a constant trend. Below: A function with a quadratic trend.

For our purposes, we consider objective functions of only dimension 2, 3, 4, or 5. We have also bounded the feasible region; the krigifier functions are only defined on a range from 0 to 10 for each variable in the domain. We handle infeasible points by returning a function value of infinity for any x with at least one component less than 0 or greater than 10. Thus we define our objective function F such that for a given dimension n, with $x \in \mathbb{R}^n$,

$$F(x) = \begin{cases} f(x), & \text{if } 0 \leq x_i \leq 10; \text{ for all } i = 1, ..., n, \\ \infty, & \text{otherwise} \end{cases} \tag{4.1}$$

where $f(x)$ denotes the value for any feasible x returned by a given realization of f produced by the krigifier. We set the value of $F(x)$ to infinity for any infeasible x so that during testing, if any simplex vertex is located within this infeasible region, the search will immediately throw it out as a possible minimum, since we start with at least one feasible vertex in the initial simplex. This will force the best vertex of every simplex to remain within in the feasible region.

For each of the four dimensions considered, we look at five different constant trends and their five corresponding quadratic trends. For a given dimension, a constant trend function with its corresponding quadratic trend function is called one realization of the dimension. We therefore consider five different realizations for each dimension. Since we are looking at four dimensions, we have a total of 20 realizations, or 40 total objective functions.

4.1.2 Description of the Tests

We consider five search algorithms for testing: two versions of the Spendley, Hext, and Himsworth algorithm, two versions of the Nelder-Mead algorithm, and one version of the Sequential Multi-Directional Search algorithm. The two variants of the Spendley, Hext, and Himsworth algorithms are as described in section 2.1.2. In particular, recall that for a given dimension n, one variant will shrink the simplex if the age of any simplex point exceeds $2(n+1)$, while the other variant will shrink the simplex if the age of any simplex point exceeds $(n+1)$. Our two variants of the Nelder-Mead simplex algorithms are almost identical; however, one variant is as described by Avriel [1] and thus is subject to the anomaly described in Chapter 3, while the other variant, due to [6] and discussed in section 2.2.2, is not subject to the anomaly described in Chapter 3. Finally, the Sequential Multi-Directional Search algorithm is as described in section 2.3.2.

Each of the five algorithms was run 1,000 times, with a different randomly generated starting point, on each of the 40 objective functions. For each of the 1,000 starting points, we ran tests with each of four different function call budgets: 20, 30, 40, and 50. Each run would count the number of function calls made for the run thus far, and if at any time the given limit on the function budget was reached, the particular run would stop, and the search would return its current best function value. While it is likely to assume that the algorithms will not get highly accurate results with a 20 function call budget on a 5-dimensional objective function, using these budgets we can draw some preliminary conclusions as to which algorithms tend

to perform better for more or less equivalent amounts of computational work. (For the simplex searches, function evaluations are the dominant computational cost.)

For each function, each run began by starting with a fixed-length right-angled simplex. By fixed-length, we mean that there is a primary vertex, and every other simplex vertex is at the same fixed distance away from the primary vertex. The starting vertex locations for the initial simplex of each run were chosen in the following manner. One primary vertex for the simplex was chosen by a pseudo-random number generator [12]. (Actually, the random number generator generates a number between 0 and 10 for each component of the initial point: a 4-dimensional point will require generating 4 random numbers.) The remaining vertices for the initial simplex were found by stepping a distance of 2.0 from the primary vertex in each of the positive coordinate directions. This completely determines $(n+1)$ initial vertices, thus creating a fixed-length right-angled simplex as desired.

Again, to allow for comparisons across the five algorithms, the starting positions of the simplices were random within a given algorithm, but uniform across the algorithms. In other words, the starting simplex location for the first test was the same for every algorithm on every objective function, and so on. If results are obtained for a given test run of the algorithms for the same objective function, it is safe to assume during further examination that the algorithms all started with the same initial simplex.

4.1.3 The Function Minima Density Plots

Upon the termination of a given run for a given algorithm, the algorithm returns the best function value it has found thus far. Note that for these tests, only the *value* of the best point is returned, not the location. Therefore, for a given function budget on a given objective function, each algorithm records 1,000 function values, each representing the best result for a given run. These 1,000 values are then combined to make a density plot, which is briefly described below. One density plot represents 1,000 runs by a given algorithm on a given objective function with a given budget. Figure 4.2 shows three examples of density plots.

Above each density plot is a title associating it with a given testing scenario. The "2-D" states that this is a density plot for a 2-dimensional objective function. The other dimensions would be written in a similar fashion. Next is "Realization 2c." As mentioned above, there are two trends, constant and quadratic, and there are five realizations of these two trends for each dimension. The "2c" means this is the second realization for the dimension, constant trend. Quadratic trends would be designated with a "q."

Next is the name of the algorithm that generated the results which are represented by the density plot. The two variants of the Spendley, Hext and Himsworth algorithm are noted by "Spendley Hext & Himsworth" followed by either "2(n+1)" or "n+1," depending on the choice of shrinking criterion. For the two variants of the Nelder-Mead algorithm, both are prefaced with "Nelder Mead", however the variant based on Avriel's description has the suffix "With Equal Sign Ambiguities," while the variant due to Lagarias, Reeds, Wright, and Wright has the suffix "No Ambiguities." The

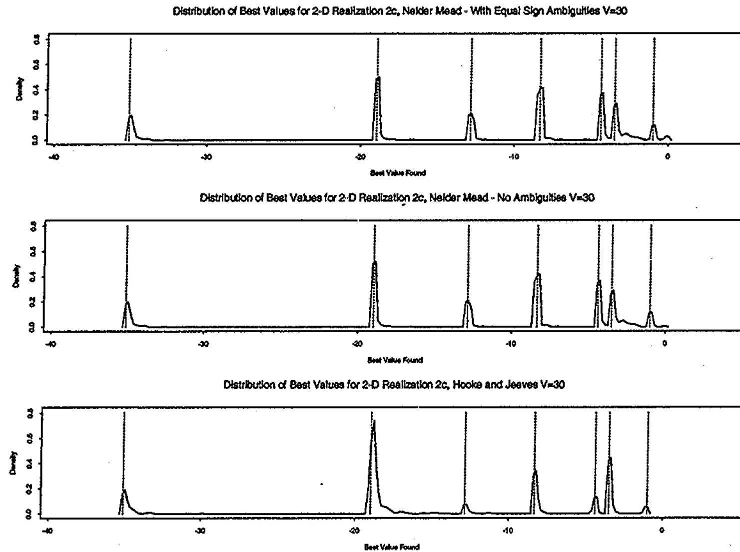


Figure 4.2: Three examples of density plots.

Sequential Multi-Directional Search is labeled as such. Finally, we have density plots for the Hooke and Jeeves pattern search algorithm [4] to compare against the simplex searches.

The final piece of information on the title bar says "V=30." The value of V indicates that the function call budget was set at 30 for this particular example. Other budget values are noted in a similar fashion.

Looking now at the plot itself, there are segmented vertical lines representing the function values of the known minima of the given objective function, which run along the bottom of the plot. In this example, we see that the function has seven known minima, with the global minimum having a function value of approximately −35. Note that the location of these minima were actually determined by Siefert [16] through empirical means by running 1,000 pattern searches on each objective function until the searches converged.

As mentioned above, one plot represents 1,000 runs from different starting points, each of which returns the lowest function value found. If a given function value is returned by a search, it is represented on the density plot by drawing a small Gaussian curve at that value. This is done for each of the 1,000 solutions. The Gaussian curves are then added together via superposition to yield the overall curve shown in the plot. Looking at the first example, there are clear peaks around the function minima lines, indicating that the search was able to reach the vicinity of the true function minima. The highest peak indicates the most likely convergence location, which in this case is the minimum around −19. Note that these plots are *not* histograms due to the Gaussian nature of the curves used to build them.

These plots are a useful way to compare the general performance of our search algorithms, and all of the analysis in the next section is based on them. Again, remember that these plots contain data on function values only and say nothing about the location of the points which generate these values. For more information

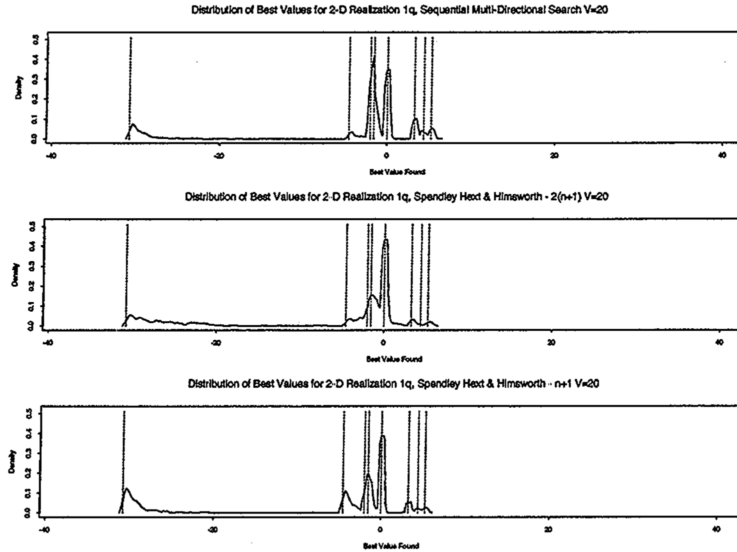


Figure 4.3: The lower two density plots show the advantage of the $(n+1)$ shrinking criterion over the $2(n+1)$ criterion for the Spendley, Hext and Himsworth algorithm.

about these type of plots, as well as the scripts used to build them from the data, see [20].

4.2 General Comparison of the Search Methods

This section contains general trends we noted while examining the density plots for all of the tests. It is important to note that these trends are solely the result of observation, and should therefore be looked upon as suggestions for future investigations, rather than as definitive evidence.

First, we address the difference between our two proposed shrinking criteria for Spendley, Hext, and Himsworth: $2(n+1)$ and $(n+1)$. For this, consider the two sets of density plots shown in Figures 4.3 and 4.4. From these, and similar results, we conclude that a shrinking criterion of age exceeding $2(n+1)$ is too conservative. A criterion of $(n+1)$ should be sufficient and should approach a minimum function value significantly faster than the $2(n+1)$ case, which appears to be wasting too many function calls before shrinking. Returning to Figures 4.3 and 4.4 we can see that both versions of Spendley, Hext, and Himsworth are likely to return a similar distribution of function values. However, the $(n+1)$ case appears to be identifying the function values at the known minimizers much faster, and it has only used 20 function calls. In Figure 4.4, after 50 function calls to a 4-dimensional function, the peaks of the $(n+1)$ case shows clear signs of identifying the function values associated with the known minimizers, while the $2(n+1)$ case has only achieved a relatively even distribution of values and it not very useful at this budget level.

Next we consider the issue of performance across the algorithms. While it is not possible to make a definitive observation with only the density plots, the following

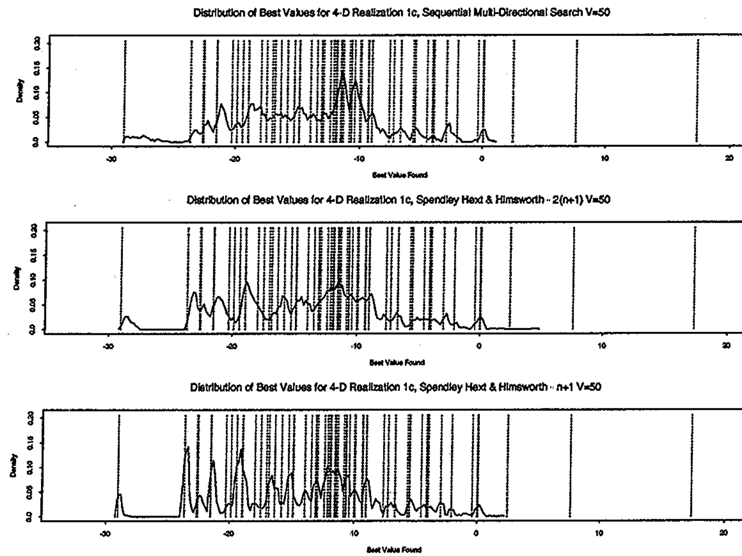


Figure 4.4: An example similar to Figure 4.3, but for a different objective function.

example represents a general trend observed across our tests of the simplex searches. Consider the six density plots in Figure 4.5. Sequential Multi-Directional Search and Hooke and Jeeves have much shorter "tails" in the upper range for the function value than either of the Spendley, Hext and Himsworth or Nelder-Mead algorithms. This suggests that given a finite budget of function calls, the former two algorithms are more likely to identify the function values associated with the known minimizers, while the latter algorithms appear to be much slower to arrive at similar conclusions. This is interesting because Hooke and Jeeves is a pattern search, and although Sequential Multi-Directional Search uses follows a simplex method, it is also a pattern search. This would suggest that given a fixed budget, patterns searches may be more effective than the simplex searches.

Having made the previous observation, we now consider what happens as we increase the "budget" for the number of function evaluations allowed. Figures 4.6 and 4.7 show density plots of the six algorithms on the same 3-dimensional objective function with function call budgets of 20 and 50 respectively. The question we would like to answer is: Given enough function calls, which, if any, of the algorithms will identify the known minimizers? Our density plots for the preliminary tests we ran do not provide a definitive answer, but they invite conjecture. Looking first at Figure 4.6, it is clear that all of the algorithms do not yet have sufficient information to allow a proper identification. This is evident from the long "tails" at the right end of each plot (though, once again, we see a clear advantage enjoyed by the pattern searches). Looking now at Figure 4.7, we see that for the most part, all of the algorithms are much closer to identifications, with greatly reduced tails to the right. Avriel's version of Nelder-Mead, however (the upper-most density plot), has not decreased the length of its tail at all, suggesting that the algorithm may be prematurely stuck at a nonstationary point.

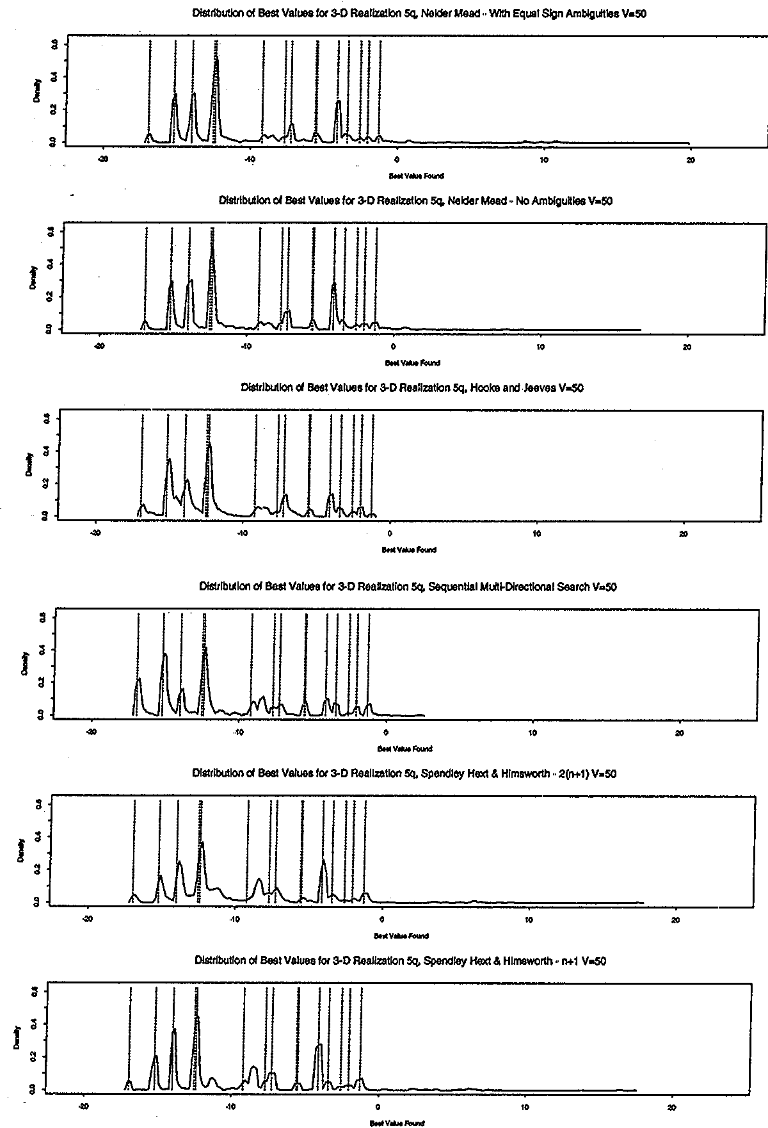


Figure 4.5: These density plots representing each algorithm suggest that the pattern searches show more satisfactory results than the simplex searches.

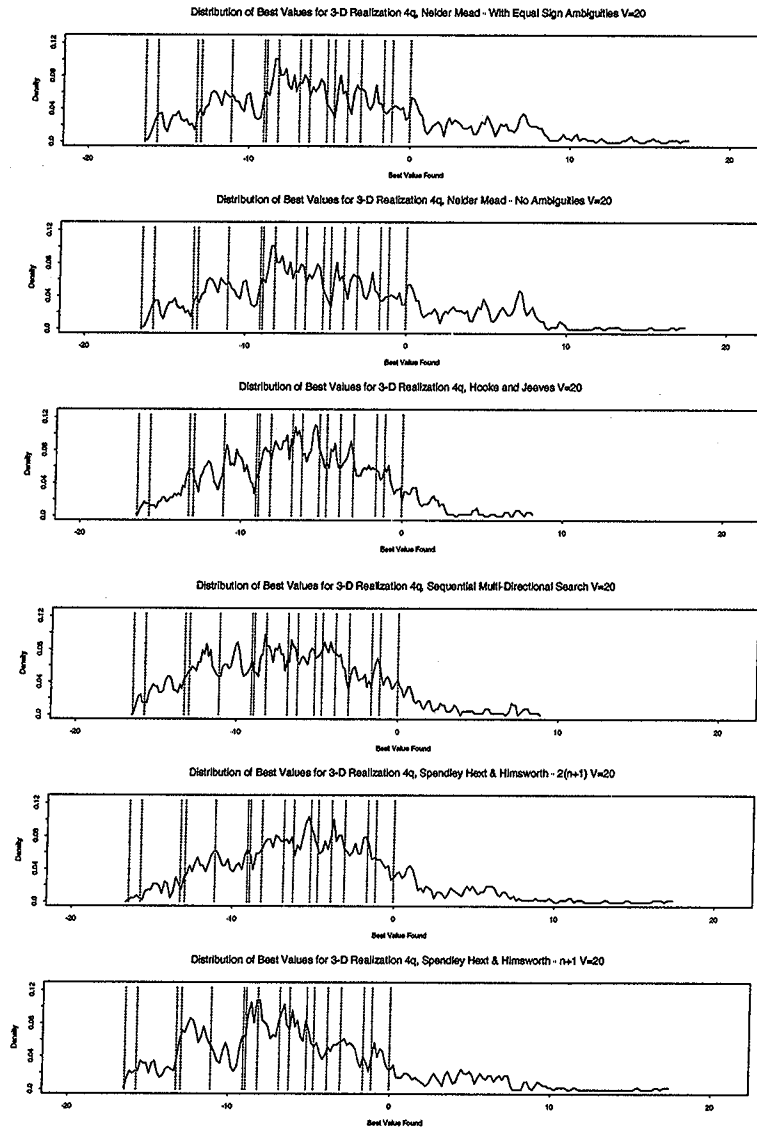


Figure 4.6: Density plots representing each algorithm on a 3-dimensional function given a budget of 20.

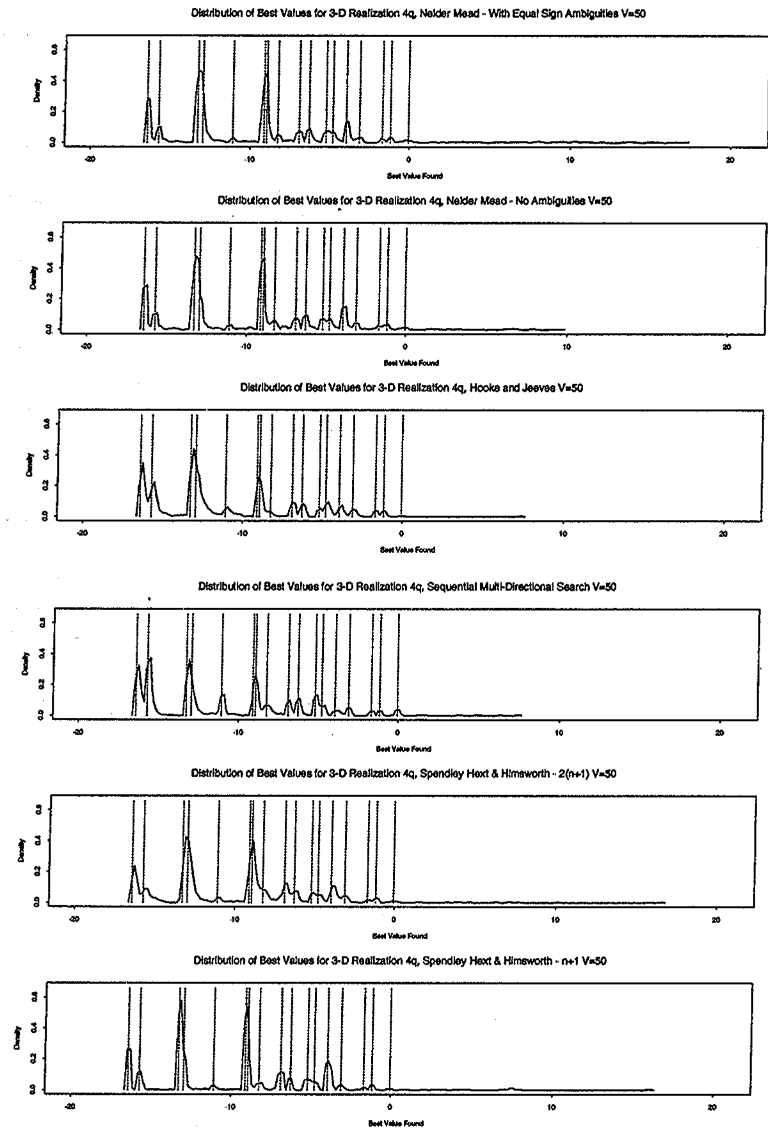


Figure 4.7: Density plot representing the same objective function as Figure 4.6, but with a budget of 50.

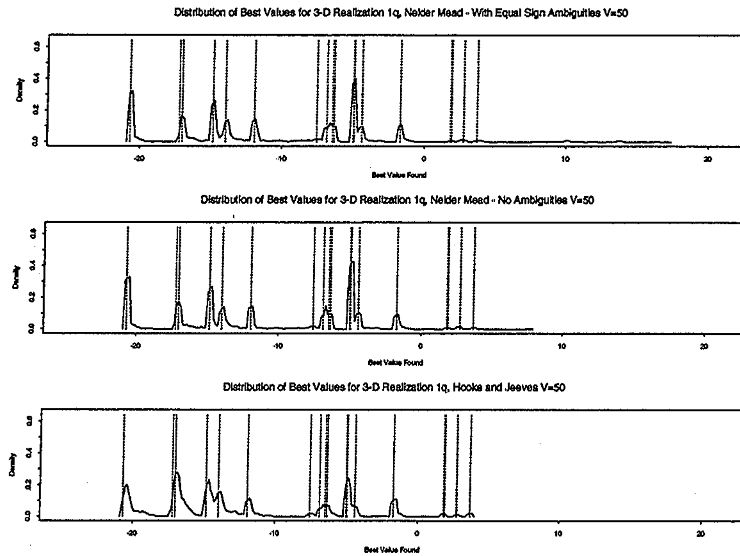


Figure 4.8: These plots show the clear advantage of the Nelder-Mead implementation based on Lagarias, Reeds, Wright, and Wright.

4.3 Visual Results of Nelder-Mead Improvement

In this section we present some results to demonstrate the effectiveness of the corrections made to the Nelder-Mead algorithm discussed in Chapter 3, where we noted that ambiguities in the original specification for the algorithm could lead to situations in which the simplex collapses. This collapses forces the search to return a nonstationary point near the boundary of the feasible region. However, the Lagarias, Reeds, Wright, and Wright description avoids this pitfall.

It would be a good assumption that these results should be apparent in the density plots for our two implementations of Nelder-Mead. This was, in fact, the case, for practically all of the problems tested. Two examples are shown in Figures 4.8 and 4.9. The difference between the algorithms is clearly shown by the extreme tails to the right ends of the density plots for the implementation that does not resolve the ambiguities. Similar tails are not visible in the density plots for the implementation that does resolve the ambiguities. As mentioned in Chapter 3, a few carefully placed equal signs can have a significant impact on the reliability of the Nelder-Mead algorithm.

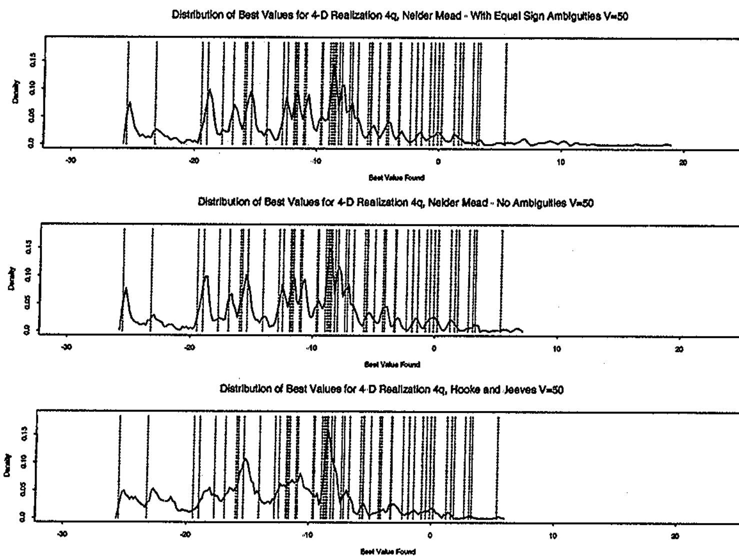


Figure 4.9: Another example of the clear advantage of the Nelder-Mead implementation based on Lagarias, Reeds, Wright, and Wright, over that described by Avriel.

Chapter 5

Conclusion

We have presented three popular algorithms used for nonlinear function optimization. The common folklore for these algorithms, particularly the very popular algorithm of Nelder and Mead, is that they are fairly straightforward and simple to implement. We have shown that, in reality, this is not the case; great care must be taken to implement these algorithms.

For each of the three simplex algorithms, we have given thorough modern algorithmic descriptions. For the simplex search of Spendley, Hext, and Himsworth, we finalized a well-defined description based on the incorporation of point age for use in the shrinking criterion, addressed the removal of simplex oscillation, and presented various versions of stopping criteria. We presented a description of the Nelder-Mead simplex algorithm that would prevent certain instances of unnecessary simplex collapse. We also created and described a successful sequential version of Torczon's Multi-Directional Search (MDS), explaining the methods used to reformulate MDS, which was originally designed to be executed on multiple processors, for use with single processor machines.

We carefully examined the Nelder-Mead simplex algorithm, an extremely popular non-derivative-based search method. We explained the importance of clarifying small ambiguities regarding the effect the placement of tests for equalities have on simplex contraction. We also examined four popular Nelder-Mead implementations in use today to determine that two did not prevent premature simplex collapse of the sort we identified. The success of the Lagarias, Reeds, Wright, and Wright variant of the algorithm [6] over the Avriel variant of the algorithm [1] was verified in the function minima density plots. We also used the density plots to make some preliminary general observations regarding the overall performance of these searches, both against variants of one another, as well as against one of the classic pattern search methods.

Finally, we have implemented all three simplex searches using C++ classes. These classes, which will be made publicly available, provide an easy means of use. Further, they will enable additional research into the behavior of the simplex searches.

While many questions have been answered, this research has provided many suggestions for further experimentation. All of the trends across the algorithms that were discussed were based solely on the density plots. Utilizing the actual data used to create the plots, it should be possible to provide a more quantitative analysis of

the overall behavior of these algorithms. Additional work could be accomplished regarding the local behavior of these searches in the vicinity of a minimizer. This would suggest analyzing their algorithmic efficiency using simple quadratic objective functions, which would provide a description of their success in converging to an isolated point.

It was also mentioned that the simplex searches were more likely to locate function minima located in the corners of the feasible region. This suggests a further study of the behavior of these algorithms along the boundaries of the feasible region. This should encompass situations where minimizers were either near or at the boundary.

After preliminary research in all of these areas has been achieved, it would also be interesting to compare and contrast in a quantitative fashion the behavior of simplex searches against pattern searches. Many of these algorithms are available for use, but there is little along the lines of suggestion for choosing a given search algorithm over another for a specific objective function.

There is a general desire to use these simplex searches for research purposes, but many find that there is a great deal of literature to wade through, and that there is no one place to look for descriptions, implementations, and suggestions on how to proceed. We feel that our research has provided a clear and consistent description for the simplex searches. This should help to make the simplex searches easier to use. Further, this should expedite new research projects in the field, as well as allow for a better interpretation of research that has already been completed.

Appendix A

The C++ Code

This chapter contains the C++ code for the three classes created to execute the simplex search of Spendley, Hext and Himsworth, the simplex search of Nelder and Mead, and the Sequential Multi-Directional Search. The code requires special matrix definitions defined in the header files `vec.h`, `cmat.h`, and `subscrpt.h`. These files require no modification by the user and are explained to a greater extent in [16].

This code implements general-use simplex search algorithms for either unconstrained or bound constrained problems. It is therefore possible to run these searches on any choice of nonlinear objective functions. It is up to the user to define the objective function on which a search is to be run. The objective function should be defined by the user in `objective.cc` in the function

```
fcn(int dimension, double *point, double& value, int& flag),
```

where `dimension` contains the dimension of the search; `point` is a pointer to an array of doubles of size `dimension`, which represents the point at which the function is to be evaluated; `value` will contain f(`point`) upon the function's termination; and `flag` is 1 if the function evaluation was successful, 0 otherwise.

The user may also choose to modify the respective header file: `objective.h`. While it is possible to define the objective function directly within `objective.cc`, that would require recompiling the source code every time a search were to be used on a new function. Therefore, the suggested method of defining the objective function, as well as the method used in this research, is to compile the objective function separately as its own binary. Then `fcn()` would simply execute this separate binary every time a function evaluation occurs. If this method is used, changing the objective function simply requires the creation of a new function binary.

Our tests used objective functions created by a function generator know as the krigifier [19]. More information on the krigifier can be found in [11]. Further information on the scripts used to incorporate the objective functions into the search code can be found in [16].

For all of the three searches, the code is meant to be used in three stages. First, the user should call one of the several initialization functions provided to set up an initial simplex. Then, the user should call the `ExploratoryMoves()` function to execute the actual simplex search. Finally, any of the query functions can be used to retrieve the results of a search. If at any time the user would like to start over with a

new initial simplex, this can be achieved by re-calling the initialization routines and running through the cycle described above again.

A.1 C++ Code for the Simplex Search of Spendley, Hext, and Himsworth

A.1.1 SHH - Header File

```
/*SHHSearch.h
 *declarations of Spendley, Hext and Himsworth Simplex Search
 *Adam Gurson College of William & Mary 1999
 */

#ifndef _SHHSearch_
#define _SHHSearch_

#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>
#include "objective.h"
#include "vec.h"
#include "cmat.h"
#include "subscrpt.h"


#define stoppingStepLength 10e-8 /*sqrt(fabs(3.0 * (4.0/3.0 - 1.0) - 1.0))*/
#ifndef DEBUG
#define DEBUG 0
#endif
//#ifndef maxCalls
//#define maxCalls 200
//#endif


class SHHSearch
{
 public:
 // constructors and destructors

   SHHSearch(int dim);
   // default constructor

   SHHSearch(int dim, double Sigma);
   // constructor which allows shrinking coefficient initialization

   SHHSearch(const SHHSearch& Original);
   // deep copy constructor
```

```
  ~SHHSearch();
  // destructor

// algorithmic routines

  void ExploratoryMoves();
  // use Spendley, Hext and Himsworth to find function minimum

  void ReplaceSimplexPoint(int index, const Vector<double>& newPoint);
  // replaces simplex point indexed at index with newPoint

  void CalculateFunctionValue(int index);
  // finds the f(x) value for the simplex point indexed at index and
  // replaces the proper value in simplexValues

  void SetSigma(double newSigma);
  // allows the user to set a new value for
  // the shrinking coefficient

  bool Stop();
  // returns true if the stopping criteria have been satisfied

  void fcnCall(int n, double *x, double& f, int& flag);
  // indirection of function call for purposes of keeping an accurate
  // tally of the number of function calls

// Simplex-altering functions

  void InitRegularTriangularSimplex(const Vector<double> *basePoint,
                                    const double edgeLength);
  // deletes any existing simplex and replaces it with a regular
  // triangular simplex in the following manner:
  //
  // basePoint points to a point that will be the "origin" of the
  //    simplex points (it will be a part of the simplex)
  // edgeLength is the length of each edge of the "triangle"
  //
  // functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.
  //
  // NOTE: basePoint is assumed to be of proper dimension

  void InitFixedLengthRightSimplex(const Vector<double> *basePoint,
                                   const double edgeLength);
  // deletes any existing simplex and replaces it with a right-angle
```

```
    // simplex in the following manner:
    //
    // basePoint points to a point that will be the "origin" of the
    //    simplex points (it will be a part of the simplex)
    // edgeLength is to be the length of each simplex side extending
    //    from the basePoint along each positive coordinate direction.
    //
    // functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.
    //
    // NOTE: basePoint is assumed to be of proper dimension

    void InitVariableLengthRightSimplex(const Vector<double> *basePoint,
                                        const double* edgeLengths);
    // deletes any existing simplex and replaces it with a right-angle
    // simplex in the following manner:
    //
    // basePoint points to a point that will be the "origin" of the
    //    simplex points (it will be a part of the simplex)
    // edgeLengths points to an array of n doubles, where n is the
    //    dimension of the given search. x_1 will then be located
    //    a distance of edgeLengths[0] away from the basepoint along the
    //    the x_1 axis, x_2 is edgeLengths[1] away on the x_2 axis, etc.
    //
    // functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.
    //
    // NOTE: basePoint and edgeLengths are assumed to be of proper dimension

    void InitGeneralSimplex(const Matrix<double> *plex);
    // deletes any existing simplex and replaces it with the one
    // pointed to by plex
    //
    // functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.
    //
    // NOTE: THIS ASSUMES THAT plex IS OF PROPER DIMENSION

    void ReadSimplexFile(istream& fp);
    // may also pass cin as input stream if desired
    // input the values of each trial point
    // NOTE: THIS FUNCTION WILL ONLY ACCEPT n+1 POINTS
    //
    // functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.

// Query functions

  int GetFunctionCalls() const;
```

```
    // number of objective function evaluations

    void GetMinPoint(Vector<double>* &minimum) const;
    // simplex point which generates the best objective function
    // value found thus far
    // USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

    double GetMinVal() const;
    // best objective function value found thus far

    void GetCurrentSimplex(Matrix<double>* &plex) const;
    // performs a deep copy of the simplex to a Matrix pointer
    // points to a newly allocated chunk of memory upon return
    // USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

    void GetCurrentSimplexValues(double* &simValues) const;
    // performs a deep copy of the simplexValues array to a double pointer
    // points to a newly allocated chunk of memory upon return
    // USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

    void GetCurrentSimplexAges(double* &simAges) const;
    // performs a deep copy of the simplexAges array to a double pointer
    // points to a newly allocated chunk of memory upon return
    // USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

    int GetVarNo() const;
    // returns the number of dimensions

    int GetTolHit() const;
    // returns toleranceHit

    void printSimplex() const;
    // prints out the simplex points by row, their corresponding f(x)
    // values, and the number of function calls thus far

private:

    void FindMinReplacementIndices(int replacementSkipIndex);
    // sets minIndex to the simplex index of the point which generates
    // the lowest value of f(x)
    // sets replacementIndex to the simplex index of the point which
    // generates the highest value of f(x) excluding the point at
    // maxSkipIndex
    // if maxSkipIndex is set to a valid simplex index, the
    // replacement search will skip over that index during its search
```

```
    // this is used to prevent the simplex from getting stuck
    // in a "back and forth" infinite loop

    void FindCentroid();
    // finds the centroid

    void FindReflectionPt();
    // finds the reflection point and sets its f(x) value

    void ShrinkSimplex();
    // this function goes through the simplex and reduces the
    // lengths of the edges adjacent to the best vertex

    int AgesTooOld();
    // determines when to shrink the simplex based on
    // how old the individual simplex points are
    // returns 1 if true and a shrink should occur, 0 otherwise

    void UpdateAges(int newIndex);
    // increments the ages of all simplex points EXCEPT the point
    // with index newIndex, which gets an age of 1

    void ResetAges();
    // resets all simplex point ages to 1


    int dimensions;                    // the number of dimensions
                                       //    (the dimension of the problem)
    Matrix<double> *simplex;           // the current simplex
    double *simplexValues;             // their corresponding f(x) values
    double *simplexAges;               // their corresponding ages
    double sigma;                      // shrinking coefficient
    int minIndex;                      // index of point generating min f(x)
    int replacementIndex;              // index of point to be replaced
    Vector<double> *centroid;          // the current centroid
    Vector<double> *reflectionPt;      // the reflection point
    double reflectionPtValue;          // the value of f(reflectionPt)
    long functionCalls;                // tally of number of function calls
    int toleranceHit;                  // 1 if stop due to tolerance, 0 if funcCalls

    // the following vectors are simply extra storage space
    // that are used by functions that require a vector of
    // size = dimensions
    Vector<double> *scratch, *scratch2;
};
```

```
#endif
```

A.1.2 SHH - ExploratoryMoves()

```
void SHHSearch::ExploratoryMoves()
{
   toleranceHit = 0;

   replacementIndex = -1;
   do {
     FindMinReplacementIndices(replacementIndex);
     if(DEBUG) printSimplex();

     // if any point has been here for a significantly long
     // time, the simplex is most likely circling a local
     // minimum, so shrink the simplex
     if( AgesTooOld() ) {
       ShrinkSimplex();
       ResetAges();
       FindMinReplacementIndices(-1);
       if(DEBUG) printSimplex();

       // stop if at maximum function calls
       if (functionCalls >= maxCalls) {
         FindMinReplacementIndices(-1);
         return;
       }
     } // if

     FindCentroid();
     FindReflectionPt();
     ReplaceSimplexPoint(replacementIndex, *reflectionPt);
     simplexValues[replacementIndex] = reflectionPtValue;
     UpdateAges(replacementIndex);
   } while (!Stop());    // while stopping criteria is not satisfied
   FindMinReplacementIndices(-1);
} // ExploratoryMoves()
```

A.1.3 SHH - Constructors and Destructor

```
SHHSearch::SHHSearch(int dim)
{
   dimensions = dim;
   functionCalls = 0;
```

```
    minIndex = 0;
    simplex = NULL;
    simplexValues = NULL;
    simplexAges = NULL;
    centroid = new Vector<double>(dimensions,0.0);
    reflectionPt = new Vector<double>(dimensions,0.0);
    sigma = 0.5;
    scratch = new Vector<double>(dimensions,0.0);
    scratch2 = new Vector<double>(dimensions,0.0);
} // SHHSearch() (default)

SHHSearch::SHHSearch(int dim, double Sigma)
{
    dimensions = dim;
    functionCalls = 0;
    minIndex = 0;
    simplex = NULL;
    simplexValues = NULL;
    simplexAges = NULL;
    centroid = new Vector<double>(dimensions,0.0);
    reflectionPt = new Vector<double>(dimensions,0.0);
    sigma = Sigma;
    scratch = new Vector<double>(dimensions,0.0);
    scratch2 = new Vector<double>(dimensions,0.0);
} // SHHSearch() (special)

SHHSearch::SHHSearch(const SHHSearch& Original)
{
    dimensions = Original.GetVarNo();
    Original.GetCurrentSimplex(simplex);
    Original.GetCurrentSimplexValues(simplexValues);
    Original.GetCurrentSimplexAges(simplexAges);
    sigma = Original.sigma;
    minIndex = Original.minIndex;
    replacementIndex = Original.replacementIndex;
    if(centroid != NULL) delete centroid;
    centroid = new Vector<double>(*(Original.centroid));
    if(reflectionPt != NULL) delete reflectionPt;
    reflectionPt = new Vector<double>(*(Original.reflectionPt));
    reflectionPtValue = Original.reflectionPtValue;
    functionCalls = Original.functionCalls;
} // SHHSearch() (copy constructor)

SHHSearch::~SHHSearch()
{
```

```
  if(simplex != NULL) delete simplex;
  if(simplexValues != NULL) delete [] simplexValues;
  if(simplexAges != NULL) delete [] simplexAges;
  delete centroid;
  delete reflectionPt;
  delete scratch;
  delete scratch2;
  //NOTE: Matrix and Vector classes have their own destructors
} // ~SHHSearch
```

A.1.4 SHH - Simplex Initialization Routines

```
void SHHSearch::InitRegularTriangularSimplex(const Vector<double> *basePoint,
                                              const double edgeLength)
{
  //  This routine constructs a regular simplex (i.e., one in which all of
  //  the edges are of equal length) following an algorithm given by Jacoby,
  //  Kowalik, and Pizzo in "Iterative Methods for Nonlinear Optimization
  //  Problems," Prentice-Hall (1972).  This algorithm also appears in
  //  Spendley, Hext, and Himsworth, "Sequential Application of Simplex
  //  Designs in Optimisation and Evolutionary Operation," Technometrics,
  //  Vol. 4, No. 4, November 1962, pages 441--461.

   int i,j;
   double p, q, temp;
   Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions,0.0);
   for( int col = 0; col < dimensions; col++ ) {
      (*plex)[0][col] = (*basePoint)[col];
   }

   temp = dimensions + 1.0;
   q = ((sqrt(temp) - 1.0) / (dimensions * sqrt(2.0))) * edgeLength;
   p = q + ((1.0 / sqrt(2.0)) * edgeLength);

   for(i = 1; i <= dimensions; i++) {
      for(j = 0; j <= i-2; j++) {
         (*plex)[i][j] = (*plex)[0][j] + q;
      } // inner for 1
      j = i - 1;
      (*plex)[i][j] = (*plex)[0][j] + p;
      for(j = i; j < dimensions; j++) {
            (*plex)[i][j] = (*plex)[0][j] + q;
      } // inner for 2
  } // outer for
```

```
    InitGeneralSimplex(plex);
    delete plex;
} // InitRegularTriangularSimplex()

void SHHSearch::InitFixedLengthRightSimplex(const Vector<double> *basePoint,
                                             const double edgeLength)
{
  // to take advantage of code reuse, this function simply turns
  // edgeLength into a vector of dimensions length, and then
  // calls InitVariableLengthRightSimplex()

   double* edgeLengths = new double[dimensions];
   for( int i = 0; i < dimensions; i++ ) {
      edgeLengths[i] = edgeLength;
   }
   InitVariableLengthRightSimplex(basePoint,edgeLengths);
   delete [] edgeLengths;
} // InitFixedLengthRightSimplex()

void SHHSearch::InitVariableLengthRightSimplex(const Vector<double> *basePoint,
                                                const double* edgeLengths)
{
   Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions,0.0);
   for( int i = 0; i < dimensions; i++ ) {
      // we're building the basePoint component-by-component into
      // the (n+1)st row
      (*plex)[dimensions][i] = (*basePoint)[i];

      // now fill in the ith row with the proper point
      for( int j = 0; j < dimensions; j++ ) {
         (*plex)[i][j] = (*basePoint)[j];
         if( i == j )
            (*plex)[i][j] += edgeLengths[i];
      }
   } // for
   InitGeneralSimplex(plex);
   delete plex;
} // InitVariableLengthRightSimplex()

void SHHSearch::InitGeneralSimplex(const Matrix<double> *plex)
{
   functionCalls = 0;
   if( simplex != NULL ) { delete simplex; }
   if( simplexValues != NULL ) { delete [] simplexValues;}
   simplex = new Matrix<double>((*plex));
```

```
    simplexValues = new double[dimensions+1];
    simplexAges = new double[dimensions+1];
    ResetAges();

    int success;
    for( int i = 0; i <= dimensions; i++ ) {
       *scratch = (*plex).row(i);
       fcnCall(dimensions, (*scratch).begin(), simplexValues[i], success);
       if(!success) cerr<<"Error with point #"<<i<<" in initial simplex.\n";
    } // for
    FindMinReplacementIndices(-1);
} // InitGeneralSimplex()
```

A.1.5 SHH - Other Unique Functions

```
void SHHSearch::ReadSimplexFile(istream& fp)
{
   if(fp == NULL) {
      cerr<<"No Input Stream in ReadSimplexFile()!\n";
      return; // There's no file handle!!
   }

   Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions);
   for( int i = 0; i <= dimensions; i++ ) {
      for ( int j = 0; j < dimensions; j++ ) {
         fp >> (*plex)[i][j];
      } // inner for
   } // outer for
   InitGeneralSimplex(plex);
   delete plex;
} // ReadSimplexFile()

bool SHHSearch::Stop()
{
   if(maxCalls > -1) {
      if(functionCalls >= maxCalls)
         return true;
   }

   double mean = 0.0;

   for( int i = 0; i <= dimensions; i++) {
      if( i != minIndex ) {
         mean += simplexValues[i];
      } // if
```

```
    } //for

    mean /= (double)dimensions;

    // Test for the suggested Nelder-Mead stopping criteria
    double total = 0.0;
    for( int i = 0; i <= dimensions; i++ ) {
       total += pow( simplexValues[i] - mean ,2);
    } //for
    total /= ((double)dimensions + 1.0);
    total = sqrt(total);

    if(total < stoppingStepLength) {
       toleranceHit = 1;
       return true;
    }
    else
       return false;
} // Stop()

void SHHSearch::GetMinPoint(Vector<double>* &minimum) const
{
   minimum = new Vector<double>((*simplex).row(minIndex));
} // GetMinPoint()

double SHHSearch::GetMinVal() const
{
   return simplexValues[minIndex];
} // GetMinVal()

void SHHSearch::GetCurrentSimplexAges(double* &simAges) const
{
   simAges = new double[dimensions+1];
   for( int i = 0; i <= dimensions; i++ ) {
      simAges[i] = simplexAges[i];
   } // for
} // GetCurrentSimplexAges()

void SHHSearch::FindMinReplacementIndices(int replacementSkipIndex)
{
   if(simplexValues == NULL) {
      cerr << "Error in FindMinReplacementIndices() - "
           << "The vector of simplexValues is NULL!!\n";
      return;
   }
```

```
    int newMinIndex = 0;
    replacementIndex = 0;
    double min = simplexValues[0];
    double replaceVal = simplexValues[0];
    if (replacementSkipIndex == 0) {
      replacementIndex = 1;
      replaceVal = simplexValues[1];
    }
    for( int i = 1; i <= dimensions; i++ ) {
       if( simplexValues[i] < min ) {
          min = simplexValues[i];
          newMinIndex = i;
       } // if
       if( (i != replacementSkipIndex) && (simplexValues[i] > replaceVal) ) {
          replaceVal = simplexValues[i];
          replacementIndex = i;
       } // if
    } // for
    if (simplexValues[newMinIndex] < simplexValues[minIndex]) {
      minIndex = newMinIndex;
      ResetAges();
    }
} // FindMinReplacementIndices()

void SHHSearch::FindCentroid()
{
   (*centroid) = 0.0;
   for( int i = 0; i <= dimensions; i++ ) {
      if( i != replacementIndex ) {
         (*centroid) = (*centroid) + (*simplex).row(i);
      } // if
   } // for
   (*centroid) = (*centroid) * ( 1.0 / (double)dimensions );
} // FindCentroid()

void SHHSearch::FindReflectionPt()
{
   (*reflectionPt) = 0.0;
   (*reflectionPt) = ( (*centroid) * 2.0 ) - (*simplex).row(replacementIndex);
   int success;
   fcnCall(dimensions, (*reflectionPt).begin(), reflectionPtValue, success);
   if(!success) {
      cerr << "Error finding f(x) for reflection point at"
           << "function call #" << functionCalls << ".\n";
   } // if
```

```
} // FindReflectionPt()

void SHHSearch::ShrinkSimplex()
{
   Vector<double> *lowestPt = scratch;
   *lowestPt = (*simplex).row(minIndex);
   Vector<double> *tempPt = scratch2;
   int success;
   for( int i = 0; i <= dimensions; i++ ) {
      if( i != minIndex ) {
         *tempPt = (*simplex).row(i);
         (*tempPt) = (*tempPt) + ( sigma * ( (*lowestPt)-(*tempPt) ) );
         for( int j = 0; j < dimensions; j++ ) {
            (*simplex)[i][j] = (*tempPt)[j];
         } // inner for
         fcnCall(dimensions,(*tempPt).begin(),simplexValues[i],success);
         if (!success) cerr << "Error shrinking the simplex.\n";

         // stop if at maximum function calls
         if (functionCalls >= maxCalls) {return;}

      } // if
   } // outer for
} // ShrinkSimplex()

int SHHSearch::AgesTooOld()
{
  if( simplexAges[minIndex] > (dimensions+1) )
    return 1;
  else
    return 0;
} // AgesTooOld()

void SHHSearch::UpdateAges(int newIndex)
{
  for( int i = 0; i <= dimensions; i++ ) {
    if( i == newIndex )
      simplexAges[i] = 1;
    else
      simplexAges[i]++;
  } // for
} // ResetAges()

void SHHSearch::ResetAges()
{
```

```
    for( int i = 0; i <= dimensions; i++ )
       simplexAges[i] = 1;
} // ResetAges()

void SHHSearch::printSimplex() const
{
  for( int i = 0; i <= dimensions; i++ ) {
     cout << "Point: ";
     for ( int j = 0; j < dimensions; j++ ) {
       cout << (*simplex)[i][j] << " ";
     } // inner for
     cout << "    Value: " << simplexValues[i]
          << "    Age: " << simplexAges[i] << "\n";
  } // outer for

  cout << "\nFCalls: " << functionCalls << "\n\n";
} // printSimplex()
```

A.2 C++ Code for the Simplex Search of Nelder and Mead

A.2.1 NM - Header File

```
/*NMSearch.h
 *declarations of Nelder Mead Simplex Search
 *Adam Gurson College of William & Mary 1999
 */

#ifndef _NMSearch_
#define _NMSearch_

#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>
#include "objective.h"
#include "vec.h"
#include "cmat.h"
#include "subscrpt.h"

//  default definitions: should be defined by user in objective.h,
//  but just in case...
/*#ifndef maxCalls
#define maxCalls -1 // means program will only terminate
```

```
                          // based on stopping criteria
#endif*/
#define stoppingStepLength 10e-8 /*sqrt(fabs(3.0 * (4.0/3.0 - 1.0) - 1.0))*/
#ifndef DEBUG
#define DEBUG 0
#endif

class NMSearch
{
 public:
 // constructors and destructors

   NMSearch(int dim);
   // default constructor

   NMSearch(int dim, double Alpha, double Beta,
                     double Gamma, double Sigma);
   // constructor which allows coefficient initialization

   NMSearch(const NMSearch& Original);
   // deep copy constructor

   ~NMSearch();
   // destructor

// algorithmic routines

   void ExploratoryMoves();
   // use Nelder Mead to find function minimum

   void ReplaceSimplexPoint(int index, const Vector<double>& newPoint);
   // replaces simplex point indexed at index with newPoint

   void CalculateFunctionValue(int index);
   // finds the f(x) value for the simplex point indexed at index and
   // replaces the proper value in simplexValues

   void SetAlpha(double newAlpha);
   void SetBeta(double newBeta);
   void SetGamma(double newGamma);
   void SetSigma(double newSigma);
   // these functions allow the user to set values of the reflection,
   // contraction, expansion, and shrinking coefficients

   bool Stop();
```

```
  // returns true if the stopping criteria have been satisfied

  void fcnCall(int n, double *x, double& f, int& flag);
  // indirection of function call for purposes of keeping an accurate
  // tally of the number of function calls

// Simplex-altering functions

  void InitRegularTriangularSimplex(const Vector<double> *basePoint,
                                    const double edgeLength);
  // deletes any existing simplex and replaces it with a regular
  // triangular simplex in the following manner:
  //
  // basePoint points to a point that will be the "origin" of the
  //    simplex points (it will be a part of the simplex)
  // edgeLength is the length of each edge of the "triangle"
  //
  // functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.
  //
  // NOTE: basePoint is assumed to be of proper dimension

  void InitFixedLengthRightSimplex(const Vector<double> *basePoint,
                                   const double edgeLength);
  // deletes any existing simplex and replaces it with a right-angle
  // simplex in the following manner:
  //
  // basePoint points to a point that will be the "origin" of the
  //    simplex points (it will be a part of the simplex)
  // edgeLength is to be the length of each simplex side extending
  //    from the basePoint along each positive coordinate direction.
  //
  // functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.
  //
  // NOTE: basePoint is assumed to be of proper dimension

  void InitVariableLengthRightSimplex(const Vector<double> *basePoint,
                                      const double* edgeLengths);
  // deletes any existing simplex and replaces it with a right-angle
  // simplex in the following manner:
  //
  // basePoint points to a point that will be the "origin" of the
  //    simplex points (it will be a part of the simplex)
  // edgeLengths points to an array of n doubles, where n is the
  //    dimension of the given search. x_1 will then be located
  //    a distance of edgeLengths[0] away from the basepoint along the
```

```
//    the x_1 axis, x_2 is edgeLengths[1] away on the x_2 axis, etc.
//
// functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.
//
// NOTE: basePoint and edgeLengths are assumed to be of proper dimension

void InitGeneralSimplex(const Matrix<double> *plex);
// deletes any existing simplex and replaces it with the one
// pointed to by plex
//
// functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.
//
// NOTE: THIS ASSUMES THAT plex IS OF PROPER DIMENSION

void ReadSimplexFile(istream& fp);
// may also pass cin as input stream if desired
// input the values of each trial point
// NOTE: THIS FUNCTION WILL ONLY ACCEPT n+1 POINTS
//
// functionCalls is reset to 0 and ALL FUNCTION VALUES ARE CALCULATED.

// Query functions

int GetFunctionCalls() const;
// number of objective function evaluations

void GetMinPoint(Vector<double>* &minimum) const;
// simplex point which generates the best objective function
// value found thus far
// USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

double GetMinVal() const;
// best objective function value found thus far

void GetCurrentSimplex(Matrix<double>* &plex) const;
// performs a deep copy of the simplex to a Matrix pointer
// points to a newly allocated chunk of memory upon return
// USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

void GetCurrentSimplexValues(double* &simValues) const;
// performs a deep copy of the simplexValues array to a double pointer
// points to a newly allocated chunk of memory upon return
// USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

int GetVarNo() const;
```

```
  // returns the dimension of the problem

  int GetTolHit() const;
  // returns toleranceHit

  void printSimplex() const;
  // prints out the simplex points by row, their corresponding f(x)
  // values, and the number of function calls thus far

private:

  void FindMinMaxIndices();
  // sets minIndex to the simplex index of the point which generates
  // the lowest value of f(x)
  // sets maxIndex to the simplex index of the point which generates
  // the highest value of f(x)

  int SecondHighestPtIndex();
  // returns simplex index of the point which
  // generates the second highest value of f(x)

  void FindCentroid();
  // finds the centroid

  void FindReflectionPt();
  // finds the reflection point and sets its f(x) value

  void FindExpansionPt();
  // finds the expansion point and sets its f(x) value

  void FindContractionPt();
  // finds the contraction point and sets its f(x) value

  void ShrinkSimplex();
  // this function goes through the simplex and reduces the
  // lengths of the edges adjacent to the best vertex


  int dimensions;                    // the number of dimensions
                                     //    (the dimension of the problem)
  Matrix<double> *simplex;           // the current simplex
  double *simplexValues;             // their corresponding f(x) values
  double alpha;                      // reflection coefficient
  double beta;                       // contraction coefficient
  double gamma;                      // expansion coefficient
```

```
    double sigma;                      // shrinking coefficient
    int minIndex;                      // index of point generating min f(x)
    int maxIndex;                      // index of point generating max f(x)
    Vector<double> *centroid;          // the current centroid
    Vector<double> *reflectionPt;      // the reflection point
    double reflectionPtValue;          // the value of f(reflectionPt)
    Vector<double> *expansionPt;       // the expansion point
    double expansionPtValue;           // the value of f(expansionPt)
    Vector<double> *contractionPt;     // the contraction point
    double contractionPtValue;         // the value of f(contractionPt)
    double maxPrimePtValue;            // min(f(maxIndexPoint),reflectionPtValue)
    long functionCalls;                // tally of number of function calls
    int toleranceHit;                  // 1 if stop due to tolerance, 0 if funcCalls
    int maxPrimePtId;                  // set by FindContractionPt() and used in
                                       // ExploratoryMoves() to branch in pos. 3

    // the following vectors are simply extra storage space
    // that are used by functions that require a vector of
    // size = dimensions
    Vector<double> *scratch, *scratch2;
};

#endif
```

A.2.2 NM - ExploratoryMoves()

```
void NMSearch::ExploratoryMoves()
{
   double secondHighestPtValue; // used for contraction/reflection decision
   toleranceHit = 0;

   FindMinMaxIndices();
   do {
      if(DEBUG) printSimplex();
      FindCentroid();
      secondHighestPtValue = simplexValues[SecondHighestPtIndex()];
    // reflection step
      FindReflectionPt();

      // stop if at maximum function calls and update the simplex
      if (functionCalls >= maxCalls) {
         FindMinMaxIndices();
         ReplaceSimplexPoint(maxIndex, *reflectionPt);
         simplexValues[maxIndex] = reflectionPtValue;
         FindMinMaxIndices();
```

```
      return;
    }

  // possibility 1

    if(simplexValues[minIndex] > reflectionPtValue) {
      FindExpansionPt(); // expansion step

      if (reflectionPtValue > expansionPtValue) {
        ReplaceSimplexPoint(maxIndex, *expansionPt);
        simplexValues[maxIndex] = expansionPtValue;
      } // inner if
      else {
        ReplaceSimplexPoint(maxIndex, *reflectionPt);
        simplexValues[maxIndex] = reflectionPtValue;
      } // else
    } // if for possibility 1

  // possibility 2

    else if( (secondHighestPtValue > reflectionPtValue         ) &&
             (   reflectionPtValue >= simplexValues[minIndex]) ) {
      ReplaceSimplexPoint(maxIndex, *reflectionPt);
      simplexValues[maxIndex] = reflectionPtValue;
    } // else if for possibility 2

  // possibility 3
    else if( reflectionPtValue >= secondHighestPtValue ) {
      FindContractionPt(); // contraction step
      if(maxPrimePtId == 0) {
        if( contractionPtValue > maxPrimePtValue ) {
          ShrinkSimplex();
        } // inner if
        else {
          ReplaceSimplexPoint(maxIndex, *contractionPt);
          simplexValues[maxIndex] = contractionPtValue;
        } // inner else
      } // maxPrimePtId == 0
      else if(maxPrimePtId == 1) {
        if( contractionPtValue >= maxPrimePtValue ) {
          ShrinkSimplex();
        } // inner if
        else {
          ReplaceSimplexPoint(maxIndex, *contractionPt);
          simplexValues[maxIndex] = contractionPtValue;
```

```
        } // inner else
      } // maxPrimePtId == 1
    } // else if for possibility 3

  // if we haven't taken care of the current simplex, something's wrong
    else {
      cerr << "Error in ExploratoryMoves() - "
           << "Unaccounted for case.\nTerminating.\n";
      return;
    }
  FindMinMaxIndices();
  } while (!Stop());   // while stopping criteria is not satisfied
} // ExploratoryMoves()
```

A.2.3 NM - Constructors and Destructor

```
NMSearch::NMSearch(int dim)
{
   dimensions = dim;
   functionCalls = 0;
   simplex = NULL;
   simplexValues = NULL;
   centroid = new Vector<double>(dimensions,0.0);
   reflectionPt = new Vector<double>(dimensions,0.0);
   expansionPt = new Vector<double>(dimensions,0.0);
   contractionPt = new Vector<double>(dimensions,0.0);
   alpha = 1.0;
   beta = 0.5;
   gamma = 2.0;
   sigma = 0.5;
   scratch = new Vector<double>(dimensions,0.0);
   scratch2 = new Vector<double>(dimensions,0.0);
} // NMSearch() (default)

NMSearch::NMSearch(int dim, double Alpha, double Beta,
                            double Gamma, double Sigma)
{
   dimensions = dim;
   functionCalls = 0;
   simplex = NULL;
   simplexValues = NULL;
   centroid = new Vector<double>(dimensions,0.0);
   reflectionPt = new Vector<double>(dimensions,0.0);
   expansionPt = new Vector<double>(dimensions,0.0);
   contractionPt = new Vector<double>(dimensions,0.0);
```

```
    alpha = Alpha;
    beta = Beta;
    gamma = Gamma;
    sigma = Sigma;
    scratch = new Vector<double>(dimensions,0.0);
    scratch2 = new Vector<double>(dimensions,0.0);
} // NMSearch() (special)

NMSearch::NMSearch(const NMSearch& Original)
{
    dimensions = Original.GetVarNo();
    Original.GetCurrentSimplex(simplex);
    Original.GetCurrentSimplexValues(simplexValues);
    alpha = Original.alpha;
    beta = Original.beta;
    gamma = Original.gamma;
    sigma = Original.sigma;
    minIndex = Original.minIndex;
    maxIndex = Original.maxIndex;
    if(centroid != NULL) delete centroid;
    centroid = new Vector<double>(*(Original.centroid));
    if(reflectionPt != NULL) delete reflectionPt;
    reflectionPt = new Vector<double>(*(Original.reflectionPt));
    reflectionPtValue = Original.reflectionPtValue;
    if(expansionPt != NULL) delete expansionPt;
    expansionPt = new Vector<double>(*(Original.expansionPt));
    expansionPtValue = Original.expansionPtValue;
    if(contractionPt != NULL) delete contractionPt;
    contractionPt = new Vector<double>(*(Original.contractionPt));
    contractionPtValue = Original.contractionPtValue;
    functionCalls = Original.functionCalls;
} // NMSearch() (copy constructor)

NMSearch::~NMSearch()
{
    if(simplex != NULL) delete simplex;
    if(simplexValues != NULL) delete [] simplexValues;
    delete centroid;
    delete reflectionPt;
    delete expansionPt;
    delete contractionPt;
    delete scratch;
    delete scratch2;
    //NOTE: Matrix and Vector classes have their own destructors
} // ~NMSearch
```

A.2.4 NM - Simplex Initialization Routines

```
void NMSearch::InitRegularTriangularSimplex(const Vector<double> *basePoint,
                                            const double edgeLength)
{
  //  This routine constructs a regular simplex (i.e., one in which all of
  //  the edges are of equal length) following an algorithm given by Jacoby,
  //  Kowalik, and Pizzo in "Iterative Methods for Nonlinear Optimization
  //  Problems," Prentice-Hall (1972).  This algorithm also appears in
  //  Spendley, Hext, and Himsworth, "Sequential Application of Simplex
  //  Designs in Optimisation and Evolutionary Operation," Technometrics,
  //  Vol. 4, No. 4, November 1962, pages 441--461.

   int i,j;
   double p, q, temp;
   Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions,0.0);
   for( int col = 0; col < dimensions; col++ ) {
      (*plex)[0][col] = (*basePoint)[col];
   }

   temp = dimensions + 1.0;
   q = ((sqrt(temp) - 1.0) / (dimensions * sqrt(2.0))) * edgeLength;
   p = q + ((1.0 / sqrt(2.0)) * edgeLength);

   for(i = 1; i <= dimensions; i++) {
      for(j = 0; j <= i-2; j++) {
         (*plex)[i][j] = (*plex)[0][j] + q;
      } // inner for 1
      j = i - 1;
      (*plex)[i][j] = (*plex)[0][j] + p;
      for(j = i; j < dimensions; j++) {
            (*plex)[i][j] = (*plex)[0][j] + q;
      } // inner for 2
   } // outer for

   InitGeneralSimplex(plex);
   delete plex;
} // InitRegularTriangularSimplex()

void NMSearch::InitFixedLengthRightSimplex(const Vector<double> *basePoint,
                                           const double edgeLength)
{
  // to take advantage of code reuse, this function simply turns
  // edgeLength into a vector of dimensions length, and then
  // calls InitVariableLengthRightSimplex()
```

```
    double* edgeLengths = new double[dimensions];
    for( int i = 0; i < dimensions; i++ ) {
       edgeLengths[i] = edgeLength;
    }
    InitVariableLengthRightSimplex(basePoint,edgeLengths);
    delete [] edgeLengths;
 } // InitFixedLengthRightSimplex()

 void NMSearch::InitVariableLengthRightSimplex(const Vector<double> *basePoint,
                                               const double* edgeLengths)
 {
    Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions,0.0);
    for( int i = 0; i < dimensions; i++ ) {
       // we're building the basePoint component-by-component into
       // the (n+1)st row
       (*plex)[dimensions][i] = (*basePoint)[i];

       // now fill in the ith row with the proper point
       for( int j = 0; j < dimensions; j++ ) {
          (*plex)[i][j] = (*basePoint)[j];
          if( i == j )
             (*plex)[i][j] += edgeLengths[i];
       }
    } // for
    InitGeneralSimplex(plex);
    delete plex;
 } // InitVariableLengthRightSimplex()

 void NMSearch::InitGeneralSimplex(const Matrix<double> *plex)
 {
    functionCalls = 0;
    if( simplex != NULL ) { delete simplex; }
    if( simplexValues != NULL ) { delete [] simplexValues;}
    simplex = new Matrix<double>((*plex));
    simplexValues = new double[dimensions+1];

    int success;
    for( int i = 0; i <= dimensions; i++ ) {
       *scratch = (*plex).row(i);
       fcnCall(dimensions, (*scratch).begin(), simplexValues[i], success);
       if(!success) cerr<<"Error with point #"<<i<<" in initial simplex.\n";
    } // for
    FindMinMaxIndices();
 } // InitGeneralSimplex()
```

A.2.5 NM - Other Unique Functions

```
void NMSearch::ReadSimplexFile(istream& fp)
{
   if(fp == NULL) {
      cerr<<"No Input Stream in ReadSimplexFile()!\n";
      return; // There's no file handle!!
   }

   Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions);
   for( int i = 0; i <= dimensions; i++ ) {
      for ( int j = 0; j < dimensions; j++ ) {
         fp >> (*plex)[i][j];
      } // inner for
   } // outer for
   InitGeneralSimplex(plex);
   delete plex;
} // ReadSimplexFile()

bool NMSearch::Stop()
{
   if(maxCalls > -1) {
      if(functionCalls >= maxCalls)
         return true;
   }

   double mean = 0.0;

   for( int i = 0; i <= dimensions; i++) {
      if( i != minIndex ) {
         mean += simplexValues[i];
      } // if
   } //for

   mean /= (double)dimensions;

   // Test for the suggested Nelder-Mead stopping criteria
   double total = 0.0;
   for( int i = 0; i <= dimensions; i++ ) {
      total += pow( simplexValues[i] - mean ,2);
   } //for
   total /= ((double)dimensions + 1.0);
   total = sqrt(total);

   if(total < stoppingStepLength) {
```

```
      toleranceHit = 1;
      return true;
   }
   else
      return false;
} // Stop()

void NMSearch::GetMinPoint(Vector<double>* &minimum) const
{
   minimum = new Vector<double>((*simplex).row(minIndex));
} // GetMinPoint()

double NMSearch::GetMinVal() const
{
   return simplexValues[minIndex];
} // GetMinVal()

void NMSearch::FindMinMaxIndices()
{
   if(simplexValues == NULL) {
      cerr << "Error in FindMinMaxIndices() - "
           << "The vector of simplexValues is NULL!!\n";
      return;
   }
   minIndex = 0;
   maxIndex = dimensions;
   double min = simplexValues[0];
   double max = simplexValues[dimensions];
   for( int i = 1; i <= dimensions; i++ ) {
      if( simplexValues[i] < min ) {
         min = simplexValues[i];
         minIndex = i;
      } // if
      if( simplexValues[dimensions-i] > max ) {
         max = simplexValues[dimensions-i];
         maxIndex = dimensions - i;
      } // if
   } // for
} // FindMinMaxIndices()

int NMSearch::SecondHighestPtIndex()
{
   if(simplexValues == NULL) {
      cerr << "Error in SecondHighestPtValue() - "
           << "The vector of simplexValues is NULL!!\n";
```

```
      return -1;
    }
    int secondMaxIndex = minIndex;
    double secondMax = simplexValues[minIndex];
    for( int i = 0; i <= dimensions; i++ ) {
      if(i != maxIndex) {
        if( simplexValues[i] > secondMax ) {
          secondMax = simplexValues[i];
          secondMaxIndex = i;
        } // inner if
      } // outer if
    } // for
    return secondMaxIndex;
} // SecondHighestPtValue()

void NMSearch::FindCentroid()
{
    (*centroid) = 0.0;
    for( int i = 0; i <= dimensions; i++ ) {
      if( i != maxIndex ) {
        (*centroid) = (*centroid) + (*simplex).row(i);
      } // if
    } // for
    (*centroid) = (*centroid) * ( 1.0 / (double)dimensions );
} // FindCentroid()

void NMSearch::FindReflectionPt()
{
    (*reflectionPt) = 0.0;
    (*reflectionPt) = ( (*centroid) * (1.0 + alpha) ) -
                      ( alpha * (*simplex).row(maxIndex) );
    int success;
    fcnCall(dimensions, (*reflectionPt).begin(), reflectionPtValue, success);
    if(!success) {
      cerr << "Error finding f(x) for reflection point at"
           << "function call #" << functionCalls << ".\n";
    } // if
} // FindReflectionPt()

void NMSearch::FindExpansionPt()
{
    (*expansionPt) = 0.0;
    (*expansionPt) = ( (*centroid) * (1.0 - gamma) ) +
                     ( gamma * (*reflectionPt) );
    int success;
```

```
   fcnCall(dimensions, (*expansionPt).begin(), expansionPtValue, success);
   if(!success) {
      cerr << "Error finding f(x) for expansion point at"
           << "function call #" << functionCalls << ".\n";
   } // if
} // FindExpansionPt()

void NMSearch::FindContractionPt()
{
   // need to first define maxPrimePt
   Vector<double> *maxPrimePt = scratch;
   if(simplexValues[maxIndex] <= reflectionPtValue) {
      *maxPrimePt = (*simplex).row(maxIndex);
      maxPrimePtValue = simplexValues[maxIndex];
      maxPrimePtId = 1;
   } // if
   else {
      maxPrimePt = reflectionPt;
      maxPrimePtValue = reflectionPtValue;
      maxPrimePtId = 0;
   } // else

   (*contractionPt) = ( (*centroid) * (1.0 - beta) ) +
                      ( beta * (*maxPrimePt) );
   int success;
   fcnCall(dimensions, (*contractionPt).begin(), contractionPtValue, success);
   if(!success) {
      cerr << "Error finding f(x) for contraction point at"
           << "function call #" << functionCalls << ".\n";
   } // if
} // FindContractionPt()

void NMSearch::ShrinkSimplex()
{
   // stop if at maximum function calls
   if (functionCalls >= maxCalls) {return;}

   Vector<double> *lowestPt = scratch;
   *lowestPt = (*simplex).row(minIndex);
   Vector<double> *tempPt = scratch2;
   int success;
   for( int i = 0; i <= dimensions; i++ ) {
      if( i != minIndex ) {
         *tempPt = (*simplex).row(i);
         (*tempPt) = (*tempPt) + ( sigma * ( (*lowestPt)-(*tempPt) ) );
```

```
            for( int j = 0; j < dimensions; j++ ) {
               (*simplex)[i][j] = (*tempPt)[j];
            } // inner for
            fcnCall(dimensions,(*tempPt).begin(),simplexValues[i],success);
            if (!success) cerr << "Error shrinking the simplex.\n";

            // stop if at maximum function calls
            if (functionCalls >= maxCalls) {return;}

         } // if
      } // outer for
   } // ShrinkSimplex()

   void NMSearch::SetAlpha(double newAlpha)
   {
      alpha = newAlpha;
   } // SetAlpha()

   void NMSearch::SetBeta(double newBeta)
   {
      beta = newBeta;
   } // SetBeta()

   void NMSearch::SetGamma(double newGamma)
   {
      gamma = newGamma;
   } // SetGamma()

   void NMSearch::printSimplex() const
   {
     for( int i = 0; i <= dimensions; i++ ) {
        cout << "Point: ";
        for ( int j = 0; j < dimensions; j++ ) {
          cout << (*simplex)[i][j] << " ";
        } // inner for
        cout << "   Value: " << simplexValues[i]
             << "   Age: " << simplexAges[i] << "\n";
     } // outer for

     cout << "\nFCalls: " << functionCalls << "\n\n";
   } // printSimplex()
```

A.3 C++ Code for the Sequential Multi-Directional Search

A.3.1 SMDS - Header File

```
/*SMDSearch.h
 *Declarations of Sequential version of Torczon's Multi-Directional Search
 *Adam Gurson College of William & Mary 2000
 */

#ifndef _SMDSearch_
#define _SMDSearch_

#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>
#include "objective.h"
#include "vec.h"
#include "cmat.h"
#include "subscrpt.h"


#define stoppingStepLength 10e-8 /*sqrt(fabs(3.0 * (4.0/3.0 - 1.0) - 1.0))*/
#ifndef DEBUG
#define DEBUG 0
#endif
//#ifndef maxCalls
//#define maxCalls 200
//#endif


class SMDSearch
{
 public:
 // constructors and destructors

   SMDSearch(int dim);
   // default constructor

   SMDSearch(int dim, double Sigma);
   // constructor which allows shrinking coefficient initialization

   SMDSearch(const SMDSearch& Original);
   // deep copy constructor
```

```
  ~SMDSearch();
  // destructor

// algorithmic routines

  void ExploratoryMoves();
  // use SMD to find function minimum

  void ReplaceSimplexPoint(int index, const Vector<double>& newPoint);
  // replaces simplex point indexed at index with newPoint

  void CalculateFunctionValue(int index);
  // finds the f(x) value for the simplex point indexed at index and
  // replaces the proper value in simplexValues

  void SetSigma(double newSigma);
  // allows the user to set a new value for
  // the shrinking coefficient

  bool Stop();
  // returns true if the stopping criteria have been satisfied

  void fcnCall(int n, double *x, double& f, int& flag);
  // indirection of function call for purposes of keeping an accurate
  // tally of the number of function calls

// Simplex-altering functions

  void InitRegularTriangularSimplex(const Vector<double> *basePoint,
                                    const double edgeLength);
  // deletes any existing simplex and replaces it with a regular
  // triangular simplex in the following manner:
  //
  // basePoint points to a point that will be the "origin" of the
  //    simplex points (it will be a part of the simplex and
  //    its function value is found here)
  // edgeLength is the length of each edge of the "triangle"
  //
  // functionCalls is reset to 0
  // delta is set to edgeLength
  //
  // NOTE: basePoint is assumed to be of proper dimension

  void InitFixedLengthRightSimplex(const Vector<double> *basePoint,
```

```
                                     const double edgeLength);
 // deletes any existing simplex and replaces it with a right-angle
 // simplex in the following manner:
 //
 // basePoint points to a point that will be the "origin" of the
 //    simplex points (it will be a part of the simplex and
 //    its function value is found here)
 // edgeLength is to be the length of each simplex side extending
 //    from the basePoint along each positive coordinate direction.
 //
 // functionCalls is reset to 0
 // delta is set to edgeLength
 //
 // NOTE: basePoint is assumed to be of proper dimension

void InitVariableLengthRightSimplex(const Vector<double> *basePoint,
                                     const double* edgeLengths);
 // deletes any existing simplex and replaces it with a right-angle
 // simplex in the following manner:
 //
 // basePoint points to a point that will be the "origin" of the
 //    simplex points (it will be a part of the simplex and
 //    its function value is found here)
 // edgeLengths points to an array of n doubles, where n is the
 //    dimension of the given search. x_1 will then be located
 //    a distance of edgeLengths[0] away from the basepoint along the
 //    the x_1 axis, x_2 is edgeLengths[1] away on the x_2 axis, etc.
 //
 // functionCalls is reset to 0
 // delta is set to the largest value in edgeLength[]
 //
 // NOTE: basePoint and edgeLengths are assumed to be of proper dimension

void InitGeneralSimplex(const Matrix<double> *plex);
 // deletes any existing simplex and replaces it with the one
 // pointed to by plex
 //
 // functionCalls is reset to 0
 // delta is set to the length of the longest simplex side
 //
 // NOTE: THIS ASSUMES THAT plex IS OF PROPER DIMENSION
 //        AND THAT basePoint is in the LAST row of plex
 //
 // The basePoint is a point that will be the "origin" of the
 //    simplex points (it will be a part of the simplex and
```

```
  //    its function value is calculated here)

  void ReadSimplexFile(istream& fp);
  // may also pass cin as input stream if desired
  // input the values of each trial point
  // NOTE: THIS FUNCTION WILL ONLY ACCEPT n+1 POINTS
  //
  // NOTE: assumes that the basePoint is the last point entered
  //
  // functionCalls is reset to 0
  // delta is set to the length of the longest simplex side

// Query functions

  int GetFunctionCalls() const;
  // number of objective function evaluations

  void GetMinPoint(Vector<double>* &minimum) const;
  // simplex point which generates the best objective function
  // value found thus far
  // USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

  double GetMinVal() const;
  // best objective function value found thus far

  void GetCurrentSimplex(Matrix<double>* &plex) const;
  // performs a deep copy of the simplex to a Matrix pointer
  // points to a newly allocated chunk of memory upon return
  // USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

  void GetCurrentSimplexValues(double* &simValues) const;
  // performs a deep copy of the simplexValues array to a double pointer
  // points to a newly allocated chunk of memory upon return
  // USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

  void GetCurrentSimplexVBits(int* &simVBits) const;
  // performs a deep copy of the simplexVBits array to a double pointer
  // points to a newly allocated chunk of memory upon return
  // USER SHOULD PASS JUST A NULL POINTER, WITHOUT PREALLOCATED MEMORY

  int GetVarNo() const;
  // returns the dimension of the problem

  int GetTolHit() const;
  // returns toleranceHit
```

```
  void printSimplex() const;
  // prints out the primary simplex points by row,
  // their corresponding f(x) values, their validity status,
  // and the number of function calls thus far

  void printRefSimplex() const;
  // prints out the reflection simplex points by row,
  // their corresponding f(x) values, their validity status,
  // and the number of function calls thus far

private:

  void CreateRefSimplex();
  // creates a reflection simplex from the primary simplex

  void SwitchSimplices();
  // swaps the primary and reflection simplices

  void ShrinkSimplex();
  // this function goes through the primary simplex and reduces the
  // lengths of the edges adjacent to the best vertex

  int GetAnotherIndex(int& index, int*& validBits);
  // this is used to find another INVALID point in the simplex
  // if all points are VALID, returns 0, otherwise 1

  void CalculateRefFunctionValue(int index);
  // Like CalculateFunctionValue(), but for the Reflection Simplex
  // (A user should not directly manipulate the reflection simplex,
  // hence the private status of this function)


  int dimensions;                     // the number of variables
                                      //    (the dimension of the problem)
  Matrix<double> *simplex;            // the current simplex
  double *simplexValues;              // their corresponding f(x) values
  int *simplexVBits;                  // valid bits for the simplex values
  int currentIndex;                   // used to step through the arrays

  Matrix<double> *refSimplex;            // the reflection simplex
  double *refSimplexValues;              // their corresponding f(x) values
  int *refSimplexVBits;                  // valid bits for the simplex values
  int refCurrentIndex;                   // used to step through the arrays
```

```
    Vector<double> *minPt;         // the actual minimizer (i.e. best point seen)
    double minValue;               // f(minimizer)
    int minIndex;                  // the row index of minPt in the main simplex
    double delta;                  // our stopping criterion
    double sigma;                  // shrinking coefficient
    long functionCalls;            // tally of number of function calls
    int toleranceHit;              // 1 if stop due to tolerance, 0 if funcCalls

    // the following vectors are simply extra storage space
    // that are used by functions that require a vector of
    // size = dimensions
    Vector<double> *scratch, *scratch2;
};

#endif
```

A.3.2 SMDS - ExploratoryMoves()

```
void SMDSearch::ExploratoryMoves()
{
   int done;
   int lastMinIndex = minIndex;
   toleranceHit = 0;

   do {
     done = 0;
     CreateRefSimplex();

     if(DEBUG) {
       printSimplex();
       printRefSimplex();
     }

     // Go through the Reflection Simplex First
     refCurrentIndex = lastMinIndex;
     while( !done && GetAnotherIndex(refCurrentIndex, refSimplexVBits) ) {
       CalculateRefFunctionValue(refCurrentIndex);
       refSimplexVBits[refCurrentIndex] = 1;

       if(DEBUG) printRefSimplex();

       if( refSimplexValues[refCurrentIndex] < minValue ) {
         (*minPt) = (*refSimplex).row(refCurrentIndex);
         minValue = refSimplexValues[refCurrentIndex];
         lastMinIndex = minIndex;
```

```
          minIndex = refCurrentIndex;
          SwitchSimplices();
          done = 1;
        } // if

        if( functionCalls >= maxCalls ) return;
      } // while (reflection search)

      // Go through the Primary Simplex Next
      while( !done && GetAnotherIndex(currentIndex, simplexVBits) ) {
        CalculateFunctionValue(currentIndex);
        simplexVBits[currentIndex] = 1;
        // NOTE: currentIndex initialized in InitGeneralSimplex()

        if(DEBUG) printSimplex();

        if( simplexValues[currentIndex] < minValue ) {
          (*minPt) = (*simplex).row(currentIndex);
          minValue = simplexValues[currentIndex];
          lastMinIndex = minIndex;
          minIndex = currentIndex;
          done = 1;
        } // if

        if( functionCalls >= maxCalls ) return;
      } // while (primary search)

      // Still there's no new min now, shrink
      if( !done )
        ShrinkSimplex();

   } while (!Stop());   // while stopping criteria is not satisfied
} // ExploratoryMoves()
```

A.3.3 SMDS - Constructors and Destructor

```
SMDSearch::SMDSearch(int dim)
{
   dimensions = dim;
   simplex = new Matrix<double>(dimensions+1,dimensions,0.0);
   simplexValues = new double[dimensions+1];
   simplexVBits = new int[dimensions+1];
   refSimplex = new Matrix<double>(dimensions+1,dimensions,0.0);
   refSimplexValues = new double[dimensions+1];
   refSimplexVBits = new int[dimensions+1];
```

```
    minPt = new Vector<double>(dimensions,0.0);
    delta = -1.0;
    sigma = 0.5;
    functionCalls = 0;
    scratch = new Vector<double>(dimensions,0.0);
    scratch2 = new Vector<double>(dimensions,0.0);
} // SMDSearch() (default)

SMDSearch::SMDSearch(int dim, double Sigma)
{
    dimensions = dim;
    simplex = new Matrix<double>(dimensions+1,dimensions,0.0);
    simplexValues = new double[dimensions+1];
    simplexVBits = new int[dimensions+1];
    refSimplex = new Matrix<double>(dimensions+1,dimensions,0.0);
    refSimplexValues = new double[dimensions+1];
    refSimplexVBits = new int[dimensions+1];
    minPt = new Vector<double>(dimensions,0.0);
    delta = -1.0;
    sigma = Sigma;
    functionCalls = 0;
    scratch = new Vector<double>(dimensions,0.0);
    scratch2 = new Vector<double>(dimensions,0.0);
} // SMDSearch() (special)

SMDSearch::SMDSearch(const SMDSearch& Original)
{
    dimensions = Original.GetVarNo();
    Original.GetCurrentSimplex(simplex);
    Original.GetCurrentSimplexValues(simplexValues);
    Original.GetCurrentSimplexVBits(simplexVBits);
    if(minPt != NULL) delete minPt;
    minPt = new Vector<double>(*(Original.minPt));
    minValue = Original.minValue;
    delta = Original.delta;
    sigma = Original.sigma;
    functionCalls = Original.functionCalls;
} // SMDSearch() (copy constructor)

SMDSearch::~SMDSearch()
{
    delete simplex;
    delete [] simplexValues;
    delete [] simplexVBits;
    delete refSimplex;
```

```
  delete [] refSimplexValues;
  delete [] refSimplexVBits;
  delete minPt;
  delete scratch;
  delete scratch2;
  //NOTE: Matrix and Vector classes have their own destructors
} // ~SMDSearch
```

A.3.4 SMDS - Simplex Initialization Routines

```
void SMDSearch::InitRegularTriangularSimplex(const Vector<double> *basePoint,
                                               const double edgeLength)
{
  //  This routine constructs a regular simplex (i.e., one in which all of
  //  the edges are of equal length) following an algorithm given by Jacoby,
  //  Kowalik, and Pizzo in "Iterative Methods for Nonlinear Optimization
  //  Problems," Prentice-Hall (1972).  This algorithm also appears in
  //  Spendley, Hext, and Himsworth, "Sequential Application of Simplex
  //  Designs in Optimisation and Evolutionary Operation," Technometrics,
  //  Vol. 4, No. 4, November 1962, pages 441--461.

   int i,j;
   double p, q, temp;
   Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions,0.0);
   for( int col = 0; col < dimensions; col++ ) {
      (*plex)[0][col] = (*basePoint)[col];
   }

   temp = dimensions + 1.0;
   q = ((sqrt(temp) - 1.0) / (dimensions * sqrt(2.0))) * edgeLength;
   p = q + ((1.0 / sqrt(2.0)) * edgeLength);

   for(i = 1; i <= dimensions; i++) {
      for(j = 0; j <= i-2; j++) {
         (*plex)[i][j] = (*plex)[0][j] + q;
      } // inner for 1
      j = i - 1; // is this line necessary (redundant)
      (*plex)[i][j] = (*plex)[0][j] + p;
      for(j = i; j < dimensions; j++) {
            (*plex)[i][j] = (*plex)[0][j] + q;
      } // inner for 2
  } // outer for

  delta = edgeLength;
  InitGeneralSimplex(plex);
```

```
    delete plex;
 } // InitRegularTriangularSimplex()

 void SMDSearch::InitFixedLengthRightSimplex(const Vector<double> *basePoint,
                                             const double edgeLength)
 {
   // to take advantage of code reuse, this function simply turns
   // edgeLength into a vector of dimensions length, and then
   // calls InitVariableLengthRightSimplex()

    double* edgeLengths = new double[dimensions];
    for( int i = 0; i < dimensions; i++ ) {
       edgeLengths[i] = edgeLength;
    }
    InitVariableLengthRightSimplex(basePoint,edgeLengths);
    delete [] edgeLengths;
 } // InitFixedLengthRightSimplex()

 void SMDSearch::InitVariableLengthRightSimplex(const Vector<double> *basePoint,
                                                const double* edgeLengths)
 {
   Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions,0.0);
   for( int i = 0; i < dimensions; i++ ) {
      // we're building the basePoint component-by-component into
      // the (n+1)st row
      (*plex)[dimensions][i] = (*basePoint)[i];

      // now fill in the ith row with the proper point
      for( int j = 0; j < dimensions; j++ ) {
         (*plex)[i][j] = (*basePoint)[j];
         if( i == j )
            (*plex)[i][j] += edgeLengths[i];
      }

      if( edgeLengths[i] > delta ) delta = edgeLengths[i];
   }
   InitGeneralSimplex(plex);
   delete plex;
} // InitVariableLengthRightSimplex()

void SMDSearch::InitGeneralSimplex(const Matrix<double> *plex)
{
   functionCalls = 0;
   (*simplex) = (*plex);
```

```
    // zero out the valid bits
    for(int i = 0; i < dimensions; i++)
      simplexVBits[i] = 0;

    // NOTE: basePoint MUST be located in the last row of plex
    Vector<double> basePoint = (*plex).row(dimensions);

    // evaluate f(basePoint) and initialize it as the min
    int success;
    fcnCall(dimensions, (basePoint).begin(), simplexValues[dimensions], success);
    if(!success) cerr<<"Error with basePoint in initial simplex.\n";
    simplexVBits[dimensions] = 1;
    (*minPt) = (basePoint);
    minValue = simplexValues[dimensions];
    currentIndex = minIndex = dimensions;

    // if we still haven't defined delta, go through the simplex and
    // define delta to be the length of the LONGEST simplex side
    double temp;
    if( delta < 0.0 ) {
      for( int j = 0; j < dimensions; j++ ) {
        for ( int k = j+1; k <= dimensions; k++ ) {
          temp = ( ((*simplex).row(j)) - ((*simplex).row(k)) ).l2norm();
          if( temp > delta ) delta = temp;
        } // inner for
      } // outer for
    } // outer if

    // if delta is still not defined, there is a definite problem
    if( delta < 0.0 )
      cout << "Error in simplex initialization: delta not set.\n";
} // InitGeneralSimplex()
```

A.3.5 SMDS - Other Unique Functions

```
void SMDSearch::ReadSimplexFile(istream& fp)
{
   if(fp == NULL) {
      cerr<<"No Input Stream in ReadSimplexFile()!\n";
      return; // There's no file handle!!
   }

   Vector<double> *basePoint = new Vector<double>(dimensions);
   Matrix<double> *plex = new Matrix<double>(dimensions+1,dimensions);
   for( int i = 0; i <= dimensions; i++ ) {
```

```
        for ( int j = 0; j < dimensions; j++ ) {
           fp >> (*plex)[i][j];
        } // inner for
     } // outer for
     (*basePoint) = (*plex).row(dimensions);
     InitGeneralSimplex(plex);
     delete basePoint;
     delete plex;
  } // ReadSimplexFile()

  bool SMDSearch::Stop()
  {
     if(maxCalls > -1) {
        if(functionCalls >= maxCalls)
           return true;
     }

     if(delta < stoppingStepLength) {
        toleranceHit = 1;
        return true;
     }
     else
        return false;
  } // Stop()

  void SMDSearch::GetMinPoint(Vector<double>* &minimum) const
  {
     minimum = new Vector<double>((*minPt));
  } // GetMinPoint()

  double SMDSearch::GetMinVal() const
  {
     return minValue;
  } // GetMinVal()

  void SMDSearch::GetCurrentSimplexVBits(int* &simVBits) const
  {
     simVBits = new int[dimensions+1];
     for( int i = 0; i <= dimensions; i++ ) {
        simVBits[i] = simplexVBits[i];
     } // for
  } // GetCurrentSimplexValues()

  void SMDSearch::CreateRefSimplex()
  {
```

```
    // copy the known flip point over
    for( int i = 0; i < dimensions; i++ )
      (*refSimplex)[currentIndex][i] = (*simplex)[currentIndex][i];
    refSimplexValues[currentIndex] = simplexValues[currentIndex];
    refSimplexVBits[currentIndex] = simplexVBits[currentIndex];
    refCurrentIndex = currentIndex;

    // reflect the remaining points
    for( int j = 0; j <= dimensions; j++ ) {
      if( j != currentIndex ) {
        refSimplexVBits[j] = 0;
        (*scratch) = ( (*simplex).row(currentIndex) * 2.0 ) - (*simplex).row(j);
        for( int k = 0; k < dimensions; k++ )
          (*refSimplex)[j][k] = (*scratch)[k];
      } // if
    } // outer for
  } // CreateRefSimplex()

  void SMDSearch::SwitchSimplices()
  {
    // this allows us to remove the need to delete and
    // reallocate memory by simply swapping pointers
    // and using the same two "simplex memory slots"
    // for the entire search

    Matrix<double> *tmp1 = simplex;
    double         *tmp2 = simplexValues;
    int            *tmp3 = simplexVBits;
    int             tmp4 = currentIndex;

    simplex = refSimplex;
    simplexValues = refSimplexValues;
    simplexVBits = refSimplexVBits;
    currentIndex = refCurrentIndex;

    refSimplex = tmp1;
    refSimplexValues = tmp2;
    refSimplexVBits = tmp3;
    refCurrentIndex = tmp4;
  } // SwitchSimplices()

  void SMDSearch::ShrinkSimplex()
  {
    if(DEBUG) cout << "Shrinking Simplex.\n\n";
```

```
    delta *= sigma;
    currentIndex = minIndex;
    Vector<double> *lowestPt = scratch;
    *lowestPt = (*simplex).row(minIndex);
    Vector<double> *tempPt = scratch2;

    for( int i = 0; i <= dimensions; i++ ) {
       if( i != minIndex ) {
          *tempPt = (*simplex).row(i);
          (*tempPt) = (*tempPt) + ( sigma * ( (*lowestPt)-(*tempPt) ) );
          for( int j = 0; j < dimensions; j++ ) {
             (*simplex)[i][j] = (*tempPt)[j];
          } // inner for

          simplexVBits[i] = 0;

       } // if
    } // outer for
} // ShrinkSimplex()

int SMDSearch::GetAnotherIndex(int& index, int*& validBits)
{
  if ( !validBits[index] ) return 1;

  int initialIndex = index;

  do {
    index++;
    if( index > dimensions ) index = 0;
  } while ( ( index != initialIndex) &&
            ( validBits[index] ) );

  if( index == initialIndex )
    return 0;
  else
    return 1;
} // GetAnotherIndex()

void SMDSearch::CalculateRefFunctionValue(int index)
{
   *scratch = (*refSimplex).row(index);
   int success;
   fcnCall(dimensions, (*scratch).begin(),
           refSimplexValues[index], success);
   if(!success) cerr<<"Error calculating point at index "
```

```
                    << index << "in CalculateFunctionValue().\n";
} // CalculateFunctionValue()

void SMDSearch::printSimplex() const
{
  cout << "Primary Simplex:\n";

  for( int i = 0; i <= dimensions; i++ ) {
     cout << "Point: ";
     for ( int j = 0; j < dimensions; j++ ) {
       cout << (*simplex)[i][j] << " ";
     } // inner for
     cout << "   Value: " << simplexValues[i];

     if( simplexVBits[i] )
       cout << "   Valid\n";
     else
       cout << "   Invalid\n";
  } // outer for

  cout << "FCalls: " << functionCalls
       << "   Delta: " << delta << "\n\n";
} // printSimplex()

void SMDSearch::printRefSimplex() const
{
  cout << "Reflection Simplex:\n";

  for( int i = 0; i <= dimensions; i++ ) {
     cout << "Point: ";
     for ( int j = 0; j < dimensions; j++ ) {
       cout << (*refSimplex)[i][j] << " ";
     } // inner for
     cout << "   Value: " << refSimplexValues[i];

     if( refSimplexVBits[i] )
       cout << "   Valid\n";
     else
       cout << "   Invalid\n";
  } // outer for

  cout << "FCalls: " << functionCalls
       << "   Delta: " << delta << "\n\n";
} // printRefSimplex()
```

A.4 Functions Common to All Three Searches

```
void Search::ReplaceSimplexPoint(int index, const Vector<double>& newPoint)
{
   for( int i = 0; i < dimensions; i++ ) {
      (*simplex)[index][i] = newPoint[i];
   } // for
} // ReplaceSimplexPoint()

void Search::CalculateFunctionValue(int index)
{
   *scratch = (*simplex).row(index);
   int success;
   fcnCall(dimensions, (*scratch).begin(), simplexValues[index], success);
   if(!success) cerr<<"Error calculating point in CalculateFunctionValue().\n";
} // CalculateFunctionValue()

void Search::fcnCall(int n, double *x, double& f, int& flag)
{
   fcn(n, x, f, flag);
   functionCalls++;
} // fcnCall()

int Search::GetFunctionCalls() const
{
   return functionCalls;
} // GetFunctionCalls()

void Search::GetCurrentSimplex(Matrix<double>* &plex) const
{
   plex = new Matrix<double>((*simplex));
} // GetCurrentSimplex()

void Search::GetCurrentSimplexValues(double* &simValues) const
{
   simValues = new double[dimensions+1];
   for( int i = 0; i <= dimensions; i++ ) {
      simValues[i] = simplexValues[i];
   } // for
} // GetCurrentSimplexValues()

int Search::GetVarNo() const
{
   return dimensions;
} // GetVarNo()
```

```
int Search::GetTolHit() const
{
   return toleranceHit;
} // GetTolHit()

void Search::SetSigma(double newSigma)
{
   sigma = newSigma;
} // SetSigma()
```

Bibliography

[1] Mordecai Avriel. *Nonlinear Programming: Analysis and Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[2] P. R. Benyon. Remark AS R15. Function minimization using a simplex procedure. *Applied Statistics*, 25(1):97, 1976.

[3] John M. Chambers and J. E. Ertel. Remark AS R11. A remark on algorithm AS 47 'Function minimization using a simplex procedure'. *Applied Statistics*, 23(2):250–251, 1974.

[4] Elizabeth D. Dolan. Pattern search behavior in nonlinear optimization. Honors Thesis, Department of Computer Science, College of William & Mary, Williamsburg, Virginia 23187–8795, May 1999. Accepted with Highest Honors.

[5] I. D. Hill. Remark AS R28. A remark on algorithm AS 47: Function minimization using a simplex procedure. *Applied Statistics*, 27(3):380–382, 1978.

[6] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1):112–147, 1998.

[7] K. I. M. McKinnon. Convergence of the Nelder–Mead simplex method to a nonstationary point. *SIAM Journal on Optimization*, 9(1):148–158, 1998.

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

[9] O'Neill, Chambers, Ertel, Benyon, and Hill. `nelmin.f`. Found in Statlib at `http://lib.stat.cmu.edu/apstat/47`, 1971.

[10] R. O'Neill. Algorithm AS 47. Function minimization using a simplex procedure. *Applied Statistics*, 20(3):338–345, 1971.

[11] Anthony Padula. Interpolation and pseudorandom function generation. Honors Thesis, May 2000. Department of Mathematics, College of William & Mary, Williamsburg, Virginia 23185–8795. Accepted with High Honors.

[12] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31:1192–1201, 1988.

[13] Michael J. D. Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in Optimization and Numerical Analysis, Proceedings of the 6th Workshop on Optimization and Numerical Analysis, Oaxaca, Mexico*, volume 275, pages 51–67, Dordrecht, 1994. Kluwer Academic Publishers.

[14] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes In C: The Art of Scientific Computing: Second Edition.* Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, 1992.

[15] D. E. Shaw, R. W. M. Wedderburn, and Alan Miller. `minim.f`. Found in Statlib at `http://lib.stat.cmu.edu/apstat/47`, August 1991.

[16] Christopher Siefert. Model-assisted pattern search. Honors Thesis, May 2000. Department of Computer Science, College of William & Mary, Williamsburg, Virginia 23185–8795. Accepted with Highest Honors.

[17] W. Spendley, G. R. Hext, and F. R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. *Technometrics*, 4(4):441–461, November 1962.

[18] Virginia Torczon. *Multi-Directional Search: A Direct Search Algorithm for Parallel Machines.* PhD thesis, Department of Mathematical Sciences, Rice University, Houston, Texas, 1989; available as Tech. Rep. 90-07, Department of Computational and Applied Mathematics, Rice University, Houston, Texas 77005-1892.

[19] Michael W. Trosset. The krigifier: A procedure for generating pseudorandom nonlinear objective functions for computational experimentation. Technical Report ICASE Interim Report 35, Institute for Computer Applications in Science and Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia 23681–2199, 1999.

[20] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S-PLUS.* Statistics and computing. Springer-Verlag, New York, 1999.