Clamping-Simplex Methods:
Improved Direct Search Simplex Algorithms

by
Raymond W. L. Szeto

Submitted to the Departments of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Masters of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
August 30, 2000

© 2000 Raymond W. L. Szeto. All rights reserved.

The author hereby grants to M.I.T. permissions to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant other the right to do so.

Author

Department of Electrical Engineering and Computer Science
August 30, 2000

Certified by

Elazer R. Edelman
Thomas D and Virginia W Cabot Associate Professor of HST
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Clamping-Simplex Methods:
Improved Direct Search Simplex Algorithms
by
Raymond W. L. Szeto

Submitted to the
Departments of Electrical Engineering and Computer Science
August 30, 2000
In Partial Fulfillment of the Requirements for the Degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

It has long been suspected, and recently proven, that the Nelder and Mead simplex method for function minimization can converge on false minima. Currently, there does not exist any method that corrects the problem. Described herein is an approach for developing methods that help correct this problem. The approach introduces the concept of function clamping. The clamping-simplex methods, uses dynamic clamping to avoid situation that can cause these false convergences. In particular, two methods (dmins and stmins) are developed. The former, dmins, was described in design-level detail; the latter, stmins, was also implemented and studied. stmins demonstrated potential not only for reducing or avoiding false convergences but also for doubling as a method for finding better starting positions for the Nelder-Mead method. Additionally, the performance feature, derived from clamping, of partial function evaluation was investigated.

Thesis Supervisor: Elazer Edelman

Title: Thomas D and Virginia W Cabot Associate Professor of HST
Acknowledgements

I am especially indebted to my thesis supervisor, Elazer Edelman, for his guidance, mentorship, support, and never-ending patience and understanding throughout this project. In addition, I would also like to thank Elazer for giving me the freedom to pursue this topic.

I would also like to thank the members of the Edelman Lab group for their insightful comments, wonderful hospitality, and general helpfulness during my time with the group. It has been a privilege to work with such a talented group of people.
Contents

CHAPTER 1: INTRODUCTION ........................................................................................................... 6

CHAPTER 2: BACKGROUND, MOTIVATION, & THE ALGORITHMS ............................................... 8

2.1 FUNCTIONAL MINIMIZATION VERSUS OPTIMIZATION .................................................... 8

2.2 BACKGROUND OF OPTIMIZATION AND RESEARCH MOTIVATION .............................. 11

   Golden Section Search ......................................................................................................... 11

   Parabolic Interpolation Method ......................................................................................... 14

   Conjugate Gradient Methods .............................................................................................. 15

   Multidimensional Direct Search and the Nelder-Mead Simplex ....................................... 16

2.3 NELDER AND MEAD’S DOWNHILL SIMPLEX METHOD ................................................. 17

   The Algorithm ...................................................................................................................... 17

   The Problems ....................................................................................................................... 23

2.4 CLAMPING-SIMPLEX ......................................................................................................... 25

   Problem, Theory on the Cause .......................................................................................... 25

   Clamping-Simplex, the Concept ......................................................................................... 28

   stmins: Theory and Implementation .................................................................................. 28

   stmins: Example .................................................................................................................. 30

   dmins: Theory ..................................................................................................................... 32

   Taking Full Advantage of Clamping .................................................................................. 32

CHAPTER 3: ALGORITHM REQUIREMENTS ................................................................................. 33

3.1 FUNCTIONALITY .................................................................................................................. 33

3.2 PERFORMANCE .................................................................................................................. 33

3.3 ACCESSIBILITY ................................................................................................................ 35

CHAPTER 4: DATA AND ANALYSIS ......................................................................................... 36

4.1 UNITS OF MEASURE .......................................................................................................... 36

4.2 DATA ................................................................................................................................. 37

   Test Set 1: Finding the Optimal ρ and ν ............................................................................. 40

   Test Set 2: Effects of Clamping and stmin Functionality .................................................. 46

   Test Set 3: Comparing stmins with Nelder-Mead (alone) .................................................. 49
Chapter 1: Introduction

This thesis describes the development and analysis of a significant improved direct search simplex algorithm. This algorithm, hereafter referred to as the clamping-simplex or as stmins, incorporates the concept of dynamic partial function evaluations into the Nelder and Mead's downhill simplex method. We are using the mathematical definition of simplex. "A simplex is a geometrical figure consisting, in N dimensions, of N + 1 points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, not necessarily the regular tetrahedron." This will become more apparent as we go along, especially after the example in section 2.3. The algorithm we now present combines the theory of two understood methods for finding local minima to produce a much better, combined method. In result, the clamping-simplex gives significant improvements in runtime over the original Nelder-Mead's method.

The Nelder and Mead's simplex method of system optimization, hereafter referred to as Nelder-Mead, is robust as it can be used when little is known about the system, and has the ability to solve NP-complete problems. At the same time, Nelder and Mead's method is computationally burdensome. In fact, in almost all instances, algorithms that use derivatives, such as Powell's method, are faster. Derivative algorithms achieve its speed by calculating the function's slope at all points instead of looking at just the local landscape as Nelder-Mead and direct search methods do. This thesis analyzes what aspects of the Nelder-Mead that retard its computational speed, and describes and analyzes how the newly developed clamping-simplex can improve performance in certain environments.

This thesis is organized into six chapters. Chapter 1 contains a general statement of the problem. Chapter 2 lays down the ground works and the theory of this project. It gives the general reader the background, the concepts, and the logic necessary for understanding this research. This chapter also describes the operation of the Nelder-Mead and the clamping-simplex methods. Chapter 3 discusses a plan for analyzing the clamping-simplex and the reasoning for the approach. "Design" requirements are

---

1 W. Press et al. pp. 408-409
presented, that insure the two algorithms are similar enough to be compared and that the clamping-simplex is algorithmically correct\textsuperscript{2}. "An algorithm is said to be \textit{correct} if, for every input instance, it halts with the correct output." (from T. Cormen \textit{et al.} p2. See endnote) Chapter 4 presents the data and the analysis of these two algorithms under the various defined test circumstances. Finally, Chapter 5 draws the conclusions and presents the applications of this work to this field as well as to other fields. Potential future work and questions are presented.

\textsuperscript{2} GR: Correct is used in the algorithms context. "An algorithm is said to be \textit{correct} if, for every input instance, it halts with the correct output." (from T. Cormen \textit{et al.} p2. See endnote) Further explained in Chapter 3.
Chapter 2: Background, Motivation, & the Algorithms

This chapter describes and analyzes the advantages and disadvantages of the Nelder-Mead algorithm. Section 2.1 starts with a brief description of the relevance of function minimization. The significance of direct search approach is presented in Section 2.2. Section 2.3 describes the Nelder-Mead algorithm and concludes with illustrative examples of the limits of its performance. Sections 2.1 through 2.3 are primarily for readers with little to no background in direct search algorithms or the Nelder-Mead algorithm. Section 2.4 presents the clamping-simplex with significant detail of the theory and the algorithm.

2.1 Functional Minimization versus Optimization

This thesis encompasses the convention employed by other publications wherein function minimization and optimization are conceptually equivalent and used interchangeably. Their equivalence might best be appreciated through the following scenario. Assume we have an old fashion radio and we wish to listen to a local radio station, WMBR, which is 88.1 FM. In this example, the goal is the get the best reception possible—the optimal reception. We start by adjusting the radio’s tuner knob. As the dial on the tuner approaches 88.1, the static turns into words and music. Unfortunately, simply adjusting the tuner knob is not enough to give a clear reception. We need to adjust the antenna. If our antenna, like most radio antennas, looks like a pole that’s attached only at the base (Figure 2.1.1a) it can be adjusted in two ways. The pitch can be manipulated such that the antenna is vertical, horizontal, or at any angle in between (denoted by θ in Figure 2.1.1b). Moreover, the angular direction of pole can be fully rotated about its base to point in any direction (Figure 2.1.1c).

3 WMBR is MIT’s own radio station, located in the basement of Walker Memorial (the acronym stands for Walker Memorial Basement Radio).
Figure 2.2.1a – A small radio with a mast antenna

Figure 2.1.1b (left) – Shows the motion described in the text as *pitch*.

Figure 2.1.1c (right) – Shows the motion described in the text as *direction*.
After ones best with the tuner, the antenna pitch and direction should be adjusted. At first
the antenna is moved around wildly and mental notes of the effects are made. After
getting a general idea of the best way to point the antenna, the antenna is pointed in that
direction and fine-tuning of the antenna's position can commence. We start by moving
just the pitch, then adjusting the direction, back to adjusting pitch, again the direction,
and so forth. At some point, any further motion of the antennae will simply worsen
reception. This is a local minimum.

Attaining this minimum static position is our final goal in our radio model, and
this effort can be expressed mathematically and generalized such that all optimization
problems can be considered function minimization problem, and vice versa.

To translate all optimization problems into function minimization problems
consider the system we wish to optimize—\( f(x) \). If in optimizing this system, we are
minimizing another characteristic of the system (such as static or error), we are done—we
have a function to minimize, the error function. If optimizing this system does not mean
minimizing error, then there is only one possibility left\(^4\): we are trying to get as close to
some number, \( K \), as possible (in the case of getting the biggest number possible, we can
express it as getting close to positive infinity). In this case, we rewrite the problem as
\( g(x) = |K - f(x)| \). We have expressed optimizing \( f(x) \) as minimizing \( g(x) \). We have just
shown that optimization problems are also function minimization problems. The other
direction follows a similar argument.

\[ \epsilon \]

The paradigm of this scenario and this proof is to point out that functional
minimization and system optimization are the same. When you optimize a system, you
can re-express it as trying to minimize another function, such as error. Now, whenever
we have an optimization problem, such as simulating protein folding, calculating
chemical reactions, or deciding the optimal transistor layout for a microprocessor, we can
rest assured that we can use algorithms that find a function's minimum to give us a
correct solution.

\(^4\) If neither of these two cases describes the system you are trying to "optimize", then we may be speaking
about two entirely different problems. The paper is currently using the conventional meaning of
optimization as used in operations research.
2.2 Background of Optimization and Research Motivation

There are many optimization algorithms, each with different capabilities, characteristics, and tradeoffs. Because of tradeoffs, there is no single "best" algorithm. Instead, engineers often have to choose their algorithms based on the situation. The following paragraphs present a sampling of these algorithms and place the Nelder-Mead algorithm within context. This will give an idea why Nelder-Mead is considered, in many situations, the method of choice.

Golden Section Search

The Golden Section Search is a minimization method that uses what is called "bracketing". This is the function minimization analog to the bisection method for finding zeros of roots. The algorithm starts with three points \{a, b, c\} called "brackets" (Figure 2.2.1a). The two outer "brackets" \[a, c\] define the boundaries of the problem with the prerequisite that somewhere between these two brackets is a solution. The middle "bracket", \(b\), is chosen a distance \(k\), as determined by the golden ratio, from one of the outer "brackets". The search's bracket interval placement, \(k\), is optimal at \((3 - \sqrt{5})/2\) of the distance from one end of the bracket. This interval placement guarantees that with each iteration, the search space is reduced to 0.618 that of the previous as we shall see. These three points divide the problem into two regions. The solution is either the interval \(\langle a, b\rangle\) or the interval \(\langle b, c\rangle\). After setting up these "brackets", another point is place in the larger of the two intervals. For now, let's say \(\langle b, c\rangle\) is the larger interval. Depending on the situation, this new point between \(b\) and \(c\) will determine whether we replace \(c\) with this new point (Figure 2.2.1b), or replace \(b\) with this new point and replace \(a\) with the old \(b\) (Figure 2.2.1c). This gives us a new set of brackets, \{a, b, c\}. After a few iterations, we might get something that looks like Figure 2.2.1d. We stop when our interval \(\langle a, c\rangle\) is smaller than some preset value.

---

5 GR: In following mathematics convention, this paper uses braces "{\}" to denote sets. Brackets and parentheses are used in the following manner.

"By the segment \(\langle a, b\rangle\) we mean the set of all real numbers \(x\) such that \(a < x < b\). By the interval \([a, b]\) we mean the set of all real numbers \(x\) such that \(a \leq x \leq b\). Occasionally we shall also encounter 'half-open intervals' \([a, b)\) and \((a, b]\); the first consists of all \(x\) such that \(a \leq x < b\), the second of all \(x\) such that \(a < x \leq b\)." (from W. Ruden, p. 31. See endnote)
Figure 2.2.1a – Bracketing in the Golden Section Search

Figure 2.2.1b – First iteration of Golden Section Search. A point between $b$ and $c$ is tested. Upon finding the new point to be better than $c$, $c$ is replaced by the new point to form the new interval.
This algorithm is very simple and self-contained. It falls into a family of algorithms called direct search algorithms. Direct search algorithms are algorithms that require only function evaluations. No other knowledge about the system is used. The Golden Section Search's greatest limitation is it can solve only single dimensional problems. Brackets on a one-dimensional function can help us isolate a region; reducing our search space each time. Unfortunately, there is no method available to isolate such a region in multiple dimensions.
Parabolic Interpolation Method

The Parabolic Interpolation method is another direct search method, which starts by choosing three points \( \{p, q, r\} \) and evaluating the function at these points. The points describe a unique parabola. Recall that we can uniquely describe any parabola with the equation \( y = ax^2 + bx + c \), and three coefficients. Since we have three unknowns \( \{a, b, c\} \) and we have three equations \( y_p = ap^2 + bp + c; \ y_q = aq^2 + bq + c; \ y_r = ar^2 + br + c \)}, we can solve the system of equations for \( a, b, \) and \( c \). Since the vertex is the minimum point in a parabola, we make a guess that the function’s minimum is also there. We replace the worst of our original three points with the vertex and start over again (see Figure 2.2.2a). The loop stops when the three points are within some preset range. If we ever find the case where the parabola predicts the function exactly, we just move the two outer points closer together.

Figure 2.2.2a – The Parabolic method assumes the vertex of the parabola is a good estimate for a possible minimum
The Parabolic Interpolation method has two primary limitations. First, it operates only in one dimension. Attempts to scale the method to multiple dimensions usually make the algorithm unwieldy. Second, the method is relatively unstable, and can just as easily jump to a parabolic maximum as it can to a minimum. It can even jump past the solution (See Figure 2.2.2b for an illustration of this situation). There are remedies for the second problem. Brent’s Method is one such. It involves switching between Parabolic Interpolation and the Golden Section Method. Unfortunately, these remedies are still restricted by the first limitation.

Conjugate Gradient Methods

The methods considered above can only solve one-dimensional problems. We will now look at methods that solve multidimensional problems. The Conjugate Gradient Method is a multidimensional algorithm that makes use of a one-dimensional sub-algorithm. This algorithm evaluates a function \( f(v) \) and its gradient \( \nabla f(v) \). The gradient is used to approximate the function’s curvature. N gradients are calculated, each using any one-dimension optimization algorithm, and inserted into an N dimensional quadratic

\[ \text{The directional derivative is a maximum in the direction of the vector } \nabla f' \text{ (from H. Greenspan, p. 509. See endnote). It is a vector comprised of the first partial derivatives in each of the unit directions. In 2-space, } \nabla f = f_i + f_j. \]
approximation formula \( f(x) = c - b \cdot x + \frac{1}{2} x \cdot A \cdot x \). Because this formula requires \( N^2 \) numbers and each gradient evaluation provides \( N \) numbers, approximately \( N \) gradient evaluations will suffice.

Although the gradient methods are generally faster than direct search methods, derivatives are not always obtainable as not every continuous function is differentiable. A simple example is the absolute value function. Not all differentiable functions are continuous. These types of functions can easily stump gradient methods or render them useless.

**Multidimensional Direct Search and the Nelder-Mead Simplex**

The following discussions list Nelder-Mead’s limitations and requirements. A detailed description of the mechanics is presented in section 2.3; it will not be included here. In its most simple form, Nelder-Mead can be compared to the movement of an amoeba on a slope. This is a smart amoeba—when it encounters a steep slope, it takes big, aggressive steps; when it encounters a shallow slope, it takes small, cautious steps. If it encounters a bottleneck, like a funnel, the amoeba shrinks and oozes down the funnel. This is how Nelder-Mead simplex method works. It uses a geometric shape called a simplex that morphs much like our amoeba. The simplex uses clues based on its current environment to determine its next step. In result, the Nelder-Mead oozes itself to the solution.

Since its publication in 1965, Nelder-Mead has been widely used and studied primarily for the following reasons. First, it is an elegantly self-contained, direct search method that works in any number of dimensions. Most multidimensional algorithms make use of a one-dimensional sub-algorithm\(^7\). This is not true with Nelder-Mead, which can function in one-dimensional space as well as it can in ten-dimensional space. Second, Nelder-Mead is robust. Although it is not very fast compared to the derivative dependent counterparts, it is considered the best method when starting conditions are vague. In cases where the problem is not differentiable, Nelder-Mead is one of the few options available. Third, among multidimensional direct search methods, Nelder-Mead is one of the fastest\(^8\). There are special situations when others are faster but these are

---

\(^7\) The Conjugate Gradient Method is an example of such an algorithm.

\(^8\) Although Nelder-Mead is one of the fastest, it is still slow compared to its gradient counterparts.
infrequent. Because of these advantages, Nelder-Mead is the algorithm that is implemented in most optimization programs.⁹

There are deficiencies. Nelder-Mead can fail to converge, and speed is a significant limitation especially for higher dimension, computationally intensive problems. It is common to find optimization problems that take hours or weeks to run on a computer using Nelder-Mead. Worse, where one starts the simplex can have a large impact on how long it takes to find a solution. If one starts at a “bad” location, the method may take much longer to run! We will see examples of this in Chapter 4.

2.3 **Nelder and Mead’s Downhill Simplex Method**

**The Algorithm**

In 1962, Spendley et al. introduced an ingenious idea for finding functional minimums. Their method starts by creating and placing a simplex in the N-plane. This N-plane exists in the minimization space, which spans (N+1)-space. After creating this simplex, Spendley compares each of the vertices and moves the simplex downhill without explicitly calculating a gradient. In 1965, Nelder and Mead published an algorithm that used Spendley’s simplex. They further refined and enhanced the theory on how the simplex should move. This section describes how Nelder-Mead works.

In the radio scenario introduced in section 2.1, reception is maximized and static minimized by adjustments to the antenna pitch and direction (Figure 2.3.1). Assume the following equation describes the level of static near our starting position:

\[ f(x, y) = (x + y)^2 + 4(x - y)^2 \]

---

⁹ Just to name a few: MATLAB® implemented Nelder-Mead as the function minimization algorithm. MultiSimplex® uses Nelder-Mead is the bases for their commercial optimization software. Numerical Recipes in C by W. Press presents Nelder-Mead as one of their multidimensional optimization algorithms.
Figure 2.3.1 – Reprint of Figure 2.1.1b-c. Pitch (left) and Direction (right).

Figure 2.3.2 – A plot of the function, $f(x, y) = (x + y)^2 + 4(x - y)^2$

Here, $x$ denotes the position of the antenna’s pitch, $y$ the antenna’s direction, and $f(x, y)$ denotes the level of static—the larger the $f(x, y)$, the more static. These variables are an
arbitrary quantification of our system; the actual scale is irrelevant to our example. For simplicity, an increase in pitch is up and direction increases clockwise. As noted before, we are limiting ourselves to the region near our starting position.

If the antenna starts at \( x = 15, y = 10 \), the model provides a static level \( f(x, y) = (x + y)^2 + 4(x - y)^2 = (15 + 10)^2 + 4(15 - 10)^2 = 25^2 + 4(5^2) = 725 \). This still tells us nothing about how to adjust our antenna. We need to create a simplex to utilize Nelder-Mead, a geometrical figure consisting, in \( N \) dimensions, of \( N + 1 \) points (or vertices). Since we have only two free variables \( \{x, y\} \), we have an \( N = 2 \) dimensional problem. Our simplex will have three points/vertices—a triangle. At the start, we only have one point, our starting position \((15, 10)\). Our initial simplex’s vertices can be arbitrary so long as the simplex spans the \( N \) dimensions. We recommend starting with points close to each other. Let us choose the set \( \{(15,10), (16, 10), (15, 11)\} \) as our starting simplex. In our radio scenario, this translates to adjusting our antenna twice. First, the pitch was increased one unit to \((16, 10)\), producing a static level of \( f(x, y) = (16 + 10)^2 + 4(16 - 10)^2 = 26^2 + 4(6^2) = 820 \). Next, the antenna is returned to \((15, 10)\), and rotated clockwise one unit to \((15, 11)\). The static is now defined as \( f(x, y) = (15 + 11)^2 + 4(15 - 11)^2 = 27^2 + 4(4^2) = 740 \).

Three static measurements were recorded \( \{725, 820, 740\} \), such that the minimum was at the starting position \((15, 10)\). From here, Nelder-Mead looks primarily at the \textit{worst} position, \( f(16,10) = 820 \), and the average of the other \( N \) vertices \( \left( \frac{[15+15]/2, [10+11]/2} \right) = (15,10.5) \). Before continuing, let’s understand why Nelder-Mead looks at these two points. Please refer to Figures 2.3.3a and 2.3.3b. Let’s call the simplex’s worst point the \textit{tail} and the other points comprise the \textit{face}. Of the \( N + 1 \) points (here, the three points of the triangle), all \( N \) face points are better than the tail. Therefore, it is reasonable to believe the solution lies near the face. We start by testing a point call the \textit{reflection point}. This point is created by taking the vector that goes from the center of the face to the tail, and flipping the vector \( 180^\circ \). The \textit{reflection point} is calculated by: \( \mathbf{v}_r = (1 + \rho)[(\mathbf{v}_1 + \ldots + \mathbf{v}_N) / N] - \rho \mathbf{v}_{N+1} \). Here, \( \rho \) stands for the amount we should stretch this reflection vector. If \( \rho = 1 \) (the convention), then it’s simply a \( 180^\circ \) flip. If \( \rho > 1 \), we move further away from the face. If \( \rho < 1 \), we move closer. In our example, the reflection point \( \mathbf{v}_r \) would be \((14, 11)\). Evaluating our system at \( \mathbf{v}_r \), we find \( f(14, 11) = 661 \). This is better than the best point in the simplex, and as such, Nelder-Mead will
move with a large step in that direction. This bigger step expands to an expansion point calculated by: \( v_e = (1 + \rho \chi)(v_1 + ... + v_N) / N - \rho \chi v_{N+1} \). Here, \( \chi \) stands for the amount beyond the reflection scaling that we should stretch the vector. Nelder and Mead requires \( \chi > 1 \). In most implementations, \( \chi = 2 \), extending the \( v_r \) vector twice as far. In our scenario, \( v_e \) is \((13, 11.5)\). Evaluating \( f(v_r) \) produces 609.25, better than \( v_r \). Now, Nelder-Mead replaces the original worst point in the simplex, \((16, 10)\), with \( v_e \). The new simplex is \{\( (15, 10) \) \((13, 11.5) \)(15, 11)\} with the respective static of \{725, 609.25, 740\}. This completes an iteration of Nelder-Mead.

![Figure 2.3.3a (left) - The initial simplex formed. The simplex placed on a contour plot.](image)

![Figure 2.3.3b (right) - The points available for replacing the tail.](image)

![Figure 2.3.3a (left) - The second simplex formed.](image)

![Figure 2.3.3b (right) - The points available for replacing the tail.](image)
The second iteration starts with this new simplex. From Figures 2.3.4a and 2.3.4b, we see that we have a new tail (15, 11) and a new face. The face’s center is \(\left(\frac{15+13}{2}, \frac{10+11.5}{2}\right) = (14, 10.75)\). The reflection point \(v_r\) is (13, 10.5) with a radio static level of \(f(v_r) = 577.25\). Again, since \(v_r\) is better than our simplex’s best point, we try an expansion point \(v_e\). Evaluating \(v_e = (12, 10.25)\) gives us \(f(v_e) = 507.3125\). Again, we replace our worst point with \(v_e\) giving us new simplex \{(15, 10) (13, 11.5) (12, 10.25)\} and static of approximately \{725, 609.25, 503.3\}, respectively. This concludes our second iteration.

The Nelder-Mead method consists of a series of such iterations. It always solves for \(v_r\) first and compares that against the best point. In both our trials, \(v_r\) was better. In the case where \(v_r\) is not better, i.e. the same or worse, our logic proceeds differently. In this situation, we may replace our worst point with the reflection point or with what is called a contraction point. There are two types of contraction points—inside and outside. Both are points that lie somewhere between the worst point and \(v_r\). The simplex can also shrink. Shrinking is simply shrinking all sides of the simplex to create a smaller version of the original. Figure 2.3.5 summarizes the logic of each iteration and Table 2.3.1 describes the calculation of the points depicted in Figure 2.3.5.

<table>
<thead>
<tr>
<th>Face’s center</th>
<th>Formula</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflection</td>
<td>(v_r = (1 + \rho)v_b - \rho v_{\text{worst}})</td>
<td>(i \in {1...n+1} \land v_i \neq v_{\text{worst}})</td>
</tr>
<tr>
<td>Expansion</td>
<td>(v_e = (1 + \rho \chi)v_b - \rho \chi v_{\text{worst}})</td>
<td></td>
</tr>
<tr>
<td>Outside Contraction</td>
<td>(v_{oc} = (1 - \psi \rho)v_b - \psi \rho v_{\text{worst}})</td>
<td></td>
</tr>
<tr>
<td>Inside Contraction</td>
<td>(v_{ic} = (1 - \psi)v_b - \psi v_{\text{worst}})</td>
<td></td>
</tr>
<tr>
<td>Shrink</td>
<td>(v_i = v_{\text{best}} + \sigma(v_i - v_{\text{best}}))</td>
<td>(i \in {1...n+1} \land v_i \neq v_{\text{worst}})</td>
</tr>
</tbody>
</table>

Table 2.3.1 – The equations for each point mentioned in Figure 2.3.5
Figure 2.3.5 – Nelder-Mead logic diagram for formation of a new simplex.
At the beginning of each iteration, the algorithm checks to see if it should terminate operation by determining if the following threshold has been exceeded.

\[
\tau > \sqrt{\frac{\sum f(v_i) - (\sum f(v))/n}{n}}^2/n
\]

This equation intuitively ceases fit analysis if the function evaluated at the vertices does not differ by much\textsuperscript{10}. In our radio example, if \( \tau = 10^{-4} \), Nelder-Mead would terminate after forming \(-50\) new simplexes. Our final solution would be \((2.52 \times 10^{-5}, -1.56 \times 10^{-5})\), very close the theoretical answer \((0, 0)\).

The Problems

Eventually, after a series of tens, hundreds, thousands, or millions of such iterations, Nelder-Mead terminates. Termination does not necessarily mean that the generated solution is correct or within some distance \( \delta \) of the correct solution. For example, let us run the algorithm on the simple system:

\[
f(x_1, \ldots, x_9) = \Sigma |x_i|
\]

We know the origin is the correct solution (see Figure 2.3.6a and 2.3.6b for a plot). The first time we run Nelder-Mead (\( \tau = 10^{-4} \)) with a randomly generate starting position.

---

\textsuperscript{10} This termination criterion is from Nelder-Mead’s original paper: if the “standard error” of the \( f(v) \)’s are small, stop. Note that MATLAB® implemented the termination criteria differently. They use two thresholds and if \textit{either} one is met, the algorithm stops. The first threshold is met if the simplex fits inside an \( N+1 \) dimensional cube of side length \( \tau_x \). This is for handling a recently proven degeneracy problem in the original Nelder-Mead. It does not prevent the problem but stops the algorithm from wasting precious time in such situations. The second threshold is met if \( \tau_F > f(v_{\text{Worst}}) - f(v_{\text{Best}}) \). \( \tau_x \) and \( \tau_F \) are preset values. Both implementations work; both experience the same general problems we will now describe.
(361.88, 198.73, 330.33, 221.53, 90.69, 10.18, 10.26, 260.73, 147.94)\textsuperscript{11}, the algorithm stops at \((-3.30\times10^{-6}, 6.63, 2.70, -4.66\times10^{-1}, 97.07, 25.49, 20.00, 112.23, 89.20)\textsuperscript{12}.

Although the algorithm did bring us closer to the correct solution, clearly we are not at the solution.

A few questions probably are crossing your mind right now:

- Why does it give an incorrect solution?
- Why did it stop prematurely?
- What good is an algorithm that gives an incorrect solution?
- Didn’t you say that Nelder-Mead always get the job done?

All these questions will be answered shortly. The answers to the first two are discussed in the section 2.4, when this thesis describes the theory behind the clamping-simplex. The last two questions are answered in the next paragraph.

Even though the initial run of Nelder-Mead does not give a correct solution immediately, the algorithm brought us closer. Of the nine free variables in the problem, one is acceptably close, three others came within 10, and only two variables moved away from the solution. Overall, most of the variables made progress. This is important because in practice, the way we use Nelder-Mead is to take the new “suspected” solution and call this our new starting point. Here, we will run the algorithm again. This time, even more variables will approach the correct solution. Those that were already at the correct solution will tend to stay there. We stop running Nelder-Mead when our starting point and our ending point is identical. This final run usually takes very little time because we are already starting at the solution. With most simple problems, one to three runs should be sufficient to give us the correct solution.

\textsuperscript{11} These numbers were rounded for readable. The actually starting position is \(3.6188264\times10^{2}, 1.9872863\times10^{2}, 3.3033056\times10^{2}, 2.2152814\times10^{2}, 9.0693486\times10^{1}, 1.0179749\times10^{1}, 1.0258997\times10^{1}, 2.6073098\times10^{2}, 1.4793935\times10^{2}\)

\textsuperscript{12} Again, the numbers were rounded off. The actual terminating position is \((-3.2963054\times10^{-6}, 6.6337504\times10^{0}, 2.7012288\times10^{0}, -4.6562688\times10^{-1}, 9.7073129\times10^{1}, 2.5485775\times10^{1}, 2.0001499\times10^{2}, 1.1223377\times10^{2}, 8.9205549\times10^{1}\). These numbers were generated with the MATLAB\textsuperscript{®} version of Nelder-Mead. As noted in a previous footnote, the termination criterion for the original Nelder-Mead’s and the MATLAB\textsuperscript{®} implementation differs. Either version will generate this same problem. Chapter 4 gives a justification for using the MATLAB\textsuperscript{®} version.
2.4 Clamping-Simplex

This section is divided into two subsections. The first describes relevant hypotheses for Nelder-Mead's speed. The second half describes the clamping-simplex, and how it may help solve some of Nelder-Mead's speed problems.

Problem, Theory on the Cause

We have seen in section 2.3 a case where running of Nelder-Mead once is not sufficient to give a correct solution. This problem is common, especially in large dimension problems. The following hypotheses are relevant to the clamping-simplex and to the cause of Nelder-Mead's non-convergence:

1. All variables affect each other, even if the variables are linearly independent.
2. Starting with variable \( x_i \) near the solution does not imply it will remain near the solution at the end.
3. Variables do not approach their respective solution together.
4. Starting position significantly affects the time required to find a solution.
5. The Nelder-Mead's use of a simplex guarantees the termination of the algorithm even if the answer is incorrect.

These five hypotheses were drawn based either on observations or on logic. We elaborate on each in the following paragraphs.

"All variables affect each other, even if the variables are linearly independent." Whenever Nelder-Mead is used to solve a problem, any variable independence that may have existed is destroyed. One variable cannot be improved without changing all others. Co-existing in the simplex relationship links variables together. To illustrate, consider the linearly independent problem: \( f(x, y) = x^2 + y^2 \). Because the variables are linearly independent, minimizing each variable separately can give the correct solution. In a Nelder-Mead simplex, this relationship no longer holds. Let's say the starting simplex was \{(0.01, 10) (0.02, 10) (0.01, 11)\}. Clearly, \( x \) is much closer to the solution than \( y \). Unfortunately, using a simplex, we no longer can solve for just \( y \) without moving \( x \). Per Nelder-Mead, the next point would be (0.025, 8). (Figure 2.4.1) Although \( y \) improved, it
did so at the cost of moving $x$ away from the solution. A simplex links all variables
together—constructively or not. This concept also explains the second hypothesis—“
Starting with variable $x_1$ near the solution does not imply it will remain near the solution
at the end.” If this relationship occurs frequently enough, $x$ will journey so far away
from the solution that it may be “caught” in a false solution.

![Figure 2.4.1 - A very thin simplex on a function. Here, an improvement on $y$ forces $x$ to be moved to
a worse location.](image)

“Variables do not approach their respective solution together.” Since we have
control over the shape and position of our initial simplex, in theory, we can conceivably
find a starting position where all the variables approach the solution at the same rate. In
practice, this is highly unlikely. A more likely scenario is a subset of the variables,
comprised of the variables with the greatest change, dominates the beginning of the
algorithm. As the subset approaches the bottom of its respective pseudo-valley, $\Delta f(x_i)$
decreases. At some point, another subset of variables will begin to dominate. In most
Nelder-Mead executions, we normally find only a few variables making significant
progress at any moment. See Figure 2.4.2.
Figure 2.4.2 - The progress of each variable for the system, $\Sigma |x_d|$, is plotted as a function of the number of simplexes formed. The highlighted region near 100 shows that only three variables (red, green, and blue) are making significant progress towards their respective solution.

"Starting position significantly affects the time required to find a solution." Since variables approach the solution relatively independently, and since variables affect each other, choosing a good starting point can be crucial to the algorithm running time. A small change in the starting position can change a cooperative variable into a destructive one. Therefore, if we are able determine a good starting position for the algorithm and move our starting simplex there, we may be able to reduce the running time of the algorithm.

"The Nelder-Mead’s use of a simplex guarantees the termination of the algorithm even if the answer is incorrect." This last concept is based on a Nelder-Mead property addressed by MATLAB® (v5.2). In recent publication, it was proven that there exist situations where Nelder-Mead is guaranteed not to converge on the correct solution. The case is that one dimension of the simplex becomes too small that it will never changes in future iterations. The variable varies from vertex to vertex by a minuscule amount $\varepsilon$ that it weakly "locks" the variable to that incorrect value. This does further damage because
all future evaluations on the simplex will be based on this incorrect value, throwing off other variables as well. The Nelder-Mead implementation in MATLAB® handles this problem by limiting the minimal size of the simplex to minimize the number of wasted iteration.

**Clamping-Simplex, the Concept**

The clamping-simplex evaluates under normal Nelder-Mead methodology until some variable(s) \( x \) has potential to destructively interfere with the progress of another, \( y \). "Clamps" are then placed on the variables \( (y) \) that could be hindered. Placing clamps is a method of temporarily assigning a constant value to a variable. This eliminates potential for further negative side effects from \( x \). When the danger has passed, the clamped variable is "unclamped", and allowed to continue its minimization process.

The implementation difficulties include complex logic used to determine when variables should be clamped. There are many heuristics to help identify situations when a variable will start hindering its peers. The two clamping-simplex methods (stmins and dmins) described in this paper are based on the following heuristics:

- The variable’s start position is the most likely time destructive interference will occur.
- Large change in one variable \( x \) increases the potential for it to interfere with another variable \( y \).

The remaining sections in this chapter describe these two clamping-simplex methods and how they each make use of the heuristics.

**stmins: Theory and Implementation**

Stmins clamps exclusively in the beginning. Here, clamping is used to find a better starting position for the Nelder-Mead simplex.

When a simplex starts, it is unlikely that any of its \( N \) variables will be at the solution. More likely, each will be a different distant from the solution, and is initially reside on the side of a **pseudo-hill**. A pseudo-hill is the hill analog of the pseudo-valley defined earlier. It is a hill in less than \( (N+1) \)-space. Because each variable will be sitting on a different slope, the variable on initially the steepest slope has the greatest potential to
interfere with the other variables. In a different scenario, if each variable were resting near the floor of their respective 1-D pseudo-valley (the valley when all other variables are treated as constants), none would be on a slope great enough to significantly affect other variables.

We can remove the potential destructiveness of the starting position by moving it closer to the floors of pseudo-valleys. The clamping-simplex method can help us find this better starting position—a better guess. This version of the clamping-simplex (called stmins) does all clamping in the beginning. The primarily intent is to quickly bring the simplex to a better starting position, placing each variable in a pseudo-valley.

Stmins starts with the given starting point (supplied by user) and clamps all but the first variable. The immediate goal is to lodge this variable in the floor a pseudo-valley. Since the situation is reduced to a one variable problem, stmins can make use of any direct search algorithm. Nelder-Mead in one dimension was chosen since nothing is known about the problem's bounds.

The logic for deciding when the point is near a valley floor (a minimum) can be implemented as follows. One heuristic is the direction of steps taken. Let the sequence of best points be denoted \([b_1, b_2, b_3, \ldots, b_n]\). Let the scalar \(d_n = b_{n+1} - b_n\). The best points sequence can be expressed as \([d_1, d_2, d_3, \ldots, d_n, 1]\). The heuristic is to stop when the signs \((+/-)\) of \(d_i\) changes at least twice (or \(v\) times where \(v \geq 2\)). As the simplex (1-D) travels toward the valley floor, its step size increases and moves in a uniform direction. When the simplex passes the minimum, it reverses direction through a contraction (the first sign change). A second sign change suggests that the simplex is very close to the minimum, possibly bracketing the minimum.

A different heuristic is the number of expansions versus the number of contractions. A counter is initialized to zero. Whenever the simplex expands, stmins increments the counter by one. Whenever it contracts, the counter is decremented by \(\rho\) (\(\rho > 0\)). If the counter goes below some preset value \(u\) (\(u \leq 0\)), stmins completed it work on this variable. The heuristic is based on the following logic. Nelder-Mead expands when it is far away from the minimum—to approach the vicinity. It contracts when it is near—to refine the location. When the counter is negative, stmins has

---

13 inner contraction, outer contraction, shrinking
contracted $1/\rho - \psi$ more times than it had expanded. This process quickly brings the simplex near the valley floor (if Nelder-Mead is implemented conventionally\(^{14}\)). Stmins was implemented on this second heuristic.

Up to this point, stmins has brought only the first variable near a pseudo-valley floor. This process is repeated with each variable. For each variable, clamps are placed on all other variables except the current one. When it has completed this for every variable, it will have found a new starting point. Every variable in this new starting position is close to a pseudo-valley floor. No single variable can perturb another very much before losing its dominance. It is now less likely for any single variable to be permanently disrupted by others because of a bad starting point. This places the new staring simplex in a controlled environment. After stmins finds this better starting position, it continues the solution search using Nelder-Mead to complete the process.

**stmins: Example**

To illustrate this, let us work through an example. Consider, again, the radio scenario. The function is $f(x, y) = (x + y)^2 + 4(x - y)^2$ with the initial starting point at $(15, 10)$. Let us set $\rho = 1$ and $\psi = 0$. Stmins starts by clamping all variables except the first. Here, $y$ will always be evaluated as 10 so long as it is clamped. This creates an initial simplex $\{(15, 10), (16, 10)\}$, which evaluates to $\{725, 820\}$, respectively. Calculating the reflection point $r = (14, 10)$ gives $f(r) = 640$. Since $r$ does better than $(15, 10)$, stmins tries the expansion point $e = (13, 10)$, $f(e) = 565$. It keeps the expansion point; the new simplex is $\{(13, 10), (15, 10)\}$. Since this iteration carried out an expansion, stmins increments the counter from 0 to 1. Figure 2.4.2 illustrates.

\(^{14}\) $\rho = 1, \chi = 2, \psi = 0.5, \sigma = 0.5$
When the simplex reaches \((6, 10) (5.5, 10)\), it is resting near the bottom of a valley. Here, \texttt{stmins} clamps the first variable \(x\) at 6, and unclamps the next variable, \(y\). Since \(y\) also happens to be the last variable, clamping is done after this. \texttt{stmins} carries out the same process; see Figure 2.4.3 for an illustration. When all the independent variables have been worked on, the new location will be the new starting point for Nelder-Mead. \texttt{stmins} proceeds with the conventional Nelder-Mead routine for the remainder of the search. Chapter 4 presents test results of \texttt{stmins}.

\textbf{Figure 2.4.4} – Similar to Figure 2.4.3 in notation. The \(y\) variable is being optimized while \(x\) is clamped at 6.
**dmins: Theory**

Clearly, clamping can also take place in the mist of executing Nelder-Mead. Here, a dynamic approach to clamping is explored. The method involves maintaining a record of the points explored thus far, and determining when clamping is appropriate for which variables. *dmins* starts with three preset values, $\tau$, $\beta$, and $\rho$ ($\tau \gg \beta$, $\rho \approx 3$). It then creates and executes simplex calculation in the same manner as Nelder-Mead. At each stage, when a new simplex is created, *dmins* determine whether any variables $\{x_i\}$ are changing by more than $\tau$ for each of the last $\rho$ iterations. If so, it searches to see if any variables $\{x_i\}$ have changed by less than $\beta$ in the last $\rho$ iterations. *dmins* places clamps on the variables $\{x_i\}$ to keep these variables constant while $\{x_i\}$ is minimized.

Like *stmins*, clamp release can be implemented in various ways. One is to leave the clamps in place so long as $\{x_i\}$ is changing by more than $\tau$. When the change falls below $\tau$, the clamps are released (simultaneously or one per iteration). Another method is to use a heuristic similar to that in *stmins*. The number of expansion and contracts is tallied at each iteration. When the counter falls below $\nu$, the clamps on $\{x_i\}$ are released.

By dynamically clamping and unclamping, the clamping-simplex method can adjust to the various topographies and situations, and react accordingly. Disruption potential caused by peer variables is kept under constant supervision.

**Taking Full Advantage of Clamping**

Although previously not viable, since the clamping-simplex methods treat and evaluate each variable independently, it is now possible to avoid reevaluating variables that are clamped. Instead, former values from previous executions are substituted accordingly. For example, if $f(x, y) = x^2 + y^2$ and $y$ were clamped at 6, all evaluations of $f(x, y)$ can be treated as $f(x, 6) = x^2 + 36$. This is desirable because floating-point operations on computers are expensive, usually 200%-500% as expensive as normal instructions. For problems where such substitutions are possible, much computation required for each function evaluation can be avoided.

---

15 This depends primarily on the processor's architecture. Floating-point operations on CISC processors (e.g. Intel x86) is considerably more expensive than on the RISC processor. (J.D. Ruley, see endnote)
Chapter 3: Algorithm Requirements

Algorithm requirements need to be explicitly stated because algorithms serve such a large array of use that requirements differ for each. This chapter describes the requirements for stmins. Section 3.1 describes the functionality requirements. Section 3.2 defines the runtime performance requirements of the clamping-simplex. Section 3.3 covers any other requirements for the clamping-simplex method.

3.1 Functionality

Algorithmic correctness is achieving the desired input/output relationship at all times. According to this definition, Nelder-Mead is not correct. McKinnon recently proved that Nelder-Mead does not necessarily halt with the optimal solution for some two-dimensional, strictly convex problems. (See endnote) On the other hand, algorithmic functionality is defined by the user. If an output that is better than the input is considered satisfactory, then Nelder-Mead is always functional. If the functional requirement is coming within ε of the answer, then Nelder-Mead fails at times.

Given these definitions, we define functionality for the clamping-simplex family as follows. Since Nelder-Mead is imbedded in the clamping-simplex and since the clamping-simplex is being compared to Nelder-Mead, functionality is defined according to the same standards as Nelder-Mead:

- The final point ("solution") must be an improvement compared to the start point.
- Multiple runs must produce a final correct answer.
- The algorithm always halts.

By meeting these requirements, the clamping-simplex methods are ensured to be functionally no worse than the Nelder-Mead method.

3.2 Performance

Performance is a benchmark that commonly involves the measurement and comparison of time or space. With time, the amount of computation time saved is compared. More important than isolated case comparisons is the comparison of the rate of growth of the algorithm’s runtime. Runtime is the number of atomic calculations required to arrive at
an acceptable output. Often, the runtime increases as a function of complexity. One example of complexity for Nelder-Mead is increasing the number of independent variables. Another is increasing the starting point’s distance from the solution. As the complexity increases, the rate of growth always becomes the dominant factor in algorithm runtime.

Algorithms usually have overhead (time) costs. In Nelder-Mead, this is the evaluation of N+1 vertices to build the initial simplex. Even if the problem halts from the start, this is the minimal cost. As problems increase in complexity, these overhead become insignificant compared to the overall cost. When measuring time performance, algorithms are usually tested multiple times, each using increasing problem complexity. This helps determine the rate of growth without the discrepancies created by the overhead.

The performance requirement for stmins is to demonstrate that it can run faster than Nelder-Mead in many instances. The following questions will be answer during the testing.

- Which method requires less function evaluations?
- Which method requires less floating-point operations?
- Under what condition does the clamping-simplex method works best?

When, measuring speed, time will be recorded but will not be used as the performance indicator because discrepancies in computing environments can skew measurements.

By space, we measure the amount of computer memory required by the algorithm. Again, more important is the rate of growth of space requirements. In modern computing, space is of less importance than speed and so the focus for stmins will be on speed.

Presently, no explicit rate of growth formula for Nelder-Mead’s runtime exists. Since Nelder-Mead is imbedded in the clamping-simplex methods, the rate of growth for stmins will not be given. Instead, performance will be demonstrated by test cases, as done by Nelder and Mead in their 1965 publication. The choice of test case and the results are presented in the following chapter.
3.3 \textit{Accessibility}

The last standard required of the clamping-simplex methods is accessibility. Accessibility is the limitation imposed by current computer technology. The method must run in reasonable time on computer technology available to the average research laboratory. A method that requires technology available to a select few will be of little use. To insure this criterion is met, the algorithm was developed independent of platform. Algorithm development and testing was carried out on computers with Intel® (Pentium™ 120 MHz), AMD® (K6-III™ 380 MHz; Athlon™ 500 MHz), and Sun (Ultra2™ 333 MHz) processors. Memory and Operating System installment for these computers were 16 MB/MS Windows 95™, 64 MB/Windows 98™, 96 MB/Windows 98™, and 256 MB/MIT Athena Version 8.4 (UNIX), respectively.
Chapter 4: Data and Analysis

Chapter 4 details the testing methodology, the results, and the analysis. Section 4.1 explains the choice for the unit of measurement. Section 4.2 presents the test cases, the reason for each test case, the results, and the analysis of the results.

4.1 Units of Measure

To compare runtime performance of the clamping-simplex methods with MATLAB's Nelder-Mead, three values were recorded and compared: the number of function evaluations, the number of floating-point operations (flops), and the actual real time taken (in seconds). Historically, functional evaluations were considered the atomic units of measure (see definition of runtime, § 3.2). The algorithm's execution time is directly related to the number of times the function was evaluated. If the function were evaluated twice as often, the algorithm would take approximately twice as long to execute. Direct search methods were compared using this unit of measure because the computational burden of every function evaluation was the same. This is not the case with the clamping-simplex methods—function evaluations are no longer equal. Functions with more variables clamped usually requires less machine time to evaluate than those with fewer clamped.

Since function evaluations are no longer an atomic unit, a new unit of measure is required. Flops were chosen because unlike function evaluations, flops maintain a stronger direct relationship with actual machine runtime. A problem that requires 1000 flops to execute requires approximately the same number of computer clock cycles, regardless of whether they came from executing a function 100 times (10 flops/execution) or half a function 200 times (5 flops/execution). This unit is a better performance indicator for comparing the clamping-simplex methods’ runtime (especially dmins) with that of Nelder-Mead. It can also be shown that the function evaluations standard is an upper bound to the to the flops measurement standard. Since each evaluation requires at most $m$ floating point operations (the function fully evaluated), a clamping-simplex method that requires the same number of function evaluations as Nelder-Mead would require the same number of flops (algorithm logic not included). On
present day computers, floating point operations are 200-500% more computationally expensive than simple instructions such as branch logic. Although it is important to develop theoretical algorithms that have very slow rates of growth, in practice, it is also important to have algorithms that have reasonable overhead as well. As such, some attention was placed on *flops*.

Many of the test cases herein take advantage of problem re-grouping. Grouping is defined here as a way of expressing a function in chunks (described in detail in § 2.4). This was done to demonstrate the flexibility of the clamping concept. Performance improvements by grouping for clamping-simplex methods were also studied in § 4.2. These tests were performed on a very ungroupable function to demonstrate possible lower bounds.

Real time can and was measured but time measurements are usually less accurate because computing environments vary from test to test. The results showed approximately a 1:1 correlation between real time and flops. Because the data shed no new insight and the inherent unreliability problems with using time as the unit of measure, real time is not further discussed in this chapter.

### 4.2 Data

Sixty-one functions were used to test *stmins*. Three of these functions have been previously used to test other optimization algorithms. Fifty-four of these functions (8-61) are comprised of variations on three functions. The functions, most of which have solutions at the origin (exceptions noted below), are:

1. Rosenbrock’s parabolic valley (Rosenbrock (1960); Nelder and Mead (1965))
   \[ y = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \]
   Solution at (1, 1)

2. Powell’s quartic function (Powell (1962); Nelder and Mead (1965))
   \[ y = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4 \]

3. Fletcher and Powell’s helical valley (Fletcher and Powell (1963); Nelder and Mead (1965))
   \[ y = 100[x_3 - 100(x_1, x_2)]^2 + [\sqrt{(x_1^2 + x_2^2)} - 1] + x_3^2 \]
where \(2\pi \theta (x_1, x_2) = \tan^{-1} (x_2/x_1), \quad x_1 > 0\)
\[\tan^{-1} (x_2/x_1), \quad x_1 < 0\]

\[\text{Solution at } (1, 0, 0)\]

\[
y = (x_1^2 + x_2^2)^2 + (x_2^2 + x_3^2)^2
\]  \hspace{1cm} (4)

\[
y = (2 + x_1^2)(4 + x_2^4) + 4(x_1 - x_2)^2 + (x_1 + x_2)^2 + x_3^6 + 2(|x_4| + |x_5|)
\]  \hspace{1cm} (5)

\[
y = 4(x_1 - x_2)^2 + (x_1 + x_2)^2
\]  \hspace{1cm} (6)

\[
y = [2 + (x_1 - 10)^2][4 + (x_2 + 5)^4] + 4[(x_1 - 10) - (x_2 + 5)]^2 + [(x_1 - 10) + (x_2 + 5)]^2 + x_3^6 + 2(|x_4| + |x_5|)
\]  \hspace{1cm} (7)

\[\text{Solution at } (10, -5, 0, 0, 0)\]

\[
\Sigma (100 - \cos^7 x_i) \quad \text{where } i \in \{1, \ldots, 9\}
\]  \hspace{1cm} (8-16)

\[
\Pi (100 - \cos^7 x_i) \quad \text{where } i \in \{1, \ldots, 9\}
\]  \hspace{1cm} (17-25)

\[
\Sigma [10 + e^{(x_1^2/18000)} \cos 4x_i] \quad \text{where } i \in \{1, \ldots, 9\}
\]  \hspace{1cm} (26-34)

\[
\Pi [10 - e^{(x_1^2/18000)} \cos 4x_i] \quad \text{where } i \in \{1, \ldots, 9\}
\]  \hspace{1cm} (35-43)

\[
\Sigma [10 + x_i^2 \cos (|x_i|/1000)] \quad \text{where } i \in \{1, \ldots, 9\}
\]  \hspace{1cm} (44-52)

\[
\Pi [10 + x_i^2 \cos (|x_i|/1000)] \quad \text{where } i \in \{1, \ldots, 9\}
\]  \hspace{1cm} (53-61)

The starting positions were generated randomly. For functions with multiple minima (i.e. trigonometric functions), bounds were placed to ensure the algorithm yielded only one solution. For example, in one of the trigonometric cases, the simplex was contained within the first half period. The stopping criteria were based on the default values of MATLAB®’s Nelder-Mead’s algorithm. The algorithm halts when either the simplex fits within an N+1 dimensional cube of side length \(\tau_X\), or when \(f(v_{\text{WORST}}) - f(v_{\text{BEST}}) < \tau_F\) (\(\tau_X = \tau_F = 10^{-4}\)). These stopping criteria should the output to within \(10^{-4}\) normalized units of the actual solution. If the algorithm does not terminate because of function degeneration, the algorithm is programmed to stop after \(10^5\) function evaluations. Restarting the algorithm at this new point normally solves the degeneration problem.\(^{16}\)

The primary difficulty encountered was choosing a starting position. Slight deviation (noise) on the starting point can have significant impact on the running time of

---

\(^{16}\) There are a few rare situations where the function will degenerate (Lagarias, & al., see Endnote) and simple restarts do not solve the problem. These exceptions are not important to this study at the moment.
Nelder-Mead (See Fig. 4.2.0). To account for this discrepancy, each starting location was tested with a sequence of 10 or 20 starting positions. Twenty were used if the function had a large number of independent variables, or if the size of the numbers involved could exceed the machine’s floating-point capabilities. Each point $p_n$ in the sequence varied from the previous point $p_{n-1}$ by $\sim 0.01\%$ the normalized distance from the origin. $p_n = (\sim 0.01\%) \times (|p_{n-1} - \theta|) \text{ s.t. } \theta = \text{zero vector}^{17}$. When the outcome is studied in aggregate, by the law of large numbers, the noise caused by starting position discrepancies would be suppressed.

**Input Noise Impact on Nelder-Mead’s Output**

![Input Noise Impact on Nelder-Mead’s Output](image)

**Figure 4.2.0** – 100 randomly generated starting points in 10 sets of 10 were tested. Each subsequent set averages $\epsilon$ times further away than the previous. Because $\tau_F = 10^{-4}$, solutions should be within $10^{-4}/n$ ($n = \# \text{ of dimensions}$) of each other. Instead, slight input deviations (noise) produced significant output deviations (~1). Since the output’s deviation is ~1 regardless of the test set, this suggests that the input’s noise has a great impact on the output’s noise than input’s distance from solution does.

$^{17}$Because these points are intended to be randomly generated, whether the solution happened to be at the origin is irrelevant; “s.t.” is a mathematic abbreviation for “such that”.
The sixty functions were divided into four test sets. The first test set (functions (1) – (5)) investigated what values for $p$ and $\nu$ are optimal. The values $\{\frac{1}{2}, 1, 2, 4\}$ were tested for $p$ and $\{0, -3, -6\}$ for $\nu$. The second test set (functions (4), (6), and (7)) helped confirmed three algorithm properties:

1. Different ways of expressing the problem had no effect on the solution or the number of evaluations required.
2. Shifting the solution away from the origin affects stmins' performance no more than it does NM's performance.
3. Expressing the problem in discrete pieces (grouping) can reduce computation (flops) required to execute the algorithm.

The third test set (functions (1) – (5)) compared stmins with Nelder-Mead using the best values of $p$ and $\nu$ (as determined from the first test set). The fourth test set (functions (8) – (61)) studied how the number of free variables affects performance.

**Test Set 1: Finding the Optimal $p$ and $\nu$**

The initial trial for the first test set was performed on two test functions $\{(1), (2)\}$. Each function tested 100 starting points. The points were grouped into series of 10, each subsequent group at least $e / i$ further away from the solution than the previous group ($i$ = number of free variables). For each starting point, all combinations of $p \in \{\frac{1}{2}, 1, 2\} \times \nu \in \{0, -3, -6\}$ were tested. The results are summarized in Figure 4.2.1A(1), Table 4.2.1A(1), Figure 4.2.1A(2), and Table 4.2.1A(2). The results suggest that the best strategies were $p = 2$ with $\nu \in \{0, -3\}$. They also suggest that $p$ has the greater impact on performance as expected; $p$ changes the size of the simplex by approximately $2^p$.

After the initial findings, all five functions were tested again, each function with the same starting positions. In this second trial, $p \in \{2\} \times \nu \in \{0, -3, -6\}$ were tested against $p = 4, \nu = 0$. The results are summarized in Figure 4.2.1B and Table 4.2.1B. Although on average $p = 4$ gave slightly better results, it did so inconsistently. For example, the strategy $\{p = 4, \nu = 0\}$ did considerably worse with function (2) whereas strategy $p = 2, \nu = 0$ consistently yields better results over $p \in \{\frac{1}{2}, 1\}$. This run analyzed 2000 random starting positions.
Figure 4.2.1A(1) - Different values for $p$ and $u$ are tested on Function (1) to find the optimal parameter strategy. Each data group consists of 10 test points. The average runtime (number of evaluations) for each data group is depicted. A smaller runtime means better performance.

**TABLE 4.2.1A(1)**

<table>
<thead>
<tr>
<th>Test Groups</th>
<th>$p=0.5$</th>
<th>$p=0.5$</th>
<th>$p=0.5$</th>
<th>$p=0.5$</th>
<th>$p=1$</th>
<th>$p=1$</th>
<th>$p=1$</th>
<th>$p=2$</th>
<th>$p=2$</th>
<th>$p=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$u=0$</td>
<td>$u=-3$</td>
<td>$u=-6$</td>
<td>$u=0$</td>
<td>$u=-3$</td>
<td>$u=-6$</td>
<td>$u=0$</td>
<td>$u=-3$</td>
<td>$u=-6$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>199.2</td>
<td>199.2</td>
<td>229.4</td>
<td>178.3</td>
<td>178.3</td>
<td>194.2</td>
<td>172.5</td>
<td>172.5</td>
<td>174.5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>259.8</td>
<td>259.8</td>
<td>269.4</td>
<td>223.9</td>
<td>223.9</td>
<td>249.1</td>
<td>245.4</td>
<td>245.4</td>
<td>219.5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>299.5</td>
<td>299.5</td>
<td>342.2</td>
<td>279.1</td>
<td>279.1</td>
<td>292.4</td>
<td>264.4</td>
<td>264.4</td>
<td>272.7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>355.7</td>
<td>355.7</td>
<td>368.0</td>
<td>333.2</td>
<td>333.2</td>
<td>346.7</td>
<td>291.6</td>
<td>291.6</td>
<td>302.3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>806.3</td>
<td>806.3</td>
<td>803.0</td>
<td>765.7</td>
<td>765.7</td>
<td>796.6</td>
<td>649.1</td>
<td>649.1</td>
<td>741.9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>769.1</td>
<td>769.1</td>
<td>779.2</td>
<td>603.7</td>
<td>603.7</td>
<td>760.7</td>
<td>181.0</td>
<td>181.0</td>
<td>185.0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2,221.5</td>
<td>2,221.5</td>
<td>2,242.3</td>
<td>2,217.0</td>
<td>2,217.0</td>
<td>2,211.2</td>
<td>181.1</td>
<td>181.1</td>
<td>1,872.0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2,100.0</td>
<td>2,100.0</td>
<td>2,079.9</td>
<td>2,524.1</td>
<td>2,524.1</td>
<td>2,151.5</td>
<td>181.5</td>
<td>181.5</td>
<td>185.5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>218.7</td>
<td>218.7</td>
<td>242.7</td>
<td>194.7</td>
<td>194.7</td>
<td>206.7</td>
<td>182.7</td>
<td>182.7</td>
<td>186.7</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4,737.5</td>
<td>4,737.5</td>
<td>4,788.1</td>
<td>195.8</td>
<td>195.8</td>
<td>4,946.6</td>
<td>183.8</td>
<td>183.8</td>
<td>187.8</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2.1A(1) - The average of the number of evaluations needed to solve Function (1) using different $p$ and $u$ values. These numbers were used to graph Figure 4.2.1A(1).
Figure 4.2.1A(2) – Different values for $\rho$ and $\nu$ are tested on Function (2) to find the optimal parameter strategy. Each data group consists of 10 test points. The average runtime (number of evaluations) for each data group is depicted. A smaller the runtime means better performance.

**TABLE 4.2.1A(2)**

<table>
<thead>
<tr>
<th>Test Groups</th>
<th>$\rho=0.5$</th>
<th>$\rho=0.5$</th>
<th>$\rho=0.5$</th>
<th>$\rho=1$</th>
<th>$\rho=1$</th>
<th>$\rho=1$</th>
<th>$\rho=2$</th>
<th>$\rho=2$</th>
<th>$\rho=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\nu=0$</td>
<td>574.1</td>
<td>574.1</td>
<td>586.7</td>
<td>489.7</td>
<td>489.7</td>
<td>527.3</td>
<td>493.6</td>
<td>493.6</td>
<td>468.7</td>
</tr>
<tr>
<td>$\nu=-3$</td>
<td>560</td>
<td>560</td>
<td>602.9</td>
<td>549.4</td>
<td>549.4</td>
<td>553.3</td>
<td>505.6</td>
<td>505.6</td>
<td>532.3</td>
</tr>
<tr>
<td>$\nu=-6$</td>
<td>598</td>
<td>598</td>
<td>654.1</td>
<td>557</td>
<td>557</td>
<td>606.2</td>
<td>489.9</td>
<td>489.9</td>
<td>531.1</td>
</tr>
<tr>
<td>$\nu=0$</td>
<td>732.7</td>
<td>732.7</td>
<td>767.2</td>
<td>671.4</td>
<td>671.4</td>
<td>691</td>
<td>585.4</td>
<td>585.4</td>
<td>617.1</td>
</tr>
<tr>
<td>$\nu=-3$</td>
<td>735.4</td>
<td>735.4</td>
<td>744.4</td>
<td>706</td>
<td>706</td>
<td>693.7</td>
<td>652.6</td>
<td>652.6</td>
<td>666.3</td>
</tr>
<tr>
<td>$\nu=-6$</td>
<td>715</td>
<td>715</td>
<td>766</td>
<td>699.6</td>
<td>699.6</td>
<td>698.6</td>
<td>662.8</td>
<td>662.8</td>
<td>694.6</td>
</tr>
<tr>
<td>$\nu=0$</td>
<td>775.2</td>
<td>775.2</td>
<td>799.4</td>
<td>720.9</td>
<td>720.9</td>
<td>763.5</td>
<td>703.7</td>
<td>703.7</td>
<td>848.1</td>
</tr>
<tr>
<td>$\nu=-3$</td>
<td>824.9</td>
<td>824.9</td>
<td>897.6</td>
<td>828.8</td>
<td>828.8</td>
<td>819.5</td>
<td>753.3</td>
<td>753.3</td>
<td>752.1</td>
</tr>
<tr>
<td>$\nu=-6$</td>
<td>886.4</td>
<td>886.4</td>
<td>899.2</td>
<td>825.7</td>
<td>825.7</td>
<td>866.6</td>
<td>805.5</td>
<td>805.5</td>
<td>849.7</td>
</tr>
<tr>
<td>$\nu=0$</td>
<td>997.6</td>
<td>997.6</td>
<td>1070.7</td>
<td>913.8</td>
<td>913.8</td>
<td>996.9</td>
<td>918.4</td>
<td>918.4</td>
<td>908.8</td>
</tr>
</tbody>
</table>

Table 4.2.1A(2) – The average of the number of evaluations needed to solve Function (2) using different $\rho$ and $\nu$ values. These numbers were used to graph Figure 4.2.1A(2).
Figure 4.2.1B – Different values for $\rho$ and $u$ are tested on Functions (1) – (5) to confirm that $\rho = 2$, $u = 0$ is the optimal parameter strategy. Each data group consists of 10 test points (except in Function (5) with 20). The average of the runtime of all test points for each function is depicted. A smaller runtime means better performance.

**TABLE 4.2.1B**

<table>
<thead>
<tr>
<th>Function Tested</th>
<th>$\rho = 4$</th>
<th>$\rho = 2$</th>
<th>$\rho = 2$</th>
<th>$\rho = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$u = 0$</td>
<td>$u = 0$</td>
<td>$u = -3$</td>
<td>$u = -6$</td>
</tr>
<tr>
<td>Function (1)</td>
<td>176.9</td>
<td>253.3</td>
<td>253.3</td>
<td>432.8</td>
</tr>
<tr>
<td>Function (2)</td>
<td>671.3</td>
<td>657.1</td>
<td>657.1</td>
<td>666.9</td>
</tr>
<tr>
<td>Function (3)</td>
<td>309.5</td>
<td>331.0</td>
<td>331.0</td>
<td>364.0</td>
</tr>
<tr>
<td>Function (4)</td>
<td>57.0</td>
<td>69.0</td>
<td>69.0</td>
<td>75.0</td>
</tr>
<tr>
<td>Function (5)</td>
<td>617.0</td>
<td>643.1</td>
<td>643.1</td>
<td>668.5</td>
</tr>
</tbody>
</table>

Table 4.2.1B – The average of the number of evaluations needed to solve Functions (1) – (5) using different $\rho$ and $u$ values. These numbers corresponds to those graphed in Figure 4.2.1B.
The second trial initially tested only four parameter strategies: $\rho = 2$, $\nu \in \{0, -3, -6\}$, and $\rho = 4$, $\nu = 0$. Again, with the exception of function (5), each function was tested with 100 starting points. Because the numbers in function (5) can easily exceed the machine's floating-point accuracy and capacity, only 60 points in three groups of 20 were tested. The results reconfirmed our previous findings that $\rho = 2$, $\nu = 0$ is near optimal. False minima were found on several occasions for function (3) regardless of the strategy.

For the latter half of the second trials, the remaining six strategies ($\rho \in \{1/2, 1\} \times \nu \in \{0, -3, -6\}$) were used to confirm these findings (Figure 4.2.1C and Table 4.2.1C). $\rho = 2$, $\nu = 0$ indeed is the best strategy. It was clear that $\rho$ had the greatest impact on finding an improved starting location. It is also concluded that maintaining some distance from the floor of the pseudo-valley is more desirable than being at the actual floor. This is suggested by the better performance of $\rho = 2$ compared with $\rho < 1$. 
As with the first trials, different values for $\rho$ and $\psi$ are tested on the remaining Functions (3), (4), & (5) to confirm that $\rho = 2$, $\psi = 0$ (blue) is indeed the optimal parameter strategy. The average of the runtime of all test points for each function is depicted. A smaller runtime means better performance.

**TABLE 4.2.1C**

<table>
<thead>
<tr>
<th>Function Tested</th>
<th>$\rho = 0.5$</th>
<th>$\rho = 0.5$</th>
<th>$\rho = 0.5$</th>
<th>$\rho = 1$</th>
<th>$\rho = 1$</th>
<th>$\rho = 1$</th>
<th>$\rho = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\psi = 0$</td>
<td>551.5</td>
<td>551.5</td>
<td>614.7</td>
<td>427.5</td>
<td>427.5</td>
<td>482.2</td>
<td>331.0</td>
</tr>
<tr>
<td>$\psi = -3$</td>
<td>123.0</td>
<td>123.0</td>
<td>159.0</td>
<td>87.0</td>
<td>87.0</td>
<td>105.0</td>
<td>69.0</td>
</tr>
<tr>
<td>$\psi = -6$</td>
<td>769.6</td>
<td>769.6</td>
<td>902.6</td>
<td>653.1</td>
<td>653.1</td>
<td>721.0</td>
<td>643.1</td>
</tr>
</tbody>
</table>

Table 4.2.1C – The average of the number of evaluations needed to solve Functions (3) – (5) using different $\rho$ and $\psi$ values. These numbers corresponds to those graphed in Figure 4.2.1C.
Test Set 2: Effects of Clamping and stmin Functionality

The second test set investigated the effects of clamping and of solution location. For this test set, the parameters for \( stmins \) were \( \{ \rho = 2, \nu = 0 \} \), and for algorithm termination, \( (r_X = r_F = 10^{-4}) \). These were chosen because they represent the best configuration for \( stmins \). Particular interest was placed insuring that \( stmins \) met the functionality requirements set forth earlier (§ 3.1). These investigations are important because the concept of clamping gives users an additional degree of freedom on problem expression.

The first trial (for Test Set 2) insured that different way of expressing the function did not affect the correctness of \( stmins \). Function (6) was used. The function was group in two ways:

Function (6a): \( y = [4(x_1 - x_2)^2 + (x_1 + x_2)^2] \)
Function (6b): \( y = [5x_1^2 - 6x_1x_2 + 5x_2^2] \)

Function (6a) and (6b) are equivalent mathematically. 100 test points in 10 groups were tried on the two problem groupings. No differences in the solution, in the runtime (Figure 4.2.2A), or in the simplex formation were found. The only difference was in the number of floating point operations (flops). The longer expression that involved explicit summations and multiplications required more flops.
Figure 4.2.2A – Functions (6a) and (6b) were tested with the same 100 randomly generated starting points. These 100 points are grouped into 10 groups of 10. The number of evaluations required to find a solution is recorded. This figure displays the average number of evaluations required by each group. The error bars indicate the range of the runtime.

The second trial examined the result of shifting the solution away from the origin. This is a glass box test case from software engineering that insures the functionality of stmins' implementation. Given the complexity of the function used (Function (7)), only 60 test points in groups of 20 were utilized. The outcome resembled those of shifting the starting point by a similar amount.

The third trial for test set 2 examined the benefits of clamping. Function (4) was expressed in two manners—the entire function as a single group, and as two groups:

Function (4a): \[ y = [(x_1^2 + x_2^2)^2 + (x_2^2 + x_3^2)^2] \]
Function (4b): \[ y = [(x_1^2 + x_2^2)^2] + [(x_2^2 + x_3^2)^2] \]
100 starting points in groups of 10 were tested and the number of flops for each expression was totaled. Converting the function from one grouping to two groupings produced a savings of 7.7% in the initial guesses (Figure 4.2.2B). For problems of larger dimensions and/or more groupings, higher savings can be expected.

Figure 4.2.2B – The average number of flops required to solve Function (4a) and (4b) in our test runs. Function (4b) (red) is where the function was expressed as a summation of two sub-parts.

This second test set demonstrates the effectiveness and usefulness of clamping. Clamping effects only the number of floating point operations, not the number of evaluations. Spending a few extra clock cycles on inexpensive logic yields an overall savings in computation time by reducing the number of expensive floating-point operations needed.
Test Set 3: Comparing stmins with Nelder-Mead (alone)

The third test set compared the performance of stmins with Nelder-Mead. The parameters for Nelder-Mead were \((p = 1, \chi = 2, \psi = 0.5, \sigma = 0.5)\), for stmins \((\rho = 2, \nu = 0)\), and for algorithm termination \((\tau_X = \tau_Y = 10^{-4})\). These parameters were chosen because they represented the best configuration for each respective method. Because stmins had been thoroughly tested on Functions (1) – (5) previously in Test Set 1 and there was no need to retest it, Nelder-Mead was simply tested on Functions (1) – (5) using the same starting points from Test Set 1. The tests were run under identical environments and conditions. The only difference is that Nelder-Mead is used instead of stmins. The results are summarized in Figure 4.2.3.

![Runtime Comparison of stmins and NM](image)

Figure 4.2.3 – Nelder-Mead and stmins’ runtime were compared on Functions (1) – (5). Each function ran on 100 test points in groups of 10 (function (5) ran on 60 points in 3 groups of 20). The graph shows the average runtime of the two algorithms for each function. The error bars indicate 1 \(\sigma\) of runtime range.
TABLE 4.2.3

<table>
<thead>
<tr>
<th>FUNCTIONS</th>
<th>DISTANCE FROM REAL SOLUTION</th>
<th>Nelder Mead (1σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>stmins (1σ)</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>2.12E-05 1.55E-05</td>
<td>2.68E-05 1.67E-05</td>
</tr>
<tr>
<td>(2)</td>
<td>7.34E-05 6.53E-04</td>
<td>5.71E-04 9.01E-04</td>
</tr>
<tr>
<td>(3)</td>
<td>1.89E+00 4.33E+00</td>
<td>1.20E+00 9.68E-01</td>
</tr>
<tr>
<td>(4)</td>
<td>1.04E-09 2.58E-09</td>
<td>5.15E-05 9.01E-06</td>
</tr>
<tr>
<td>(5)</td>
<td>1.32E-02 1.86E-02</td>
<td>3.63E+01 4.93E+01</td>
</tr>
</tbody>
</table>

Table 4.2.3 – The success rate is measured by the average stopping distance from the correct solution. The smaller the value, the close the output is to the correct answer.

On average, the stmins method for finding a better starting position produced both better results (Table 4.2.3 and Appendix Figures A.1–A.5) and runtime performance (Figure 4.2.3). For the majority of the tests, stmins’ suggested solutions are closer to the actual answer and required a fewer number of function evaluations. Often, in the first run alone, stmins suggests solutions that are a couple orders of magnitude closer to the correct solution than Nelder-Mead. The only exception is function (3) whose first run often yielded unacceptable solutions from both Nelder-Mead and stmins. This is the result of simplex degeneration from both algorithms. As with all simplex degeneration situations, a simple solution is restarting the algorithm at the suggested solution. These restarts yielded acceptable solutions without any significant performance differences between the two algorithms.

Despite the extra overhead cost required to find the better starting point, the saving usually exceeds this cost. Nelder-Mead converged on false solution slightly more frequently than stmins. When both give incorrect solutions, the number of variables that are incorrect in stmins are fewer. This means if the algorithm were restarted at this new location, the algorithm would converge on a correct solution faster.

As we move further and further away from the solution, the gap between stmins's improved runtime and that of Nelder-Mead widens. These results suggest that for more complicated problems or those where absolutely nothing is know about the problem (thus, starting position is likely to be far away from the solution), stmins is likely to perform better than Nelder-Mead.
Test Set 4: Effects of Number of Variables
This last test set studies the effects of increasing the number of variables. Three base functions were used (see functions (8)-(60)). The functions were either a summation of the trigonometric base functions or a product. The number of variables ranged from 2-9. This methodology tested the effects of the number of variable. It was found that as the number of free variables increased, the time spent increased only linearly for stmins compared to Nelder-Mead's time increasing at least polynomially (Appendix Figures A6–A11). We should note that this is likely to be a best-case scenario instead of an average case. Because each variable in these functions can be optimized independent of the others, the guessing methodology of stmins would perform extremely well.
Chapter 5: Discussion

Section 5.1 covers the discussion of our modified algorithm and the implications of the data generated in its validation (Chapter 4). Section 5.2 describes future work and direction for further development of these ideas.

5.1 Discussion of Analysis and Conclusions

The clamping-simplex methods significantly differ from other direct search and simplex methods in that partial evaluation of the function is often sufficient to arrive at a local minimum. Initially, stmins treats the \((N+1)\)-space function as \(N+1\) one-dimensional sub-problems and solves these smaller dimension problems first. Later, the problem is retreated as a \((N+1)\)-spaced problem. In dmins, the sub-problem at any given moment is determined based on the topography of the current point. Per iteration, the clamping-simplex concept is computationally not much more burdensome than Nelder-Mead. Nelder-Mead’s ability to locate the minimum with any desired level of accuracy is retained while situations leading to false convergences are often avoided. On the degeneracy of Nelder-Mead, one may incorrectly assume that degeneracy is undesired (Wright 2000), and should be avoid whenever possible. We have just seen (in stmins and dmins) that intentional controlled degeneration, by reducing the problem’s dimensions temporarily, can help avoid some situations for numerical degeneracy.

Unlike Nelder-Mead, the clamping-simplex is not an opportunistic method—it requires some knowledge of recent simplex actions. Although the clamping-simplex methods have this additional memory requirement, it can be implemented in \(O(N)\) (space linearly proportionally the number of variables)—same as Nelder-Mead.

Depending on the particular implementation, some clamping-simplex methods may exhibit more similarities Nelder-Mead’s characteristics than others may. Since a substantial portion of stmins uses Nelder-Mead logic (after the new starting point is located), the curvature of space affects stmins similarly to that on Nelder-Mead. Since dmins executes only part of Nelder-Mead’s logic, it should be expected to react differently at times. Overall, curvature of space will have similar impact—under moderately curved space is best.
The problem of converging on false minima has been partially corrected with \textit{stmins}. \textit{stmins} achieves this by avoiding problems caused by bad starting points. Drastic fluctuations in late problem topography may lead to false convergences, as it would to Nelder-Mead. Further study on \textit{dmins} will be needed to determine its success at avoiding false convergences. The same solution for false convergence used with Nelder-Mead can be applied to these methods. If restarting the problem at the "suspected" point brings us back to the same point (a computationally inexpensive process), the suspected point is likely to be a true minimum.

In conclusion, the concept of clamping and the clamping-simplex methods has been successfully demonstrated. Functionally, testing has shown that clamping successfully meets the requirements set forth in Chapter 3. Additionally, it has been shown that the method is capable of avoiding bad starting points that can lead to simplex degeneracy. Results suggest that on average, this method demonstrated better performance than Nelder-Mead—it can converge on a solution with fewer calculations and \textit{iterations} than Nelder-Mead. Since the clamping-simplex methods have the same problem requirements as Nelder-Mead, adapting the clamping-simplex to current uses of Nelder-Mead, such as chemistry, chemical engineering, and medicine (e.g. reaction modeling, engineering and cost parameter optimization, biological modeling), should be simple.

5.2 Future Work

Merit of the clamping concept has been demonstrated in the implementation and testing of \textit{stmins}. By clamping, not only has the algorithm gained greater control over what happens, and thus, avoiding false convergences, but also has given users control over algorithm efficiency through grouping. Presented earlier was another clamping-simplex method, \textit{dmins}. This method's more sophisticated logic may help avoid a suspected cause of simplex degeneracy.

Clamping gives us much more control over the simplex's path than before; it allows us to avoid cases of simplex degeneracy through logic. Unfortunately, despite wide use of Nelder-Mead for over 30 years, simplex degeneracy theories and proofs are
very recent. Further study of problems that lead to degeneracy will need to be researched in parallel with those for clamping-simplex method development.

Savings through grouping has been demonstrated but difficult to express in a closed mathematical form. As further theoretical understanding for Nelder-Mead and the clamping-simplex methods are developed, it should become increasingly possible to express this closed form formula.
Appendix

Figure A.1 – Nelder-Mead and stmins' runtime were compared on Function (1). The function ran on 100 test points in groups of 10. The starting points for each subsequent group are \( \sim e^{\sqrt{2}} \) times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The error bars indicate 1\( \sigma \) of runtime range.
Figure A.2 – Nelder-Mead and stmins’ runtime were compared on Function (2). The function ran on 100 test points in groups of 10. The starting points for each subsequent group are \( \sim 2\epsilon \) times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The error bars indicate 1 \( \sigma \) of runtime range.
NM & $\text{stmins}$ runtime were compared on Function (3). The function ran on 100 test points in groups of 10. The starting points for each subsequent group are $-e^{\sqrt{3}}$ times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The error bars indicate $1\sigma$ of runtime range.
Figure A.4 – Nelder-Mead and stmins’ runtime were compared on Function (4). The function ran on 100 test points in groups of 10. The starting points for each subsequent group are $-e^{\sqrt{3}}$ times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The error bars indicate $1\sigma$ of runtime range.
Figure A.5 – Nelder-Mead and stmins' runtime were compared on Function (5). The function ran on 60 test points in 3 groups of 20. The starting points for each subsequent group are $-e\sqrt{5}$ times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The error bars indicate $1\sigma$ of runtime range.
Figure A.6 – Nelder-Mead and stmins’ runtime were compared on Functions (8) – (16). Each function ran on 100 test points in groups of 10. The starting points for each subsequent group are \( \sim e \) times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The trend lines suggest possible growth rates for each respective algorithm.
Runtime Comparison of stmins & NM on $\Pi(100 - \cos^7 x_i)$

Figure A.7 – Nelder-Mead and stmins’ runtime were compared on Functions (17) – (25). Each function ran on 100 test points in groups of 10. The starting points for each subsequent group are $e$ times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The trend lines suggest possible growth rates for each the respective algorithm.
Figure A.8 – Nelder-Mead and stmins' runtime were compared on Functions (26) – (34). Each function ran on 100 test points in groups of 10. The starting points for each subsequent group are $e$ times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The trend lines suggest possible growth rates for each respective algorithm.
Runtime Comparison of stmins & NM on

\[ \Pi(10 + e^{(-x^2/18000)} \cos 4x) \]

Figure A.9 - Nelder-Mead and stmins' runtime were compared on Functions (35) – (43). Each function ran on 100 test points in groups of 10. The starting points for each subsequent group are \( e \) times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The trend lines suggest possible growth rates for each respective algorithm.
Runtime Comparison of stmins & NM on 
\[ \Sigma [10 + x_i^2 \cos(|x_i|/1000)] \]

Figure A.10 – Nelder-Mead and stmins’ runtime were compared on Functions (44) – (52). Each function ran on 100 test points in groups of 10. The starting points for each subsequent group are \( \approx e \) times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The trend lines suggest possible growth rates for each respective algorithm.
Figure A.11 - Nelder-Mead and stmins’ runtime were compared on Functions (53)–(61). Each function ran on 100 test points in groups of 10. The starting points for each subsequent group are $e$ times further from the origin than the previous group. The graph shows the average runtime within each group of the two algorithms for each function. The trend lines suggest possible growth rates for each respective algorithm. Although the trend lines suggest linear growth, the data is deceiving because both stmins and NM encountered the threshold tolerance $\tau_f$ numerous times. As a result, we see what appears to be linear growth. Additionally, stmins’ overhead for finding a new starting position in addition to the threshold adds to stmins’ evaluations making it appear the slower of the two algorithm although the solution was often closer to the correct answer.
Bibliography


RULEY, J. D. *Networking Windows NT 4.0, Workstation and Server*, John Wiley & Sons, Inc.
