Pattern Search Behavior in Nonlinear Optimization

A thesis submitted in partial fulfillment of the requirements for a

Bachelor of Science with Honors in Computer Science

from the College of William & Mary in Virginia,

by

Elizabeth D. Dolan

Accepted for ______________________

Thesis Advisor: ______________________
Virginia J. Torczon

____
Stephen K. Park

____
Michael W. Trosset

1 Acknowledgments

My investigations were made possible through the help and support of a number of individuals. Michael Lewis, Michael Trosset, and Steve Park gain my much deserved appreciation for their contributions mentioned within the text and for their time spent with me discussing problems small and large. I want here especially to thank friends not mentioned elsewhere. Lisa, Brian, Chris, and Sean helped me by reading through the thesis with me line by line, engaging in long discussions of pattern searches, plotting tedious graphs for me when I was running up on deadlines, and just being generally supportive. Their behind-the-scenes contributions kept me sane and motivated. Another individual who I have never met has earned my gratitude and respect: Danny Thorne at the University of Kentucky. I emailed him out of the blue after finding a reference on one of his old web pages to a function that had become a stumbling block to my work. He responded the following day with the answer and the explanation for the problem I was having, and I want to thank him for his act of kindness to a stranger. And Virginia Torczon, for all of your patience, wisdom and good humor, someday I'd like to take *you* to Beethoven's Inn.

Contents

1 Acknowledgments **2**

2 Motivation **7**

3 Theory Defining A Pattern Search **9**

4 Pattern Search Implementation Suite **10**

4.1 Pattern Search Base Class 11

4.2 `CompassSearch` 14

4.3 `NLessSearch` 15

4.3.1 Constructing the `NLess` Pattern 15

4.4 The Hooke and Jeeves Pattern Search 17

4.5 Optimism in Hooke and Jeeves – Birth of EdHJ Search 18

5 Test Functions **21**

6 Test and Implementation **22**

6.1 Creating the Test Functions 24

7 Results **25**

7.1 Halting Criteria 28

7.1.1 Variance vs Δ 32

7.1.2 Simplex Gradient vs Δ 33

7.1.3 Summary 33

7.2 Numerical Error in the HJ Search 34

7.2.1 History 35

7.2.2 Example 36

7.2.3 Explaining the Error 37

7.2.4 The Simple Fix 40

7.2.5 Regarding the Theory 41

7.2.6 Further Observations 42

8 Further Study **42**

9 Displays of Results **49**

9.1 Halting Criteria 49

9.2 Efficiency 51

9.3 Function Value Accuracy 59

9.4 Distance From the Optimum 68

10 Source Code **77**

10.1 Source Code for Generating Test Parameters 77

10.1.1 objective.h 77

10.1.2 objective.cc 77

10.2 Source Code for Pattern Search Base Class 80

10.2.1 PatternSearch.h 80

10.2.2 PatternSearch.cc 81

10.3 Source Code for Derived Classes 86

10.3.1 CompassSearch.h 86

10.3.2 CompassSearch.cc 87

10.3.3 NLessSearch.h 89

10.3.4 NLessSearch.cc 90

10.3.5 HJSearch.h 93

10.3.6 HJSearch.cc 93

10.3.7 EdHJSearch.h 96

10.3.8 EdHJSearch.cc 96

List of Figures

1 A simple example of pattern search 11

2 Building the `NLessSearch` Pattern from a Regular Simplex 16

3 `HJSearch` Behavior At Δ Reduction 19

4 Halting `tolerance` Values 22

5 HJSearch Numerical Error—first signs of trouble 38

6 HJSearch Numerical Error—the final bog down 39

7 Hooke and Jeeves Search and Edited Hooke and Jeeves Search. . .

0 in two variables and trial 9 in five variables 52

8 trial 0 and trial 1 in two variables 52

9 trial 2 and trial 3 in two variables 52

10 trial 4 and trial 5 in two variables 53

11 trial 6 and trial 7 in two variables 53

12 trial 8 and trial 9 in two variables 53

13 trial 0 and trial 1 in three variables 54

14 trial 2 and trial 3 in three variables 54
15 trial 4 and trial 5 in three variables 54
16 trial 6 and trial 7 in three variables 55
17 trial 8 and trial 9 in three variables 55
18 trial 0 and trial 1 in four variables 55
19 trial 2 and trial 3 in four variables 56
20 trial 4 and trial 5 in four variables 56
21 trial 6 and trial 7 in four variables 56
22 trial 8 and trial 9 in four variables 57
23 trial 0 and trial 1 in five variables 57
24 trial 2 and trial 3 in five variables 57
25 trial 4 and trial 5 in five variables 58
26 trial 6 and trial 7 in five variables 58
27 trial 8 and trial 9 in five variables 58

List of Tables

1 **Stopping Criteria vs Function Value** (with correct digits in red) 30
2 **Stopping Criteria vs Function Value** (with correct digits in red) 31

2 Motivation

Pattern searches have existed since at least the 1950's [4], but their heuristic approach to optimization had not been taken seriously until the recent development of mathematical theory supporting their use. Now that researchers have shown that pattern search algorithms do converge globally [18, 13], the goal of this project is to study the algorithms' behavior through the process of implementation and experimentation. The results should help scientists in many fields to make informed decisions when considering available optimization methods.

In the course of my investigation, I have studied several pattern search algorithms, the convergence theory behind them, and the problems for which they are most applicable. I have implemented three of the standard sequential pattern search algorithms as well as two of my own variations developed in response to my curiosity about these algorithms and the flexibility afforded by the convergence theory. I have performed extensive tests to study the efficiency and accuracy of these methods and have noted some of the obstacles inherent in creating a reasonable testing model. I have also examined multiple halting criteria commonly advocated for pattern searches and compared their abilities to track improvement in the function value. I have ferreted out numerical error that can arise in direct implementation of the canonical pattern search algorithm [8], explained the ways in which the error is generated within the algorithm, and shown the relationship between this issue of implementation and the convergence theory. While these investigations have raised more questions in my mind than they have answered, I have tried to establish a positive precedent for the

further study of the practicality of pattern search optimization methods.

While the theory supporting pattern search convergence has been improving steadily, proponents of pattern search methods have little numerical evidence to substantiate the corresponding theoretical results. Specifically lacking have been implementations of some of the most basic methods in their sequential forms. Because many pattern searches appeal to the user's common sense in their strategic simplicity, over the years, programmers have produced straightforward implementations of what can now be classified as pattern search methods; however, few of these implementations can be found in published form. Also, no suite of pattern search implementations exists under a single umbrella for ease of comparison, probably for the simple reason that research into the convergence of these algorithms has only recently provided a description of the necessary and sufficient attributes that classify a search as a pattern search with the accompanying convergence guarantees. The optimization community now has the resources to determine whether an algorithm qualifies as a pattern search and to maintain the framework of pattern searches while gearing methods toward solving particular kinds of problems. In response to the need for basic sequential implementations, as well as the desire for a foundation that may be extended easily to create a variety of pattern searches, I have implemented a basic pattern search suite using C++ classes.

3 Theory Defining A Pattern Search

Pattern searches minimize a real-valued function,

$$\text{minimize } f(x), \text{ where } x \in \mathbf{R}^n.$$

Assume the continuous differentiability of the function on an open neighborhood of the compact level set $L(x_0) \subset \mathbf{R}^n$ where $L(x_0) = \{x : f(x) \leq f(x_0)\}$, and x_0 is the initial point at which the search begins. Problems for which pattern searches would be most applicable include ones for which reliable derivative information cannot be obtained.

Pattern search algorithms direct the search for a minimum through a pattern containing at least $n + 1$ points per iteration, where the vectors representing the direction and distance of each point relative to the current iterate form a positive basis in $\mathbf{R}^n$. Any set of positively independent vectors that positively span the search space provides a positive basis. Other vectors may be included in the pattern as trial steps in addition to those forming the positive basis, allowing for a fair amount of flexibility in devising a search pattern. Further, an iteration of a pattern search algorithm may require as few as one function evaluation because the search requires only simple decrease to accept a new point. In that way even large patterns may be used sparingly.

The lengths of trial steps may change between iterations, but the location of the trial points relative to one another must maintain a particular structure. That is, each trial point in the pattern lies on a vertex of a rational lattice or grid, as does the current iterate. The term *rational lattice* reflects the requirement that the

distance between vertices remain a rational factor of the initial search step length, Δ_0, specified by the user. In this way, restrictions on the trial steps replace the conditions of sufficient decrease usually imposed to prove global convergence for optimization methods.

An iteration of a pattern search algorithm calls for a comparison of the objective function values of at least some subset of the points in the pattern to the value of the function at the current iterate. At each iteration either a comparison shows improvement (simple decrease) in the function value and the improving point becomes the new iterate or no decrease in value is found at any point in the pattern. In the latter case, the scale factor of the lattice is reduced so that the next iteration continues the search on a finer grid. Typically, this process continues until the resolution of the grid is deemed fine enough for the user. A discussion of various stopping criteria and their effectiveness in signaling appropriate termination appears later in section 7.1. For further insight into the details of the convergence theory not presented here, I refer the reader to either [18] or [13].

4 Pattern Search Implementation Suite

I have included a visual of a simple pattern search in Figure 1 to help illustrate the algorithmic concepts discussed with the implementation. In this example of a `CompassSearch`, the search makes exploratory moves from the current iterate x_k to trial points in the coordinate directions. The trial steps are based on the length of Δ and step across the theoretical rational lattice. If these exploratory moves locate no

improving points, the iteration is deemed unsuccessful or failed; and the size of Δ is scaled back to allow for a refinement of the search lattice. The search halts when Δ falls below a predefined `tolerance` value.

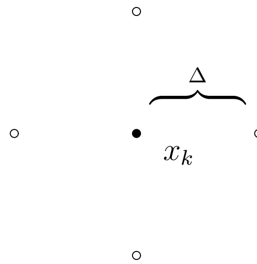


Figure 1: A simple example of pattern search

4.1 Pattern Search Base Class

My suite of `PatternSearch` software takes advantage of the class structure of C++ by deriving four separate searches from one base class. The simple `CompassSearch` executes a form of coordinate search. The minimal positive basis `NLessSearch` explores the flexibility in the convergence theory. An implementation of the original "pattern search" algorithm by Hooke and Jeeves [8], called `HJSearch`, and an edited version of the Hooke and Jeeves method I have devised, called `EdHJSearch`, join the other searches in that they all call procedures introduced in their parent class, `PatternSearch`. The `PatternSearch` class is intended to provide programmers with the tools to create a variety of simple and complex pattern searches.

The `PatternSearch` base class attempts to ease the creation of future pattern search software in two ways: first by providing some default data types and functions and second by requiring that the programmer implement the pure virtual function,

`ExploratoryMoves`, as a necessary component of any pattern search algorithm. By implementing a base class and encouraging derived classes to use the same name for the most basic general function inherent to all pattern searches, I also simplify my testing implementation. The symmetry of the user's required object declarations and function executions allows for readability and ease of implementation.

While many successful implementations of pattern searches do not store their patterns in a rigid structure, I provide for the possibility of using direct matrix-like storage of the pattern vectors by declaring as a private data member a pointer to an optional array of `rml_Vector`s, the vector class objects I have gratefully inherited and modified from the work of Dr. Robert Michael Lewis [12]. Search classes of my own implementation that make use of this structure include `CompassSearch` and `NLessSearch`. The accompanying functions for use with explicitly stored patterns are optional in the sense that their implementation is not required (although default implementations have been provided).

As the algorithm first coined a "pattern search" by Hooke and Jeeves exemplifies, patterns of an amorphous nature may also prove successful without the rigidity of a stored pattern matrix. My own twist on the original Hooke and Jeeves algorithm, `EdHJSearch`, advocates an even more dynamic pattern that adjusts in response to successful and unsuccessful exploratory moves.

The common feature of any optimization method lies in its need to interface with a function supplied by the user. The information this suite absolutely requires includes the dimension, n, of the search space, a function returning an initial iterate x_0, and a function $f(x)$ to return values for given $x \in \mathbf{R}^n$. My format for passing

arguments to this last function is derived mostly from the structure required by the *NEOS* server [6, 7]. The *Network-Enabled Optimization System (NEOS)* at Argonne National Laboratory provides an existing interface on the Web, currently located at *http://www-unix.mcs.anl.gov/neos/Server/*, that allows people access to various optimization implementations. I structure the arguments to my user functions after theirs except that I also require a success flag in the call to evaluate the function $f(x)$. The success flag allows for the possibility of a function subroutine failing for certain choices of x so that a search would not necessarily be crippled by the absence of data at one, possibly unimportant, point.

Additionally, I offer optional variables for the user to specify, including the initial choice of Δ; a limit on the number of function calls allowed such that the search will halt when the number has been exceeded upon a call to the `PatternSearch::Stop` function; and the desired stopping `tolerance` so that the search will not proceed once Δ has been reduced below `tolerance`. I refer to these variables as optional, not because I do not mean for them to be provided by the user, but because I have provided default definitions in case the user knows so little about the function that they do not care to hazard a guess as to appropriate choices. My initial step length is 1.0, and the default value for `tolerance` is approximately equal to the square root of machine epsilon. The search is designed to run indefinitely if the user provides no limit on the number of objective function calls and the search fails to halt on Δ. As I will discuss in section 7, my test results support the practicality of halting the search when Δ falls below the value of `tolerance`; users may need (or wish) to experiment with other variable values to obtain the accuracy they desire.

Indeed, the stopping mechanism plays a crucial role in implementing a convergent pattern search, as is standard in nonlinear optimization. There are many possible and debated options for this key aspect of any pattern search. Choices include testing the simplex gradient of the pattern trial points [3] or measuring the variance in the function values returned at those points [15] as well as tracking Δ. The `PatternSearch::Stop` function's declaration reads `virtual` so as to leave room for varying interpretations in use, yet all of my class implementations halt based on the size of Δ as a reflection of the mesh resolution. Experimentation with some of the alternative stopping criteria serve to justify this choice, as related in section 7.1.

4.2 CompassSearch

The `CompassSearch` implementation represents a simple form of coordinate search. The storage of the pattern remains explicit throughout the search in that the pattern pointer provided in the `PatternSearch` base class points to an array of $2n$ `rml_Vector` pointers. The private data member representative of Δ plays a direct role as the scaling factor of the pattern. The search adds the vectors in the pattern to the current iterate in the order they are stored so that the components of the current iterate are first shifted by Δ and then by $-\Delta$ to obtain the trial points, as illustrated in Figure 1, where the solid circle indicates the current iterate and the open circles indicate the four possible moves (N,S,E, W) possible in $\mathbf{R}^2$.

If the search finds simple decrease at any trial point, the exploratory move relocates the current iterate to that trial point and begins a new iteration. Otherwise, an iteration finding no improvement after trying all $2n$ points in the pattern multiplies

Δ by a relatively conservative scaling factor of 0.5 (the scaling factor 1/2 is used in all of my search implementations) and begins another iteration with this new trial step length.

4.3 NLessSearch

The NLessSearch implementation seeks to take advantage of the insight provided by the sufficient conditions for a positive basis [14]. These conditions determine that a minimal positive basis may be formed with only $n+1$ vectors ($n-1$ *less* than that of a CompassSearch). It would be easy to build a minimal positive basis by first choosing the n unit coordinate vectors. The final vector in the pattern would then be the vector $(-1, -1, \ldots, -1)^T$. Because I feel that this approach would bias the search in favor of optima lying in the positive coordinate directions relative to the current iterate, I invest in the extra computational effort of building a minimal positive basis where the angles between any two unique vectors are equal. For insight into creating this type of pattern, I turn to the discussion on building a regular simplex in [9].

4.3.1 Constructing the NLess Pattern

A *simplex* is a set of $n+1$ points in $\mathbf{R}^n$ (e.g. a triangle in $\mathbf{R}^2$, a tetrahedron in $\mathbf{R}^3$, etc.). A *regular* simplex is a simplex with edges of equal length (e.g. an equilateral triangle in $\mathbf{R}^2$). A method for creating a regular simplex starting from one initial vertex as a base point is given in [9]. The difficulty from there lies in restructuring the creation of the simplex so that it may be built relative to a given centroid that represents the initial point of the search, rather than being constructed relative to

one of the vertices. The $n+1$ pattern vectors from the centroid to each vertex of the simplex must also have length Δ.

To construct a centroid-based pattern, I calculate the location of the base vertex for the simplex algorithm in [9] given that each component of the centroid, x_0, should represent the average of the corresponding components of the simplex vectors. Having shifted the base vertex of the simplex appropriately relative to my initial iterate x_0, I may then follow the algorithm in [9], subtracting the centroid from each vector to create a pattern of vectors from the centroid to the vertices of the simplex. I then scale the vectors to length Δ as a separate subprocess. I have rendered a general notion of the transition applied through this process in Figure 2, showing the transformation from an algorithm for a regular simplex to one for a minimal basis pattern in two dimensions. The C++ code to accomplish this task is given in section 10.3.4.

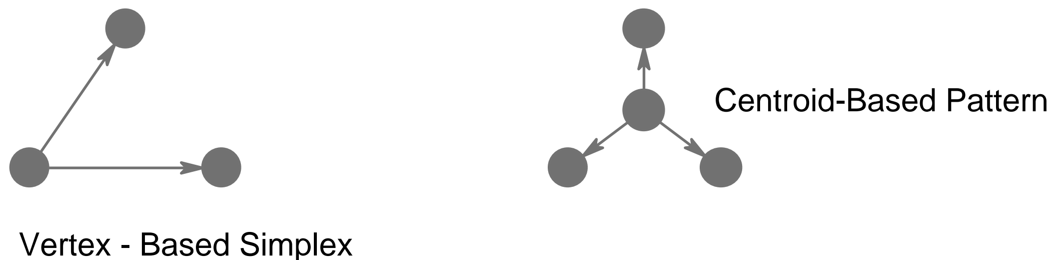


Figure 2: Building the `NLessSearch` Pattern from a Regular Simplex

The most striking feature of the `NLessSearch` is, of course, that it employs the smallest possible number of function evaluations per standard iteration through the pattern. I have attempted to respond to the open question of whether a pattern search that requires fewer total function evaluations per iteration (in the worst case) might prove more efficient than a pattern search with more trial points in the pattern. The

results of my tests comparing the numbers of function evaluations required amongst various pattern search implementations will be presented in section 7. Whether an `NLessSearch` might reveal benefits other than efficiency remains a topic for future investigation.

4.4 The Hooke and Jeeves Pattern Search

The Hooke and Jeeves pattern search algorithm [8] represents the first search I implemented that includes a sense of search history. This search augments its knowledge of the objective function's behavior by adapting the first trial step of an iteration in response to successful trial steps in the previous iteration. That adapting step, which I will refer to as the *pattern extending step*[1], serves to imbed in the algorithm a sense of iteration history.

When first playing with iterative direct search methods by hand, I applied the description of the original pattern search method of Hooke and Jeeves found in [5] with help from the explanations in [1]. The essence of the search is as follows. From a starting base point b_1 with function value f_1, the search increases the first component of the b_1 vector by some step length Δ. If simple decrease in the function value is found, this trial point becomes the current iterate and the search continues on the next component of b_1. If no decrease appears, then the next trial point is formed by subtracting Δ from that component of b_1. Again, if improvement is found at the trial point, the trial point becomes the current iterate of the search. Whether or not the

[1]Hooke and Jeeves simply call it the *pattern step*, but I want to avoid confusing it with a trial step in the search pattern.

current iterate relocates for each b_1 component, the process continues for all of the dimensions of the search space.

Upon completion of the coordinate search on all of the dimensions, the current iterate becomes the base point b_2 with its value f_2. If $f_2 < f_1$, a pattern extending step relocates the current point to $2b_1 - b_2$. This new point acts as a temporary base point in the sense that it becomes the current iterate whether or not its value is less than f_2. The `HJSearch` carries out a coordinate search in all dimensions as described above relative to the temporary base point, b_{temp}. If, at the conclusion of the coordinate search about b_{temp}, the value of the current iterate is less than f_2, then the current iterate becomes b_3 with value f_3; and a pattern extending move is made to $2b_3 - b_2$, where the search continues. If, however, the current value at the termination of the coordinate search about b_{temp} fails to improve on f_2, the extending move is judged a failure, b_2 becomes the current iterate, and the `HJSearch` conducts a coordinate search about b_2. If at any time a coordinate search about a permanent base point yields no improvement, Δ is reduced to apply another coordinate search to the base point. The Hooke and Jeeves algorithm continues until some halting criterion is met.

4.5 Optimism in Hooke and Jeeves – Birth of EdHJ Search

I consider the algorithm presented by Hooke and Jeeves to be a transformation of the traditional coordinate search based on intuitive ideas about the behavior of many functions. Achieving greater efficiency through a pattern extending step relies on the premise that if one step in a particular direction produces function decrease, then

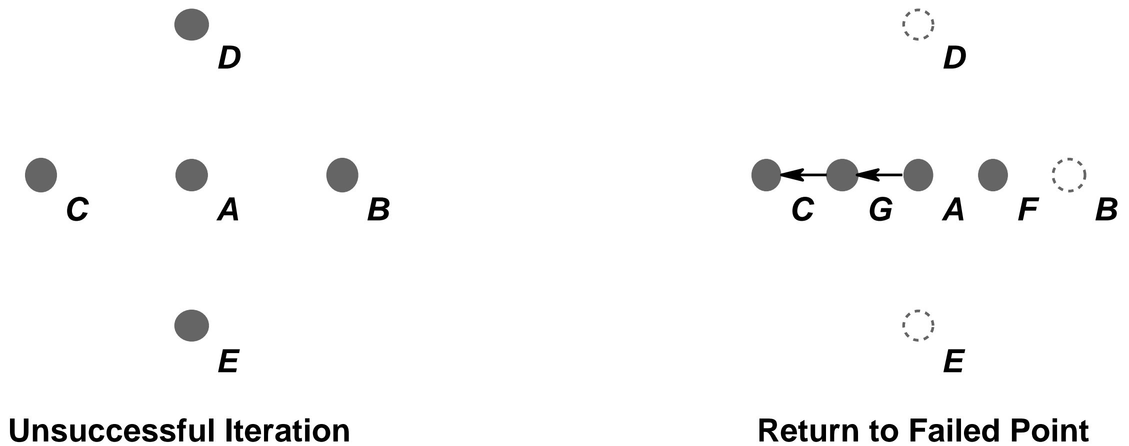


Figure 3: `HJSearch` Behavior At Δ Reduction

another step in that direction might produce more still. The intuition that motivates the Hooke and Jeeves pattern extending step is that if improvement is seen moving from b_k to b_{k+1}, the step should be doubled to produce b_{temp}. The search continues on the assumption that the first improving direction it finds is golden even if the move to b_{temp} initially yields no decrease. The search discards the pattern extending move to b_{temp} only after a coordinate search about b_{temp} fails to improve further upon the value found at the base point b_{k+1}. My results in section 7 make clear that these intuitive notions provide valuable alterations to the search algorithm on the functions I tested. The optimism inherent in the pattern extending step and the amorphous pattern structure it introduces seem to steer sequential pattern searching in a healthy direction.

One insight into the Hooke and Jeeves pattern search algorithm that first occurred to me as I drew examples of the method on graph paper seems out of character with the search's intuitive foundation. As shown in Figure 3, after an unsuccessful iteration, typically Δ is reduced by a factor of 1/2. At the next iteration, the pattern

extending step may then return the search to a trial point whose lack-luster function value has already been evaluated and discarded. In the example shown in Figure 3, the search finds improvement at point **G** after contracting the step size from the previous iteration. The search then takes a pattern extending step to point **C**, which was already determined to be an unpromising step during the preceding iteration. Intuitively, the search has already established that the coordinate directions about the current iterate fail to show promise before the search contracts Δ. Thus it makes little sense to focus on immediately moving away from the most promising point in that smaller neighborhood, especially when the temporary base point reached by the pattern extending step may have a larger function value than the current base point.

I have extended the original Hooke and Jeeves algorithm in an edited version, `EdHJSearch`, that allows the search to learn from *failed* iterations as well as successful ones. In their same spirit of optimism, I have applied the method's fundamental behavior to hoping that if we are reducing Δ, we are in the neighborhood of the minimum and should focus out current search on that neighborhood. A move outside of that immediate neighborhood is allowed only if the next iteration after a contraction finds an improving trial step away. My edited version, `EdHJSearch`, thus omits the pattern extending step directly after a reduction in Δ. This is instrumented via a flag that tracks whether the previous iteration contracted Δ (i.e. reduced Δ by $1/2$). The results of `EdHJSearch`, as discussed further in section 7, have borne out my hopes for improved search efficiency.

5 Test Functions

I have followed the suggestion of Dr. Michael Trosset of creating random convex quadratics for my objective functions. While I would like to experiment with more general functions at a future date, I have discovered that the numerous advantages of quadratic functions far exceed my earlier expectations.

By creating my own random quadratic functions I am able to address issues of accuracy as well as efficiency, which moves beyond the initial scope of my testing plans. Bearing in mind the simple form of a quadratic

$$f(x) = x^T Ax + b^T x + c,$$

if I set $b = 0$ and record the value of c as one distinguishing characteristic for referencing amongst tests, I am able to proceed with the knowledge that the real location of the optimum x_* is at the origin and that $f(x_*)$ is equal to the value of c. With this knowledge, I am able to compare the results produced by the different pattern searches with the known answer and then compare the experimental accuracies among searches. Comparing the true theoretical and experimentally returned function values becomes a matter of a single subtraction followed by a possible change of sign to obtain the absolute difference. In retrospect, this process would have been one floating point operation shorter had I set c to zero along with b. However, even as such, the rounding error remains negligible. Since x_* is at the origin, I can assess the quality of the solutions returned by my experiments simply by taking the 2-norm of the experimental result. The version of a 2-norm function found in my code comes from an `f2c` conversion of the LAPACK function `dnrm2.f`. Through these means I

attempt to conserve as many significant digits of the experimental result as possible while carefully measuring its accuracy.

6 Test and Implementation

I have constructed one main testing program, which I execute in ten separate trials to measure the searches' efficiency and accuracy. The dimensions of the search spaces for testing range from two to five inclusive. Each of the ten trials generates ten thousand sets of testing parameters per testing dimension, using a separate stream of pseudo-random numbers for each of the three parameters. The testing parameters include the quadratic objective function, the initial delta length Δ_0, and the starting point x_0 for each repetition of the testing loop. Each of the ten trials uses the same ten thousand sets of test parameters, and each parameter set remains constant for testing each of the four pattern searches: `CompassSearch`, `NLessSearch`, `HJSearch`, and `EdHJSearch`. The distinguishing characteristic between trials is the value of the halting `tolerance`, decreased between trials as shown in Figure 4.

Figure 4: Halting `tolerance` Values

trial 0	$3.12500000000000e-02$	trial 5	$9.53674316406250e-07$
trial 1	$3.90625000000000e-03$	trial 6	$1.19209289550781e-07$
trial 2	$4.88281250000000e-05$	trial 7	$3.72529029846191e-09$
trial 3	$6.10351562500000e-05$	trial 8	$1.16415321826935e-10$
trial 4	$7.62939453125000e-06$	trial 9	$3.63797880709172e-12$

I track and store the number of function calls required by each search, the function value returned as the minimum, and the point returned as the optimum. I record

the initial iterate of the search, whose components are randomly generated from a normal distribution with a mean of zero and a standard deviation of one, and the Δ_0 value, which is generated from an exponential distribution with a mean of one. While the number of function evaluations alone can be used to compare relative efficiency, measuring accuracy for the functions I create requires that I store the constant term c, which represents $f(x_*)$. Along with the H matrices, I also store the initial seeds used to generate the matrices, initial points, and initial step lengths from their respective random number streams. Because I then have easy access to this information, I have been able to replicate individual runs from amongst the ten thousand by simply reseeding the random number streams. This test structure proved invaluable in tracking the numerical error I discuss in section 7.2.

The implementation of my test program relies on the use of streams of randomly generated numbers from the pseudo-random number generator library `rngs` and the random variate generator library `rvgs` I obtained from Steve Park [16, 17]. I generate new test parameters within both the objective function implementation and the function for retrieving an initial point, as well as within the main body of the test program. In my software, I rely on the presence of global variables to signal these function implementations of the need to generate new testing parameters. Although the use of global variables is discouraged in C++, I feel that the test structure must communicate via global variables. As an example of `PatternSearch` use, my test program should not require the restructuring of the arguments passed to the objective function and starting point returning function or the overloading of success flags because I mean for the function calls here to demonstrate their actual use. Dynamic

user functions, while invaluable for systematic testing, fail to represent common usage of pattern search methods.

6.1 Creating the Test Functions

When the evaluation of the aforementioned global variables within the call to my dynamic objective function signals the creation of a new quadratic function, the function fills an $(n+2) \times n$ matrix, denoted H, with the random numbers returned from calls to a `Normal` random variate with a mean of zero and standard deviation of one. The positive constant term, c, is generated as an `Exponential` random variate with a mean of one [16, 17]. Computing

$$f(x) = x^T H^T H x + c,$$

upon subsequent calls to the function implements a positive semi-definite quadratic function.

By filling the H matrix to create the matrix A where $A = H^T H$, I avoid biasing $f(x)$ in favor of search patterns based solely on steps in the coordinate directions. Therefore, simple searches such as `CompassSearch` and even the core patterns of more complicated Hooke and Jeeves searches gain no unfair advantage over, for example, `NLessSearch`. The intent of implementing a quadratic function in this manner is to represent the behavior of the neighborhood of the solution of a general objective function translated to the origin. In that sense, I hope that the results of experiments on quadratic functions can add to the understanding of the local behavior of pattern searches when applied to more general functions.

I have added `CleanSlate` member functions to each of the search classes. The

`CleanSlate` function erases all traces of a previous search and allows a new search to begin without necessarily declaring a new search class object. The testing may proceed on a search algorithm using a particular initial step length, starting point, and objective function only to have all of those parameters replaced in the next repetition of the test loop without losing track of dynamically allocated memory, allowing time—rather than memory—to act as the only barrier on the number of test trials.

7 Results

Much of the data I have obtained from these experiments[2] is richly complex in form. I can make a few simple analytic statements, but other factors require more exact study of the trends revealed in the data.

I can say that for my test functions the `EdHJSearch` consistently requires the smallest average number of function calls in all dimensions and with all choices of `tolerance`, with the `HJSearch` running a close second. Halting on most restrictive choices of `tolerance` (on the order of 10^{-5} or less for these tests) leaves the `NLessSearch` with the highest average total of function calls, but the less exacting stopping criteria leave little distinction between the efficiency of `NLess` and `Compass` searches. Asymptotically, the searches finish in terms of function calls in the order `EdHJSearch`, `HJSearch`, `CompassSearch`, and `NLessSearch`. The median efficiency behavior of the searches adds even more interest to the results as graphically displayed

[2]Experiments were run on a Pentium II-400 MHz with 384 MB RAM - SDRAM DIMMs and a 512K L2-Cache with the Intel 440 BX Chipset, running Linux version 2.0.35 and compiling the C++ programs with g++ version 2.7.2.3.

in section 9.2. I have also run the experiments with the crude time estimates provided by Unix timing commands. While I hesitate to print these—at best—approximate times, the time required by each of the searches maps to the required number of function calls, which reinforces the point that the dominant computational cost for a pattern search lies in the evaluation of the function. The enhanced performance provided by my editing of the Hooke and Jeeves algorithm heartens me, and I would like to examine further the possibilities of adjusting pattern models.

The notion that decreasing the total number of function calls required before determining that the conditions for a decrease in Δ have been met might decrease the total number of function calls required by the algorithm proves false, on average, for my tests. While the `NLessSearch` is most efficient for some quadratic functions and starting points, these cases appear to be exceptional. Imagine, for example, a case where the location of the optimum relative to the initial iterate lies nearly in the direction of one of the pattern vectors. Then the `NLessSearch` may have an advantage over a coordinate-based search in that it would hone in on the minimum with each iteration while the coordinate search would need to stair step across and back over multiple iterations. Indeed, the `NLessSearch` appears to fare better than the `CompassSearch` in terms of the number of function calls when the choice of `tolerance` remains relatively large for small numbers of dimensions. The difference in the accuracies obtained by `CompassSearch` over `NLessSearch` in those trials may help to shed some light on the true value of requiring less functions calls, though, as shown in sections 9.3 and 9.4. Beyond the second trial (with `tolerance` $= 10^{-4}$) in any dimension, the `CompassSearch` gains in average efficiency with its $2n$ vectors

over the $n+1$ vectors of the `NLessSearch` without compromising accuracy.

The results that come as the greatest surprise to me involve the accuracy of the different searches. My original records of the returned function values and optima arose as a method of verifying that I had implemented the searches correctly. As I began to notice particular trends in that data, however, I decided to maintain more extensive data sets. Because the interval estimates in sections 9.3 and 9.4 cannot isolate a clear winner in terms of accuracy (owing largely to the variance in the simplex pattern searches), I have computed Hotelling's T^2 test statistics for the data from trial 8 under the guidance of Michael Trosset, who has analyzed the statistics for significance. First of all, the Hooke and Jeeves searches require not only the least number of function calls but also produce the best average accuracy for small `tolerance` to a highly significant extent. While Dr. Trosset cautions that statistical significance does not equate to importance, I still find it fascinating that the Hooke and Jeeves searches can return an order of magnitude improvement on the accuracy in some cases, especially since they require the least total number of function evaluations.

I would caution the reader, however, that the Hooke and Jeeves searches only show improved accuracy over the others with smaller `tolerance` values. Specific to my tests, the distinction only becomes relevant beyond trial 3, where `tolerance` $= O(10^{-5})$. In section 9, I include extensive tables of the results to illustrate the patterns created by the accuracy trends.

7.1 Halting Criteria

The halting criteria represent probably the least well-defined aspect of a pattern search. My searches all halt based on the length of Δ as a reflection of the refinement of the lattice. Finding Δ incurs no additional computational cost to the pattern searches and comparing it to some predefined halting `tolerance` proves to be a reliable signal for termination.

Another common criterion for a direct search [15] involves calculating the variance among the function values found at all of the trial steps in an iteration. I calculate the variance

$$\sigma^2 = \frac{\sum_{i=0}^{l}(f(x_i) - \mu)^2}{l},$$

where μ equals the average function value in the iteration and $n+1 \leq l \leq 2n$ is the length of the core pattern, via Welford's algorithm. The search then halts when this variance falls beneath a predetermined threshold.

A third proposal, found in [3], involves halting on a sufficiently small simplex gradient $D(f : S) = V^{-T}\delta(f : S)$, where $x_1, \ldots, x_{n+1}$ denote the $n+1$ vertices of an n-dimensional simplex S,

$$\delta(f : S) = \begin{pmatrix} f(x_2) - f(x_1) \\ f(x_3) - f(x_1) \\ \vdots \\ f(x_{n+1}) - f(x_1) \end{pmatrix}$$

and

$$V(S) = (x_2 - x_1, x_3 - x_1, \ldots, x_{n+1} - x_1)$$

$$= (v_1, v_2, \ldots, v_n).$$

S must be a nonsingular matrix with vertices $\{x_j\}_{j=1}^n$ for the simplex gradient to be well-defined. I have applied this idea to the simple `CompassSearch` pattern, with its $2n$ vectors, for comparison with the other two stopping criteria.

I include calculations of the variance and simplex gradient just before the point in the code where Δ will be reduced due to an absence of improvement at all trial steps in the iteration. The convergence theory guides the choice of this placement to the one point where theoretical results may be applied to the current iterate and stopping criteria. In this manner, I may examine the behavior of the variance and the simplex gradient relative to Δ and the function value of the current iterate. It should be noted that my searches still halt when Δ has fallen below `tolerance`; the other calculations were only made for comparison.

Two representative examples of the results in two dimensional `CompassSearch` tests are shown in Tables 1 and 2. Table 1 presents the results for the objective function with

$$H = \begin{pmatrix} -1.2139420049718712 & 1.4996742225832957 \\ -1.2143611698729968 & -1.4711004206481226 \\ -0.0592565088141217 & -1.7023524789549154 \\ 0.1534197890377110 & -0.2318145721980239 \end{pmatrix}$$

where the constant term $c = 0.0297737224361487$. The initial size of Δ for the first example is 2.5014767071524533, and the randomly generated initial iterate is

Table 1: **Stopping Criteria vs Function Value** (with correct digits in red)

variance	simplex gradient	Δ	function value
3.0780276899464747	2.8564077075567220	0.7379498442749530	2.4314866861505493
0.0840886692951946	0.4587722065355163	0.3689749221374765	1.5908180895957142
0.0034923677292313	0.0335358610519773	0.1844874610687383	1.4773040257429713
0.0009624353196831	0.3867907025135328	0.0922437305343691	1.4772932399797003
0.0000198901150653	0.0895411558448952	0.0461218652671846	1.4704383882086089
0.0000028553411229	0.0895411558448976	0.0230609326335923	1.4704383882086089
0.0000000607727084	0.0146150464379905	0.0115304663167961	1.4697557553897906
0.0000000069229129	0.0146150464379905	0.0057652331583981	1.4697557553897906
0.0000000003717269	0.0041164809137459	0.0028826165791990	1.4697238142244098
0.0000000000128574	0.0014031259841643	0.0014413082895995	1.4697199463307293
0.0000000000011952	0.0014031259840873	0.0007206541447998	1.4697199463307293
0.0000000000001726	0.0014031259839332	0.0003603270723999	1.4697199463307293
0.0000000000000044	0.0002324055244136	0.0001801635361999	1.4697198389374599
0.0000000000000003	0.0001125699065903	0.0000900817681000	1.4697198337782784
0.0000000000000000	0.0000599178070629	0.0000450408840500	1.4697198322913623
0.0000000000000000	0.0000263260460663	0.0000225204420250	1.4697198320181937
0.0000000000000000	0.0000167958854282	0.0000112602210125	1.4697198319006213
0.0000000000000000	0.0000047650704594	0.0000056301105062	1.4697198318958682
0.0000000000000000	0.0000060153976247	0.0000028150552531	1.4697198318823599
0.0000000000000000	0.0000060154370635	0.0000014075276266	1.4697198318823599
0.0000000000000000	0.0000014421967569	0.0000007037638133	1.4697198318803744
0.0000000000000000	0.0000000946530359	0.0000003518819066	1.4697198318802809
0.0000000000000000	0.0000007685826514	0.0000001759409533	1.4697198318802656
0.0000000000000000	0.0000001400864931	0.0000000879704767	1.4697198318802380
0.0000000000000000	0.0000001388244526	0.0000000439852383	1.4697198318802380
0.0000000000000000	0.0000000858187525	0.0000000219926192	1.4697198318802374
0.0000000000000000	0.0000000302889715	0.0000000109963096	1.4697198318802369
0.0000000000000000	0.0000000201926477	0.0000000054981548	1.4697198318802369
0.0000000000000000	0.0000000403852953	0.0000000027490774	1.4697198318802369
0.0000000000000000	0.0000000807705906	0.0000000013745387	1.4697198318802369
0.0000000000000000	0.0000001615411813	0.0000000006872693	1.4697198318802369
0.0000000000000000	0.0000003230823625	0.0000000003436347	1.4697198318802369
0.0000000000000000	0.0000006461647250	0.0000000001718173	1.4697198318802369
0.0000000000000000	0.0000012923294500	0.0000000000859087	1.4697198318802369
0.0000000000000000	0.0000025846589001	0.0000000000429543	1.4697198318802369
0.0000000000000000	0.0000000000000000	0.0000000000214772	1.4697198318802369
0.0000000000000000	0.0000000000000000	0.0000000000107386	1.4697198318802369

Table 2: **Stopping Criteria vs Function Value** (with correct digits in red)

std. deviation	simplex gradient	Δ	function value
22.9183504010289383	2.9743637607191737	2.5014767071524533	4.1464661873603124
8.6832469203393288	3.0525104362787010	1.2507383535762266	3.2386547175689682
0.9742534137216189	0.7079812916976593	0.6253691767881133	0.0913657370406688
0.3148834895917984	0.7079812916976596	0.3126845883940567	0.0913657370406688
0.1022307895726505	0.2223733058514898	0.1563422941970283	0.0534052037468748
0.0285932325107842	0.2272574730739602	0.0781711470985142	0.0393810907743447
0.0060453625492211	0.0077732599245623	0.0390855735492571	0.0309003117979746
0.0016903021570746	0.0065522181189447	0.0195427867746285	0.0301593652865318
0.0003390992412607	0.0071627390217534	0.0097713933873143	0.0298266368295058
0.0001309365647018	0.0068574785703491	0.0048856966936571	0.0298172006298319
0.0000181464864324	0.0070101087960513	0.0024428483468286	0.0297779686226221
0.0000036521102920	0.0002582864973020	0.0012214241734143	0.0297738452031370
0.0000011507666769	0.0002582864973006	0.0006107120867071	0.0297738452031370
0.0000004532412662	0.0002582864973063	0.0003053560433536	0.0297738452031370
0.0000000725351831	0.0002678258864045	0.0001526780216768	0.0297737336536433
0.0000000265700146	0.0001864488194325	0.0000763390108384	0.0297737305475209
0.0000000057042037	0.0000383036861745	0.0000381695054192	0.0297737235541039
0.0000000017720984	0.0000394961098672	0.0000190847527096	0.0297737229774112
0.0000000003126184	0.0000178844401704	0.0000095423763548	0.0297737224948635
0.0000000001197997	0.0000105077288229	0.0000047711881774	0.0297737224772656
0.0000000000157681	0.0000035393027747	0.0000023855940887	0.0297737224378923
0.0000000000057364	0.0000035393027747	0.0000011927970443	0.0297737224378923
0.0000000000019638	0.0000000275450570	0.0000005963985222	0.0297737224367684
0.0000000000001968	0.0000000089121494	0.0000002981992611	0.0297737224361492
0.0000000000000504	0.0000000089121494	0.0000001490996305	0.0297737224361492
0.0000000000000137	0.0000000089121494	0.0000000745498153	0.0297737224361492
0.0000000000000044	0.0000000088888801	0.0000000372749076	0.0297737224361492
0.0000000000000018	0.0000000089354187	0.0000000186374538	0.0297737224361492
0.0000000000000002	0.0000000094938824	0.0000000093187269	0.0297737224361487
0.0000000000000001	0.0000000096800369	0.0000000046593635	0.0297737224361487
0.0000000000000000	0.0000000044677094	0.0000000023296817	0.0297737224361487
0.0000000000000000	0.0000000044677094	0.0000000011648409	0.0297737224361487
0.0000000000000000	0.0000000059569458	0.0000000005824204	0.0297737224361487
0.0000000000000000	0.0000000059569458	0.0000000002912102	0.0297737224361487
0.0000000000000000	0.0000000119138916	0.0000000001456051	0.0297737224361487
0.0000000000000000	0.0000000238277832	0.0000000000728026	0.0297737224361487
0.0000000000000000	0.0000000476555665	0.0000000000364013	0.0297737224361487
0.0000000000000000	0.0000000953111330	0.0000000000182006	0.0297737224361487
0.0000000000000000	0.0000000000000000	0.0000000000091003	0.0297737224361487
0.0000000000000000	0.0000000000000000	0.0000000000045502	0.0297737224361487

$$x_0 = \begin{pmatrix} 0.5069357602729747 \\ -0.6767951222352641 \end{pmatrix}.$$

The matrix used for the second test function is

$$H = \begin{pmatrix} -2.0024276947358515 & 1.0800605002607353 \\ -0.5864902477646186 & 0.3970139246658717 \\ 0.6689094408010875 & -0.0235658431216256 \\ 0.6022562572585082 & 0.8245007702218383 \end{pmatrix}$$

where the constant term $c = 1.4697198318802369$. The initial size of Δ for this example is 0.7379498442749530, and the randomly generated initial iterate is

$$x_0 = \begin{pmatrix} -0.2160152988897405 \\ 0.6616820877821726 \end{pmatrix}.$$

7.1.1 Variance vs Δ

One of the most noteworthy features of the results in Tables 1 and 2 can be seen in the direct correlation between the variance and Δ. The variance decreases at a faster rate, yet it always shows decrease with decrease in Δ. The variance has already diminished to below machine epsilon, however, before the potential accuracy of the search has been completely fulfilled. In order to compare more similar metrics, the results in Table 2 replace the variance with the square root of the variance (i.e. the standard deviation). Section 9.1 displays tables of the comparison with the `HJSearch` and the `NLessSearch` in five variables. Even the standard deviation may fall well below machine epsilon before the potential accuracy has been exploited as shown in

the tables in section 9.1. More to the point, though, finding the standard deviation represents even more computation for the search while producing no readily apparent benefit. Considering the fact that tracking Δ requires no additional computation for the pattern search, I would need a convincing argument in favor of computing the variance or the standard deviation to employ as my halting criterion.

7.1.2 Simplex Gradient vs Δ

The behavior of the simplex gradient as I apply it is not as satisfactory as that of Δ in that it fails to produce a monotonicly decreasing sequence. Perhaps I could forgive the increase in the simplex gradient at the tail end of the search if only the decrease were smooth before that point. Because this criterion does not behave cleanly in even the simple quadratic case, I would be hesitant to use it without stronger arguments in its favor, especially considering the amount of linear algebra involved to compute the simplex gradient.

7.1.3 Summary

While the simplex gradient begins to misbehave in a basic coordinate search, comparison of the variance and standard deviation with Δ in my other searches on quadratics with search spaces of dimensions two through five yields results similar to those in Tables 1 and 2 and in section 9.1. I examined the output for hundreds of trials without seeing a case where the variance failed to decrease with Δ, and I would offer this as evidence that the two measures are in fact closely related. Again, I see no compelling reason to provide a default stopping criterion other than the size of Δ in my base

class, and I feel that these results support that decision.

Further, the results from all of my trials also share the common feature that the search has obtained the limit on the accuracy it can reach when Δ is approximately 10^{-9}. Referring to sections 9.3 and 9.4 of the results, the searches attain no greater accuracy by decreasing `tolerance` between trial 8 and trial 9. Notably, searches such as `NLess` do not obtain the same accuracy as the Hooke and Jeeves searches even with smaller stopping length values. While none of the searches die when smaller stopping values of Δ are requested, even values below machine epsilon, I see no reason to continue a search beyond a step length of about 10^{-9} for any of the quadratic functions I have tested since it brings no further improvement.

7.2 Numerical Error in the HJ Search

I encountered difficulties with my first implementation of the algorithm by Hooke and Jeeves and, as a consequence, rediscovered early observations of rounding error reported in [2], in response to the algorithm in [11]. Through my familiarity with pattern search theory, I was able to pinpoint the exact nature of the numerical error and its relationship to the theory. Hopefully some of the discussion below will help guide others in implementing pattern searches.

What I noticed as I began to test my implementation of the test program was that the `HJSearch` would occasionally become trapped. By *occasionally*, I mean that every few hundred trials it would halt based on the maximum number of function evaluations allowed rather than because Δ had dropped below `tolerance`. The algorithm would become trapped so that the value of the current iterate was still not

particularly close to the real solution even at the conclusion of all that computational effort in evaluating function calls. Moreover, the search would have decreased Δ only a few times or not at all. Because the other search methods that I had implemented at that time all found the optimum at the origin with function value equal to c for those trials, the test program appeared to be working correctly. With that in mind, I began to review the `HJSearch` for implementation errors.

7.2.1 History

My investigation included first stepping through the execution of my implementation to find the root of the problem. I traced the problem to floating point error caused by certain pattern extending steps, as I explain below. I then searched to see if anyone else had encountered this numerical error and was able to find an example of another implementation of the Hooke and Jeeves algorithm with the following intriguing comments [10]:

> make sure that the differences between the new and the old [base] points are due to actual displacements; beware of roundoff errors . . .

Curious, I followed the references in [10] to [2]. There I found the following remark:

> The algorithm compiled and ran after these modifications had been made but for a number of problems took a prodigious amount of computing owing to a flaw in the algorithm caused by rounding error.

This statement was followed by a 1-dimensional example that illustrates the problem I was seeing. The authors of [2] closed the discussion with a simple kludge to avoid

the error, one implemented in [10] and that I was able to incorporate into my implementation. Before I discuss the solution, I offer the following illustration of one of the examples I tracked from my own experimentation.

7.2.2 Example

I have selected an example of the process that leads to search degradation in Figures 5 and 6 for illustration. The H matrix for the function is

$$H = \begin{pmatrix} 1.1250434984182105 & 2.9393523736291134 \\ 0.3018243999955224 & 0.7723233840336674 \\ -0.7046745157561507 & -0.0382939899828596 \\ -0.7075833763502262 & 0.9484561354579317 \end{pmatrix}$$

where the constant term equals 0.0025602323710517 and the function value differs from earlier description of the test structure in that it is divided by $n + 2 = 4$ before adding the constant term, c. The initial size of Δ for the example is 6.0, which just happens to be the Δ_0 I was using at this early stage in the implementation and testing process, and the initial point, as generated, is

$$x = \begin{pmatrix} 0.1123135875305460 \\ 0.9266765162475219 \end{pmatrix}.$$

The accompanying Figures 5 and 6 show an artistic rendition of the exploratory moves for my first Hooke and Jeeves implementation. Clearly, the dots and their relative positions are not exactly to scale, but the points as listed in the margins supply the sixteen digits to the right of the decimal point required to demonstrate the nature of the problem. The bug that eventually leads to degeneration in this

example actually appears in the first few moves. The pattern extending step to point C and the improving exploratory move in the positive x coordinate direction creates a situation where the new base point, D, is within rounding error of the previous base point, B. Recall that only simple decrease acts as a required condition on accepting a trial step. The pattern extending step to point E moves outside of the radius of the rounding error created via the exploratory move additions. While I must accept some slight shifting off of the lattice via numerical error, perpetuating and exacerbating that error contributes nothing to the search. The Hooke and Jeeves search abandons the lattice only momentarily at this point because the search then finds improvement at point F and continues the search from there.

This sort of cycle appears many times in various tests when using my first Hooke and Jeeves implementation, but this example is one of the relatively few that finds itself in a cycle from which it cannot break free. From point Y until the search halts on the maximum function calls, the pattern steps shift the current point in minute steps until it has moved from point Y at

$$\begin{pmatrix} -0.075186412469451014 \\ -0.010823483752478102 \end{pmatrix}$$

to

$$\begin{pmatrix} -0.075186412425045646 \\ -0.010823483752478102 \end{pmatrix}.$$

7.2.3 Explaining the Error

Because the pattern extending step moves to a point that is not necessarily improving relative to the last base point, a particular pattern extending step may find

HJ Search Numerical Error

...first signs of trouble.

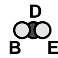
initial point: A (value 2.3369848213768507)
(0.11231358753054604,
0.9266765162475219)

good move to: B (value 1.4501926486662426)
(-1.38766864124694539,
0.9266765162475219)

pattern extended step to: C (value
(-2.8876864124694537,
0.9266765162475219)

good move to: D (value 1.4501926486662424)
(-1.38766864124694537,
0.9266765162475219)

pattern extended step to: E (value 1.4501926486662426)
(-1.38766864124694535,
0.9266765162475219)



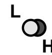
good move to: F (value 0.85996306512627974)
(-1.38766864124694535,
0.1766765162475219)

pattern extended step to: G (value 3.1208522724603247)
(-1.38766864124694532,
-0.5733234837524781)

good move to: H (value 0.15785350974275647)
(-0.63768641246945323,
0.1766765162475219)

pattern extended step to: I value 0.11782368075554148)
(0.11231358753054699,
0.1766765162475219)

pattern extended step to: J (value 0.73987357816463484)
(0.86231358753054721,
0.1766765162475219)



good move to: K (value 0.11782368075554157)
(0.11231358753054721,
0.1766765162475219)

good move to: L (value 0.055078629449610443)
(-0.26268641246945279,
0.1766765162475219)

pattern step to: M (value 0.15785350974275617)
(-0.63768641246945257,
0.1766765162475219)

good move to: N (value 0.055078629449610429)
(-0.26268641246945257,
0.1766765162475219)

pattern step to: O (value 0.055078629449610415)
(-0.26268641246945235,
0.1766765162475219)

pattern step to: P (value 0.055078629449610408)
(-0.26268641246945212,
0.1766765162475219)

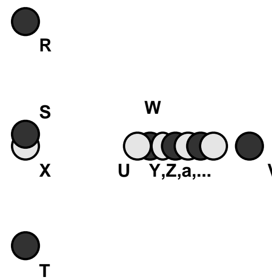
pattern step to: Q (value 0.055078629449610388)
(-0.2626864124694519,
0.1766765162475219)

pattern step to: R (value 0.055078629449610388)
(-0.26268641246945168,
0.1766765162475219)

Figure 5: HJSearch Numerical Error—first signs of trouble

HJ Search Numerical Error

...the final bog down.



good move to: S (value 0.047583877685930392)
(-0.26268641246945168,
-0.010823483752478102)
pattern step to: T (value 0.21828405035187581)
(-0.26268641246945146,
-0.1983234837524781)
good move to: U (value 0.0073623007302194444)
(-0.075186412469451458,
-0.010823483752478102)
pattern step to: V (value 0.0085207066742778292)
(0.11231358753054876.
-0.010823483752478102)
good move to: W (value 0.0073623007302194201)
(-0.075186412469451236,
-0.010823483752478102)
pattern step to: X (value 0.0073623007302193975)
(-0.075186412469451014,
-0.010823483752478102)
pattern step to: Y (value 0.007362300730219375)
(-0.075186412469450792,
-0.010823483752478102)

The pattern steps continue to move off of the lattice until stopping due to the maximum function calls condition at 1000004 calls.

By then the current point has shifted to (-0.075186412425045646, -0.010823483752478102).

Figure 6: HJSearch Numerical Error—the final bog down

improvement at a trial step located within floating point error of the last base point. Particular pattern extending steps that may create this problem step a distance Δ along a coordinate direction. Floating point error may then create an "echo" of the last base point of the form $b_k + \epsilon e_i$, where ϵ is a value very near machine epsilon. If this echo of the last base point should happen to have a better function value than the base point, the implementation finds the distance between the two to calculate the next pattern extending step. The length of the next pattern extending step is then on the order of machine epsilon, but still large enough for a well-conditioned function to show improvement over the value at the base point. In that way, the slight discrepancy created by rounding error becomes the basis for a degeneration of the search into a virtually infinite series of small steps off of the rational lattice, which in effect tricks the implementation into believing that the moves being conducted are successful.

7.2.4 The Simple Fix

Again, I would point out that the symptoms and solution of this numerical error had been noted previously in [2]. The relatively simple solution to the problem proposed in [2] requires comparing the locations of the permanent base points to the current iterate. If all of the components of the current iterate are within 0.5Δ of the last permanent base point, the pattern extending step is deemed a failure and the last base point becomes the location of the new current iterate, which amounts to a slight shift. I also implement storage vectors to represent the current iterate at the start of each iteration so as to reduce the effects of possible numerical error introduced by adding

and subtracting Δ. I would remark here that this last security measure is performed automatically when the pattern is stored explicitly in a matrix so that any numerical error introduced in my implementations of `CompassSearch` and `NLessSearch` is not detrimental to the convergence. Refer to section 10 to view the code.

7.2.5 Regarding the Theory

Of course, nothing in the convergence theory would allow for the introduction of such problems because the mathematical theory requires the search to remain on a rational lattice. From [18], the hypothesis on exploratory moves begins by stating that the trial step is $s_k \in \Delta_k P_k$ where P_k is the pattern matrix. My first implementation fails to ensure that the trial point is in fact located on that theoretical lattice by ignoring the possibility of numerical error. The numerical shifting in a slightly improving direction also effectively nullifies the algorithms ability to find iterations that "fail", i.e. discover no function decrease at any of the trial points. Failing to find improvement at any trial point in the pattern represents a fundamental step toward convergence. Without it, the search cannot reduce Δ and thus cannot gain access to the trial points that exist at a higher resolution of the lattice. The search then loses its guarantee that there are a finite number of improving trial points and abandons the proof of its convergence. The gap between theory and implementation yawns wide here, yet the theory in its strictest sense may provide remedies for this sort of numerical error as well.

7.2.6 Further Observations

Having experienced the possibility of floating point error in these relatively simple optimization algorithms, I am curious whether a practical set of implementations could be crafted to follow the convergence results explicitly in calculating trial points by the formulae presented in [18]. Torczon defines a pattern matrix P through $P = BC_k$ with a real basis matrix $B \in \mathbf{R}^{n \times n}$ and a generating matrix C_k. The trial step may then be calculated according to $s_k^i = \Delta_k B c_k^i$ where c_k^i represents a column of $C_k = [c_k^1 \dots c_k^p]$. The idea would be to store and update the *integer* matrix C_k and compute each trial step as $s_k^i = \Delta_k B_k c_k^i$ as a way to at least minimize—if not eliminate—the possibility of floating point error.

Whether a closer marriage of the theory and the implementation of pattern searches might help prevent numerical error or whether such a connection would encumber the algorithms inordinately is a question still to be answered.

8 Further Study

I have mentioned that my project has spawned more questions than answers about the behavior of pattern searches. Particularly of interest to me are outcomes of incorporating search history into pattern search algorithms (such as one sees with the Hooke and Jeeves pattern search algorithm) and the possibility of finding better local mimima on functions with multiple minima.

In searching for a simple explanation of the greatly improved efficiency found in the Hooke and Jeeves searches, I had hoped to see that the pattern extending

step would begin to approximate the Newton direction but found no evidence of this. Baffled, I then calculated the angles between the Newton step from the current iterate and all of the other trial steps in the pattern. Again, I found nothing that implied these angles were converging to zero. When I compared the smallest angles between the Newton direction and a trial step or pattern extending step in the `HJSearch` with the smallest angles between the Newton Step and a trial step in the `CompassSearch`, I noted that the average minimum angle in the `HJSearch` actually exceeded that of the `CompassSearch` pattern for a series of search iterations on the same objective functions. Apparently the effects of incorporating a search history on the Hooke and Jeeves method's efficiency is more complex than would have been convenient.

Another issue of specific interest to my `EdHJSearch` implementation lies in the efficiency of the searches when a factor other than 1/2 is used to reduce Δ more aggressively. I ran preliminary tests with my existing testing structure and discovered that the `EdHJSearch` still gave somewhat improved efficiency over the `HJSearch` for factors of 1/4 and 1/8, which alleviated my concern that the improved efficiency for my tests relied solely on the fact that the `HJSearch` returned *exactly* to points already visited. A more extensive test, including all of the search variants, remains to be undertaken.

I feel strongly attracted to the idea of running tests on functions created via kriging [19], a method of interpolation that can be applied to randomly generated points, yielding a virtually unlimited supply of complicated test functions more closely resembling the general functions for which the pattern search method would be most applicable in practice. Particularly appealing is the notion of experimenting with test

functions with multiple minima in order to explore a pattern search's ability to find the best, rather than merely the closest, local optimum and the parameters that might boost that ability. I would like to see whether the krigifier [19] could help to structure a general testing environment that would allow more complicated objective functions while maintaining some of the advantages that creating one's own test functions has to offer.

I even implemented a search class I call `RandomSearch`, which attempts to boost the search's ability to find the global optimum by adding randomly generated search vectors to the end of an `NLess` pattern. Unlike the other searches I have implemented, the `RandomSearch` tries every point in the pattern before selecting the best as the current iterate of the next search. The lengths of the random vectors grow with the distance the search has traveled from the initial iterate. Also, if a random vector does find improvement, that vector is kept in the pattern when the other random vectors are reinitialized to new values. Both of these characteristics are based on my intuitive notions that a search that has traveled far from its origin may be optimizing a function rife with local minima and that, like in a Hooke and Jeeves search, optimism regarding improving directions may be rewarded. Work remains to ensure that the `RandomSearch` fulfills the requirements of a pattern search. Specifically, the random points should be shifted back onto the rational lattice, as suggested by Michael Trosset and Virginia Torczon, to ensure that the number of potential improving trial points is bounded. More thorough testing of such a heuristic search on complicated objective functions is required before judging the success of something like a `RandomSearch`.

I would hope to see some of these questions resolved in the future. My project's

testing phase offers an early attempt at addressing complex issues involved in studying optimization methods. In the best case, future research by myself and others will greatly improve on this model. I await with enthusiasm the publication of other resolutions to the implementation issues with which I have struggled and the creation of yet simpler sets of building blocks for pattern search implementations. Most of all, I anticipate enjoying the creativity of the pattern search algorithms other minds might yield. I hope that my work will offer other researchers an excuse to study further the potential of these algorithms.

References

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

[2] M. Bell and M. C. Pike, *Remark on Agorithm 178 [E4] Direct Search*, Communications of the Association for Computing Machinery, 9 (1966).

[3] D. M. Bortz and C. T. Kelley, *The simplex gradient and noisy optimization problems*, in Computational Methods for Optimal Design and Control, J. Borggaard, J. Burns, E. Cliff, and S. Schreck, eds., Birkhauser, 1998.

[4] G. E. P. Box, *Evolutionary operation: A method for increasing industrial productivity*, Applied Statistics, 6 (1957), pp. 81–101.

[5] M. Box, D. Davies, and W. Swann, *NON-LINEAR OPTIMIZATION TECHNIQUES*, Oliver and Boyd Ltd, Edinburgh, 1969.

[6] J. Czyzyk, M. P. Mesnier, and J. J. Moré, *The network-enabled optimization system (NEOS) server*, Tech. Rep. 97/02, Optimization Technology Center, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60139, February 1997.

[7] J. Czyzyk, J. H. Owen, and S. J. Wright, *NEOS: Optimization on the internet*, Tech. Rep. 97/04, Optimization Technology Center, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60139, June 1997.

[8] R. Hooke and T. A. Jeeves, *Direct search solution of numerical and statistical problems*, Journal of the Association for Computing Machinery, 8 (1961), pp. 212–229.

[9] S. Jocoby, J. Kowalik, and J. Pizzo, *Iterative Methods for Nonlinear Optimization Problems*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

[10] M. G. Johnson, `hooke.c`. Found in Netlib at `www.netlib.org/opt/hooke.c`, February 1994.

[11] A. F. Kaupe, Jr., *Algorithm 178 Direct Search*, Communications of the Association for Computing Machinery, 6 (1963).

[12] R. M. Lewis, `vector.C`, 1998. ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-0001.

[13] R. M. Lewis, V. Torczon, and M. W. Trosset, *Why pattern search works*, Optima, (1998), pp. 1–7. Also available as ICASE technical report 98-57.

[14] R. M. Lewis and V. J. Torczon, *Rank ordering and positive bases in pattern search algorithms*, Tech. Rep. 96–71, Institute for Computer Applications in Science and Engineering, Mail Stop 403, NASA Langley Research Center, Hampton, Virginia 23681–2199, 1996. In revision for *Mathematical Programming*.

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

[16] S. Park and L. Leemis, *Discrete-event simulation: A first course*. College of William and Mary. Class notes for a simulation course in the Computer Science department.

[17] S. K. Park and K. W. Miller, *Random number generators: Good ones are hard to find*, Communications of the ACM, 31 (1988).

[18] V. Torczon, *On the convergence of pattern search algorithms*, SIAM Journal on Optimization, 7 (1997), pp. 1–25.

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

9 Displays of Results

9.1 Halting Criteria

Stopping Criteria vs Returned Function Value and Distance from the Optimum
(`NLessSearch` in 5 variables with correct digits in red)

std. deviation	Δ	function value	2norm
6.5486433983619570	0.6962268138239019	2.4483717410566310	0.2777276094251399
0.8963664167841120	0.3481134069119510	1.7640603407930924	0.2176549125833814
0.0864294848558236	0.1740567034559755	1.6145138133840160	0.1084697941566830
0.0019435277139807	0.0870283517279877	1.4960316981463304	0.1159039741974479
0.0001572197295223	0.0435141758639939	1.4842980031542836	0.0704179184654578
0.0000095818143977	0.0217570879319969	1.4743470866911506	0.0405719037218861
0.0000002451147079	0.0108785439659985	1.4704546621276158	0.0347609196921366
0.0000000233681897	0.0054392719829992	1.4704546621276158	0.0347609196921366
0.0000000014623741	0.0027196359914996	1.4700532980495444	0.0248115028149709
0.0000000000553148	0.0013598179957498	1.4699489450489640	0.0218490362789150
0.0000000000164568	0.0006799089978749	1.4698667243929047	0.0175966242272939
0.0000000000017077	0.0003399544989375	1.4697677960311799	0.0100138238099774
0.0000000000000594	0.0001699772494687	1.4697270352791718	0.0038904854860887
0.0000000000000036	0.0000849886247344	1.4697216248598952	0.0019382378095814
0.0000000000000001	0.0000424943123672	1.4697201725880154	0.0008493561483688
0.0000000000000000	0.0000212471561836	1.4697199201379811	0.0004321719657023
0.0000000000000000	0.0000106235780918	1.4697198459928282	0.0001688909052103
0.0000000000000000	0.0000053117890459	1.4697198376569938	0.0001089433999276
0.0000000000000000	0.0000026558945229	1.4697198343587019	0.0000721440217908
0.0000000000000000	0.0000013279472615	1.4697198322765974	0.0000289467518772
0.0000000000000000	0.0000006639736307	1.4697198319422011	0.0000113059332085
0.0000000000000000	0.0000003319868154	1.4697198319057987	0.0000073254833814
0.0000000000000000	0.0000001659934077	1.4697198318911258	0.0000047849244202
0.0000000000000000	0.0000000829967038	1.4697198318815330	0.0000016498976358
0.0000000000000000	0.0000000414983519	1.4697198318805513	0.0000008062142814
0.0000000000000000	0.0000000207491760	1.4697198318803257	0.0000004305795219
0.0000000000000000	0.0000000103745880	1.4697198318802669	0.0000002459337107
0.0000000000000000	0.0000000051872940	1.4697198318802593	0.0000002175628883
0.0000000000000000	0.0000000025936470	1.4697198318802593	0.0000002175628883
0.0000000000000000	0.0000000012968235	1.4697198318802593	0.0000002175628883
0.0000000000000000	0.0000000006484117	1.4697198318802593	0.0000002175628883

Stopping Criteria vs Returned Function Value and Distance from the Optimum

(`HJSearch` in 5 variables with correct digits in red)

std. deviation	Δ	function value	2norm
9.3045897498347561	0.6962268138239019	5.4361106754128041	2.3963015589443808
0.4397293876738081	0.3481134069119510	1.9159086219448116	0.6983895923138458
0.0410391664545333	0.1740567034559755	1.6606319720281675	0.4503869030736324
0.0013106836834532	0.0870283517279877	1.4844950859730970	0.1539439700826097
0.0002155320262490	0.0435141758639939	1.4839329285424792	0.1193151775059503
0.0000103551097665	0.0217570879319969	1.4719532056197253	0.0340026098043645
0.0000004083040040	0.0108785439659985	1.4700097998460036	0.0157919106168492
0.0000000404631436	0.0054392719829992	1.4699006162510915	0.0146783468207784
0.0000000039524363	0.0027196359914996	1.4698655024813616	0.0165829903968097
0.0000000001053694	0.0013598179957498	1.4697300627272001	0.0037575966250456
0.0000000000099616	0.0006799089978749	1.4697241294437269	0.0026128528103905
0.0000000000004175	0.0003399544989375	1.4697203396318461	0.0008546098460838
0.0000000000000347	0.0001699772494687	1.4697203309184161	0.0009857505477117
0.0000000000000027	0.0000849886247344	1.4697198554491324	0.0000742441507744
0.0000000000000001	0.0000424943123672	1.4697198351311140	0.0000435103250214
0.0000000000000000	0.0000212471561836	1.4697198345462448	0.0000326892368369
0.0000000000000000	0.0000106235780918	1.4697198321413172	0.0000138786375838
0.0000000000000000	0.0000053117890459	1.4697198320338560	0.0000174583151828
0.0000000000000000	0.0000026558945229	1.4697198319056393	0.0000058036150574
0.0000000000000000	0.0000013279472615	1.4697198319056393	0.0000058036150574
0.0000000000000000	0.0000006639736307	1.4697198318827320	0.0000018108722900
0.0000000000000000	0.0000003319868154	1.4697198318803228	0.0000002150972866
0.0000000000000000	0.0000001659934077	1.4697198318802984	0.0000001662453937
0.0000000000000000	0.0000000829967038	1.4697198318802445	0.0000000439549373
0.0000000000000000	0.0000000414983519	1.4697198318802440	0.0000000712678851
0.0000000000000000	0.0000000207491760	1.4697198318802382	0.0000000444503609
0.0000000000000000	0.0000000103745880	1.4697198318802382	0.0000000444503609
0.0000000000000000	0.0000000051872940	1.4697198318802376	0.0000000391633863
0.0000000000000000	0.0000000025936470	1.4697198318802376	0.0000000391633863
0.0000000000000000	0.0000000012968235	1.4697198318802376	0.0000000391633863
0.0000000000000000	0.0000000006484117	1.4697198318802376	0.0000000391633863

9.2 Efficiency

The following bar charts offer a visual guide to the efficiency of the four searches in terms of objective function evaluations required before convergence. Note that convergence is defined by Δ falling below `tolerance`; none of the searches were halted based on the number of objective function evaluations. The horizontal lines within the bars demarcate the minimum number of function calls required for ten, twenty-five, fifty, seventy-five, and ninety percent of the ten thousand searches to converge to the desired `tolerance` value for the trial. The extremity in the differences in the number of function evaluations required by the four searches makes the lines for the `HJ` and `EdHJ` searches difficult to distinguish in some of the later graphs. The actual values are still quite distinguishable, with the values for `EdHJSearch` lower than those for `HJSearch`.

Figure 7: Hooke and Jeeves Search and Edited Hooke and Jeeves Search... trial 0 in two variables and trial 9 in five variables

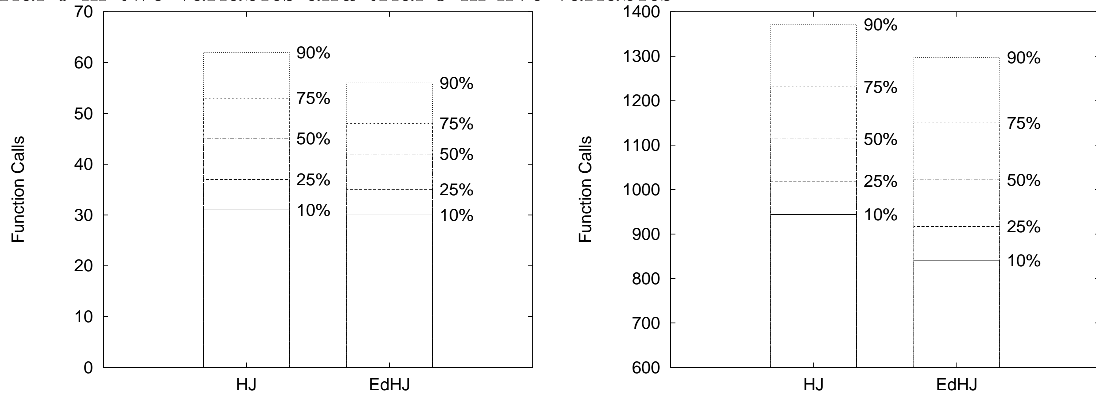


Figure 8: trial 0 and trial 1 in two variables

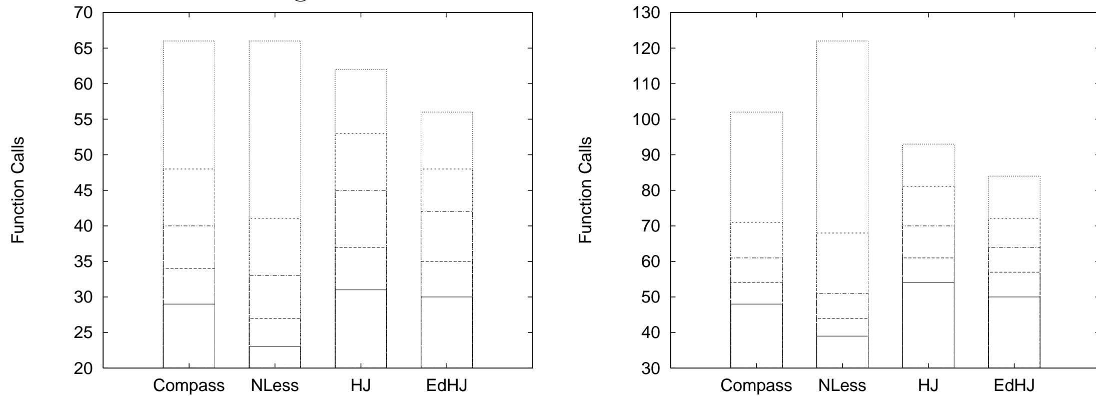


Figure 9: trial 2 and trial 3 in two variables

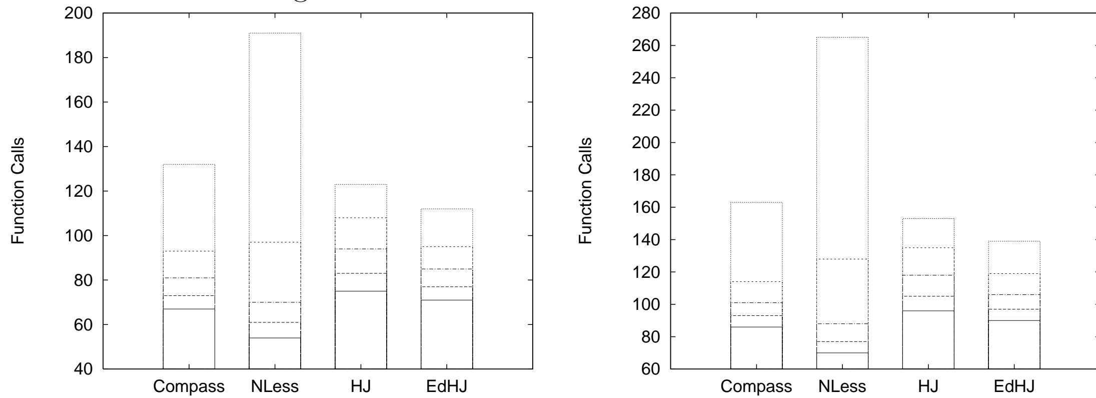


Figure 10: trial 4 and trial 5 in two variables

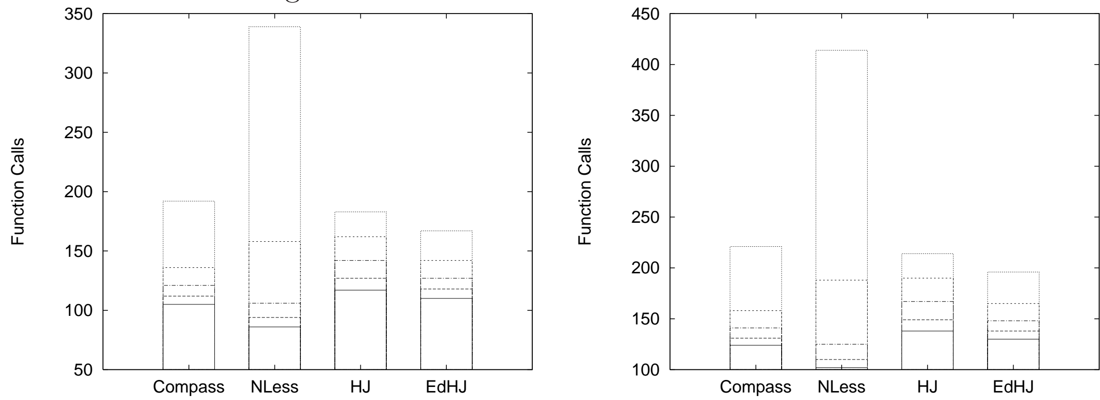


Figure 11: trial 6 and trial 7 in two variables

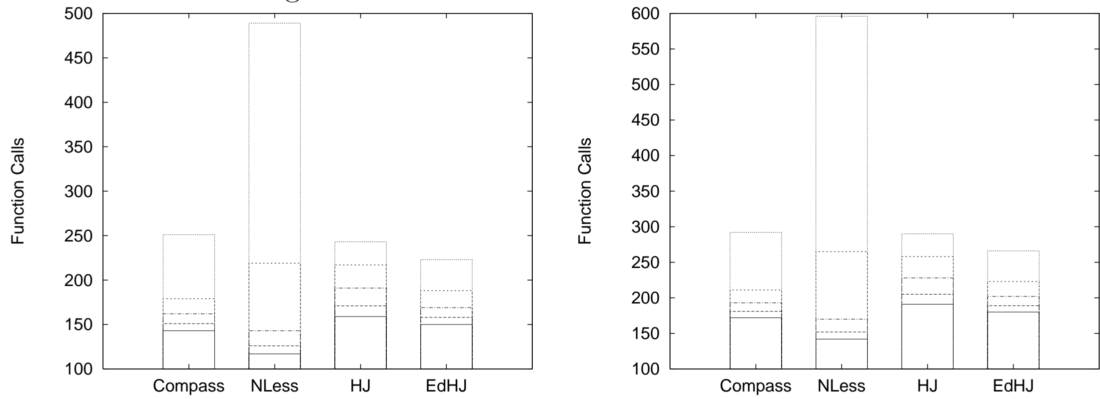


Figure 12: trial 8 and trial 9 in two variables

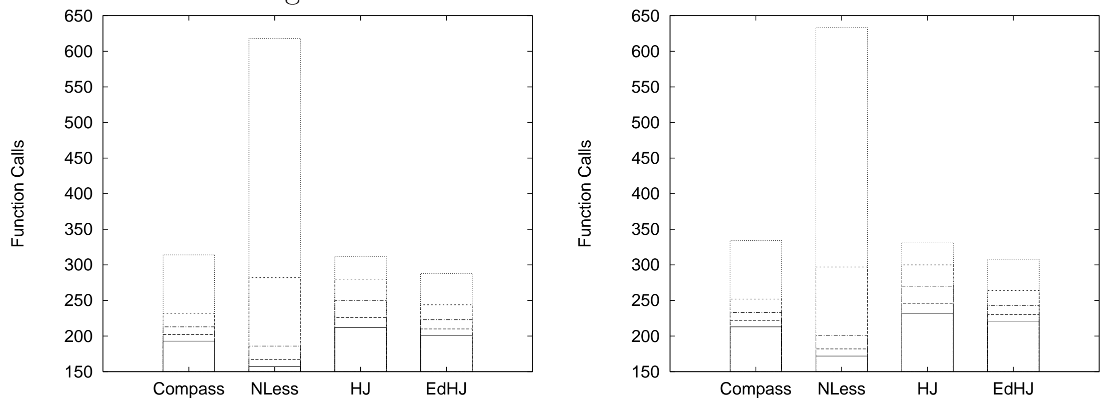


Figure 13: trial 0 and trial 1 in three variables

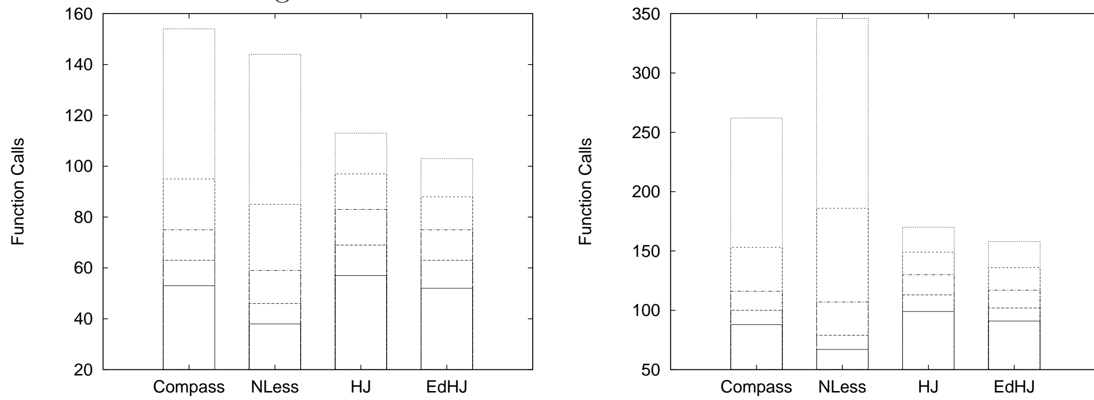


Figure 14: trial 2 and trial 3 in three variables

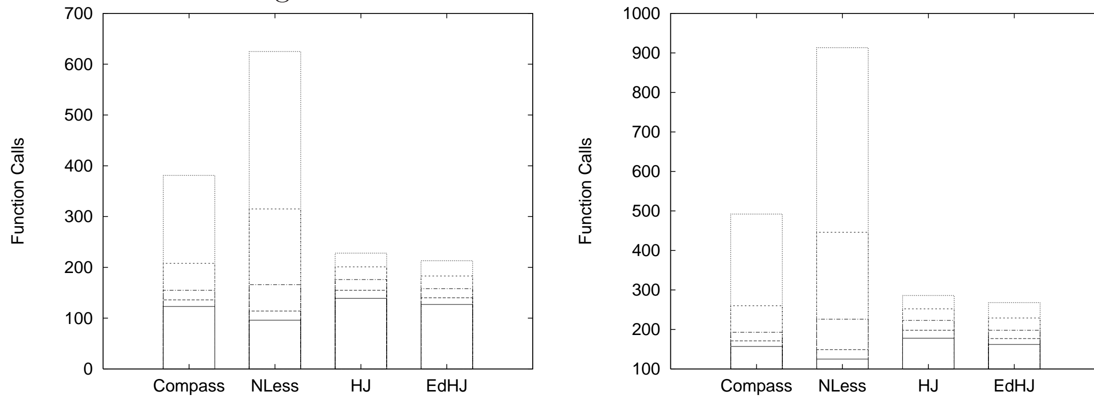


Figure 15: trial 4 and trial 5 in three variables

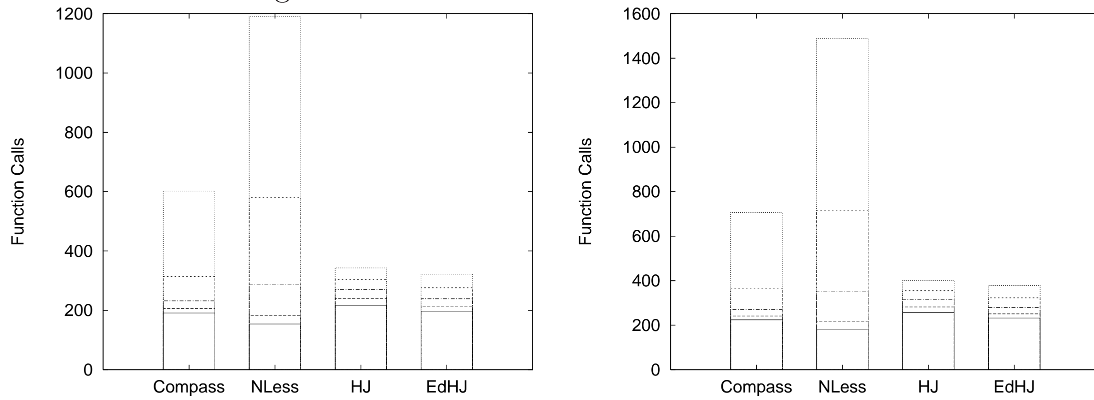


Figure 16: trial 6 and trial 7 in three variables

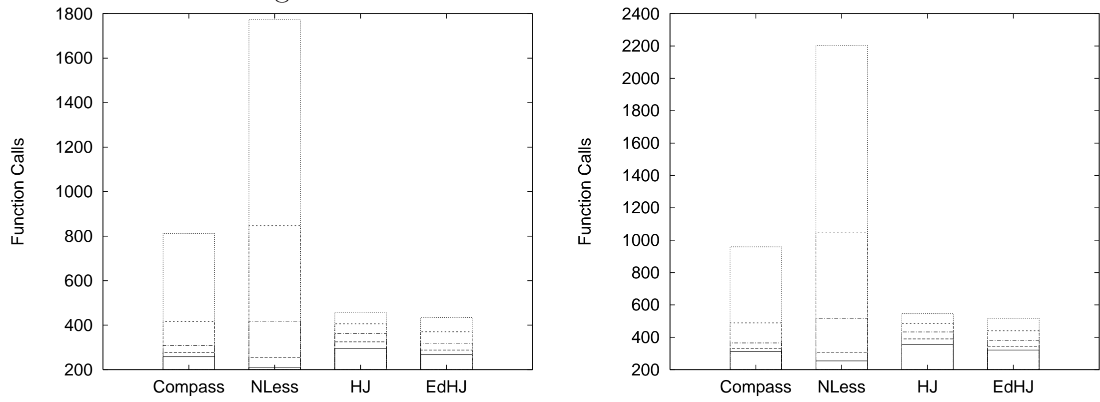


Figure 17: trial 8 and trial 9 in three variables

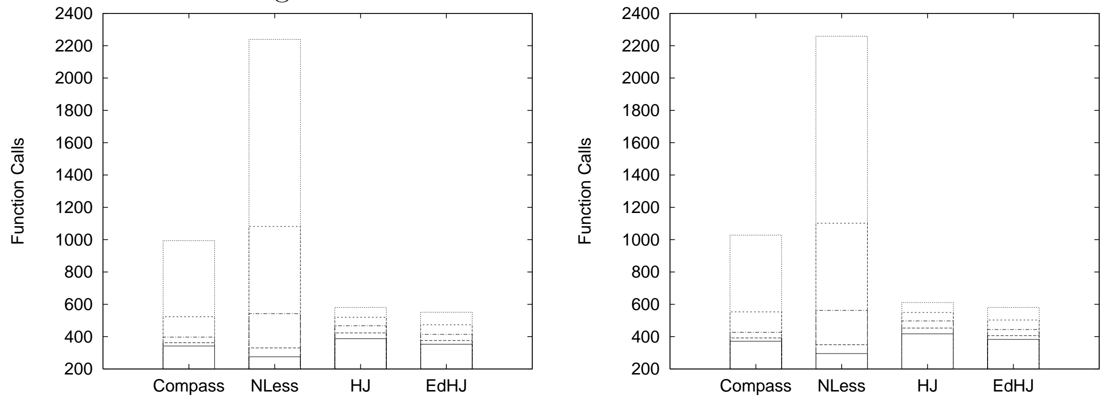


Figure 18: trial 0 and trial 1 in four variables

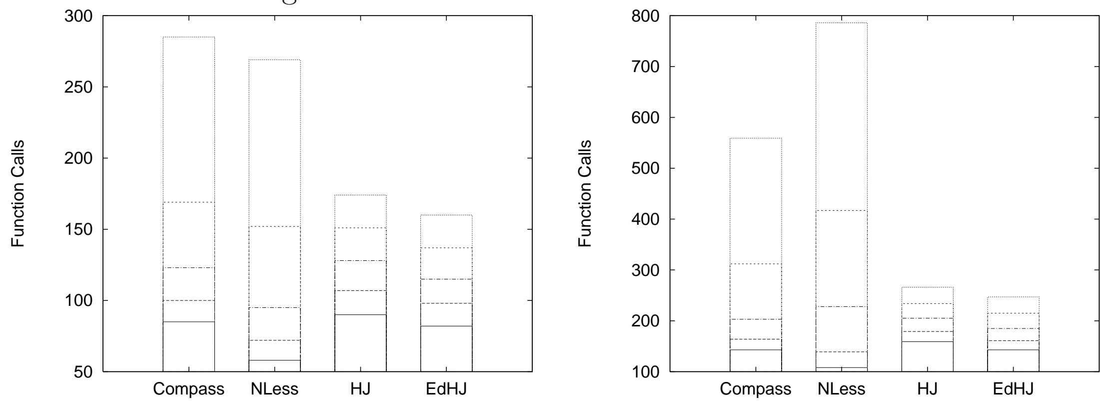


Figure 19: trial 2 and trial 3 in four variables

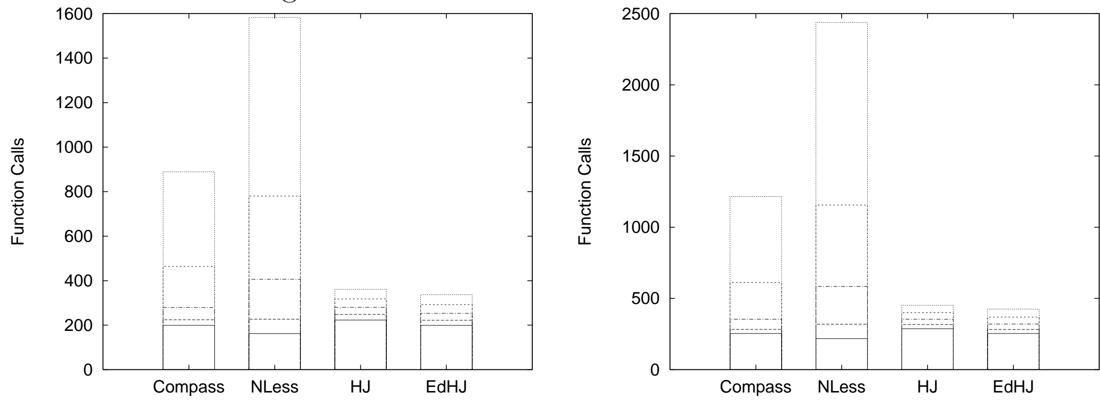


Figure 20: trial 4 and trial 5 in four variables

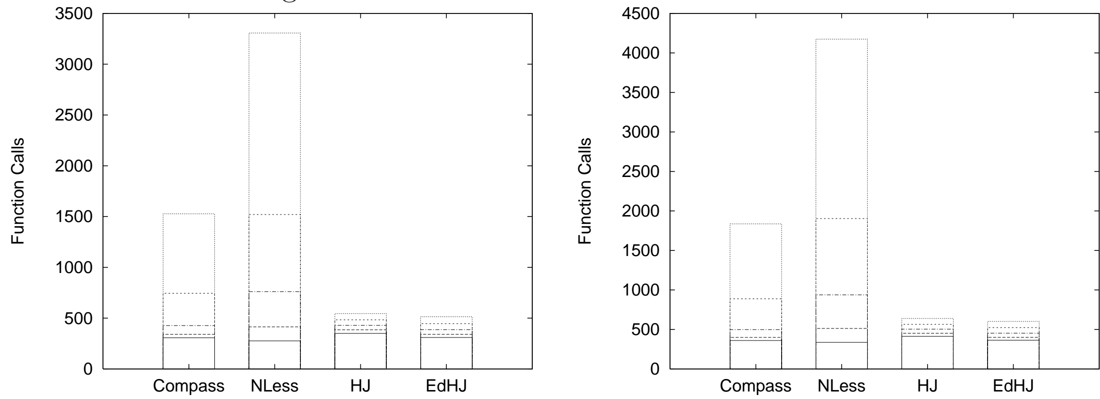


Figure 21: trial 6 and trial 7 in four variables

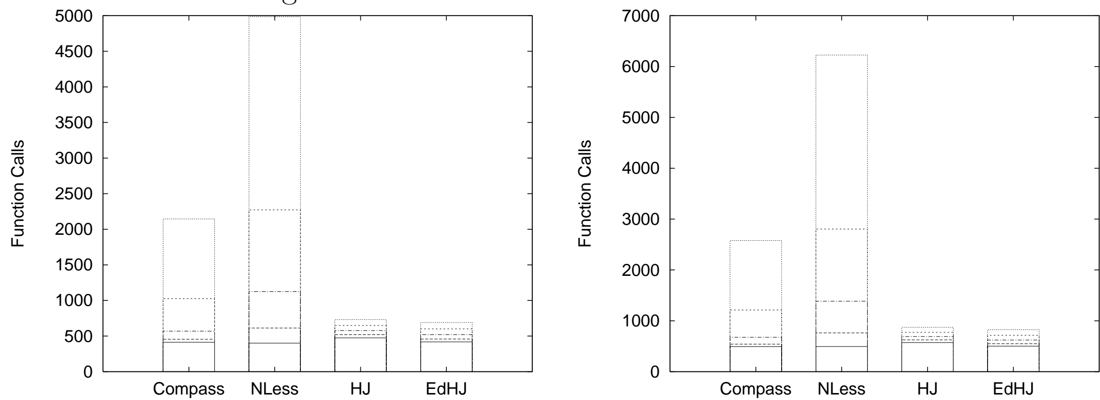


Figure 22: trial 8 and trial 9 in four variables

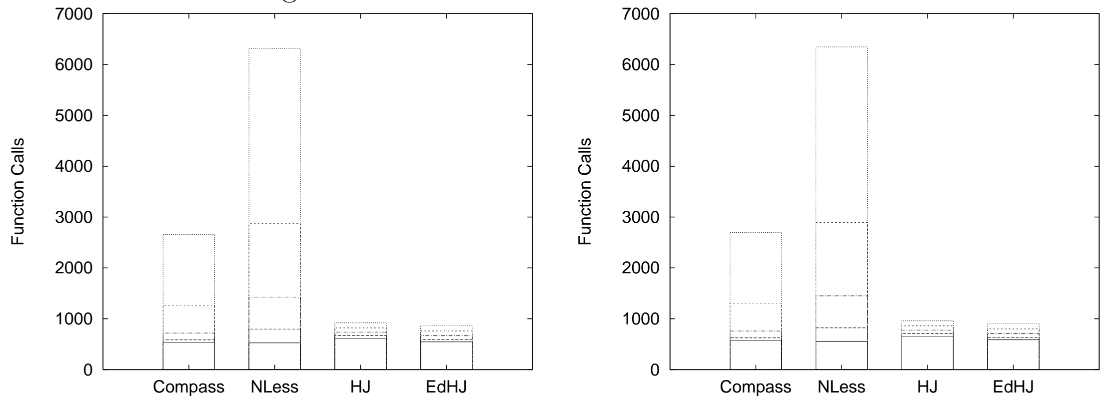


Figure 23: trial 0 and trial 1 in five variables

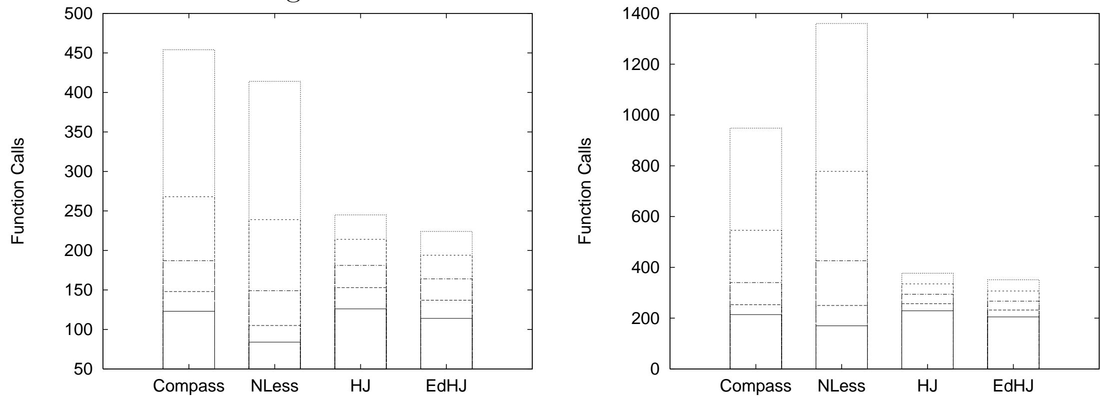


Figure 24: trial 2 and trial 3 in five variables

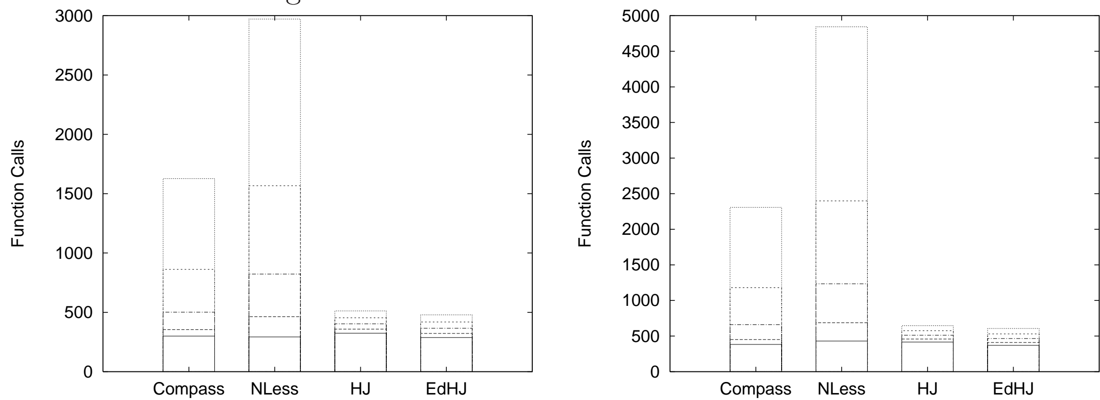


Figure 25: trial 4 and trial 5 in five variables

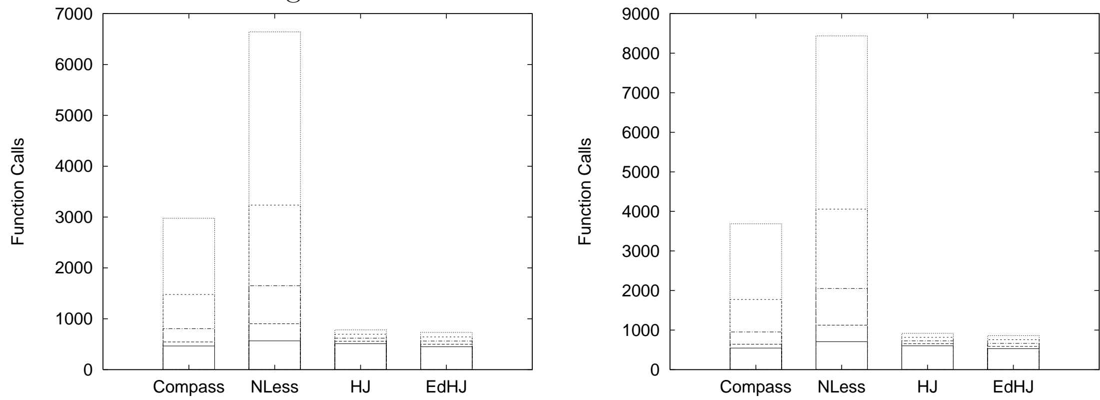


Figure 26: trial 6 and trial 7 in five variables

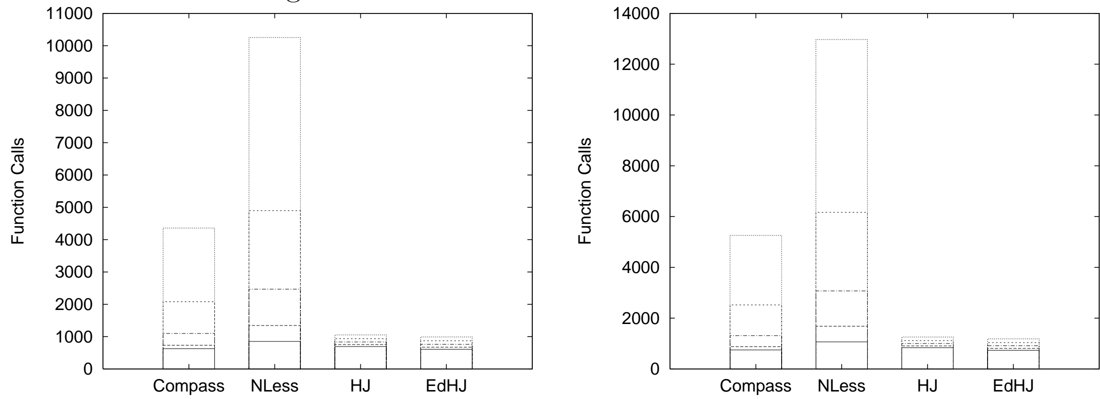
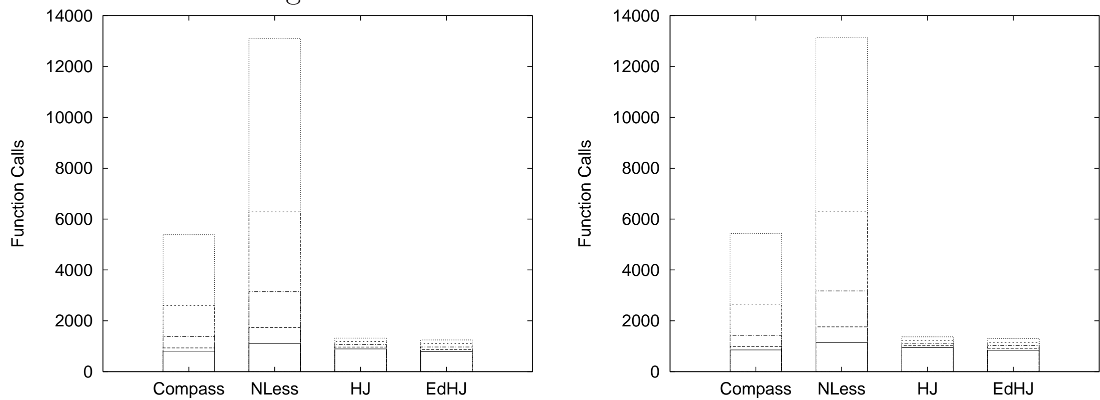


Figure 27: trial 8 and trial 9 in five variables



9.3 Function Value Accuracy

The following tables record the average difference between the function value returned as the minimum by the search and the true $f(x_*)$. Confidence intervals were taken to show the range in the mean of the accuracy within which ninety-five percent of accuracy averages for similar trials of ten thousand could be expected to fall. While perhaps not as enlightening as the previous bar charts, the tables do exhibit patterns. Note, in particular, the shift that occurs typically between trial 3 and trial 4 in the relative accuracies of the `HJSearch` and `EdHJSearch` to the `NLessSearch` and `CompassSearch`.

Function Value Accuracy Estimates in Two Dimensions

trial 0		(mean)	
Compass	0.0024429972779161	0.0026358312472298	0.0028286652165434
NLess	0.0075555956548713	0.0083266837920918	0.0090977719293123
HJ	0.1743257298389256	0.2147171469038104	0.2551085639686952
EdHJ	0.1743815591017040	0.2147728632123105	0.2551641673229170
trial 1		(mean)	
Compass	0.0000485419562679	0.0000550182932928	0.0000614946303178
NLess	0.0002055073345572	0.0002313644946987	0.0002572216548403
HJ	0.0103406819042583	0.0262490871387065	0.0421574923731548
EdHJ	0.0103415184251897	0.0262499231336132	0.0421583278420367
trial 2		(mean)	
Compass	0.0000008194177042	0.0000009388962398	0.0000010583747754
NLess	0.0000039357779546	0.0000044679466948	0.0000050001154350
HJ	0.0000461714632160	0.0039437589662255	0.0078413464692350
EdHJ	0.0000461800997175	0.0039437675993850	0.0078413550990526
trial 3		(mean)	
Compass	0.0000000129399216	0.0000000148273879	0.0000000167148543
NLess	0.0000000692098901	0.0000000823115480	0.0000000954132060
HJ	−0.0008593299999037	0.0008949620250239	0.0026492540499515
EdHJ	−0.0008593298405541	0.0008949621843423	0.0026492542092387
trial 4		(mean)	
Compass	0.0000000002044363	0.0000000002372084	0.0000000002699805
NLess	0.0000000011244400	0.0000000013378706	0.0000000015513011
HJ	0.0000000001003068	0.0000000001047632	0.0000000001092195
EdHJ	0.0000000001059643	0.0000000001108857	0.0000000001158072

Function Value Accuracy Estimates in Two Dimensions

trial 5		(mean)	
Compass	0.0000000000032144	0.0000000000037146	0.0000000000042149
NLess	0.0000000000179233	0.0000000000211529	0.0000000000243825
HJ	0.0000000000015692	0.0000000000016352	0.0000000000017011
EdHJ	0.0000000000015986	0.0000000000016695	0.0000000000017405
trial 6		(mean)	
Compass	0.0000000000000495	0.0000000000000572	0.0000000000000648
NLess	0.0000000000002790	0.0000000000003300	0.0000000000003811
HJ	0.0000000000000248	0.0000000000000263	0.0000000000000278
EdHJ	0.0000000000000245	0.0000000000000254	0.0000000000000263
trial 7		(mean)	
Compass	0.0000000000000002	0.0000000000000003	0.0000000000000004
NLess	0.0000000000000009	0.0000000000000013	0.0000000000000017
HJ	0.0000000000000000	0.0000000000000000	0.0000000000000000
EdHJ	0.0000000000000000	0.0000000000000001	0.0000000000000001
trial 8		(mean)	
Compass	0.0000000000000001	0.0000000000000002	0.0000000000000003
NLess	0.0000000000000008	0.0000000000000012	0.0000000000000016
HJ	0.0000000000000000	0.0000000000000000	0.0000000000000000
EdHJ	0.0000000000000000	0.0000000000000000	0.0000000000000001
trial 9		(mean)	
Compass	0.0000000000000001	0.0000000000000002	0.0000000000000003
NLess	0.0000000000000008	0.0000000000000012	0.0000000000000016
HJ	0.0000000000000000	0.0000000000000000	0.0000000000000000
EdHJ	0.0000000000000000	0.0000000000000000	0.0000000000000001

Function Value Accuracy Estimates in Three Dimensions

trial 0		(mean)	
Compass	0.0069701877951054	0.0073597651473794	0.0077493424996535
NLess	0.0241033436118908	0.0256655668254416	0.0272277900389924
HJ	0.3095767253516128	0.3785150504080378	0.4474533754644628
EdHJ	0.3099432027161219	0.3788808071761120	0.4478184116361021
trial 1		(mean)	
Compass	0.0001611092596185	0.0001758506821707	0.0001905921047229
NLess	0.0008102046235363	0.0008971670727973	0.0009841295220582
HJ	0.0194396257220741	0.0364623407451212	0.0534850557681682
EdHJ	0.0194499253943019	0.0364726324946375	0.0534953395949730
trial 2		(mean)	
Compass	0.0000027717696410	0.0000033573798107	0.0000039429899804
NLess	0.0000169592957031	0.0000197698970624	0.0000225804984217
HJ	−0.0002240998798550	0.0029774605022160	0.0061790208842870
EdHJ	−0.0002239923561625	0.0029775679876049	0.0061791283313723
trial 3		(mean)	
Compass	0.0000000472629794	0.0000000585394245	0.0000000698158697
NLess	0.0000002812349339	0.0000003283218606	0.0000003754087873
HJ	−0.0003750263787036	0.0003905991055643	0.0011562245898322
EdHJ	−0.0003750243697439	0.0003906011141308	0.0011562265980055
trial 4		(mean)	
Compass	0.0000000007376640	0.0000000009222864	0.0000000011069087
NLess	0.0000000045588320	0.0000000053383168	0.0000000061178016
HJ	0.0000000002069735	0.0000000002212090	0.0000000002354444
EdHJ	0.0000000002290546	0.0000000002583601	0.0000000002876656

Function Value Accuracy Estimates in Three Dimensions

trial 5		(mean)	
Compass	0.0000000000113506	0.0000000000142635	0.0000000000171764
NLess	0.0000000000712437	0.0000000000829994	0.0000000000947551
HJ	0.0000000000032647	0.0000000000035507	0.0000000000038368
EdHJ	0.0000000000036479	0.0000000000038341	0.0000000000040202
trial 6		(mean)	
Compass	0.0000000000001749	0.0000000000002178	0.0000000000002608
NLess	0.0000000000011362	0.0000000000013212	0.0000000000015063
HJ	0.0000000000000505	0.0000000000000533	0.0000000000000561
EdHJ	0.0000000000000586	0.0000000000000625	0.0000000000000664
trial 7		(mean)	
Compass	0.0000000000000007	0.0000000000000011	0.0000000000000015
NLess	0.0000000000000041	0.0000000000000052	0.0000000000000063
HJ	0.0000000000000001	0.0000000000000001	0.0000000000000001
EdHJ	0.0000000000000001	0.0000000000000002	0.0000000000000003
trial 8		(mean)	
Compass	0.0000000000000007	0.0000000000000010	0.0000000000000014
NLess	0.0000000000000039	0.0000000000000049	0.0000000000000060
HJ	0.0000000000000001	0.0000000000000001	0.0000000000000001
EdHJ	0.0000000000000001	0.0000000000000002	0.0000000000000003
trial 9		(mean)	
Compass	0.0000000000000007	0.0000000000000010	0.0000000000000014
NLess	0.0000000000000039	0.0000000000000049	0.0000000000000060
HJ	0.0000000000000001	0.0000000000000001	0.0000000000000001
EdHJ	0.0000000000000001	0.0000000000000002	0.0000000000000003

Function Value Accuracy Estimates in Four Dimensions

trial 0		(mean)	
Compass	0.0159201594769705	0.0167143403605377	0.0175085212441049
NLess	0.0566760424375808	0.0591344835853876	0.0615929247331945
HJ	0.6177154355790652	0.7338436976107635	0.8499719596424619
EdHJ	0.6187762421025244	0.7349021016528481	0.8510279612031718
trial 1		(mean)	
Compass	0.0004221854162892	0.0004618232532246	0.0005014610901599
NLess	0.0022041585340743	0.0023988389508273	0.0025935193675804
HJ	0.0410471306786834	0.0921226323285811	0.1431981339784788
EdHJ	0.0410682501478229	0.0921437374289399	0.1432192247100569
trial 2		(mean)	
Compass	0.0000078657855097	0.0000089433226894	0.0000100208598691
NLess	0.0000448131138199	0.0000494754790865	0.0000541378443531
Random	−0.0093802942067721	0.0191885355207255	0.0477573652482231
EdHJ	−0.0094350772227545	0.0191314385239991	0.0476979542707527
trial 3		(mean)	
Compass	0.0000001298800056	0.0000001492661052	0.0000001686522049
NLess	0.0000007557448806	0.0000008503091859	0.0000009448734912
HJ	−0.0008668449735012	0.0009028250036741	0.0026724949808494
EdHJ	−0.0008668402408988	0.0009028297353493	0.0026724997115973
trial 4		(mean)	
Compass	0.0000000020474356	0.0000000023343659	0.0000000026212963
NLess	0.0000000120400732	0.0000000137226687	0.0000000154052642
HJ	0.0000000003799867	0.0000000004029525	0.0000000004259183
EdHJ	0.0000000004485826	0.0000000004794656	0.0000000005103486

Function Value Accuracy Estimates in Four Dimensions

trial 5		(mean)	
Compass	0.0000000000323386	0.0000000000369915	0.0000000000416444
NLess	0.0000000001910563	0.0000000002166582	0.0000000002422601
HJ	0.0000000000056847	0.0000000000059158	0.0000000000061469
EdHJ	0.0000000000068560	0.0000000000071913	0.0000000000075267
trial 6		(mean)	
Compass	0.0000000000004937	0.0000000000005666	0.0000000000006394
NLess	0.0000000000030440	0.0000000000034493	0.0000000000038547
HJ	0.0000000000000884	0.0000000000000948	0.0000000000001012
EdHJ	0.0000000000001107	0.0000000000001167	0.0000000000001227
trial 7		(mean)	
Compass	0.0000000000000018	0.0000000000000021	0.0000000000000025
NLess	0.0000000000000103	0.0000000000000119	0.0000000000000134
HJ	0.0000000000000002	0.0000000000000002	0.0000000000000002
EdHJ	0.0000000000000003	0.0000000000000003	0.0000000000000003
trial 8		(mean)	
Compass	0.0000000000000017	0.0000000000000020	0.0000000000000023
NLess	0.0000000000000093	0.0000000000000109	0.0000000000000124
HJ	0.0000000000000002	0.0000000000000002	0.0000000000000002
EdHJ	0.0000000000000002	0.0000000000000002	0.0000000000000003
trial 9		(mean)	
Compass	0.0000000000000017	0.0000000000000020	0.0000000000000023
NLess	0.0000000000000093	0.0000000000000109	0.0000000000000124
HJ	0.0000000000000002	0.0000000000000002	0.0000000000000002
EdHJ	0.0000000000000002	0.0000000000000002	0.0000000000000003

Function Value Accuracy Estimates in Five Dimensions

trial 0		(mean)	
Compass	0.0274570867539054	0.0285950974737656	0.0297331081936259
NLess	0.0990485451476941	0.1027495921367188	0.1064506391257436
HJ	0.9003224209169278	1.0671631079235064	1.2340037949300851
EdHJ	0.9026040012195177	1.0694393896339436	1.2362747780483694
trial 1		(mean)	
Compass	0.0007864066132930	0.0008359654073355	0.0008855242013781
NLess	0.0039716913994037	0.0041779169763865	0.0043841425533693
HJ	0.0712912375515785	0.1213936683423878	0.1714960991331972
EdHJ	0.0713303435492137	0.1214327396254077	0.1715351357016018
trial 2		(mean)	
Compass	0.0000146447364491	0.0000161270082719	0.0000176092800946
NLess	0.0000865293910833	0.0000940617909494	0.0001015941908156
HJ	0.0007158605639889	0.0258243802644783	0.0509328999649678
EdHJ	0.0007163963178371	0.0258249158070007	0.0509334352961644
trial 3		(mean)	
Compass	0.0000002450972453	0.0000002761302317	0.0000003071632181
NLess	0.0000014638865221	0.0000017260182329	0.0000019881499437
HJ	−0.0055107009893715	0.0057391957444782	0.0169890924783278
EdHJ	−0.0055106957090675	0.0057392010237465	0.0169890977565604
trial 4		(mean)	
Compass	0.0000000036256379	0.0000000045795556	0.0000000055334734
NLess	0.0000000231914512	0.0000000272466341	0.0000000313018170
HJ	0.0000000005664018	0.0000000005912256	0.0000000006160494
EdHJ	0.0000000006982424	0.0000000007492825	0.0000000008003225

Function Value Accuracy Estimates in Five Dimensions

trial 5		(mean)	
Compass	0.0000000000605032	0.0000000000769118	0.0000000000933203
NLess	0.0000000003589325	0.0000000004235556	0.0000000004881788
HJ	0.0000000000091625	0.0000000000100741	0.0000000000109857
EdHJ	0.0000000000109840	0.0000000000115559	0.0000000000121279
trial 6		(mean)	
Compass	−0.0000000000029672	0.0000000000054022	0.0000000000137715
NLess	0.0000000000018992	0.0000000000107494	0.0000000000195996
HJ	0.0000000000001423	0.0000000000001489	0.0000000000001555
EdHJ	0.0000000000001765	0.0000000000001880	0.0000000000001995
trial 7		(mean)	
Compass	−0.0000000000040937	0.0000000000042722	0.0000000000126381
NLess	−0.0000000000041903	0.0000000000046433	0.0000000000134770
HJ	0.0000000000000003	0.0000000000000004	0.0000000000000004
EdHJ	0.0000000000000004	0.0000000000000005	0.0000000000000006
trial 8		(mean)	
Compass	−0.0000000000040940	0.0000000000042719	0.0000000000126378
NLess	−0.0000000000041921	0.0000000000046415	0.0000000000134752
HJ	0.0000000000000003	0.0000000000000003	0.0000000000000003
EdHJ	0.0000000000000003	0.0000000000000004	0.0000000000000005
trial 9		(mean)	
Compass	−0.0000000000040940	0.0000000000042719	0.0000000000126378
NLess	−0.0000000000041921	0.0000000000046415	0.0000000000134752
HJ	0.0000000000000003	0.0000000000000003	0.0000000000000003
EdHJ	0.0000000000000003	0.0000000000000004	0.0000000000000005

9.4 Distance From the Optimum

The following tables record the average distance between the point returned as the minimum by the search and the true x_*. If I were to run another similar trial with ten thousand tests, its average would fall within the ranges given with ninety-five percent confidence. Note again that the relative accuracies of the `HJSearch` and `EdHJSearch` begin to improve noticably over those of the `NLessSearch` and `CompassSearch` after trial 3. Particularly of interest in comparing these tables with the previous tables is to notice that the accuracies of the `HJSearch` and `EdHJSearch` in terms of locating the minimum are not particularly poor before trial 3.

Optimal Point Accuracy Estimates in Two Dimensions

trial 0		(mean)	
Compass	0.0385279511719603	0.0410500566401427	0.0435721621083250
NLess	0.0683579604348376	0.0723663207187724	0.0763746810027071
HJ	0.0577232843295159	0.0622275679510949	0.0667318515726740
EdHJ	0.0590211821177044	0.0635331127626647	0.0680450434076251
trial 1		(mean)	
Compass	0.0056092358570495	0.0061587985417031	0.0067083612263568
NLess	0.0121277629833288	0.0132287218505115	0.0143296807176942
HJ	0.0060858112575836	0.0075375104933280	0.0089892097290725
EdHJ	0.0062486716615709	0.0077015105665452	0.0091543494715196
trial 2		(mean)	
Compass	0.0007336904853225	0.0008091192100996	0.0008845479348767
NLess	0.0017427083005539	0.0019184944799659	0.0020942806593779
HJ	0.0005539914843042	0.0011881449490278	0.0018222984137514
EdHJ	0.0005699320964900	0.0012041557430172	0.0018383793895444
trial 3		(mean)	
Compass	0.0000916043557912	0.0001010706481809	0.0001105369405706
NLess	0.0002323826387481	0.0002637993415911	0.0002952160444341
HJ	−0.0001240652846731	0.0002465583027102	0.0006171818900934
EdHJ	−0.0001209316141433	0.0002496987095321	0.0006203290332075
trial 4		(mean)	
Compass	0.0000115092683997	0.0000127361307792	0.0000139629931587
NLess	0.0000296709825200	0.0000336666645594	0.0000376623465988
HJ	0.0000066652277411	0.0000070258205599	0.0000073864133787
EdHJ	0.0000071874378134	0.0000075630364406	0.0000079386350679

Optimal Point Accuracy Estimates in Two Dimensions

trial 5		(mean)	
Compass	0.0000014431887205	0.0000016043726175	0.0000017655565144
NLess	0.0000037650103370	0.0000042633753581	0.0000047617403792
HJ	0.0000008400094443	0.0000008734748677	0.0000009069402912
EdHJ	0.0000008665772821	0.0000009099442199	0.0000009533111577
trial 6		(mean)	
Compass	0.0000001798938824	0.0000002000664323	0.0000002202389822
NLess	0.0000004720895414	0.0000005342406479	0.0000005963917543
HJ	0.0000001062128827	0.0000001107493630	0.0000001152858433
EdHJ	0.0000001065542916	0.0000001102146985	0.0000001138751055
trial 7		(mean)	
Compass	0.0000000124407547	0.0000000146087018	0.0000000167766490
NLess	0.0000000305286407	0.0000000350566682	0.0000000395846957
HJ	0.0000000060979101	0.0000000063853126	0.0000000066727151
EdHJ	0.0000000066714581	0.0000000073213483	0.0000000079712386
trial 8		(mean)	
Compass	0.0000000117646541	0.0000000139274847	0.0000000160903152
NLess	0.0000000283779662	0.0000000327341601	0.0000000370903539
HJ	0.0000000056605309	0.0000000059467027	0.0000000062328745
EdHJ	0.0000000062188606	0.0000000068684281	0.0000000075179956
trial 9		(mean)	
Compass	0.0000000117646368	0.0000000139274674	0.0000000160902980
NLess	0.0000000283778612	0.0000000327340553	0.0000000370902495
HJ	0.0000000056605188	0.0000000059466907	0.0000000062328626
EdHJ	0.0000000062188485	0.0000000068684160	0.0000000075179836

Optimal Point Accuracy Estimates in Three Dimensions

trial 0		(mean)	
Compass	0.0829733865784636	0.0869186671636808	0.0908639477488979
NLess	0.1564860320751748	0.1628158316972420	0.1691456313193092
HJ	0.0875096704533737	0.0932838025452382	0.0990579346371027
EdHJ	0.0920273826374173	0.0978863407125467	0.1037452987876761
trial 1		(mean)	
Compass	0.0135291918374614	0.0145883606746941	0.0156475295119269
NLess	0.0321585637898069	0.0343408436182048	0.0365231234466026
HJ	0.0093300075497522	0.0111812405933575	0.0130324736369627
EdHJ	0.0102641828361188	0.0121579186751758	0.0140516545142327
trial 2		(mean)	
Compass	0.0018104068122112	0.0020685725681360	0.0023267383240609
NLess	0.0047420920630823	0.0052620404731615	0.0057819888832408
HJ	0.0009123568128967	0.0014416970833654	0.0019710373538341
EdHJ	0.0009913557122403	0.0015211190921268	0.0020508824720134
trial 3		(mean)	
Compass	0.0002343580810911	0.0002735482850942	0.0003127384890973
NLess	0.0006144726615291	0.0006918649163296	0.0007692571711301
HJ	0.0000211545319989	0.0001774472772456	0.0003337400224924
EdHJ	0.0000320375130973	0.0001884025504871	0.0003447675878770
trial 4		(mean)	
Compass	0.0000291798457196	0.0000341329750389	0.0000390861043581
NLess	0.0000785661505143	0.0000885129103370	0.0000984596701597
HJ	0.0000112873484380	0.0000126301034213	0.0000139728584046
EdHJ	0.0000124500644640	0.0000144073792154	0.0000163646939667

Optimal Point Accuracy Estimates in Three Dimensions

trial 5		(mean)	
Compass	0.0000036473593584	0.0000042628688985	0.0000048783784387
NLess	0.0000099000325279	0.0000111313204696	0.0000123626084113
HJ	0.0000014601454640	0.0000015647628314	0.0000016693801988
EdHJ	0.0000015972573041	0.0000016770644604	0.0000017568716166
trial 6		(mean)	
Compass	0.0000004522493837	0.0000005281146680	0.0000006039799522
NLess	0.0000012516886533	0.0000014077956604	0.0000015639026675
HJ	0.0000001798586779	0.0000001916558495	0.0000002034530210
EdHJ	0.0000002022017297	0.0000002150812050	0.0000002279606804
trial 7		(mean)	
Compass	0.0000000307396116	0.0000000383076550	0.0000000458756983
NLess	0.0000000777354139	0.0000000903227726	0.0000001029101312
HJ	0.0000000099967064	0.0000000105922831	0.0000000111878598
EdHJ	0.0000000110607634	0.0000000143878492	0.0000000177149350
trial 8		(mean)	
Compass	0.0000000291053323	0.0000000366653076	0.0000000442252829
NLess	0.0000000729891755	0.0000000855411431	0.0000000980931107
HJ	0.0000000090804080	0.0000000096662634	0.0000000102521187
EdHJ	0.0000000101815523	0.0000000135079900	0.0000000168344277
trial 9		(mean)	
Compass	0.0000000291052327	0.0000000366652082	0.0000000442251837
NLess	0.0000000729890121	0.0000000855409801	0.0000000980929481
HJ	0.0000000090803669	0.0000000096662225	0.0000000102520781
EdHJ	0.0000000101815011	0.0000000135079389	0.0000000168343767

Optimal Point Accuracy Estimates in Four Dimensions

trial 0		(mean)	
Compass	0.1501406284088255	0.1562820974602762	0.1624235665117269
NLess	0.2796997444325927	0.2886681319780744	0.2976365195235562
HJ	0.1233099655423388	0.1304298599453771	0.1375497543484154
EdHJ	0.1319907079514115	0.1392123780405315	0.1464340481296515
trial 1		(mean)	
Compass	0.0269061551752484	0.0288018839474231	0.0306976127195978
NLess	0.0633972230338815	0.0670520426813438	0.0707068623288061
HJ	0.0141759695858268	0.0165128301264647	0.0188496906671027
EdHJ	0.0156147306755179	0.0179845467361783	0.0203543627968386
trial 2		(mean)	
Compass	0.0038126145520266	0.0042311556431662	0.0046496967343057
NLess	0.0094905084690905	0.0103137131720150	0.0111369178749395
HJ	0.0012882093676841	0.0023345849113634	0.0033809604550427
EdHJ	0.0015545521316188	0.0026066528612341	0.0036587535908493
trial 3		(mean)	
Compass	0.0004914438196435	0.0005486564461442	0.0006058690726449
NLess	0.0012512817386852	0.0013738870129007	0.0014964922871162
HJ	−0.0000584475442264	0.0003774453425243	0.0008133382292750
EdHJ	−0.0000393865714460	0.0003965275569212	0.0008324416852883
trial 4		(mean)	
Compass	0.0000620078822107	0.0000691266936339	0.0000762455050571
NLess	0.0001586308375257	0.0001750884711321	0.0001915461047385
HJ	0.0000183763839519	0.0000195000863056	0.0000206237886594
EdHJ	0.0000210946993157	0.0000227291392644	0.0000243635792131

Optimal Point Accuracy Estimates in Four Dimensions

trial 5		(mean)	
Compass	0.0000077746986467	0.0000086689013112	0.0000095631039757
NLess	0.0000199827513813	0.0000220085827092	0.0000240344140371
HJ	0.0000022615009878	0.0000023870860493	0.0000025126711108
EdHJ	0.0000025355611081	0.0000026703847844	0.0000028052084607
trial 6		(mean)	
Compass	0.0000009599118544	0.0000010727004397	0.0000011854890249
NLess	0.0000025244371924	0.0000027810714627	0.0000030377057329
HJ	0.0000002764777721	0.0000002943084439	0.0000003121391156
EdHJ	0.0000003305704688	0.0000003521834736	0.0000003737964785
trial 7		(mean)	
Compass	0.0000000594762388	0.0000000668889089	0.0000000743015789
NLess	0.0000001473482363	0.0000001628301260	0.0000001783120157
HJ	0.0000000155261236	0.0000000168078289	0.0000000180895343
EdHJ	0.0000000183154285	0.0000000193955881	0.0000000204757477
trial 8		(mean)	
Compass	0.0000000552397724	0.0000000624503790	0.0000000696609855
NLess	0.0000001353518694	0.0000001502532035	0.0000001651545376
HJ	0.0000000138096768	0.0000000150346109	0.0000000162595449
EdHJ	0.0000000167210813	0.0000000177772019	0.0000000188333224
trial 9		(mean)	
Compass	0.0000000552397527	0.0000000624503594	0.0000000696609660
NLess	0.0000001353517009	0.0000001502530356	0.0000001651543703
HJ	0.0000000138096509	0.0000000150345851	0.0000000162595193
EdHJ	0.0000000167210644	0.0000000177771850	0.0000000188333057

Optimal Point Accuracy Estimates in Five Dimensions

trial 0		(mean)	
Compass	0.2178632617593219	0.2251811675124386	0.2324990732655554
NLess	0.3968574111604345	0.4071009065284119	0.4173444018963893
HJ	0.1556223350732633	0.1635135894605061	0.1714048438477490
EdHJ	0.1706465816232003	0.1786822842839827	0.1867179869447650
trial 1		(mean)	
Compass	0.0417222167910058	0.0440453326307046	0.0463684484704034
NLess	0.0942831861589735	0.0983896333895409	0.1024960806201083
HJ	0.0186107628363263	0.0215597385900391	0.0245087143437520
EdHJ	0.0207241841447710	0.0237435471102113	0.0267629100756516
trial 2		(mean)	
Compass	0.0059085807187189	0.0064323829964016	0.0069561852740844
NLess	0.0148626440115780	0.0160426448081967	0.0172226456048154
HJ	0.0020401198186421	0.0033997696581853	0.0047594194977285
EdHJ	0.0022688531266929	0.0036319512712130	0.0049950494157331
trial 3		(mean)	
Compass	0.0007661064419559	0.0008562340249422	0.0009463616079285
NLess	0.0019476489169702	0.0022074221092738	0.0024671953015774
HJ	−0.0001948888179214	0.0006601996851754	0.0015152881882723
EdHJ	−0.0001780878726949	0.0006767724870687	0.0015316328468323
trial 4		(mean)	
Compass	0.0000945577605347	0.0001099112131959	0.0001252646658571
NLess	0.0002454535885927	0.0002779677769697	0.0003104819653466
HJ	0.0000243570655416	0.0000256536797467	0.0000269502939518
EdHJ	0.0000281463832350	0.0000296518671403	0.0000311573510456

Optimal Point Accuracy Estimates in Five Dimensions

trial 5		(mean)	
Compass	0.0000119460438834	0.0000138301937755	0.0000157143436676
NLess	0.0000306689450679	0.0000347646248946	0.0000388603047213
HJ	0.0000032154761534	0.0000035476556082	0.0000038798350631
EdHJ	0.0000035702048543	0.0000038072100469	0.0000040442152395
trial 6		(mean)	
Compass	0.0000015046678201	0.0000017583383955	0.0000020120089708
NLess	0.0000034970358261	0.0000048006876900	0.0000061043395539
HJ	0.0000003879518630	0.0000004044137155	0.0000004208755680
EdHJ	0.0000004498452703	0.0000004831076701	0.0000005163700700
trial 7		(mean)	
Compass	0.0000000737196654	0.0000001367306848	0.0000001997417042
NLess	−0.0000003052778287	0.0000009647934464	0.0000022348647215
HJ	0.0000000202640331	0.0000000226205691	0.0000000249771051
EdHJ	0.0000000245434403	0.0000000277228291	0.0000000309022179
trial 8		(mean)	
Compass	0.0000000669525485	0.0000001299486707	0.0000001929447929
NLess	−0.0000003248254630	0.0000009452445159	0.0000022153144948
HJ	0.0000000181001062	0.0000000204491824	0.0000000227982587
EdHJ	0.0000000221218774	0.0000000252822759	0.0000000284426744
trial 9		(mean)	
Compass	0.0000000669525253	0.0000001299486475	0.0000001929447697
NLess	−0.0000003248255413	0.0000009452444376	0.0000022153144166
HJ	0.0000000181000924	0.0000000204491688	0.0000000227982451
EdHJ	0.0000000221218523	0.0000000252822508	0.0000000284426494

10 Source Code

10.1 Source Code for Generating Test Parameters

10.1.1 objective.h

```
/* objective.h declares the necessary objective functions
   and provides a forum for defining reasonable initialStepLength,
   stoppingStepLength, and maxCalls values.
   Default values are defined at the top of PatternSearch.h for
   reference.  Note that specific definitions are highly recommended.
   Liz Dolan
*/

#if !defined _userfile_
#define _userfile_

#include <malloc.h>
#include "rngs.h"
#include "rvgs.h"
#include <iostream.h>
#include <math.h>
#include <stdio.h>

void fcn(int , double *, double & , int & );
void initpt(int , double *&);
double initLength();
extern bool newFunction;
extern bool newInitialPoint;
extern int n;
extern double tolerance;

#define initialStepLength initLength()
#define stoppingStepLength tolerance
#define maxCalls 1000000
#endif
```

10.1.2 objective.cc

```
/* objective.cc implements the minimal required information from the user:
   the dimension of the search, n;
   the objective function, fcn, which returns a flag of success and the double
       objective function value at x;
   and the initial point at which the search should start
   Liz Dolan
*/

#include "objective.h"

int n = 5;
double tolerance = sqrt(fabs(3.0 * (4.0/3.0 -1.0) -1.0));
bool newFunction;
bool newInitialPoint;

double initLength()
{
  static double length;
  if(newInitialPoint)
```

```
    {
      length = Exponential(1.0);
      if (length < 0)
        length *=-1.0;
    }
  return length;
}

void fcn(int vars, double *x, double & f, int & flag)
  /*
    fcn evaluates a convex unimodal quadratic, creating
    a new random quadratic whenever newFunction is true
    upon a call to fcn.  Specify a seed to the random
    number generator via a call to PlantSeeds(long int)
    before calling fcn with newFunction true; to regenerate
    an earlier function, simply plan the earlier seed and
    recall fcn.  Alternatively, one may vary the function
    using the SelectStream(int i: 0<=i<=256) call

    The convext quadratic is built using the following
    formula:
    f() = x_transpose H_transpose H x + C,
    where H is an n+2 by n matrix of normally distributed
    values with a mean of zero and a std dev of one.

    fcn returns a function value of f and signifies a successful
    call by setting flag to one.
  */

{
  static double C;   //C is the constant term
  f = 0; //initialize the return value
  double xtAx = 0;
  static double (**H) = NULL; //the H matrix

  if(newFunction)
    {
      /*apply this code only once to save disk space & time
      FILE * MATRIX;
      FILE * FUNCTION;
      char matrix[252];
      sprintf (matrix, "/home/scratch4/eddola/data4/dir%d/matrix%d",n,n);
      MATRIX = fopen(matrix, "a+");
      sprintf (matrix, "/home/scratch4/eddola/data4/dir%d/func_opt%d", n, n);
      FUNCTION = fopen(matrix, "a+");
      */
      if (H!=NULL)
        {
          for (int i = 0; i < n+2; i++)
            {
              delete (H)[i];
            }
          delete H;
        }

      H = new (double*)[n+2];
      if(H!=NULL)
        {
          for (int i = 0; i < n+2; i++)
            {
              (H)[i] = new double[n];
            }
```

```
          for (int i = 0; i < n+2; i++)
            {
              for (int j = 0; j < n; j++)
                {
                  (H[i])[j] = Normal(0.0, 1.0);
                  //fprintf(MATRIX, "%20.16f\n", (H[i])[j]);
                }
            }
          C = Exponential(1.0);
          flag = 1;
        }
      else
        flag = 0;
      //fprintf(MATRIX, "%19.16f\n\n",C);
      //fprintf(FUNCTION, "%20.16f\n",C);
      //fclose(MATRIX);
      //fclose(FUNCTION);
    }//if we need to allocate the array
else if( H!= NULL )
{
  for(int i = 0; i < n+2; i++)
    {
      for (int j = 0; j < n; j++)
        {
          xtAx += ((H[i])[j] * x[j]); // while not going in the usual direction,
          // this should work because of the commutative property of addition
        }
      f += xtAx * xtAx;
      xtAx = 0;
    }
  f += C;
  flag = 1;
}
else
  flag = 0;
  return;
}

void initpt(int vars, double *&x)
  /* initpt creates a somewhat random point whose components
     are normally distributed about the origin
     I've chosen to reserve stream 0 of the random number
     streams for this purpose
  */
{
  static double * initialPoint = NULL;
  if(newInitialPoint)
    {
      if(initialPoint!=NULL)
        delete initialPoint;
      initialPoint = new double[vars];
      for (int i = 0; i < vars; i++)
        {
          initialPoint[i] = Normal(0.0, 1.0);
        }
    }
  x = new double[vars];
  for (int i = 0; i < vars; i++)
    x[i] = initialPoint[i];
}
```

10.2 Source Code for Pattern Search Base Class

10.2.1 PatternSearch.h

```
/* PatternSearch.h
 * declarations of the PatternSearch base class member
 * functions and data
   Liz Dolan
*/


#ifndef _PatternSearch_
#define _PatternSearch_

#include <iostream.h>
#include <fstream.h>
#include <malloc.h>
#include <stdlib.h>
#include "f2c.h"
#include "vector.h"
#include "objective.h"

//default definitions
//should be defined by user in objective.h
#ifndef maxCalls
#define maxCalls -1
#endif
#ifndef initialStepLength
#define initialStepLength 1.0
#endif
#ifndef stoppingStepLength
#define stoppingStepLength tolerance  //the square root of machine epsilon
#endif

extern int dgetrs_(char *trans, integer *n, integer *nrhs,
        doublereal *a, integer *lda, integer *ipiv, doublereal *b,
        integer * ldb, integer *info);
extern int dgetrf_(integer *m, integer *n, doublereal *a, integer *
        lda, integer *ipiv, integer *info);
extern doublereal dnrm2_(integer *d, doublereal *x, integer *incx);

typedef char file[32];

class PatternSearch
{

public:

 PatternSearch(int dimensions);
 ~PatternSearch();
 virtual void ExploratoryMoves() = 0;
 virtual bool Stop();
 //gives default stopping criteria based on maxCalls and stoppingStepLength
 virtual bool SimpGradient(long int cols, rml_Vector ** pat, FLOP * func_vals);
 //cols==length of the search pattern
 //pat==an array of the pattern with x2-x1, x3-x1,...,xcols-x1
 //func_vals==the values of the objective function as found
 //by f(x2)-f(x1), f(x3)-f(x1),...,f(xcols)-f(x1)
 //finds the simplex gradient per T. Kelley of the pattern matrix
 //and corresponding function values
```

```
 int GetVarNo();  //returns the number of dimensions
 int GetFunctionCalls() { return functionCalls;};
 void GetMinPoint(rml_Vector & minimum);
 //requires and rml_Vector of the correct size
 void GetMinVal(FLOP & value); //best objective function value found thus far
 void GetPattern(rml_Vector ** &pat); //deep copy of the pattern
 void GetPatternLength(int & pattern);
 //returns the number of "columns" of the pattern "matrix"
 virtual FLOP GetStepLength() {return latticeStepLength;};
 virtual void ReplaceMinimum(const rml_Vector & newPoint, FLOP  newValue);
 //replaces the minimizer & the minimum objective function value
 virtual void NextPoint(int index, const rml_Vector &
   currentPoint, rml_Vector & nextPoint);
 //calculates the next prospect point by adding the pattern vector at
 //index to the current vector
 //returns the prospect in nextPoint
 void ReadPatternStandard();
 //input first the pattern length and then the values of each vector
 void ReadPatternFile(ifstream fp, file FILENAME);
 //input first the pattern length and then the values of each vector
 void InitializePattern(int patternSize, rml_Vector * pat[]);
 //deletes any existing pattern and replaces it with the one pointed to by pat
 virtual void ScalePattern( FLOP scalar);
 //scale each pattern vector by scalar
 virtual void fcnCall(int n, double *x, double & f, int & flag);
 //indirection of function call for purposes of keeping an accurate
 //tally of the number of function calls
 void CleanSlate(int dimensions);
 //reinitialize all values

private:

 rml_Vector ** pattern;
 //a pointer to an array of pointers to vector class objects
 int variables;          //dimension of the problem
 rml_Vector * minPoint;
 //the minimizer, should be a pointer to a vector class object
 int patternLength;      //number of "columns" in pattern "matrix"
 FLOP minValue;
 //best objective function value calculated thus far
 FLOP latticeStepLength; //density of underlying lattice
 long functionCalls;     //tally of the number of function calls
};

#endif
```

10.2.2 PatternSearch.cc

```
/* PatternSearch.cc
   Liz Dolan
*/

#include "PatternSearch.h"

PatternSearch::PatternSearch(int dimensions)
{
  variables = dimensions;
  patternLength = 0;
  functionCalls = 0;
  pattern = NULL;
  //need to initialize pattern and evaluationOrder
```

```
  FLOP * startPoint = NULL;
  initpt(variables, startPoint);
  minPoint = new rml_Vector(startPoint, variables);
  int flag = 1;
  latticeStepLength = initialStepLength;
  fcn(variables, startPoint, minValue, flag);
}//constructor

PatternSearch::~PatternSearch()
{
  for(int j = 0; j < patternLength; j++)
    {
      delete pattern[j];
    }//for
  delete pattern;
  pattern = NULL;
  delete minPoint;
  minPoint = NULL;
}//destructor

void PatternSearch::CleanSlate(int dimensions)
//reinitialize all values
{
  variables = dimensions;
  functionCalls = 0;
  for (int j=0; j<patternLength; j++)
    delete pattern[j];
  delete pattern;
  patternLength = 0;
  delete minPoint;
  pattern = NULL;
  minPoint = NULL;
  //need to initialize pattern and evaluationOrder
  FLOP * startPoint = NULL;
  initpt(variables, startPoint);
  minPoint = new rml_Vector(startPoint, variables);
  int flag = 1;
  latticeStepLength = initialStepLength;
  fcn(variables, startPoint, minValue, flag);
}

bool PatternSearch::Stop()
{
  if(maxCalls>-1)
    if(functionCalls >= maxCalls)
      return true;
  if(patternLength > 0)
    {
      if (latticeStepLength <= stoppingStepLength)
        return true;
      else
        {
          return false;
        }
    }//
  return true;
  //as a default, if there is no pattern, stop
}//Stop

bool PatternSearch::SimpGradient(long int cols, rml_Vector* simpat[],
                                 FLOP * func_vals)
```

```
  /* SimpGradient finds the simplex gradient a la T. Kelley
     for comparison as a stopping criterion with the function
     variance (as is used in classic Nelder-Mead) and the
     lattice step length.  Gradient information is in no
     way neccessary to the Pattern Search; this function
     exists in this version for testing purposes only.

     The simplex gradient as proposed by T. Kelley:
     D(f:S) = V^-T d(f:S),
     where d(f:S) =
     (f(x_2) - f(x_1) ; f(x_3) - f(x_1) ;...; f(x_N+1) - f(x_1))
     where x_1,...x_N+1 denote the N+1 vertices of a
     N-dimensional simplex S and
     V(S)= (x_2-x_1, x_3-x_1, ..., x_N+1-x_1).

     Source: D.M. Bortz and C.T. Kelley.  "The Simplex
     Gradient and Noisy Optimization Problems."
     Computational Methods for Optimal Design and
     Control. Borggaard, Burns, Cliff, and Schreck.
     Birkhaeuser, Boston. 1998.

     I keep the term simplex gradient here even though I use the
     function for pattern of more than N+1 points.
  */
{
  long int M = n; //dimensions of the search space
  long int LDA = M;
  long int * Mpoint = &M;
  long int * Npoint = &cols;
  long int * LDApoint = &LDA;
  long int ipiv[n]; //will return the pivot indices
  long int flag ;  //returned 0 for success, -i for an illegal ith arg
  int nice = -1; //value returned by clapack calls
  char tran[1];  //value is N for no transpose, T for transpose, or
                 //C for conjugate transpose
  tran[0] = 'T';
  long int oneholder = 1;
  long int * one = &oneholder;
  FLOP * sneaky;
  FLOP * pat;
  pat = new FLOP[M*cols];

  /*note that the matrix should be passed by column and
    not by row as is the standard in C++ because the
    functions dgetrf and dgetrs are converted from
    FORTRAN, specifically from functions to find the
    LU factorization of a matrix and solve Ax=b for x
    from LAPACK
  */
  for(int count =0; count < cols; count++)
    {
      sneaky = (*simpat[count]).passPointer();
      for (int countess = 0; countess < M; countess++)
        {
          pat[(count*M)+countess] = sneaky[countess];
        }
    }
  nice = dgetrf_(Mpoint, Npoint, pat, LDApoint, ipiv, &flag);
  if((flag)==0)
    {
      nice = dgetrs_(tran, Mpoint, one, pat, LDApoint,
              ipiv, func_vals, Mpoint, &flag);
```

```
    }
  delete pat;
  pat = NULL;
  bool success;
  if ((flag)==0)
    success = true;
  else
    success = false;
  return true;
}//SimpGradient

int PatternSearch::GetVarNo()
{
  return variables;
}//GetVarNo

void PatternSearch::GetMinPoint( rml_Vector & minimum)
{
  minimum.copy(*minPoint);
}//GetMinPoint

void PatternSearch::GetMinVal( FLOP & value)
{
  value = minValue;
}//GetMinVal

void PatternSearch::GetPattern(rml_Vector ** &pat)
  //user should be able to pass just a pointer
{
  pat = new (rml_Vector*)[patternLength];
  for(int i = 0; i < patternLength && pattern[i]!=NULL ; i++)
    {
      pat[i] = new rml_Vector(variables);
      (*pat[i]).copy ( (*pattern[i]) );
    }//for

}//GetPattern

void PatternSearch::GetPatternLength(int & pattern)
{
  pattern = patternLength;
}//GetPatternLength

void PatternSearch::ReplaceMinimum(const rml_Vector & newPoint, FLOP newValue)
{
  (*minPoint).copy( newPoint );
  minValue = newValue;
}//UpdateMinPoint

void PatternSearch::NextPoint( int index, const rml_Vector &
currentPoint, rml_Vector & nextPoint )
{
  //To get the next point I add the currentPoint to the pattern vector at index
  if (pattern != NULL && patternLength > index)
    sum(nextPoint,currentPoint, *pattern[index]);
}

void PatternSearch::ReadPatternStandard()
{
  FLOP * buffer;
  if(patternLength != 0)
    {
```

```
      for(int j = 0; j < patternLength; j++)
        {
          delete pattern[j];
        }//for
      delete [] pattern;
      pattern = NULL;
    }//if

  cin >> patternLength;
  typedef rml_Vector* pointer;
  pattern = new pointer[patternLength];

  for(int i = 0; i < patternLength; i++)
    {
      buffer = new FLOP[variables];
      for(int j = 0; j < variables; j++)
        {
          cin >> buffer[j];
        }//inner for
        pattern[i] = new rml_Vector(buffer, variables);
    }//outer for
}//ReadPatternStandard

void PatternSearch::ReadPatternFile(ifstream fp, file FILENAME)
{
  if(patternLength != 0)
    {
      for(int j = 0; j < patternLength; j++)
        {
          delete pattern[j];
        }//for
      delete [] pattern;
      pattern = NULL;
    }//if
  if (fp==NULL) return;
  FLOP * buffer;
  fp >> patternLength;  //the length of the pattern must precede the pattern
  typedef rml_Vector* pointer;
  pattern = new pointer[patternLength];
  for(int i = 0; i < patternLength; i++)
    {
      buffer = new FLOP[variables];
      for(int j = 0; j < variables; j++)
        {
          fp >> buffer[j];
        }//inner for
      pattern[i] = new rml_Vector(buffer, variables);
    }//outer for
}//ReadPatternFile

void PatternSearch::InitializePattern(int patternSize, rml_Vector *pat[])
{
  if(patternLength != 0)
    {
      for(int j = 0; j < patternLength; j++)
        {
          delete pattern[j];
        }//for
      delete [] pattern;
      pattern = NULL;
    }//if
  typedef rml_Vector* pointer;
```

```
  pattern = new pointer[patternSize];
  patternLength = patternSize;
  for(int j = 0; j < patternLength; j++)
    {
      pattern[j] = new rml_Vector(variables);
      (*(pattern[j])).copy(*(pat[j]));
    }//for
}//InitializePattern

void PatternSearch::ScalePattern(FLOP scalar)
{
  if(pattern != NULL)
    {
      for(int i = 0; i < patternLength && pattern[i]!=NULL; i++)
        {
          (*pattern[i]) *= scalar;
        }//for
      latticeStepLength *= scalar;
    }//if
}//ScalePattern

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

10.3 Source Code for Derived Classes

10.3.1 CompassSearch.h

```
/*CompassSearch.h
  header file for a search based on the class PatternSearch
  Liz Dolan
*/
#ifndef _CompassSearch_
#define _CompassSearch_
#include "PatternSearch.h"
#include "vector.h"
class CompassSearch: public PatternSearch
{
  public :
  CompassSearch(int numberOfVariables);
  //the dimensions of the search space are required for the constructor
  ~CompassSearch();
  void ExploratoryMoves();
  //searches in compass directions for a better solution to the objective function

  private:
  void CreatePattern(FLOP);
  //initializes the pattern to one that contains one positive
  //and one negative element in each of the compass directions
  void UpdatePattern();
  //scales the pattern to search half as far in each direction
};//class CompassSearch

#endif
```

10.3.2 CompassSearch.cc

```
/* CompassSearch.cc
   implementation of the CompassSearch class to find a minimal objective function
   solution.
   Liz Dolan
*/
#include "CompassSearch.h"
CompassSearch::CompassSearch(int numberOfVariables):
PatternSearch(numberOfVariables)
{
}

CompassSearch::~CompassSearch()
{
}

void CompassSearch::ExploratoryMoves()
{
  FILE * gradient;
  char gradfile[252];
  sprintf (gradfile, "/scratch/eddola/dir%d/gradient%d",n,n);
  gradient = fopen(gradfile, "a+");
  FILE * VARIANCE;
  sprintf (gradfile, "/scratch/eddola/dir%d/variance%d",n,n);
  //  this file will become ungainly very quickly, I recommend writing to
  //  it only for small numbers of trials i.e. 100-
  //  VARIANCE = fopen(gradfile, "a+");

  FLOP pace = 0.0;
  pace = GetStepLength();
  CreatePattern(pace);
  rml_Vector currentPoint(GetVarNo());
  rml_Vector nextPoint(GetVarNo());
  FLOP value;
  FLOP nextValue;
  int length;
  int success = 0;
  GetMinPoint(currentPoint);
  GetMinVal(value);
  GetPatternLength(length);
  FLOP * knowns;
  FLOP * sendknowns;
  rml_Vector ** getpat;
  rml_Vector ** simple;
  FLOP simplexGradient = -1.0;
  FLOP minSimGrad = -1.0;
  FLOP minVariance = -1.0;
  knowns = new FLOP[length];
  sendknowns = new FLOP[length-1];
  long int incat = 1;
  long int * inc = &incat;
  long int simDimens = GetVarNo() - 1;
  double diff, mean, summer, variance;
  //used in calculating variance
  FLOP curDelta = -1; //lattice size
  do
    {
      for( int i = 0; i < length; i++)
        {
          NextPoint(i, currentPoint, nextPoint);
          FLOP * tried;
```

```
    tried = nextPoint.passPointer();

    fcnCall(GetVarNo(), tried, nextValue, success);
    knowns[i] = nextValue;
    if(success==1)
      {
        if(nextValue < value)
          {
            ReplaceMinimum(nextPoint, nextValue);
            value = nextValue;
            currentPoint.copy(nextPoint);
            i = -1; //start the compass search over at the new point
          }//if there is improvement

      }//if able to get a function value
  }//for now we know that there aren't better points around this one
UpdatePattern();
GetPattern(getpat);
for(int simp = 0; simp < length; simp++)
  {
    sum((*(getpat[simp])), (*(getpat[simp])), currentPoint);
  }

mean = 0.0;
summer = 0.0;
for (int simp = 0; simp < length; simp++) {
  /* Welford's one-pass method     */
  /* to calculate the sample mean  */
  /* and variance                  */

  diff  = knowns[simp] - mean;
  summer  += diff * diff * (simp) / (simp + 1.0);
  mean += diff / (simp + 1.0);
}
variance  = (summer / length);

if(minVariance < 0)
  minVariance = variance;
if(variance < minVariance)
  minVariance = variance;
curDelta = GetStepLength();
for(int simp = 1; simp < length; simp++)
  {
    sum((*(getpat[simp])), 1.00, (*(getpat[simp])), -1.00, (*(getpat[0])));
    sendknowns[simp-1] = knowns[simp] - knowns[0];
  }
simple = (&(getpat[1]));
if(SimpGradient(length-1, simple, sendknowns))
  {
    if(minSimGrad == -1 )
      minSimGrad = dnrm2_(&simDimens, sendknowns, inc);
    simplexGradient = dnrm2_(&simDimens, sendknowns, inc);
    if( simplexGradient < minSimGrad)
      minSimGrad = simplexGradient;
  }
simple = NULL;
for(int simp = 0; simp < length; simp++)
  {
    delete getpat[simp];
  }
delete getpat;
getpat = NULL;
```

```
    }while(!Stop());//while we haven't stopped()

  delete knowns;
  delete sendknowns;
  fprintf(gradient, "%20.16f    %20.16f   %20.16f\n", minSimGrad,
          simplexGradient, (1.0+minSimGrad) - (1.0 + simplexGradient));
  fclose(gradient);
}//ExploratoryMoves

void CompassSearch::CreatePattern(FLOP initialStepSize)
{
  typedef rml_Vector * pointer;
  int vars = GetVarNo();
  rml_Vector ** compassPattern;
  compassPattern = new pointer[2*vars];
  FLOP * temp;
  if(vars > 0)
    {
      for(int j = 0; j < vars; j++)
        {
          temp = new FLOP[vars];
          for(int i = 0; i < vars; i++)
            {
              temp[i] = 0;
            }//1st inner for
          temp[j] = initialStepSize;
          compassPattern[2*j] = new rml_Vector(temp, vars);
          temp = new FLOP[vars];
          for(int i = 0; i < vars; i++)
            {
              temp[i] = 0;
            }//2nd inner for
          temp[j] = -initialStepSize;
          compassPattern[2*j+1] = new rml_Vector(temp, vars);
        }//for
      InitializePattern(2*vars, compassPattern);
      for(int j=0; j<2*vars; j++)
        delete compassPattern[j];
      delete compassPattern;
    }//if there's anything to allocate
}//CreatePattern

void CompassSearch::UpdatePattern()
{
  ScalePattern(0.5);
}//UpdatePattern
```

10.3.3 NLessSearch.h

```
/*NLessSearch.h
  header file for a regular simplex search based on
  the PatternSearch class
  by: Liz Dolan
*/
#ifndef _NLessSearch_
#define _NLessSearch_
#include "PatternSearch.h"
#include "vector.h"
class NLessSearch: public PatternSearch
{
  public :
```

```
  NLessSearch(int numberOfVariables);
  //constructor requires only the dimensionality of the search space
  ~NLessSearch();
  void ExploratoryMoves();

  private:
  void CreatePattern(FLOP);
  void UpdatePattern();
  void SizePattern(int dimens, rml_Vector *pat[], FLOP size);
};//class NLessSearch
#endif
```

10.3.4 NLessSearch.cc

```
/* NLessSearch.cc
   implementations of the derived class NLessSearch functions
   Liz Dolan
*/
#include "NLessSearch.h"

NLessSearch::NLessSearch(int numberOfVariables): PatternSearch(numberOfVariables)
{
}

NLessSearch::~NLessSearch()
{
}

void NLessSearch::ExploratoryMoves()
{
  FLOP pace = 0.0;
  pace = GetStepLength();  //as given by the user or default
  CreatePattern(pace);     //creates a regular simplex
  rml_Vector currentPoint(GetVarNo());
  rml_Vector nextPoint(GetVarNo());
  FLOP value;
  FLOP nextValue;
  int length;
  int success = 0;
  FLOP * vectorArray;
  //vectorArray points to the array of values
  //in the rml_Vector class object
  GetMinPoint(currentPoint);  //initialize min point
  GetMinVal(value);           //and obj. func. value
  GetPatternLength(length);
  //search the pattern in each direction until improvement is found,
  //then stop and begin a new iteration at the better point
  do
    {

      for( int i = 0; i < length; i++)
        {
          NextPoint(i, currentPoint, nextPoint);
          vectorArray = nextPoint.passPointer();
          //  cout << vectorArray[0] << "  " << vectorArray[1] << endl;
          fcnCall(GetVarNo(), vectorArray, nextValue, success);
          if(success==1)
            {
              if(nextValue < value)
                {
                  ReplaceMinimum(nextPoint, nextValue);
```

```
                  value = nextValue;
                  currentPoint.copy(nextPoint);
                  i = -1; //start the search over at the new point
                }//if there is improvement
            }//if able to get a function value
        }//for now we know that there aren't better points around this one
        UpdatePattern();
    }while(!Stop());//while we haven't stopped()
}//ExploratoryMoves

void NLessSearch::CreatePattern(FLOP initialStepSize)
/*
  For information on how to build a regular
  simplex, see pages 79-81 S.L.S. Jacoby, J.S. Kowalik
  and J.T. Pizzo.
  Iterative Methods for Nonlinear Optimization Problems.
  Prentice Hall, Inc., Englewood Cliffs, NJ. 1972.
*/
{
  typedef rml_Vector * pointer;
  int vars = GetVarNo();
  FLOP * min;
  //refer to book for p and q
  FLOP p = (sqrt(vars+1) -1 + vars)*initialStepSize/(vars*sqrt(2));
  FLOP q = (sqrt(vars+1) -1)*initialStepSize/(vars*sqrt(2));
  rml_Vector ** nlessPattern; //pointer to the pattern being made
  nlessPattern = new pointer[vars+1];  //# of vectors in pattern
  FLOP * temp;
  FLOP * basis;
  //basis is the first point used to create a simplex according to the algorithm
  rml_Vector center(vars);
  //initialize temp's array and build a rml_Vector about it
  if(vars > 0)
    {
      basis = new FLOP[vars];
      temp = new FLOP[vars];
      GetMinPoint(center);  //will be the initial point given by user
      min = center.passPointer();  //array of values
      //calculate the location of the basis according to the
      //given value of the centroid
      //temp is the direction of the vertex created from min
      for(int j = 0; j < vars; j++)
        {
          basis[j] = (min[j] - (p/(vars+1)) - ((vars-1)*q/(vars+1)));
          temp[j] = basis[j] - min[j];
        }
      nlessPattern[0] = new rml_Vector(temp, vars);
      temp = new FLOP[vars];
      for(int i = 1; i < vars + 1; i++)
        {
          for(int k = 0; k < vars; k++)
            {
              temp[k] = basis[k] + q - center[k];
            }
          temp[i-1] = basis[i-1] + p - center[i-1];
          nlessPattern[i] = new rml_Vector(temp, vars);
          temp = new FLOP[vars];
        }//outer for
      delete basis;
      delete temp;
      temp = NULL;
      basis = NULL;
```

```
      //make sure that the vectors of the pattern not only point
      //in the right direction, but are also of desired length
      SizePattern(vars, nlessPattern, initialStepSize);
      InitializePattern(vars + 1, nlessPattern);
      for(int j=0; j<vars+1; j++)
        delete nlessPattern[j];
      delete nlessPattern;
    }//if there's anything to allocate
}//CreatePattern

void NLessSearch::UpdatePattern()
{
  ScalePattern(0.5);
}//UpdatePattern

void NLessSearch::SizePattern(int dimens, rml_Vector *pat[], FLOP size)
{
  FLOP squares[dimens];
  //used to find the squares of the distances in each dimension
  rml_Vector * compare;
  rml_Vector * compareTo;
  FLOP *compareArray;
  FLOP *compareToArray;
  double compDist; //distance from the centroid
  int length = dimens+1;  //length of the pattern
  compareTo = new rml_Vector(dimens);
  (*compareTo).copy(*(pat[0])); //initialize for testing
  compareToArray = (*compareTo).passPointer();
  for(int i = 0;i < length;i++)
    {
      compare = new rml_Vector(dimens);
      (*compare).copy(*(pat[i]));
      compareArray = (*compare).passPointer();
      compDist = 0;
      //sum the squares and take square root to find the distance
      for(int find = 0; find < dimens; find++)
        {
          squares[find] = (compareArray[find]*compareArray[find]);
          compDist += squares[find];
        }
      compDist = pow(compDist, 0.5);
      /*
        calculate the angles and assure that they are equal
        between compareTo and compare for each compare n.e.
        compareTo
        see page 106 of Gilbert Strang, Linear Algebra and its
        Applications

      double angle = 0;
      for(int calc = 0;calc<dimens;calc++)
        {
          angle += (compareArray[calc] * compareToArray[calc]);
        }
      angle = angle/(compDist*compDist);
      angle = acos(angle);
      */
      //resize to desired length, based on knowledge of
      //relationships in the ratios of similar triangles
      for(int j = 0;j<dimens;j++)
        {
          compareArray[j] = size*compareArray[j]/compDist;
        }
```

```
      compDist = 0;

      delete pat[i];
      pat[i] = compare;

    }//for
delete compareTo;
}
```

10.3.5 HJSearch.h

```
/* HJSearch.h
   header file for a Hooke and Jeeves search based
                     on the class PatternSearch
   For a good description of the Hooke and Jeeves search algorithm
   I recommend Non-Linear Optimization Techniques by Box, Davies,
   and Swann, 1969.
  Liz Dolan
*/
#ifndef _HJSearch_
#define _HJSearch_
#include "PatternSearch.h"
#include "vector.h"

class HJSearch: public PatternSearch
{
  public :
  HJSearch(int numberOfVariables);
  //the dimensions of the search space are required for the constructor
  ~HJSearch();
  void ExploratoryMoves();
  //searches in Hooke and Jeeves pattern for best objective function solution
  bool Stop(FLOP pace);
  //makes certain search has not exceeded maxCalls or
  //is stepping at less than stoppingStepLength
   FLOP GetStepLength();
  //overrides the PatternSearch version with an acurate length
   void CleanSlate(int dimensions);
  //reinitialize all values

  private:
  FLOP step;
};//class HJSearch
#endif
```

10.3.6 HJSearch.cc

```
/* HJSearch.cc
   implementation of the HJSearch class to find a minimal objective function
   solution.
   For a good description of the Hooke and Jeeves search algorithm
   I recommend Non-Linear Optimization Techniques by Box, Davies,
   and Swann, 1969.
   Liz Dolan
*/
#include "HJSearch.h"

HJSearch::HJSearch(int numberOfVariables): PatternSearch(numberOfVariables)
{
```

```
  step = initialStepLength;  //as provided by the user or default
}

HJSearch::~HJSearch()
{
}

void HJSearch::CleanSlate(int dimensions)
//reinitialize all values
{
  PatternSearch::CleanSlate(dimensions);
  step = initialStepLength;
}

void HJSearch::ExploratoryMoves()
{
  int dimens = GetVarNo();
  rml_Vector currentPoint(dimens);
  rml_Vector lastImprovingPoint(dimens); //last base point
  rml_Vector storage(dimens);
  //for intermediate storage to reduce rounding error
  rml_Vector direction(dimens); //direction of pattern extended step
  FLOP value;                   //objective function value
  FLOP positiveValue;           //obj.fun. value in the positive step
  FLOP negativeValue;           //" for the negative step
  FLOP lastImprovingValue;      //obj.fun.value of last base point
  FLOP *vectorArray = currentPoint.passPointer();
  //vectorArray is a pointer to the array of values inside the
  //currentPoint rml_Vector class object
  int success = 0;
  GetMinPoint(currentPoint);    //initialize to the user initial point
  GetMinPoint(lastImprovingPoint);
  GetMinPoint(storage);
  GetMinVal(value);
  GetMinVal(lastImprovingValue);
  bool foundImprove = false;
  diff(direction, currentPoint, currentPoint);
  //initialize directions to a vector of zeros (or within rounding error of zero)
  do
    {
      for(int iteration=0;iteration < dimens;iteration++)
        {
          vectorArray[iteration] += step;
          fcnCall(GetVarNo(), vectorArray, positiveValue, success);
          if(success!=1)
            {
              positiveValue = value + 1.0;
              //if the call returned unsuccessfully, set positiveValue
              //to a value that will not be improving
            }
          if(positiveValue < value)
            {
              value = positiveValue;
              foundImprove = true;
              //continue search in other dimensions from here
            }//if positive is better
          if(!foundImprove)
            {
              currentPoint.copy(storage);
              vectorArray[iteration] -= (step);
              fcnCall(GetVarNo(), vectorArray, negativeValue, success);
              if(success!=1)
```

```
                {
                  negativeValue = value + 1.0;
                  //same kludge as in positive case
                }
              if(negativeValue < value)
                {
                  value = negativeValue;
                  foundImprove = true;
                  //continue search in other dimensions from here
                }//if negative direction is better
            }//if we need to check the negative
          if(!foundImprove)
            {//reset to original position
              currentPoint.copy(storage);
            }//if neither direction gave improvement
          else
            {
              storage.copy(currentPoint);
            }
          foundImprove = false;      //reset for next iteration
        }//for
      diff(direction, currentPoint,lastImprovingPoint);
      //direction now holds the extended pattern step vector
      if(value < lastImprovingValue)
        {
          //check whether the "new" point is within .5*step of the old
          if(lastImprovingPoint.isNear(currentPoint, step))
            {
              currentPoint.copy(lastImprovingPoint);
              storage.copy(currentPoint);
              fcnCall(dimens, vectorArray, value, success);
            }
          else   //some step yielded improvement
            {
              lastImprovingValue = value;
              ReplaceMinimum(currentPoint, value);
              lastImprovingPoint.copy(currentPoint);
              //take the pattern extending step and find its value
              sum(currentPoint,direction,currentPoint);
              storage.copy(currentPoint);
              fcnCall(dimens, vectorArray, value, success);
            }
        }
      else
        {
          if(currentPoint.isNear(lastImprovingPoint, step))
            {
              step = step / 2.0;
            }
          else
            {
              //this case can only occur after an unsuccessful
              //search about a pattern-step-located point
              //move back to the point that was improving from the
              //search about the last base point
              GetMinPoint(currentPoint);
              storage.copy(currentPoint);
              fcnCall(GetVarNo(), vectorArray, value, success);
            }
        }
    } while(!Stop(step));//while we haven't stopped()
}//ExploratoryMoves
```

```
bool HJSearch::Stop(FLOP pace)
//makes certain search has not exceeded maxCalls or
//is stepping at less than stoppingStepLength
{
  if(maxCalls > -1)
    if(GetFunctionCalls() > maxCalls)
      return true;
  if(pace < stoppingStepLength)
    return true;
  return false;
}//Stop

FLOP HJSearch::GetStepLength()
{
  return step;
}
```

10.3.7 EdHJSearch.h

```
/* EdHJSearch.h
   header file for a variation on the Hooke and Jeeves search based
                        on the class PatternSearch
   For a good description of the Hooke and Jeeves search algorithm
   I recommend Non-Linear Optimization Techniques by Box, Davies,
   and Swann, 1969.
  Liz Dolan
*/
#ifndef _EdHJSearch_
#define _EdHJSearch_
#include "PatternSearch.h"
#include "vector.h"

class EdHJSearch: public PatternSearch
{
  public :
  EdHJSearch(int numberOfVariables);
  //the dimensions of the search space are required for the constructor
  ~EdHJSearch();
  void ExploratoryMoves();
  //searches in Hooke and Jeeves pattern for best objective function solution
  bool Stop(FLOP pace);
  //makes certain search has not exceeded maxCalls or
  //is stepping at less than stoppingStepLength
  FLOP GetStepLength();
  //overrides the PatternSearch version with an acurate length
  void CleanSlate(int dimensions);
  //reinitialize all values

  private:
  FLOP step;
};//class EdHJSearch
#endif
```

10.3.8 EdHJSearch.cc

```
/* EdHJSearch.cc
   implementation of the HJSearch class to find a minimal objective function
   solution.
```

```
   Includes a minor modification to the basic Hooke and Jeeves strategy to
   avoid making pattern steps directly after contractions (which mostly cover
   the same ground that was already covered in the search step preceding
   the contraction.
   Liz Dolan
*/
#include "EdHJSearch.h"

EdHJSearch::EdHJSearch(int numberOfVariables): PatternSearch(numberOfVariables)
{
  step = initialStepLength;
}

EdHJSearch::~EdHJSearch()
{
}

void EdHJSearch::CleanSlate(int dimensions)
//reinitialize all values
{
  PatternSearch::CleanSlate(dimensions);
  step = initialStepLength;
}

void EdHJSearch::ExploratoryMoves()
{
  int dimens = GetVarNo();
  rml_Vector currentPoint(dimens);
  rml_Vector lastImprovingPoint(dimens); //last base point
  rml_Vector storage(dimens);
  //for intermediate storage to reduce rounding error
  rml_Vector direction(dimens); //direction of pattern extended step
  FLOP value;                   //objective function value
  FLOP positiveValue;           //obj.fun. value in the positive step
  FLOP negativeValue;           //" for the negative step
  FLOP lastImprovingValue;      //obj.fun.value of last base point
  FLOP *vectorArray = currentPoint.passPointer();
  //vectorArray is a pointer to the array of values inside the
  //currentPoint rml_Vector class object
  int success = 0;
  GetMinPoint(currentPoint);    //initialize to the user initial point
  GetMinPoint(lastImprovingPoint);
  GetMinPoint(storage);
  GetMinVal(value);
  GetMinVal(lastImprovingValue);
  bool foundImprove = false;
  bool contracted = false;
  diff(direction, currentPoint, currentPoint);
  //initialize to a vector of zeros (or within rounding error of zero)
  do
    {
      for(int iteration=0;iteration < dimens; iteration++)
        {
          vectorArray[iteration] += step;
          fcnCall(GetVarNo(), vectorArray, positiveValue, success);

          if(success!=1)
            {
              positiveValue = value + 1.0;
              //if the call returned unsuccessfully, set positiveValue
              //to a value that will not be improving
            }
```

```
    if(positiveValue < value)
      {
        value = positiveValue;
        foundImprove = true;
        //continue search in other dimensions from here
      }//if positive is better
    if(!foundImprove)
      {
        currentPoint.copy(storage);
        vectorArray[iteration] -= (step);
        fcnCall(GetVarNo(), vectorArray, negativeValue, success);
        if(success!=1)
          {
            negativeValue = value + 1.0;
            //same kludge as in positive case
          }
        if(negativeValue < value)
          {
            value = negativeValue;
            foundImprove = true;
            //continue search in other dimensions from here
          }//if negative direction is better
      }//if we need to check the negative
    if(!foundImprove)
      {//reset to original position
        currentPoint.copy(storage);
      }//if neither direction gave improvement
    else
      {
        storage.copy(currentPoint);
      }
    foundImprove = false; //reset for next iteration
  }//for
diff(direction, currentPoint,lastImprovingPoint);
//direction now holds the extended pattern step vector
if(value < lastImprovingValue)
  {
    //check whether the "new" point is within .5*step of the old
    if(lastImprovingPoint.isNear(currentPoint, step))
      {
        currentPoint.copy(lastImprovingPoint);
        storage.copy(currentPoint);
        fcnCall(dimens, vectorArray, value, success);
      }
    else
      {
        lastImprovingValue = value;
        ReplaceMinimum(currentPoint, value);
        lastImprovingPoint.copy(currentPoint);
        if(!contracted)   //my personal modification to the algorithm
          {
            //take the pattern extending step and find its value
            sum(currentPoint,direction,currentPoint);
            storage.copy(currentPoint);
            fcnCall(dimens, vectorArray, value, success);
          }
      }
    contracted = false;
  }
else
  {
    if(currentPoint.isNear(lastImprovingPoint, step))
```

```
            {
              step = step / 2.0;
              contracted = true;
            }
          else
            {
              //this case can only occur after an unsuccessful
              //search about a pattern-step-located point
              //move back to the point that was improving from the
              //search about the last base point
              GetMinPoint(currentPoint);
              storage.copy(currentPoint);
              fcnCall(GetVarNo(), vectorArray, value, success);
              contracted = false;
            }
        }
    }while(!Stop(step));//while we haven't stopped()
}//ExploratoryMoves

bool EdHJSearch::Stop(FLOP pace)
//makes certain search has not exceeded maxCalls or
//is stepping at less than stoppingStepLength
{
  if(maxCalls > -1)
    if(GetFunctionCalls() > maxCalls)
      return true;
  if(pace < stoppingStepLength)
    return true;
  return false;
}//Stop

FLOP EdHJSearch::GetStepLength()
{
  return step;
}
```