Model-Assisted Pattern Search

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
Christopher M. Siefert

Accepted for: ______________________________

Thesis Advisor: ______________________________
Virginia J. Torczon

____
Michael W. Trosset

____
Stephen K. Park

Abstract

Computer simulations of complex physical phenomena are used in many contexts, including that of engineering design. Increasingly scientists and engineers have also been trying to optimize problems defined by such simulations (e.g. to determine design parameters for a physical product). However, these problems often have several features that hinder the use of standard optimization techniques. The lack of derivative information and numerical error induced by the simulation can cause problems for derivative-based optimization methods. Likewise, extreme computational expense can make the use of direct search methods problematic.

The Model-Assisted Pattern Search (MAPS) algorithm, which is the subject of this research, attempts to address the issue. While maintaining a pattern search framework, MAPS makes use of easily constructed surrogates to the objective function in order to speed the optimization process. Numerical results for MAPS and several other algorithms are presented here for a variety of different objective functions.

Acknowledgments

This work has been made possible through the support of various organizations[1] and individuals. First and foremost, I would like to thank Virginia Torczon for all of her wonderful help and sage advice. Throughout all of the problems and set-backs, she was always willing to encourage me and push me onward. She knew I could finish this, even when I had my doubts. Thank you very much for everything. I would also like to thank Michael Trosset for helping me to extract something meaningful from the mountains of data, and for his meticulous attention to detail. Though I may have complained at times about his suggestions, I freely admit that they made this research much better.

This work was built on the work of other students, whose contributions have made my work much easier. Elizabeth Dolan's pattern search implementations formed the basis for many of my experiments. Without Anthony Padula's implementation of the krigifier, I would have had many fewer functions on which to test MAPS. I would also like to thank Adam Gurson and Erin Parker for their help throughout this project.

Several other individuals also came to my assistance, and I would like to thank them as well. Steve Park helped me to clearly explain my plots. Juan Meza and Patricia Hough allowed me access to the T-WAFER problem to work on, which has served as an illustrative example time and time again. Special thanks go to my two favorite proofreaders Anne Shepherd and Joseph Carnahan. Anne was always happy to read this work, even when no one else had time. Joe sat through presentation after presentation, and probably knows the work as well as I do. He has offered moral support that goes beyond words, and I would like to thank him from the bottom of my heart.

When it comes to moral support, I would also like to thank my parents and the campus ministers at William and Mary. My mother and father are always there for me, and were always a source of encouragement for me. Without their loving help, I would not be writing this document. Likewise, our campus ministers Fr. Patrick Golden and MaryEllen Pitard were always available to help me through tough times. My gratitude for their support is great.

Most importantly, thanks be to God for everything, especially these wonderful people who have helped me at every step of the way. I count myself blessed to have such wonderful friends and co-workers.

Vivat Jesus!

[1] This research was supported by NSF Grant CCR-9734044, and a Batten Scholarship.

Contents

1 Introduction **7**
1.1 Motivation . . . 7
1.2 The Basic Concept of MAPS . . . 8
1.3 Issues to be Addressed . . . 8

2 Algorithmic Framework **10**
2.1 Two Illustrative Examples . . . 10
2.2 The MAPS Algorithm . . . 16
2.3 Numerical Optimization . . . 19
2.3.1 Algorithmic Considerations . . . 19
2.3.2 Computational Considerations . . . 20
2.3.3 Practical Considerations . . . 22
2.4 Parameter Estimation . . . 23
2.4.1 Mathematical Considerations . . . 23
2.4.2 Computational Considerations . . . 24
2.4.3 Practical Considerations . . . 24

3 Testing **26**
3.1 Common test functions in the literature . . . 27
3.2 Real applications . . . 28
3.3 Krigifier-generated functions . . . 28

4 Summary of Results **30**
4.1 MAPS and Optimization . . . 30
4.1.1 The Reliability of MAPS . . . 31
4.1.2 MAPS vs. One-shot Optimization . . . 32
4.1.3 The Advantage of Modeling . . . 33
4.2 MAPS and Global Optimization . . . 38
4.2.1 Global vs. Local Minimization . . . 38
4.2.2 The MAPS Search Criterion . . . 40
4.2.3 Global Optimization and the "Curse of Dimensionality" . . . 43
4.2.4 Costs of the MAPS Approach . . . 43

5 Conclusions and Future Research **45**

A Graphs of Results **50**

B Global Comparisons with DACE **59**

C "Nearness" to the Global Minimizer **61**

D Source Code for Approximation **64**
D.1 Headers 64
D.1.1 Approximate 64
D.1.2 KrigApprox 65
D.1.3 ParamEstimate 68
D.2 Code 74
D.2.1 Approximate 74
D.2.2 KrigApprox 75
D.2.3 ParamEstimate 82

E Source Code for Optimization **100**
E.1 Headers 100
E.1.1 MAPS 100
E.1.2 SearchCriterion 104
E.1.3 InitialDesign 109
E.1.4 LatinHCDesign 110
E.1.5 PSGrid 111
E.2 Code 115
E.2.1 MAPS 115
E.2.2 SearchCriterion 133
E.2.3 InitialDesign 144
E.2.4 LatinHCDesign 146
E.2.5 PSGrid 147

List of Figures

1.1 The 2-D T-WAFER problem. 8

2.1 Latin hypercube design. 11
2.2 Latin hypercube design on $f(x)$. 11
2.3 Approximation $\hat{f}_0$, 10 evaluations. 12
2.4 Approximation $\hat{f}_0$, superimposed on T-WAFER. 13
2.5 Search criterion S_0, 10 evaluations. 13
2.6 Approximation $\hat{f}_0$, with x_1 and original design sites shown. 14
2.7 New approximation $\hat{f}_1$, superimposed on T-WAFER. 14
2.8 Minimizers located by compass search. 15
2.9 Compass search minimizers along the valley. 15
2.10 Goldstein-Price function, with minimizers. 16
2.11 Goldstein-Price and the MAPS approximation at the start. 17
2.12 Goldstein-Price and the MAPS approximation after 100 evaluations. 17
2.13 Goldstein-Price and the MAPS approximation near the global minimizer. . . 18
2.14 Statement of the Model-Assisted Pattern Search Algorithm. 19

4.1 Hartman6 Function, 50 evaluations. 31
4.2 2-D T-WAFER density plots, 20 evaluations. 32
4.3 2-D T-WAFER density plots, 50 evaluations. 33
4.4 Hartman3 density plots, 20 evaluations. 34
4.5 Hartman3 density plots, 50 evaluations. 35
4.6 Shekel7 density plots, 20 evaluations. 36
4.7 Shekel7 density plots, 50 evaluations. 37
4.8 T-WAFER density plots, 20 evaluations. 39
4.9 T-WAFER density plots, 50 evaluations. 39
4.10 T-WAFER density plots, 100 evaluations. 40
4.11 Shekel10 density plots, 20 evaluations. 41
4.12 Shekel10 density plots, 50 evaluations. 42
4.13 `gprof` Results for one MAPS run on Shekel5. 44

A.1 5-D Krigifier(1C) density plots, 20 evaluations. 51
A.2 5-D Krigifier(1C) density plots, 50 evaluations. 52
A.3 4-D Krigifier(1C) density plots, 20 evaluations. 53
A.4 4-D Krigifier(1C) density plots, 50 evaluations. 54
A.5 3-D Krigifier(1C) density plots, 20 evaluations. 55

A.6 3-D Krigifier(1C) density plots, 50 evaluations. 56
A.7 3-D Krigifier(1Q) density plots, 20 evaluations. 57
A.8 3-D Krigifier(1Q) density plots, 50 evaluations. 58

B.1 Krigifier 2-D — Runs better than DACE's 25th percentile. 59
B.2 Krigifier 3-D — Runs better than DACE's 25th percentile. 59
B.3 Krigifier 4-D — Runs better than DACE's 25th percentile. 60
B.4 Krigifier 5-D — Runs better than DACE's 25th percentile. 60
B.5 Shekel Family — Runs better than DACE's 25th percentile. 60
B.6 Hartman Family — Runs better than DACE's 25th percentile. 60

C.1 Krigifier 2-D — Runs "near" the function value of the global minimizer. . . . 61
C.2 Krigifier 3-D — Runs "near" the function value of the global minimizer. . . . 62
C.3 Krigifier 4-D — Runs "near" the function value of the global minimizer. . . . 62
C.4 Krigifier 5-D — Runs "near" the function value of the global minimizer. . . . 62
C.5 Shekel Family — Runs "near" the function value of the global minimizer. . . . 62
C.6 Hartman Family — Runs "near" the function value of the global minimizer. 63
C.7 T-WAFER — Runs "near" the function value of the global minimizer. . . . 63

Chapter 1

Introduction

1.1 Motivation

Scientists and engineers in all disciplines have turned to computer simulations to supplement traditional methods of experimentation. Often these simulations replace expensive or impossible physical tests, and allow for analysis of these physical systems at a significantly reduced cost. Though the simulation may take hours or days to complete one run, the net cost of the simulation is often far less than that of its physical counterpart. Increasingly, scientists and engineers have also been using these simulations to find the "best" possible design parameters for a physical product. This process is called optimization. One example of optimizing such a simulated system is the helicopter rotor blade studied by Booker et al. [3]. The goal of optimizing the design problem represented by this computer simulation was the design of a better rotor blade by choosing certain properties of the blade. It is this issue of the optimization of problems defined by computationally expensive computer simulations, such as the rotor blade simulation, which this research addresses.

Functions represented by expensive simulations tend to have characteristics which make traditional optimization methods unsuitable. Often, derivatives are unavailable. Further, approximations to the derivative may be unreliable due to variations induced by numerical error in the computer simulation of the objective. The T-WAFER problem [16, 12] is a simulation with such variation, as shown in Figure 1.1. The "furrows" in this function have the potential to cause problems for derivative-based optimization methods. Direct search methods, which do not require derivatives, do require a relatively large number of function evaluations. For the problems of interest to us, there is a high computational expense for each evaluation of the objective function, which makes the use of direct search methods problematic. Thus we turn to other alternatives.

The Surrogate Management Framework (SMF) outlined by Booker et al. [3], offers one such alternative. The basic idea of SMF is to use a global approximation to the objective function to accelerate the search for a minimizer. The SMF uses this approximation to guide a pattern search, which is a type of direct search. The specific instantiation of the Surrogate Management Framework that is described in [27] is called the Model-Assisted Grid Search (MAGS); we have modified MAGS along the lines discussed in [24, 25, 27] and renamed it the Model-Assisted Pattern Search (MAPS). The implementation and testing of MAPS is

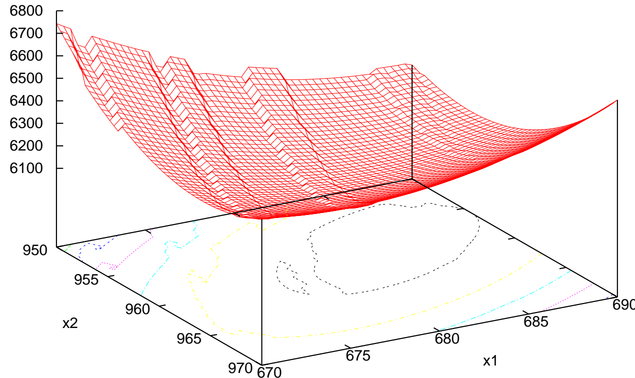

Figure 1.1: The 2-D T-WAFER problem.

the topic of this research.

1.2 The Basic Concept of MAPS

The Model-Assisted Pattern Search algorithm is designed to solve problems of the form:

$$\text{minimize } f(x) : \Re^n \to \Re, \text{ subject to } a \leq x \leq b, \text{ for some bounds } a, b \in \Re^n.$$

For engineering design problems, the goal frequently is to find a global minimizer of the function. Except in rare circumstances, no optimization algorithm can produce such a result on an extremely limited set of budget evaluations. Thus there are often two competing foals. The first is to guarantee convergence to some stationary point or minimizer, even if it is not a global minimizer. The second is to try to find a global minimizer. The Model-Assisted Pattern Search algorithm is concerned primarily with the first goal, as will be discussed further in Section 2.2. However, the MAPS methodology also addresses the the second goal. The MAPS algorithm uses a global approximation to the objective function to guide its search process tries to keep the approximation reasonably faithful to the objective function. By keeping the global model faithful to the objective function, MAPS has the potential to do a better job of identifying a global minimizer than more traditional nonlinear programming techniques, which rely only on local information about the objective.

1.3 Issues to be Addressed

This work addresses two basic sets of issues. The first set involves the development and implementation of MAPS. Chapter 2 focuses on this discussion. Mathematical, computational and practical concerns in the implementation are addressed in Section 2.3. Similar concerns about the parameter estimation process used by MAPS are addressed in Section 2.4.

The second set of issues involves empirical testing of MAPS, and are discussed in Chapter 3. By testing MAPS on functions which are known to be difficult to optimize, one can get an idea of the effectiveness of the algorithm in general. The results obtained using MAPS were compared to the results obtained using the algorithms from which MAPS was derived, namely pattern search [19] and DACE [29], both of which are described more extensively in Chapter 3. The results of these experiments show that MAPS is more efficient in terms of function evaluations than either the DACE or the pattern search algorithms tested. In addition to this efficiency, the tests show that MAPS is on average more likely to converge to the global (rather than local) minimizers than either DACE or the pattern search algorithms tested.

Chapter 2

Algorithmic Framework

The Model-Assisted Pattern Search (MAPS) algorithm walks the middle line between aggressively searching for a global minimizer and guaranteeing convergence to some minimizer or stationary point.

MAPS is a special case of the Surrogate Management Framework (SMF) outlined by Booker et al. [3]. The idea behind the SMF is to use a global approximation to the objective function f to accelerate the search for a minimizer. The SMF uses this approximation to guide a pattern search, a type of direct search described in [22, 19]. The implementation of the algorithm developed in this work uses a geostatistical process known as kriging to create an interpolatory approximation from known values of the function [17, 30].

2.1 Two Illustrative Examples

Figure 1.1 shows a plot of the two-variable version of the silicon wafer heating problem (T-WAFER) described in [12, 16]. The function $f(x)$ defined by that computer simulation will be the subject for optimization in this example. One important thing to note is that this function is somewhat "noisy." The small "waves" in the graph are the visual representation of that noise. However, the noise is not representative of the physics underlying the problem definition; rather it is the result of using a relatively large tolerance to terminate the PDE-solver within T-WAFER. Since this noise is considered to be an artifact of the computational process, we would like to see an optimization algorithm ignore the false minimizers computation may introduce. Instead, we would like to see the algorithm converge to the global minimizer in the correct part of the "valley" on the plot.

MAPS is a grid-based search, and thus some kind of initial grid Γ_0 must be chosen for a run of the MAPS algorithm. Next, some kind of initial design must be chosen to obtain the sample data used to construct the global approximation for the problem. The initial grid, Γ_0, for this problem is a simple rectangular grid with a spacing of 0.2 between grid lines, starting at the lowest point in both dimensions $(670, 950)$. The initial design sites, $x_1, \ldots, x_{10} \in \Gamma_0$, have been generated using a Latin hypercube [13], a common method for generating space-filling designs. The sites themselves are shown in Figure 2.1.

At each these initial design sites the simulation is evaluated, by running T-WAFER, to yield values $f(x_1), \ldots, f(x_N)$. They are shown superimposed on the objective function

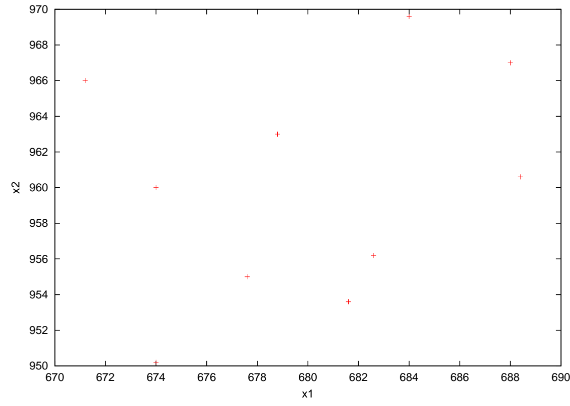

Figure 2.1: Latin hypercube design.

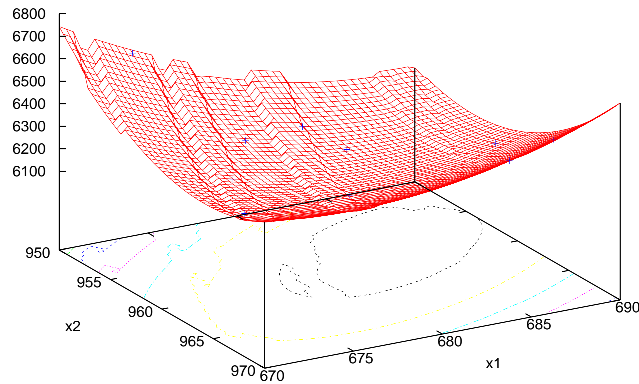

Figure 2.2: Latin hypercube design on $f(x)$.

$f(x)$ in Figure 2.2. With this information, the process of kriging is used to generate an initial approximation $\hat{f}_0$ to the objective function $f(x)$. This kriging process is discussed in Section 2.4. A graph of the initial approximation is shown in Figure 2.3. Note that this approximation smoothes out the "noise" apparent in 1.1, the graph of the simulation outputs. This smoothing effect will allow MAPS to ignore the false minimizers which are believed to be artifacts of the computational process.

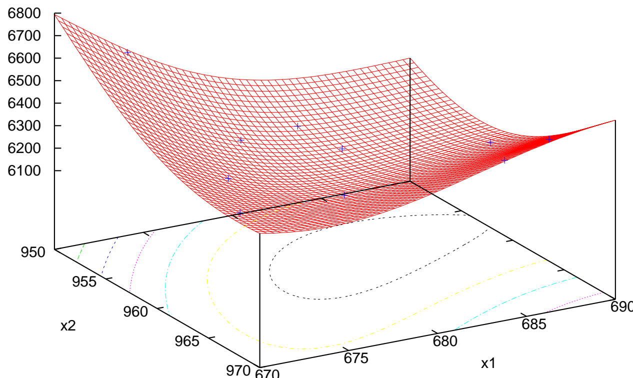

Figure 2.3: Approximation $\hat{f}_0$, 10 evaluations.

At this point, the iterative procedure of MAPS is ready to begin, and will continue until termination, which is discussed further in Section 2.3.2. Figure 2.4 shows the approximation superimposed on the graph generated by the simulation. One important thing to note is that while the approximation has captured the general shape of the simulation graph, it is imperfect. In this example, the approximation is poor near the corners of the feasible region. But what if the global minimizer is near one of the corners of the feasible region? Acting on the approximation alone, MAPS might miss it. Some method is necessary to force MAPS to "spread out" and look around in areas of the function domain where not much data exists.

The concept of a "search criterion" $S_0(x)$ was introduced in [4, 24, 25] to deal with this concern. The goal of the function $S_0(x)$ is to balance the need for exploration of the feasible region with the desire to immediately find a minimizer. A detailed discussion of our choice of a search criterion can be found in Section 2.2. For now, it suffices to say that our search criterion is based on the value of the approximation at a given site x, but decreases the weight given to the approximation's value for values of sites far from sites at which f has already been evaluated. A graph of our search criterion function for this example is shown in Figure 2.5. It is this search criterion, rather than the approximation itself, that is optimized directly to yield a prospective new point x_1 for the evaluation by T-WAFER. The point is shown on the approximation $\hat{f}_0$ in Figure 2.6.

After the evaluation of the new point x_1, the algorithm updates the model $\hat{f}_0$ to yield a new model $\hat{f}_1$, and the search criterion S_0 to yield a new search criterion S_1, both of which

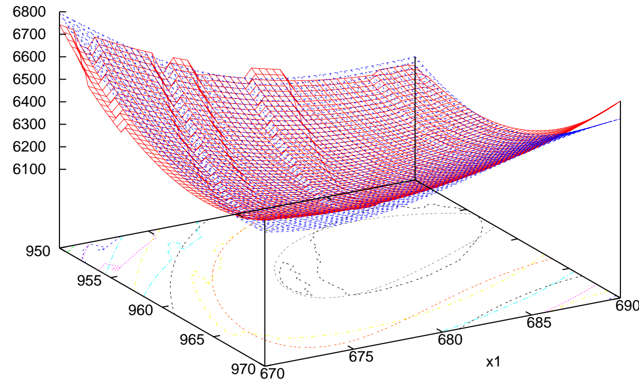

Figure 2.4: Approximation $\hat{f}_0$, superimposed on T-WAFER.

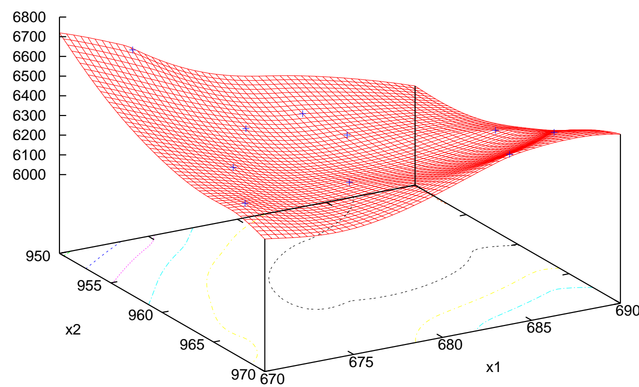

Figure 2.5: Search criterion S_0, 10 evaluations.

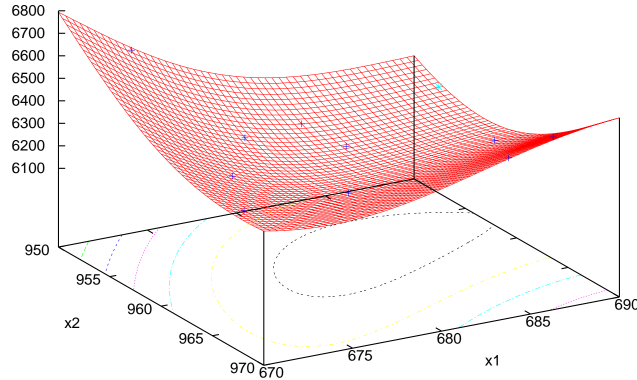

Figure 2.6: Approximation $\hat{f}_0$, with x_1 and original design sites shown.

will be used in the next round of computation. The new model, $\hat{f}_1$ is superimposed on the simulation outputs and is shown in Figure 2.7. Note that the new model changes shape to more accurately represent the simulation outputs, especially around the area of the new point at which T-WAFER was evaluated. MAPS is now prepared for the next round, and thus continues the optimization process.

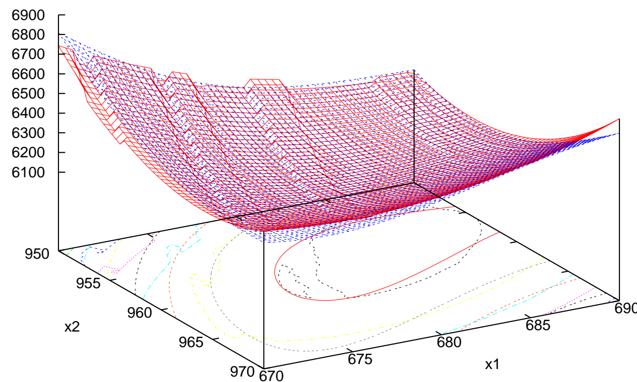

Figure 2.7: New approximation $\hat{f}_1$, superimposed on T-WAFER.

As stated before, the use of approximations in lieu of the simulation code (T-WAFER in this example) "smooths" out the function. The goal here is to try to avoid being trapped at the false minimizers created as artifacts of the simulation. Is this a legitimate concern? Should one be worried at all about this occurrence? The answer is "yes." Figure 2.8 shows

the location of the solutions reported by 1,000 runs of the compass search variant of pattern search [7], run to convergence for a fixed tolerance. Note that this algorithm was indeed caught by those artifacts for several of these runs. The valley in which compass search (see Section 3) got caught contains multiple points of discontinuity introduced by the simulation. A more detailed graph of this valley, and the subset of solutions returned by compass search, can be found in Figure 2.9. MAPS, however, did not fall victim to this problem. For 1,000 runs, with a function evaluation budget of 100, MAPS always returned a point in the neighborhood of the global minimizer. Thanks to the effects of smoothing, MAPS missed the ridge entirely.

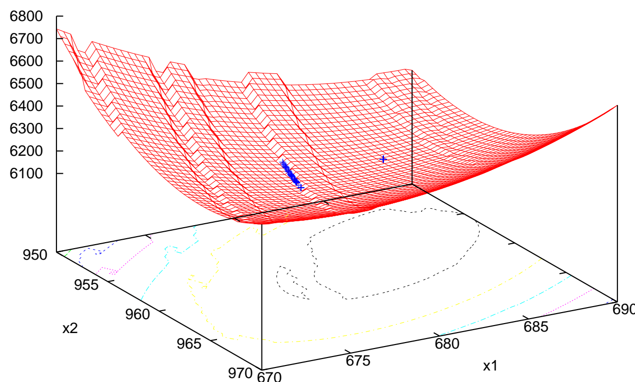

Figure 2.8: Minimizers located by compass search.

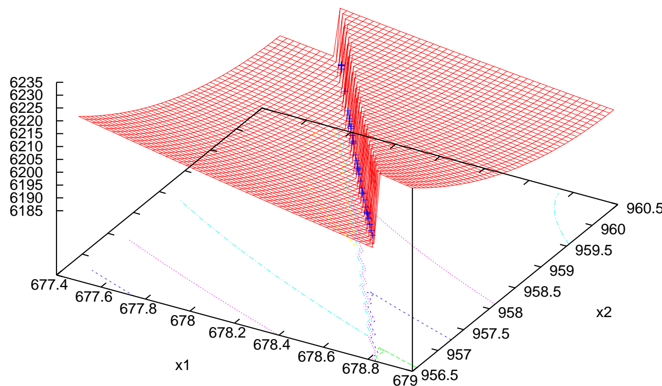

Figure 2.9: Compass search minimizers along the valley.

In addition to the smoothing out of what are believed to be spurious discontinuities, in our experiments MAPS has also been more likely to find a global minimizer. One example found in the optimization literature is the Goldstein-Price function [6], which has the domain $[-2, 2]^2$ and and four distinct minimizers. For the purpose of this example, the Goldstein-Price function has been rescaled to the domain $[-20, 20]^2$. Figure 2.10 shows a graph of the function, with the minimizers marked. It is important to note that this objective function is very poorly scaled, and possesses ridges that cannot be seen in the plot at this level of resolution.

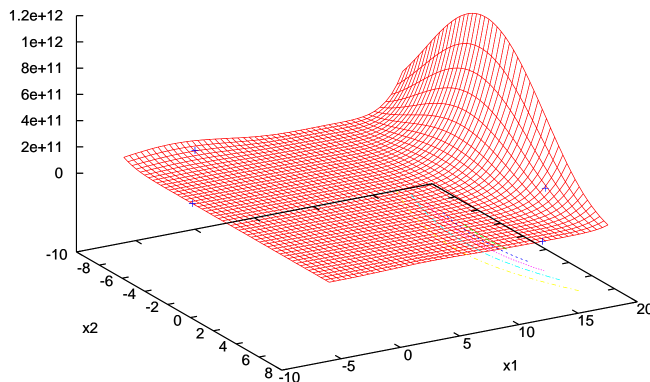

Figure 2.10: Goldstein-Price function, with minimizers.

By playing a balancing game between a single-minded pursuit of the minimizer and a desire to obtain a more accurate approximation of the function, MAPS is better able to find the global minimizer. Figure 2.11 shows the Goldstein-Price function and MAPS' approximation after 10 initial function evaluations. After a total of 100 function evaluations (including the initial 10 used to construct f_0), MAPS' approximation has improved significantly, as shown in Figure 2.12. Thanks to its search criterion which promotes exploration of the feasible region, MAPS has a very good idea of what the function looks like, at least on a macro scale. After 100 evaluations, MAPS has neared the global minimizer, and its approximation near that minimizer is shown in Figure 2.13.

2.2 The MAPS Algorithm

A statement of the MAPS algorithm can be found in Figure 2.14. One first selects a grid for the search. Next, the initial design sites are chosen using some initial design method. Our implementation of MAPS uses a Latin hypercube initial design, as Latin hypercubes are very simple to generate [13]. Then one evaluates the objective function at these design points and creates the initial approximation. The algorithm then moves sequentially, optimizing the "search criterion" to yield a new point, evaluating the objective function at that point,

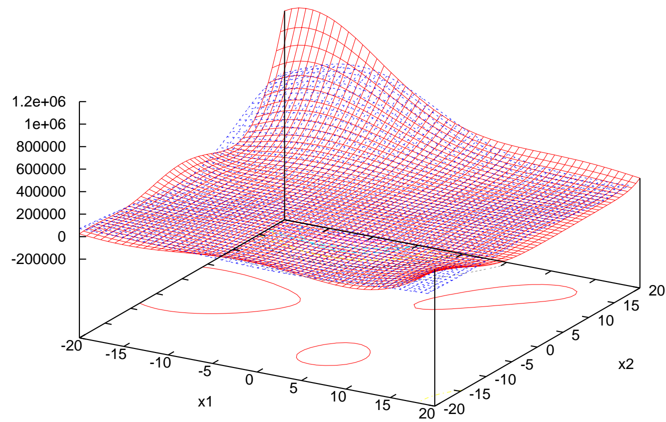

Figure 2.11: Goldstein-Price and the MAPS approximation at the start.

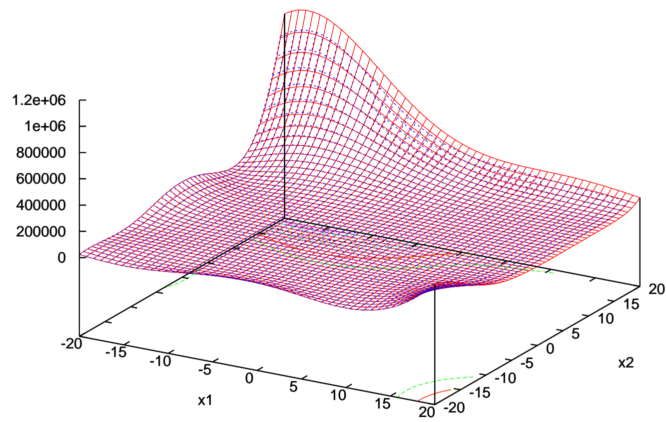

Figure 2.12: Goldstein-Price and the MAPS approximation after 100 evaluations.

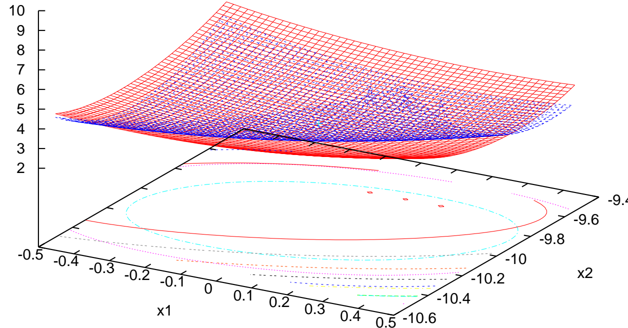

Figure 2.13: Goldstein-Price and the MAPS approximation near the global minimizer.

and finally updating the search criterion. If the "core pattern" of the best point has been evaluated, then the grid can be refined.

But what are the core pattern and the search criterion? The core pattern is specified by the pattern search upon which MAPS is based. In order for a pattern search to refine the grid underlying the search, a set of points known as its core pattern must have been evaluated. This stipulation is made in order to guarantee the convergence of the pattern search algorithm [22, 15]. Likewise, to ensure convergence, MAPS can refine its grid only when this same condition is satisfied [23]. In our implementation of MAPS, the core pattern is a simple compass pattern [7], consisting of steps in each of the positive and negative coordinated directions.

The search criterion is a function that is best stated as follows:

$$S_c(x) \quad = \quad \hat{f}_c(x) - w_c D_c(x) \tag{2.1}$$

At the current iteration c, the value of the search criterion S_c at a given point x is equal to the value of the approximation of this point $\hat{f}_c(x)$, plus a weight w_c times the "design criterion" $D_c(x)$ at that point. The weight is a positive real number selected by some sort of weighting scheme.

The design criterion used in MAPS was inspired by Cox and John [4], and can be stated as follows:

$$D_c(x) \quad = \quad \sqrt{\widehat{MSE}\left(\left[\hat{f}_c(x)\right]\right)} \tag{2.2}$$

Note that $\widehat{MSE}([\hat{f}_c(x)])$ is the estimated mean squared error of the approximation.

The goal of the search criterion is to make the trade-off between single-minded pursuit of a minimizer and a wider exploration of the feasible region to look for a more promising solution. The design criterion promotes the investigation of previously unexplored areas of

1. Specify an initial grid, Γ_0, on bounds [a,b].
2. Perform an initial computer experiment :
   (a) Select initial design sites $x_1, \ldots, x_N \in \Gamma_0$.
   (b) Compute $f(x_1), \ldots, f(x_N)$.
   (c) Construct $\hat{f}_0$ by kriging.
3. Let $\Gamma_c = \Gamma_0$. Let $x_c = \text{argmin}\,(f(x_1), \ldots, f(x_N))$.
   Let $\hat{f}_c = \hat{f}_0$ and $\text{Eval}_c = \{x_1, \ldots, x_N\}$.
4. Do until convergence:
   (a) Do until $\text{Core}(x_c) \subset \text{Eval}_c$:
      i. Apply an optimization algorithm to S_c (the search criterion) to obtain $x_t \in \Gamma_c \backslash \text{Eval}_c$.
      ii. Compute $f(x_t)$. Let $\text{Eval}_c = \text{Eval}_c \cup \{x_t\}$. Update S_c.
      iii. If $f(x_t) < f(x_c)$, then let $x_c = x_t$.
   (b) Refine Γ_c.

Figure 2.14: Statement of the Model-Assisted Pattern Search Algorithm.

the feasible region, whereas the approximation promotes the pursuit of a near-by minimizer. The original Model-Assisted Grid Search (MAGS) algorithm [27] did not make use of a design criterion. Instead, the algorithm optimized the approximation directly. Trosset [25] argues that this single-minded pursuit of the minimizer may actually hinder the search's quest for the global minimizer, and introduced the search criterion described above.

2.3 Numerical Optimization

2.3.1 Algorithmic Considerations

As outlined in 2.2, the MAPS search criterion tries to strike a balance between the value of the approximation $\hat{f}_c(x)$ and the value of the design criterion $D_c(x)$ for a given x. The important algorithmic question then becomes one of finding the right weighting scheme, or set of w_c's, that make MAPS work most effectively. Trosset [25] ruled out the scheme of choosing $w_c = 0$ for all c, as it would undermine the search for the global minimizer, and lead to potential ill-conditioning of the correlation matrix that is at the heart of the approximation. This variant of MAPS was called Model-Assisted Grid Search (MAGS) [24]. A second weighting scheme is inspired by a similar approach found in the literature. Since the MAPS' design criterion was inspired by Cox and John's SDO algorithm [4], it is only natural to use their choice of weight. While SDO differs from MAPS in several important respects, both use the square root of the mean squared error (2.2) as a measure of how

well the domain has been explored (our design criterion $D_c(x)$). Unlike MAPS, the SDO algorithm is not based on a pattern search and does not refine its initial grid. For optimizing a specific grid, MAPS and SDO both use a search criterion of the form (2.1) to encourage investigation in previously unexplored areas of the feasible region. However, SDO uses a constant weight of two for the entire search. The choice of two is a natural one, as the search criterion then approximates the lower bound of a 0.95-level confidence interval around the expected value of the objective function. Thus, the SDO-inspired weighting scheme would be $w_c = 2$, for all c.

In addition, two fixed-schedule weighting schemes of the form $w_0 = a$, $w_{c+1} = b \cdot w_c$, for some $a, b \in R^+$ have been tested. Likewise, several dynamic schemes were examined. The "Dynamic" scheme proved to be the most effective and so is the one presented here. The Dynamic scheme can be stated as follows. For each iteration c:

$$w_{c+1} = \begin{cases} m_g \cdot w_c, & \text{if } \left|f(x_c) - \hat{f}_c(x_c)\right| \leq \frac{|f(x_{best}) - \hat{f}_c(x_c)|}{2} \\ m_b \cdot w_c, & \text{otherwise} \end{cases} \quad (2.3)$$

where $w_0 = a$, for some $a \in \Re$. The value of m_b should be greater than one, and the value of m_g less than one, in order to reflect the tradeoff discussed above.

The Dynamic weighting scheme forms an interval around the expected value of the last point. If the actual value of $f(x_c)$ is in that interval, then the approximation is deemed to be accurate. In that case, the weight is shifted in favor of the approximation. If the actual value $f(x)$ is not in the interval, then the approximation is deemed to be poor and weight is shifted in favor of design criterion. Note that the weighting scheme scales well to almost any problem, as the test condition can be restated as the ratio shown below.

$$\left|\frac{f(x_c) - \hat{f}_c(x_c)}{f(x_{best}) - \hat{f}_c(x_c)}\right| \leq \frac{1}{2} \quad (2.4)$$

The choice of $\frac{1}{2}$ was arbitrary, but seems to work well in practice. The results in Chapter 4 use $m_g = 0.75$ and $m_b = 1.25$. Again the choices were arbitrary, but worked well for the problems tested.

Finally, the statement of the MAPS algorithm in Figure 2.14 lends no clue as to how the algorithm should be terminated. Obviously, if the function evaluation budget is exhausted, then the algorithm terminates. But what if the algorithm gets sufficiently close a minimizer before the budget is evaluated? A common criterion found in the literature is to stop when the step size between successive points is small. Since MAPS is a pattern search, one can use the grid resolution, the value of the step length control parameter Δ, to define a termination criterion. The algorithm would terminate when Δ became less than a certain numerical tolerance (we used 10^{-12}). The viability of this criterion was demonstrated in [7]. Other properties of this criterion, including its relation to commonly used gradient-based stopping criterions, are discussed in [8].

2.3.2 Computational Considerations

In addition to the algorithmic considerations outlined above, the MAPS implementation leads to several computational considerations as well. Trosset [27] implemented a prototype

version of the original MAGS algorithm in S-PLUS, but it suffered from three serious drawbacks. First, the S-PLUS software is restricted by license to certain machines. Hence, this prototype could not be ported to systems without a license for S-PLUS. Second, S-PLUS scripts are not fully compiled binaries, so execution times for even modestly-sized problems are unacceptably slow. Third, it could only optimize functions of two variables. Thus, Trosset's implementation, while useful as a prototype, is unsuitable either for computationally expensive problems or comprehensive algorithmic testing. Crafting a C++ implementation of MAPS required that certain key concerns be addressed.

Capacity

The standard version of C++ comes with no implementation of mathematical vectors. The standard template library does contain a vector class, but it has no mathematical operators for it. The Template Numerical Toolkit (TNT), developed by Roldan Pozo at the National Institute of Standards and Technology [20] does provide serviceable implementations of these important constructs. A modified version of TNT, including an implementation of the Euclidean vector norm adapted from the BLAS [14, 2], was used in the MAPS software.

Correctness

Throughout the construction of the computationally delicate parts of the software, specifically the kriging, the author continually compared results with Trosset's S-PLUS prototype for MAPS. The C++ implementation uses a different implementation of the SVD [1] than that used in S-PLUS[1]. In addition, MAPS uses a pattern search to find the MLE, which is used for the construction of the approximation. No such option exists within S-PLUS. Finally, the two implementations use different methods of minimizing the search criterion. In all other respects however, the two versions agree. Thus, for almost all cases tested using both implementations, the results were identical. In those cases where the results differed, the discrepancies could be traced back to either the SVD or different results from the numerical optimization software. Use of graphical methods of analysis have also shown that the processes involved in the MAPS algorithm are functioning properly in the software.

Portability

The goal of portability demanded that the software be implemented in some general-purpose programming language that is widely and freely available. Due to such features as classes, templates, inheritance and polymorphism, C++ was chosen to be the language of implementation for the MAPS software. Designed from the start with classes in mind, the software takes advantage of these features to minimize the amount of code needed. Effort was made in the development of the software to ensure that the software had enough flexibility to compile on a variety of major systems and with a variety of compilers such as egcs (Linux), g++ (SunOS) and xlC (AIX).

[1] Please examine the CLAPACK web page [5] for more details.

Efficiency

The goal of efficiency demanded that the implementation be well-written and done in a language with optimizing compilers available. Using debugging tools such as Pure Software's Purify and Electric Fence, the author has written the code in such a way to avoid memory leaks and off-by-one errors. The use of C++ makes optimizing compilers available for MAPS on almost every platform. Not too surprisingly, the speed gain over the S-PLUS version, in terms of wall clock time on similarly equipped systems, is more than an order of magnitude.

Robustness

Simulations can be written in any of a number of languages (typically some mix of C, C++, or Fortran). Thus, the MAPS software should be able to work with an optimization problem written in almost any programming language in order to achieve the goal of robustness. Thus, no system which required that MAPS compile and/or link with the objective function would be appropriate, as cross-linking is a delicate process. The author decided that the most reliable way to allow the MAPS software to work with any sort of objective function is through the use of separate executables. The MAPS software runs the executable for the objective function, using "fork" and "exec" kernel calls, and then communicates to the results to the MAPS executable via the program's standard input. The cost of this robustness is a slight drop in portability. For instance, a separate function evaluation routine would be needed for Windows operating systems. However, the software should function robustly on all UNIX or other POSIX-compliant operating systems.

2.3.3 Practical Considerations

Numerical error is far and away the largest practical obstacle to making a numerical optimization algorithm like MAPS work. The imprecise nature of floating point arithmetic posed many dangers to the successful implementation of this algorithm. This section will detail some of these more important numerical concerns.

As stated in Section 2.2, MAPS works off of a grid. Thus, a means for moving a point returned by the optimizer of the search criterion to the nearest grid point is needed. The `SnapToGrid` function in `PSGrid.cc` takes care of this with the use of the process described below. The source code is included in Appendix Section E.2.5.

Let the vector $x \in \Re^n$ be the point to be moved to the nearest grid point, and let the vector $t \in \Re^n$ represent the the grid "coarseness" in each of the n dimensions. Finally, let $a \in \Re^n$ represent the lower bound for the problem. Then let the "snapped" point $\hat{x} = \left\lfloor \frac{x-a}{t} \right\rfloor$, where the division operator represents element-wise division.

There is however, one more situation for which point verification is a concern. Assume that at a given point the function has already been evaluated. Suppose that the minimizer of the search criterion is this exact same point, but due to numerical error the program returns a point close, but not identical to the search criterion's minimizer. A test for equality would fail as they are two different points. But they should be considered the same. This implementation of MAPS uses a "non-strict" equality for such checks. It checks to see if the point returned from optimizing the search criterion is within a given tolerance of one of the

evaluated points. By asking "Is this point close to a previously evaluated point?" rather than "Is this point equal to a previously evaluated point?", MAPS can overcome potential identification and verification problems introduced by numerical error.

One of the main dangers of the modeling approach used by MAPS is the degradation of the approximation. If the approximation to the function ceases to accurately represent the function even near known points, then the approximation is useless if not outright harmful. The degradation of the approximation is a property of the parameter estimation. The reasons behind the degradation are discussed further in Section 2.4; here we address the numerical considerations specifically.

It has been found that the search criterion can yield function values for previously evaluated points that are different from the actual function values at these points. Operating without the limits of finite precision, these values would indeed be identical since the kriging-based approximation would directly interpolate these points, and the design criterion at these points would equal zero. On a finite precision machine however, this need not be the case. MAPS deals with the potential for numerical error by replacing the search value with that of the previously evaluated function value at this point. This trick prevents the search criterion's optimizer from returning a point that has already been evaluated unless that point is a minimizer of the search criterion. As an added safety precaution, if the core pattern of the point returned by the search criterion and the point itself have all been evaluated, MAPS will begin to evaluate the core pattern of the current best point.

2.4 Parameter Estimation

2.4.1 Mathematical Considerations

The kriging process used to generate the approximation requires the maximum likelihood estimation (MLE) of the kriging parameters $(\beta, \sigma^2, \theta)$, where β is a weighted function mean, σ^2 the function variance, and θ the parameter of the chosen correlation family. A comprehensive introduction to the kriging process can be found in [27] or [17], which we will not attempt to summarize here. However, it is important note that the kriging process requires two things. First, it requires some kind of optimization algorithm to use to perform the MLE. Second, determining $(\beta, \sigma^2, \theta)$, requires the inversion of a matrix.

Computing the maximum likelihood estimate of θ requires the minimization of the expression $n \log \hat{\sigma}^2(\theta) + \log \det R(\theta)$, where $\hat{\sigma}^2$ is the MLE of σ and $R(\theta)$ is the matrix representing the correlation between the previously evaluated points. This sub-problem is either a one dimensional problem or a multi-dimensional problem, depending on the correlation family used. Currently, MAPS uses a type of pattern search known as compass search [7] for this optimization.

Computing the parameters also requires matrix inversions. This is non-trivial, as the matrices that need to be inverted may be singular. MAPS uses an implementation of the Moore-Penrose generalized inverse to achieve this goal. The Moore-Penrose generalized inverse for a matrix A with singular value decomposition $A = V\Sigma W^*$ is defined as $A^\dagger = W\Sigma^\dagger V^*$, where $\Sigma^\dagger$ is the transpose of Σ with non-zero values replaced by their reciprocals [11]. Thus, the Moore-Penrose generalized inverse allows the "pseudo-inversion" of singular

matrices.

2.4.2 Computational Considerations

The computational considerations for the parameter estimation all center on inverting matrices. As noted in Section 2.3.2, C++ provides no suitable implementation of mathematical matrices and vectors. Likewise, it has no standard implementation of important matrix operations, such as the singular value decomposition. The CLAPACK library from netlib [5] does provide this functionality, however. The library was converted from the Fortran LAPACK [1] using the f2c program.

For computational reasons, we modified the Moore-Penrose matrix pseudo-inversion algorithm mentioned in Section 2.4.1 slightly. The SVD routine can return supposed singular values that are very small, but non-zero. Mathematically these values might be equal to zero, but floating-point mathematics this may not be the case. In the case of values this small, we have opted to zero out all such values under a fixed tolerance, in an effort to be as accurate as possible. Following the fairly standard practice in the literature, we have chosen $\epsilon_{mach} \cdot n \cdot \sigma_1$, where n is the size of our square matrix, σ_1 is the largest singular value of the matrix, and ϵ_{mach} is an estimate of machine epsilon. This particular tolerance is used in the MATLAB software from MathWorks [21].

2.4.3 Practical Considerations

Much like the practical concerns of the optimization process, numerical error dominated the practical concerns for the parameter estimation process. There are three main areas in this process that involve such numerical error concerns. The first is the maximum likelihood estimation (MLE) of the parameter θ, the second is the computation of the parameter σ^2, and the third is the set of computations involved with the mean squared error (MSE) required by the design criterion.

The main danger in the maximum likelihood estimation of θ is having the parameter run to either zero or infinity during the estimation process. For the Gaussian correlation family, a very large θ effectively eliminates any correlation between the points, reducing the approximation to a flat function with a few very sharp spikes at the evaluated points. Likewise, a very small θ increases the correlation between points to almost one. This makes the correlation matrix very poorly conditioned, if not numerically singular. This problem is created in one of two ways. The first cause is created by an accumulation of numerical error from previous calculations; the second is caused by the starting point for the optimizer used for the MLE. The first cause is largely unavoidable. However, the second can be dealt with in an intelligent fashion.

Initially, Trosset proposed using $\theta = -\frac{log(0.99)}{\Delta^2}$, where delta is the grid step-length, as an upper bound with the Gaussian correlation family in mind. With the Gaussian family, the upper bound forces adjacent points on the grid to have a correlation of 0.99. Practically, this proved to be too strict an upper bound, as functions with rapid function variation often have nearby points which are not so closely related. Thus we relaxed the upper bound to $\theta = -\frac{log(0.80)}{\Delta^2}$. The initial guess for the first iteration of estimation was half of the upper bound. During experimentation, we found that this was likely to promote θ to increase to

infinity. A lower starting point would not lead to such a result as often. We settled on $\frac{-100log(0.5)}{L^2}$, where L is the largest difference between the lower and upper bounds in any dimension.

Secondly, the variance (σ^2) would occasionally wind up equal to or even less than zero. This error was due to accrued numerical error. Making the necessary corrections to safeguard θ went a long way to solve this problem, but as a safety precaution, after each update of σ^2, it is assigned to be $\max\{10^{-12}, \sigma^2\}$.

In addition to the other sources of numerical error described above, the computation of the design criterion is also potentially vulnerable to numerical error. Occasionally, the computed MSE is slightly negative. This is problematic as the design criterion is the square root of the MSE. This problem occurred with the highest frequency when the point was one of the previously evaluated points. In case of this error, the entire design criterion is set to zero.

These errors might seem to be somewhat obscure, but they are indeed actual errors which occurred in the early phases of testing. We refer to these difficulties because they actually do occur in practice, and thus must be addressed in some fashion. These effects are most probably due to an ill-conditioning of the approximation. As MAPS progresses toward a minimizer, it tends to place evaluation sites close together. This has the effect of increasing the condition number of the correlation matrix, and potentially leading to the sort of errors discussed above.

Chapter 3

Testing

The primary goal of the testing is to make a rigorous comparison between MAPS and the algorithms upon which it is based. The other algorithms tested are listed below:

- **Compass Search** - one of the simplest pattern search methods. It steps one grid length in both the positive and negative coordinate directions. In two dimensions, this corresponds to steps “north,” “south,” “east,” and “west,” hence the name. A good description of compass search and other related pattern search algorithms can be found in [7].

- **Latin Compass Search** - a variant of compass search. The first ten evaluations of the objective function $f(x)$ are devoted to forming a Latin hypercube [13]. Then a compass search is started at the best point in the hypercube.

- **Hooke and Jeeves** - another pattern search methods, first introduced in [10]. See [7] for a further discussion and some comparisons with other pattern searches.

- **DACE** - a non-iterative (i.e. “one-shot”) approach to nonlinear optimization outlined in [29]. This approach involves devoting the whole function budget, barring one evaluation, to the construction of an approximation to the objective function. Then this approximation is optimized to yield the site for the last evaluation. Our implementation of DACE uses a Latin hypercube to choose points for the initial approximation, and then optimizes the approximation with a compass search.

All of the pattern search algorithms were run with an initial step length control parameter $\Delta = 2$, stopping either when the function evaluation budget was expended or when $\Delta \leq \sqrt{\epsilon_{mach}}$. For these trial runs, the budgets were always expended, and the other stopping criterion were not satisfied. The DACE algorithm generated a Latin Hypercube on a boundary-aligned mesh with resolution 10^{-7}. The approximation generated by the DACE algorithm was optimized using compass search with starting step length control parameter $\Delta = 2$ and termination when $\Delta \leq 10^{-7}$.

Three variants of MAPS were tested against these algorithms. They vary according to the search criterion used. The concept of the search criterion is discussed in Section 2.2.

- **Model-Assisted Grid Search (MAGS)** - outlined in [27]. This version of the algorithm optimized the approximation directly and put no weight on the search criterion at all.

- **MAPS-like-SDO** - uses a weighting scheme inspired by the work of Cox and John on the SDO optimization algorithm [4]. This variant fixes the weight on the design criterion at two.

- **MAPS-Dynamic** - uses the dynamic weighting scheme detailed in Section 2.3.1.

The MAPS variants were all run using the Gaussian isotropic correlation family for the kriging parameter estimation, and were starting with a grid resolution of 10^{-2}. The maximum likelihood estimator of the correlation family parameter θ was solved using a compass search. Its starting step length was 0.8 times the upper bound described in Section 2.4.3, and the stopping step length was 10^{-5} times the same upper bound. The algorithms were set to terminate if the grid were refined to a resolution of 10^{-11}, but this criterion was never reached, due to evaluation budget constraints. The MAGS algorithm optimized its search criterion using a compass search, while the MAPS-Dynamic and MAPS-SDO algorithms optimized their search criterions using a multi-start compass search. The initial step size for the appropriate pattern search was $\frac{m}{2}$, where m was the largest integer multiple of MAPS' Δ times the grid resolution of coarsest grid dimension, such that this quantity would be less than the distance between the bounds in that dimension. The optimization of the search criterion would terminate when the Δ of the pattern search reached the current MAPS Δ.

The testing methodology itself was fairly simple. Each algorithm was run 1,000 times on a given optimization problem, with each runs starting from a different point. Each run used different seeds for the random number generator in order to generate its starting points. However, the random seeds were the same across all of the algorithms. This means that the Latin Hypercubes for the MAPS variants and the Latin Compass Search were exactly the same. This consistency allows for meaningful comparisons between the algorithms. The Latin hypercubes were generated by a random shuffling algorithm using the Lehmer pseudo-random number generator described in [18].

All of the tests were run on a Intel machines running Linux 2.2.5-22 (RedHat 6.0). The algorithm implementations were compiled using the egcs-g++ compiler (version 2.95.2) using optimization level `-O`.

3.1 Common test functions in the literature

The optimization literature is full of proposed test functions for global optimization problems. Five classic problems can be found in [6]. For use in the testing of MAPS, the author desired not just single optimization problems, but rather families of such problems. Ideally, these families of problems should have similar behavior, but vary in dimension or number of minimizers. The two families of functions that were used to test MAPS were the Shekel and Hartman families [6, 9].

The general statement of the Hartman family is:

$$f(x) = -\sum_{i=1}^{m} c_i e^{(x-p_i)' A_i (x-p_i)} \tag{3.1}$$

where for $i = 1 \ldots m$, $c_i \in \Re$, $p_i \in \Re^n$, and A_i is a negative definite matrix. The choice of m determines the number of minimizers of the function; n denotes the dimension of x (i.e. $x \in \Re^n$). The two members of the Hartman family used in this research had $m = 4,\ n = 3$ and $m = 4,\ n = 6$, respectively. Values of the other parameters can be found in [6].

The Shekel family is another extensible family. The general statement of this family is:

$$f(x) = -\sum_{i=1}^{m} \frac{1}{(x-a_i)'(x-a_i) + c_i} \tag{3.2}$$

where for $i = 1 \ldots m$, $a_i \in \Re^n$ and $c_i \in \Re^+$. As in the Hartman family, m determines the number of minimizers of the function, and n denotes the dimension. The three members of the Shekel family that were used in this research had $n = 4$ and $m = 5, 7$ and 10. Values for the remaining parameters can be found in [6].

3.2 Real applications

The T-WAFER problem [12, 16] simulates silicon deposition in a hot-wall tubular reactor. The T-WAFER simulation reduces to the numerical solution, within a given tolerance, of a set of partial differential equations. The tolerance used for all of the results in this research is 10^{-3}. This relatively large tolerance appears to yield many numerical artifacts in the objective function, some of which are discussed more in depth in Section 2.1. The two-dimensional of the problem is believed to to have one "real" minimizer; many more are introduced by the simulation.

3.3 Krigifier-generated functions

The krigifier, proposed by Trosset [26] and implemented by Padula [17], is a software package for generating pseudo-random functions. It generates these functions using kriging, a method for function approximation used in geostatistics [30]. The basic methodology used for generating the objective functions used in this research is outlined below:

1. Specify a general trend; half of the functions generated used constant trends, the other half use quadratic trends.

2. Specify an underlying stochastic process; kriging assumes that the the function is the realization of a stochastic process, an infinite collection of normal random variables corresponding to $\Re^n$. The functions tested in this research all assume a Gaussian stationary stochastic process.

3. Specify the number of random points at which to "observe" the stochastic process; for an n dimensional function, 2^{2n} points were used. The more points that are used, the more minimizers a function tends to have.

4. The krigifier evaluates the stochastic process at the observation sites.

5. The krigifier uses kriging to interpolate at these sites to create "noise."

6. The noise and general trend are added together to produce the final function.

Different realizations of the krigifier are created by "observing" the stochastic process at different points. By using different seeds for the random number generator, it was easy to create these different observations. The suite of test functions for MAPS and the other optimization algorithms tested includes five different realizations for each trend (constant or quadratic), for each dimension tested (two, three, four and five). This brings the total number of krigifier-generated test functions to forty.

An important point to note is that the random numbers used to generate the krigifier functions were different from those used to create starting points or Latin hypercubes for the optimizers. This was achieved using the "streams" of independent random numbers created by the random number generator.

A second point of importance involves the relationship between the DACE algorithm and krigifier-generated functions. The DACE algorithm assumes that objective functions are realizations of some stochastic process, and uses this assumption in order to perform optimization. The krigifier-generated functions actually are realizations of a stochastic process. Thus, since the DACE assumptions match reality in this class of functions, the algorithm should perform at its best.

Chapter 4

Summary of Results

For any given optimization algorithm, one can ask two types of questions. First of all, does this algorithm perform reliably and is this revealed in the experimental results? This question deals with the algorithm as an optimization algorithm, completely apart from any concerns for global minimization. Results dealing strictly with reliability and performance are described in Section 4.1. The second set of questions that can be explored involve global optimization concerns. Is the algorithm likely to identify the function value associated with the global minimizer in practice? If not, is it likely to avoid the higher function values of poor local minimizers? Though the MAPS algorithm is not tailored specifically for global optimization, the use of a global approximation to the objective function means that these questions may still be of interest. Results pertaining to these questions are found in Section 4.2. Although additional plots are included in Appendix A, space constraints make it impossible to include the entirety of the results in this document. Arrangements are underway for these plots to appear in electronic form.

The first round of results is represented in the form of individual kernel estimates with a rectangular window [28], through the use of the S-PLUS `density` function. A sample plot of this type is shown in Figure 4.1. Each of the plots represents 1,000 runs of the given algorithm, each returning the function value of a purported minimizer. The kernel estimate for the probability density of the function values returned by the algorithm is then computed, yielding a plot that looks like a smoothed-out histogram. The horizontal axis of the plot represents the value returned by the algorithm, and the vertical axis represents the probability density at that value. The dashed vertical lines represent the function values of the known stationary points of the function. Ideally, an optimization algorithm will show a high probability density around these values, and a low density elsewhere. For global optimization, a high density at the function value representing the global minimizer (leftmost on the plot) would be ideal.

4.1 MAPS and Optimization

The question of the effectiveness of MAPS as an optimization algorithm can be broken up into three different subquestions. First, can we demonstrate, empirically, that MAPS consistently identifies the function value of a minimizer? The conformance of MAPS to the

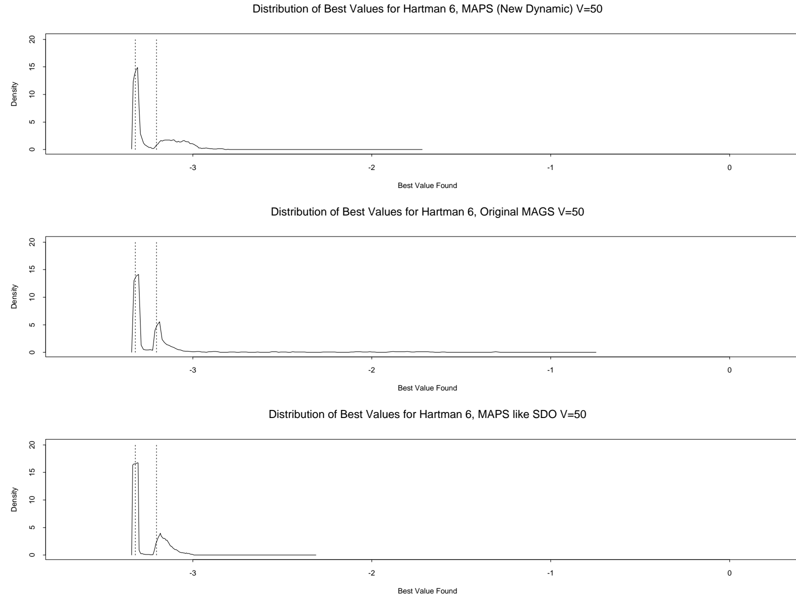

Figure 4.1: Hartman6 Function, 50 evaluations.

requirements of the pattern search analysis suggests that such convergence should occur. Second, does the iterative nature of MAPS yield any advantage over a one-shot approach represented by DACE? Rather than rely solely on the quality of a single approximation, as DACE does, we want to ascertain if there is a clear advantage to refining the approximation iteratively, as the optimization progresses. Finally, does the modeling in MAPS improve the computational cost of the overall performance? The operating assumption throughout has been that the cost of obtaining a single evaluation of the objective function dominates the computational expense. The computational overhead for standard pattern search methods [7] is negligible. Introducing the modeling aspect certainly adds appreciable computational expense to the overhead associated with the optimization. We want to determine if this additional expense is justified by the end results; i.e., leads to an appreciable reduction in the total number of function values required to realize a satisfactory result.

4.1.1 The Reliability of MAPS

One desirable property of an optimization algorithm is reliability. Regardless of the problem being solved, as more function evaluations are allocated to the algorithm, it should move closer toward a solution. To put it simply, if we do more work, we should get a better answer. Our results indicated that MAPS shows that sort of improvement for all the classes of problems tested.

Using the T-WAFER problem discussed in Section 2.1, we can illustrate the improvement MAPS makes as it is given more function evaluations to work with. The density plot in Figure 4.2 represents a budget of $V = 20$ function evaluations allotted to each of the MAPS, Compass Search and Hooke and Jeeves algorithms. Clearly, as Figure 4.2 shows, the MAPS algorithm is nearing the minimum function value with a high degree of probability. Adding

thirty more evaluations, to a total of $V = 50$, accentuates this trend. This scenario, which is shown in Figure 4.3, shows that even the worst runs are returning a function value very close to that of the global minimizer. In this case, the worst run returns a value of about 0.52 greater than the best, and the interval of width 0.015 above the minimum function value contains over 99% of the probability mass.

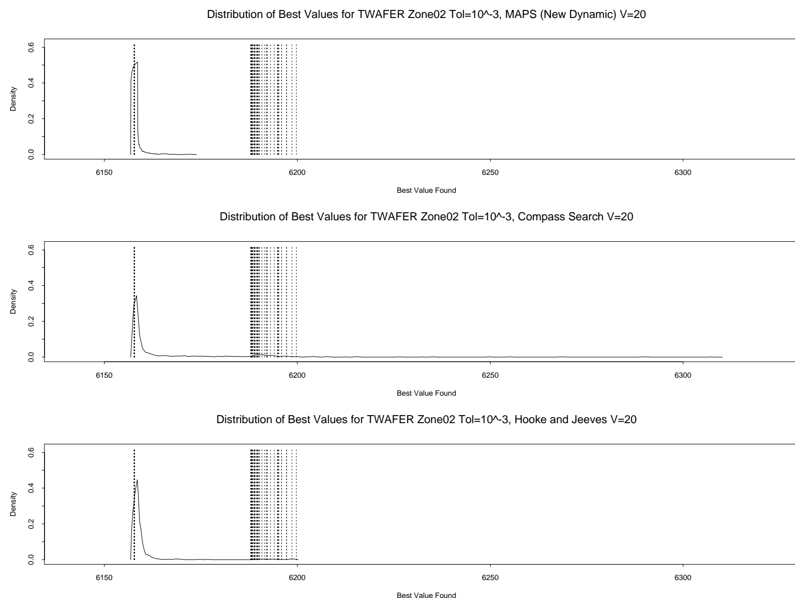

Figure 4.2: 2-D T-WAFER density plots, 20 evaluations.

These figures serve to illustrate what is typical of the experimental results, namely that the MAPS algorithm consistently identifies the function values of known minimizers. Not a single density plot for any of the over 45 functions MAPS was tested on shows the MAPS algorithm yielding worse results when it is allocated a larger budget of function evaluations. A similar example for a five-dimensional krigifier function is shown in Figures A.3 and A.4 in Appendix A. Notice that even with the large number of minimizers that the function has, increasing the function evaluation budget tends to concentrate the probability mass near the function values of known minimizers.

4.1.2 MAPS vs. One-shot Optimization

The MAPS algorithm uses modeling within an iterative framework, but is the iteration useful? The DACE algorithm, advanced in [29], is a one-shot approach that relies exclusively on a single approximation. If there is any advantage in using an iterative approach, then the MAPS algorithm should perform better than the DACE algorithm.

The results from the three-variable function of the Hartman family (Hartman3; see Section 3.1) illustrate the advantage of iterative optimization quite well. Figure 4.4 shows the density plots for the Hartman3 function with an evaluation budget of $V = 20$. Even for this fairly meager evaluation budget, all three MAPS variants are showing far more evidence of having identified two of the three function values associated with minimizers than

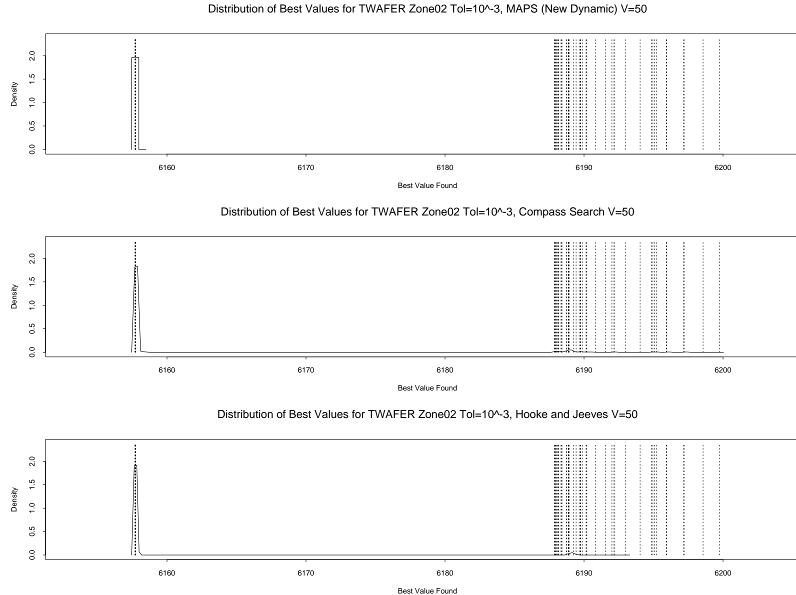

Figure 4.3: 2-D T-WAFER density plots, 50 evaluations.

DACE. With a budget of $V = 50$, the situation is even worse for DACE. Figure 4.5 shows these results. For almost all of the 1,000 runs, the MAPS variants have identified a function value associated with a minimizer. DACE, however, shows little improvement. This strongly suggests that there is an advantage to using an iterative approach.

Once again, these densities are typical of the results generated across all of the functions tested. An example for a five-dimensional krigifier function is shown in Figures A.1 and A.2 in Appendix A. While the results are not as dramatic as those for Hartman3, MAPS is clearly performing better than DACE.

4.1.3 The Advantage of Modeling

MAPS uses a modeling-based approach within a pattern search framework. A reasonable question to ask is whether there is any advantage to the modeling, or will a pattern search alone do just as well? Since our implementation of MAPS is based on a compass search, this pattern search is used as one basis for comparison. The pattern search outlined by Hooke and Jeeves [10], which has been shown to be one of the most effective of the classic pattern search algorithms [7], will also serve as basis for comparison.

The results from the seven variable function of the Shekel family (Shekel7; see Section 3.1) illustrate the potential advantage of using modeling. Figure 4.6 shows the results of the algorithms for a function evaluation budget of $V = 20$. None of the algorithms show convincing signs of having identified a function value associated with a known minimizer. However, when we increase the function evaluation budget to $V = 50$, as shown in Figure 4.7, the MAPS algorithms show clear signs having identified these function values. The same cannot be said for DACE or either of the traditional pattern searches. Results for a four-dimensional krigifier function, which shows the same trends, can be found in Figures

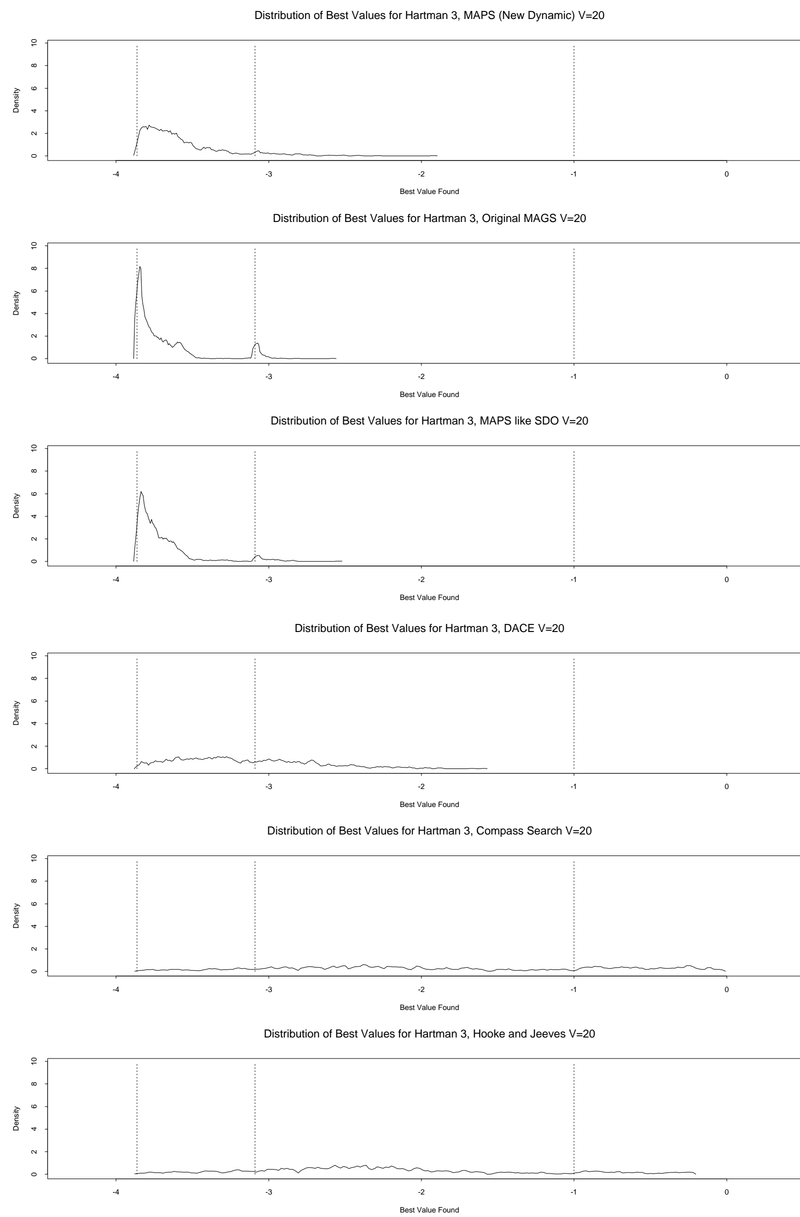

Figure 4.4: Hartman3 density plots, 20 evaluations.

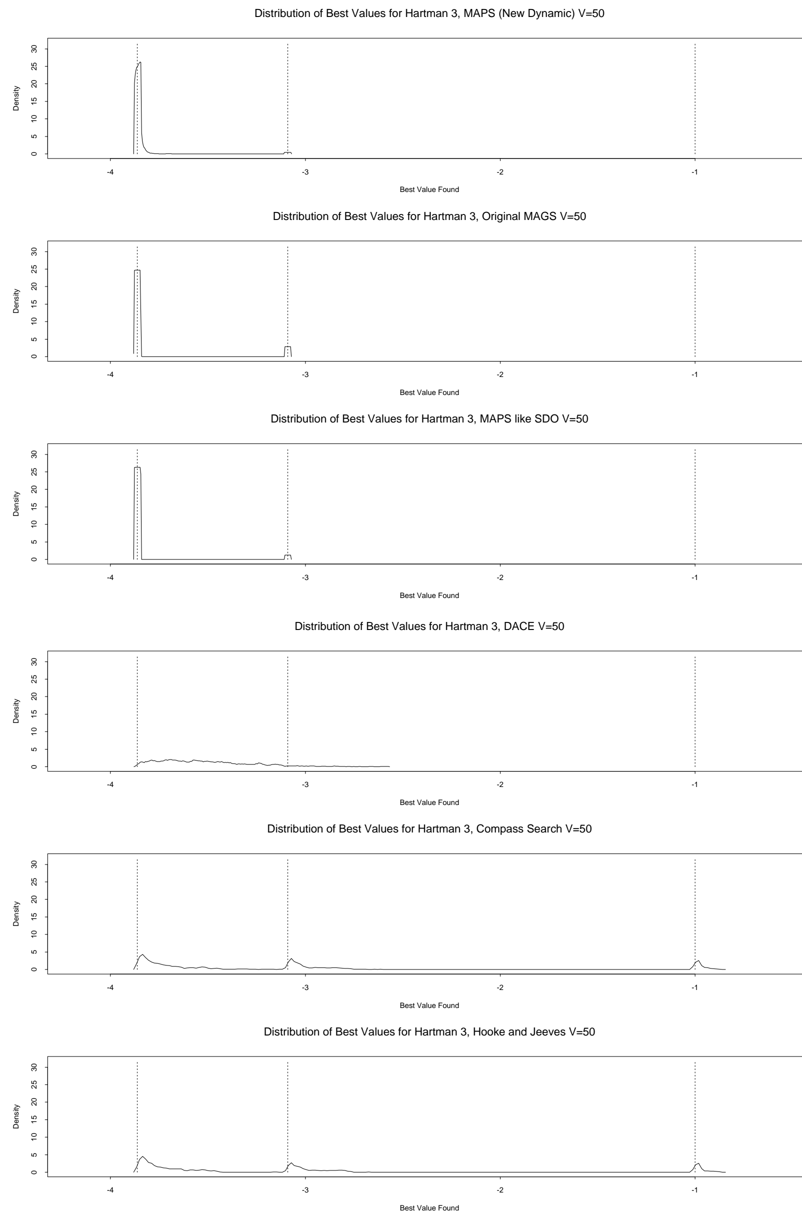

Figure 4.5: Hartman3 density plots, 50 evaluations.

A.3 and A.4 in Appendix A.

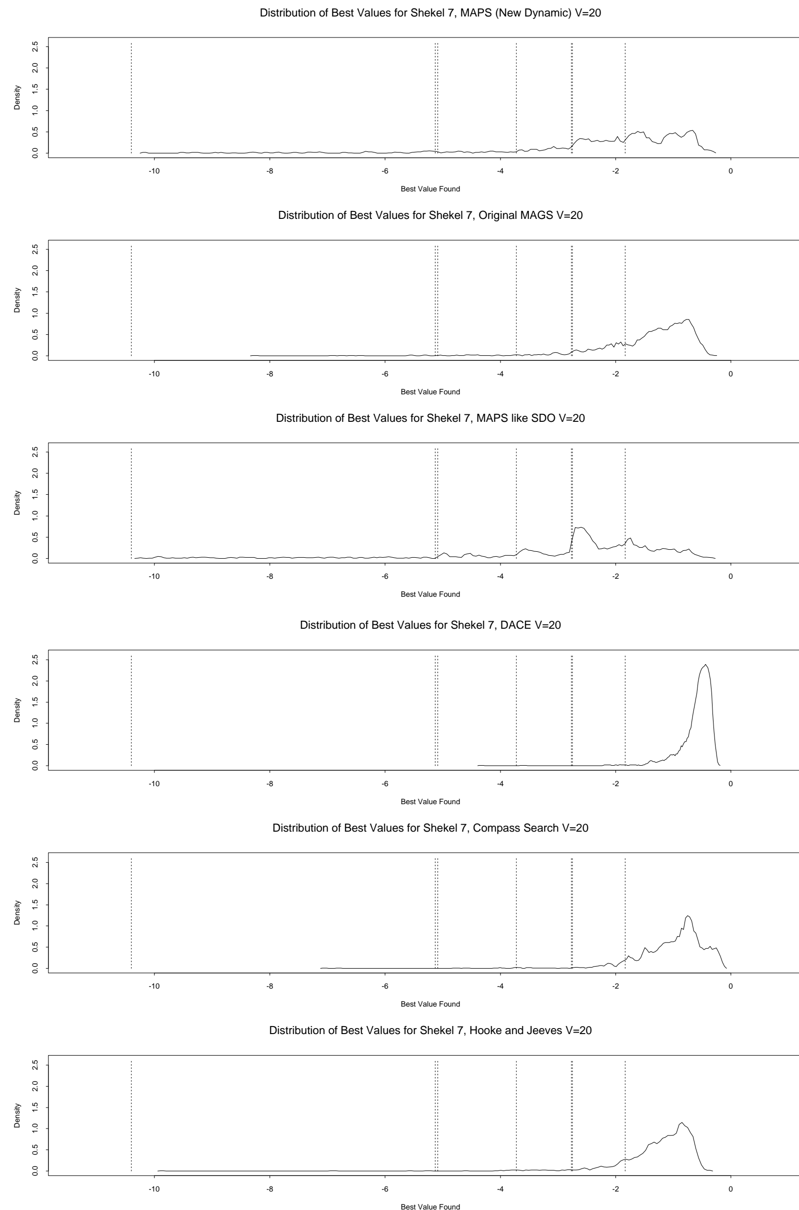

Figure 4.6: Shekel7 density plots, 20 evaluations.

This modeling, however, comes at a price. The modeling adds a substantial computational expense to the otherwise computationally inexpensive pattern search. While this cost is insignificant compared to the costs of an expensive objective function, it is still worth noting and will be discussed in further detail in Section 4.2.4. Nevertheless, these results, and similar results on other families of functions, argue strongly for the advantage of using a modeling approach over a regular pattern search when the cost of function evaluations justify the additional expense introduced by the modeling.

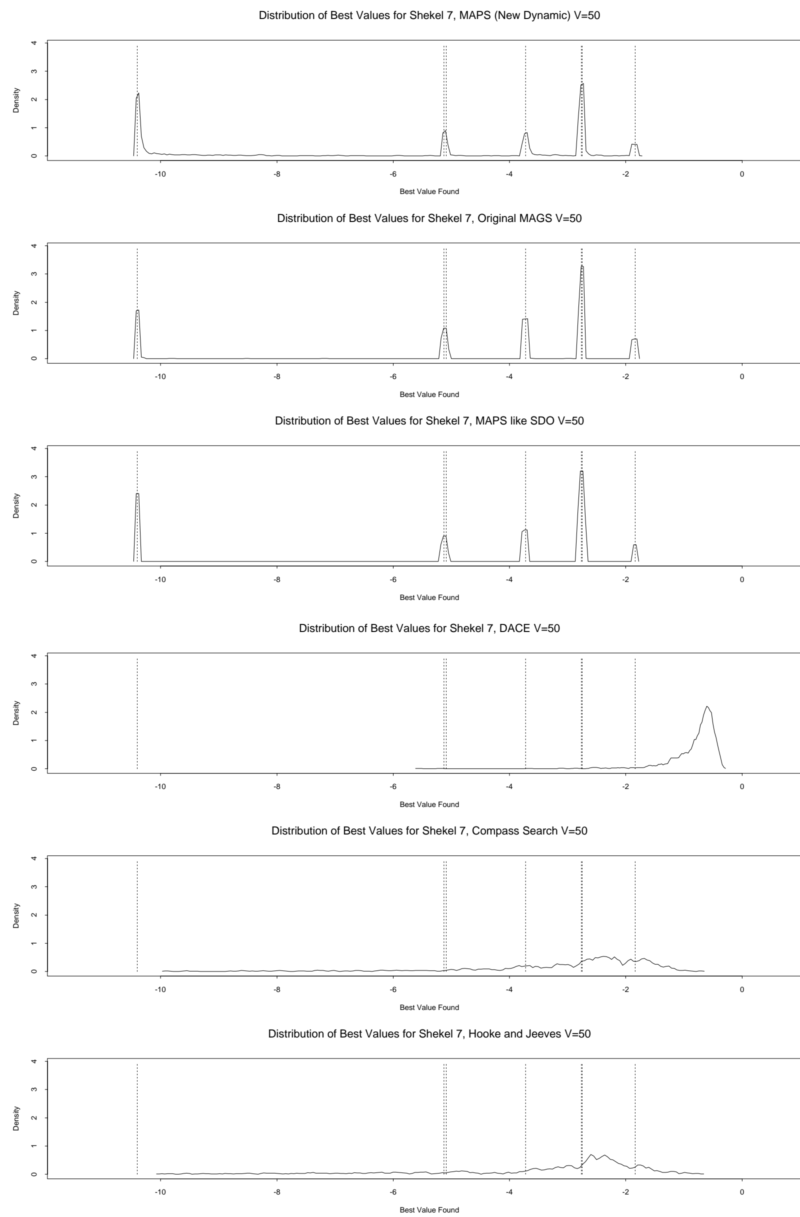

Figure 4.7: Shekel7 density plots, 50 evaluations.

4.2 MAPS and Global Optimization

Problems represented by computationally expensive computer simulations, such as the T-WAFER problem described in Chapter 1 may require that the optimization algorithm attempt to return a global minimizer. Though it is impossible for an algorithm to guarantee to find the global minimizer on an extremely limited function evaluation budget, a desirable algorithm for such problems will make some attempt to avoid the poorer minimizers, particularly those that may be no more than a computational artifact of the simulations, such as those observed in Figures 2.8 and 2.9. Though MAPS is a pattern search, and thus only guarantees convergence to a local constrained stationary point, the use of a global model suggests that it is still reasonable to investigate how well MAPS may do at global optimization. This question leads us into two major sub-discussions. First, does the modeling framework alone yield empirical results that are closer to the function value of the global minimizer? Second, does the use of a search criterion in MAPS improve the likelihood of finding a global minima?

4.2.1 Global vs. Local Minimization

From a strictly global optimization perspective, the ideal algorithm would always return the global minimizer of the functions it was asked to optimize. In some contexts (i.e. looking for "stable" minima) this need not be the case, but more often than not the global minimizer is what is sought. However, on a severely limited evaluation budget, guarantees of convergence to a global minimizer usually are not possible. What we ask of MAPS is whether its is more likely to find the function value associated with the global minimizer, or at least more likely than the other algorithms tested to avoid the local minimizers with the highest function values.

The T-WAFER problem illustrates MAPS' ability to avoid local minimizers with high function values. As noted in Section 2.1, the pattern search algorithms occasionally are caught by the local minimizers generated by a discontinuity, or simply points of discontinuity themselves created by the simulation in the T-WAFER problem. Figure 4.8 shows the function density plots for the T-WAFER problem fun with an evaluation budget of $V = 20$. While all three algorithms show signs of clear convergence to the function value associated with the global minimizer, MAPS appears to have the shortest "tail." Figures 4.9 and 4.10 show the density plots for evaluation budgets $V = 50$ and $V = 100$, respectively. Note that the pattern search algorithms both have some runs which appear to have identified the function values that are presumed to represent simulation artifacts. The MAPS algorithm on the other hand, only shows signs of having identified the function value at the global minimizer. Empirically, this avoidance of these poorer minima holds true for MAPS across all functions tested. The density plots of a three-dimension krigifier function shown in Figures A.5 and A.6 of Appendix A also illustrate this trend quite well.

The tables in Appendix B also show MAPS' ability to return "good" function values. These tables compare the results of all the algorithms tested to the 25th percentile of the value of the runs returned by DACE. These charts show that especially as dimension increases, MAPS tends to be more likely to return better values than DACE. Though MAPS is not primarily a global optimization algorithm, it seems to have performed well relative to

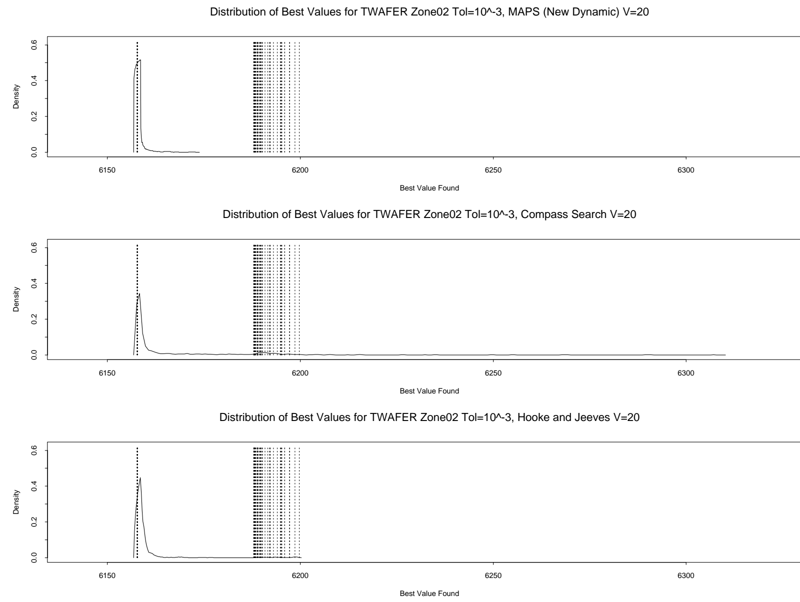

Figure 4.8: T-WAFER density plots, 20 evaluations.

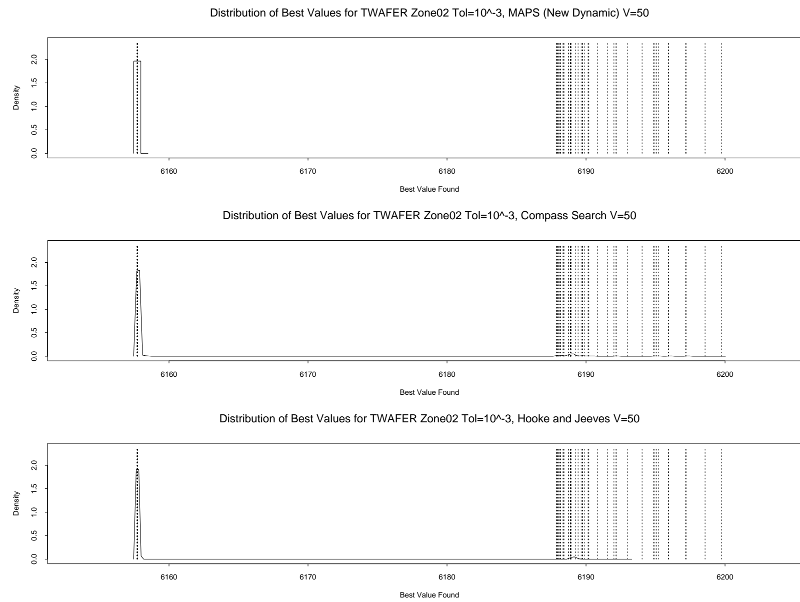

Figure 4.9: T-WAFER density plots, 50 evaluations.

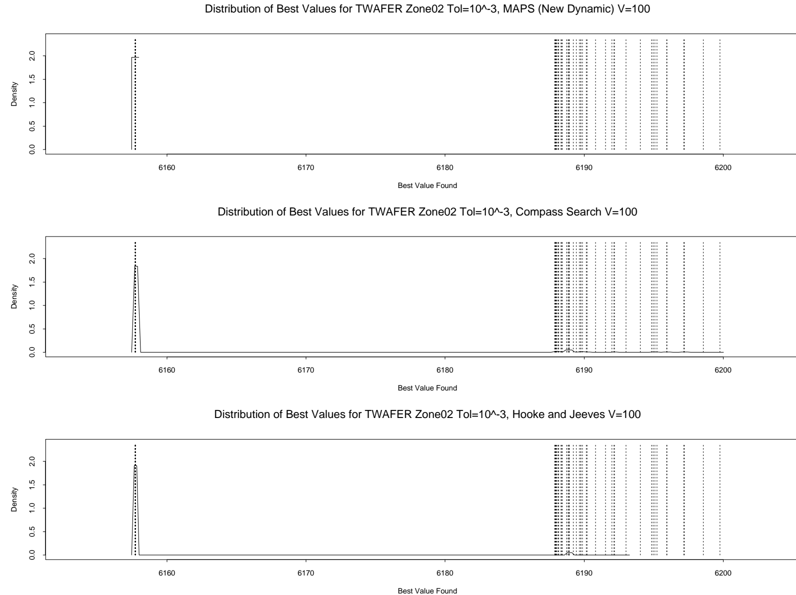

Figure 4.10: T-WAFER density plots, 100 evaluations.

DACE.

4.2.2 The MAPS Search Criterion

Throughout Section 2.1, the point that the use of a search criterion is beneficial is often made. If this is the case, then there should be some kind of demonstrable difference in results between the MAGS implementation, which uses only the approximation, and the Dynamic and SDO-style implementations which factor in a measure of the quality of the overall design. Though this difference is not as obvious as the difference between the MAPS and DACE algorithms, the difference is noticeable nonetheless.

The difference in results due to the use of the search criterion shows itself in the frequency with which a solution near the global minimizer of the function is found. The 5-minimizer function in the Shekel family is a problem that illustrates this difference quite well. Density plots for this function are shown in Figures 4.11 and 4.12. Note that after 50 evaluations (Figure 4.12), the “peak” for the function value of the global minimizer for MAPS-Dynamic and MAPS-SDO is higher than the peak for the function value of the global minimizer for MAGS. According to the statistics in Figure C.5 in Appendix C, the MAPS-Dynamic algorithm has shown itself to be, for this set of experiments, almost twice as likely as MAGS to find the function value associated with the global minimizer. Likewise for this set of experiments, MAPS-SDO proved to be roughly 50% more likely to find the function value associated with the global minimizer than MAGS. The density plots of a three-dimension krigifier function shown in Figures A.7 and A.8 of Appendix A also illustrate this trend.

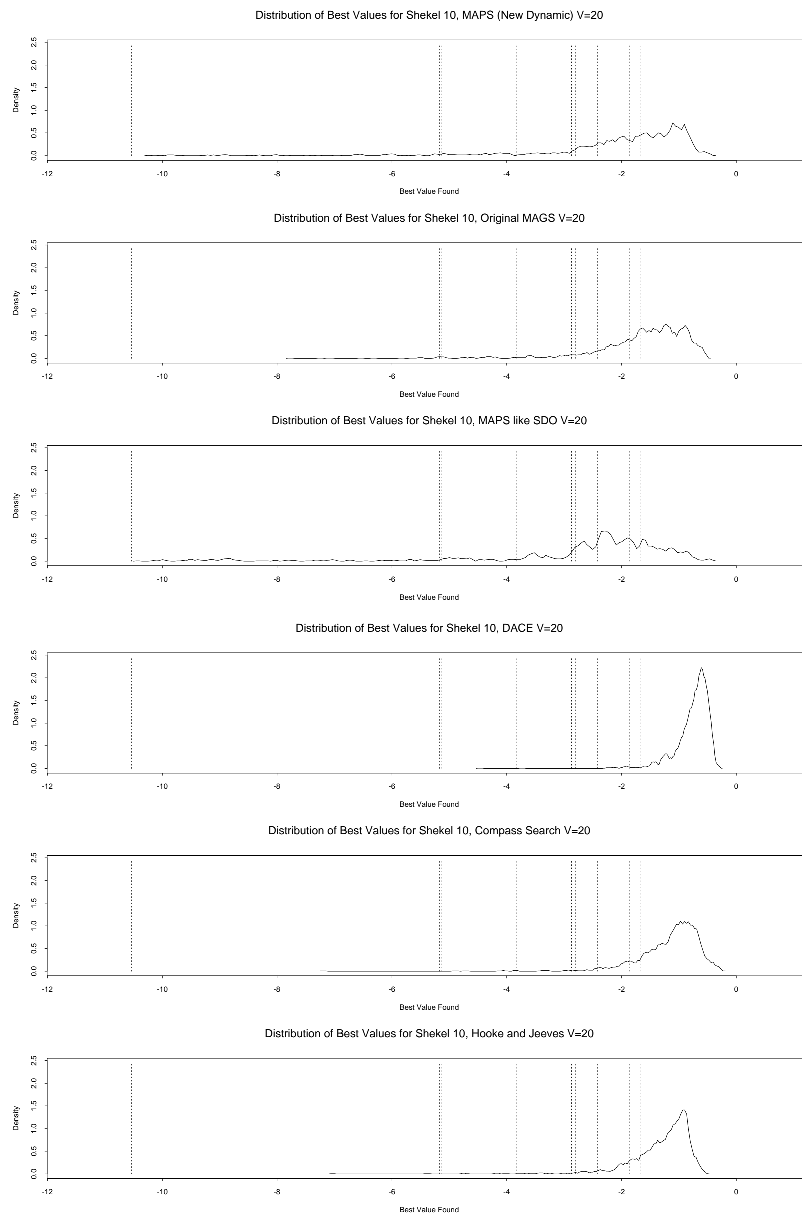

Figure 4.11: Shekel10 density plots, 20 evaluations.

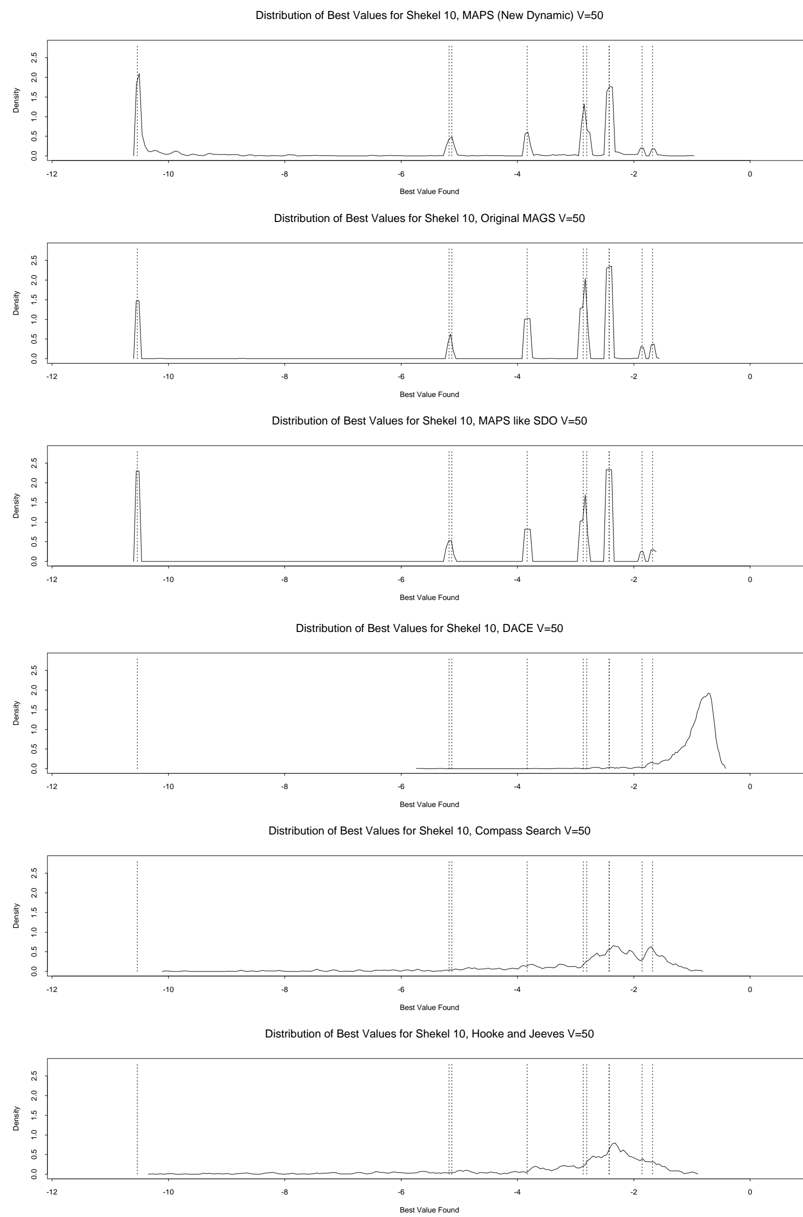

Figure 4.12: Shekel10 density plots, 50 evaluations.

4.2.3 Global Optimization and the "Curse of Dimensionality"

One of the attractive features of the krigifier family of functions is that they easily scale across a variety of dimensions. Since the functions are drawn from the same family, with a comparative density of krigifier evaluation sites among all of the dimensions, some comparison can be made across these functions. The tables in Appendix C show the number of runs returned by the algorithm that are "near" the function value of the global minimizer (the Appendix itself defines that nearness). One trend to note is that as the dimension increases, all of the algorithms tend to do worse in terms of finding the function value associated with global minimizer. This is not unexpected, as the number of minimizers in these functions have increased. What is interesting to note, however, is that in the 5-dimensional krigifier functions detailed in Figure C.4, the MAPS algorithms are the only ones identifying the function value at the global minimizer with any degree of frequency. This suggests that MAPS performs reliably as we increase the dimension of the problem to be solved. While the "curse of dimensionality" does appear to take its toll, MAPS seems to resist the "curse" much better than any of the other direct search algorithms tested.

In terms of comparisons between the MAPS algorithms, the differences are much less significant than those between MAPS and other algorithms. The choice of search criterion has an effect on how well MAPS responds to increasing dimensions. As the tables in Appendix C show, MAPS-Dynamic and MAPS-SDO improve relative to MAGS. The MAPS-Dynamic algorithm also does better than the MAPS-SDO on the lower dimension functions tested. It is our conjecture that the MAPS-Dynamic algorithm will also do better relative to SDO with an evaluation budget of say $V = 100$ on a higher dimensional function.

4.2.4 Costs of the MAPS Approach

The extra effort the Model-Assisted Pattern Search algorithm invests in building a global approximation of the function comes with a computational cost. Recalling that the algorithm is intended primarily for computationally expensive objective functions, this cost may still be reasonable.

The critical computational path in the current implementation of MAPS comes from the linear algebra routines used in the computation of the singular-value decomposition of matrices, used for the Moore-Penrose generalized inverse described in Section 2.4.1. Since parameter estimation for the kriging process requires a matrix pseudoinversion for each iteration of the estimation process, this expense accounts for the overwhelming majority of MAPS' computational cost. The time complexity for computing one of these pseudoinverses is $O(k^3)$, where k is the number of points at which the function has been evaluated. The cost grows in this fashion since we find the pseudoinverse of a $k \times k$ correlation matrix.

Figure 4.2.4 shows partial results from `gprof`, the UNIX/Linux profiling utility, for a sample run of MAPS. The algorithm was run on the 5-minimizer function of the Shekel family, with an evaluation budget of $V = 50$. For this example, the cost of evaluating the objective is insignificant, so we can concentrate on the computational overhead involved in executing MAPS. The results show quite clearly that the vast majority of time spent by the MAPS algorithm was spent in BLAS and CLAPACK routines used by the SVD routine. (The number of calls are not reported for the BLAS and CLAPACK routines, because

gathering that information would have required recompiling the BLAS and CLAPACK.) The other major contributors to the total execution time included linear algebra procedures that performed operations like vector subtraction and matrix transposition. Computation of the correlation matrix also added some computational expense as GenerateCorrelationMatrix calls the function Cor_Gauss_Isotropic in the given example. The process of generating this matrix occurs with time complexity $O(k^2p)$, where k is the number of points at which the function has been evaluated, and p is the dimension of the problem. It is precisely this matrix which is inverted during the estimation process of the kriging parameter θ. Though other computational expenses are evident, clearly the dominant factor is the cost of the pseudoinversion.

% Time	Cumulative Seconds	Self Seconds	Number of Calls	Self ms/call	Total ms/call	Function Name
20.83	2.11	2.11				dgemv_ (used by SVD)
20.04	4.14	2.03				dlasr_ (used by SVD)
14.81	5.64	1.50				dger_ (used by SVD)
11.06	6.76	1.12	1,121	1.00	1.23	pseudoinvert
6.02	7.37	0.61	597,774	0.00	0.00	Cor_Gauss_Isotropic
5.53	7.93	0.56				dlartg_ (used by SVD)
3.46	8.28	0.35				dbdsqr_ (used by SVD)
2.07	8.49	0.21				dnrm2_ (used by SVD)
1.58	8.65	0.16	599,186	0.00	0.00	Vector operator-
1.38	8.79	0.14	1,081	0.13	0.70	GenerateCorrelationMatrix
1.28	8.92	0.13	2,242	0.06	0.06	Matrix transpose
1.28	9.05	0.13	1,121	0.12	0.12	ComputeSVD

Figure 4.13: `gprof` Results for one MAPS run on Shekel5.

Chapter 5

Conclusions and Future Research

Although the results presented here are preliminary, they do strongly suggest certain conclusions that can be drawn about the MAPS algorithm. The MAPS algorithm enjoys a theoretical guarantee of asymptotic convergence to a constrained stationary point, inherited from pattern search [15] and our empirical results show this is born out in practice. On all of the problems tested, MAPS performs well, even when alloted a fairly meager budget for function evaluations.

It is also important to note that the MAPS algorithm compares favorably with the algorithms from which it draws inspiration. The iterative nature of pattern search gives MAPS an empirical advantage over a one-shot approach such as DACE. MAPS outperformed the Latin compass search, thus indicating that the empirical effectiveness shown by MAPS is more than simply a result of its initial space-filling Latin hypercube design. On the other side of the coin, the global modeling gives MAPS an advantage over the simpler model-free pattern searches. The fact that MAPS resists the "curse of dimensionality" better than both DACE and pattern search, speaks for the robustness of the algorithm. Admittedly, this advantage comes at a computational cost, but the MAPS algorithm may still be appropriate for objective functions with their own high computational cost. In this case, the effort put into modeling is recovered due to the use of fewer function evaluations.

Even though MAPS is based on a local optimization algorithm like pattern search, it has shown its worth in terms of global optimization. Our results strongly suggest that MAPS is far more capable at global minimization on a limited budgets especially as the dimension of the problem increases, than either the pattern search or DACE algorithms. The choice of the search criterion also made a difference. The MAPS-Dynamic and MAPS-SDO algorithms seemed to perform better than the MAGS algorithm. Thus, the balancing act between immediate convergence and space exploration, represented by our search criterion, does indeed seem to be worthwhile.

In terms of future research, there are several different tests we would like to see performed. First, we would like to see tests of MAPS done in higher dimensions. Five dimensional problems are still fairly small by engineering design standards, and it would be of value to see the performance of MAPS on problems of that scale. Second, we would like to see MAPS compared to other model-based optimization algorithms. Comparisons to quasi-Newton methods could certainly yield interesting results. We would also like to see MAPS compared with algorithms specifically proposed for global optimization. Though global optimization

is not MAPS' primary role, it has fared well in that context compared to pattern search and DACE. Further comparison with global optimization algorithms would thus be of interest.

Further "tweaking" of the MAPS algorithm itself could also be a line for future research. The experimental design literature has offered a variety of experimental designs which are "better" at space-filling than Latin hypercubes. Maximin and orthogonal array designs would be two such examples. Would changing the initial design effect the final results yielded by MAPS? It is certainly plausible that this may be the case. The kriging process in MAPS is currently implemented using only a constant trend. If it was known a priori that the objective function was basically quadratic, would kriging with a quadratic trend be helpful? This may be a useful endeavor, as many objective functions of interest tend to have a vaguely quadratic shape.

In terms of mitigating the computational expense of MAPS, lessening the frequency with which the estimation of the correlation parameter θ (used in the kriging), may prove to lessen the computational expense of MAPS considerably, without any appreciable deterioration in overall performance. Currently, the implementation follows the algorithm specification strictly, and re-estimates the parameter θ in each iteration. If θ does not change much between iterations, perhaps computational efficiency could be increased by simply reusing the current value of θ.

We hope to see these questions answered by future researchers. The MAPS algorithm has a great deal of potential for application to computationally expensive problems in industrial application. It is our hope that this potential is realized.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide, Third Edition.* Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[2] Basic linear algebra subprograms. `http://www.netlib.org/blas`.

[3] Andrew J. Booker, J. E. Dennis, Jr., Paul D. Frank, David B. Serafini, Virginia Torczon, and Michael W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. *Structural Optimization*, 17(1):1–13, February 1999.

[4] Dennis D. Cox and Susan John. SDO: A statistical method for global optimization. In Natalia M. Alexandrov and M. Y. Hussaini, editors, *Multidisciplinary Design Optimization: State-of-the-Art*, pages 315–329, Philadelphia, 1997. SIAM.

[5] J. Demmel, X. Li, C. Puscasiu, S. Timson, S. Ostrouchov, and J. Toth. C linear algebra package. `http://www.netlib.org/clapack`.

[6] L. C. W. Dixon and G. P. Szegö. The global optimisation problem: An introduction. In L. C. W. Dixon and G. P. Szegö, editors, *Towards Global Optimisation 2*, pages 1–15. Elsevier North-Holland, New York, 1978.

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

[8] Elizabeth D. Dolan, Robert Michael Lewis, and Virginia J. Torczon. On the local convergence properties of pattern search. In preparation.

[9] J. K. Hartman. Some experiments in global optimization. *Naval Research Logistics Quarterly*, 20:569–575, 1973.

[10] R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. *Journal of the Association for Computing Machinery (ACM)*, 8(2):212–229, April 1961.

[11] Robert A. Horn and Charles R. Johnson. *Matrix Analysis.* Cambridge University Press, Cambridge, 1985.

[12] W. G. Houf, J. F. Grcar, and W. G. Breiland. A model for low pressure chemical vapor deposition in a hot-wall tubular reactor. *Materials Science Engineering, B, Solid State Materials for Advanced Technology*, 17:163–171, 1993.

[13] J. R. Koehler and A.B. Owen. Computer experiments. *Handbook of Statistics*, 13:261–308, 1996.

[14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *Association for Computing Machinery (ACM) Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[15] Robert Michael Lewis and Virginia J. Torczon. Pattern search algorithms for bound constrained minimization. *SIAM Journal on Optimization*, 9(4):1082–1099, 1999.

[16] C. D. Moen, P. A. Spence, J. C. Meza, and T. D. Plantenga. Automatic differentiation for gradient-based optimization of radiatively heated microelectronics manufacturing equipment. In *Proceedings of the 6th AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Bellevue, WA*, September 4–6, 1996.

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

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

[19] R. M. Lewis, Virginia Torczon, and Michael W. Trosset. Why pattern search works. *Optima*, 59:1–7, 1998.

[20] Roldan Pozo. `http://www.math.nist.gov/tnt`. Template numerical toolkit: A numeric library for scientific computing in c++.

[21] MathWorks Inc. `http://www.mathworks.com`. Internal `help` for matlab version 5.3.1.29215a.

[22] Virginia Torczon. On the convergence of pattern search algorithms. *SIAM Journal on Optimization*, 7(1):1–25, February 1997.

[23] Virginia Torczon and Michael W. Trosset. From evolutionary operation to parallel direct search: Pattern search algorithms for numerical optimization. *Computing Science and Statistics*, 29:396–401, 1998.

[24] Virginia Torczon and Michael W. Trosset. Using approximations to accelerate engineering design optimization. In *Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO*, September 2–4, 1998. AIAA Paper 98-4800.

[25] Michael W. Trosset. Optimization on a limited budget. In *1998 Proceedings of the Section on Physical and Engineering Sciences*. American Statistical Association, 1998.

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

[27] Michael W. Trosset and Virginia Torczon. Numerical optimization using computer experiments. Technical Report 97–38, Institute for Computer Applications in Science and Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia 23681–2199, 1997.

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

[29] W. J. Welch and J. Sacks. A system for quality improvement via computer experiments. *Communications in Statistics — Theory and Methods*, 20:477–495, 1991.

[30] G. S. Watson. Smoothing and interpolation by kriging and with splines. *Mathematical Geology*, 16:601–615, 1984.

Appendix A

Graphs of Results

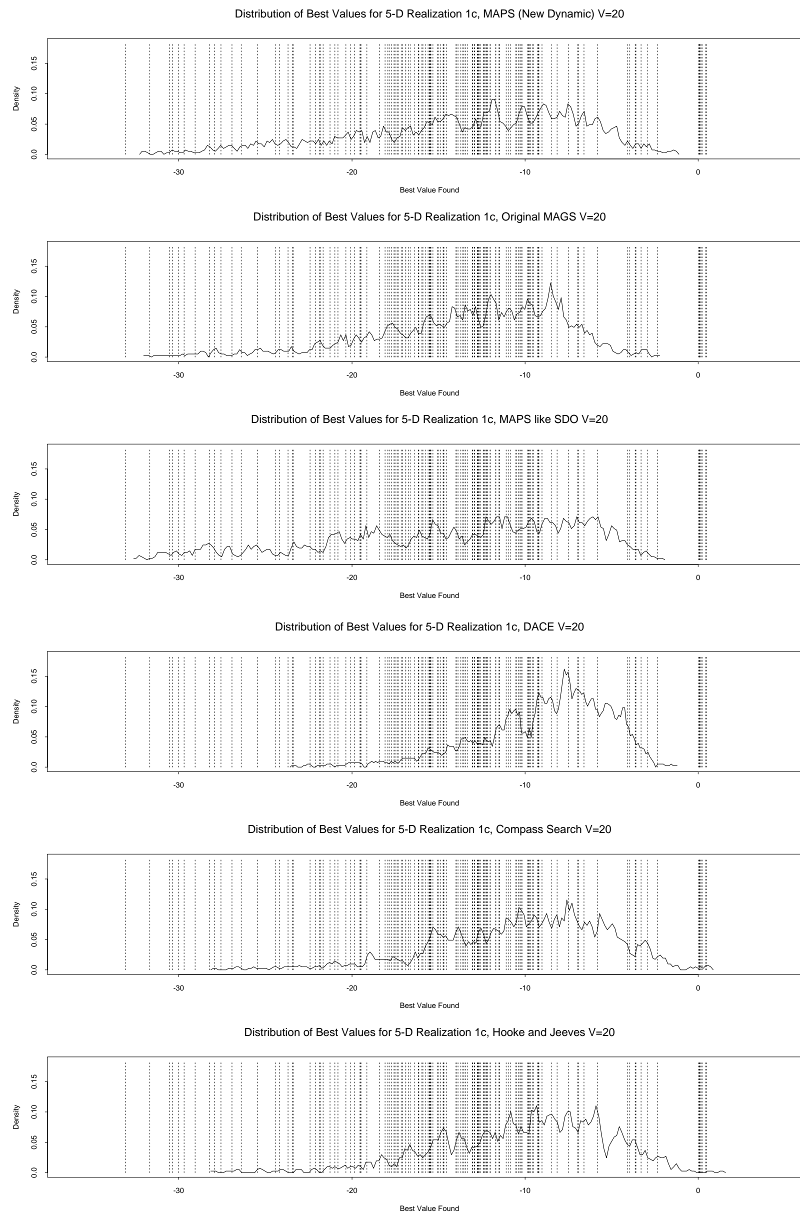

Figure A.1: 5-D Krigifier(1C) density plots, 20 evaluations.

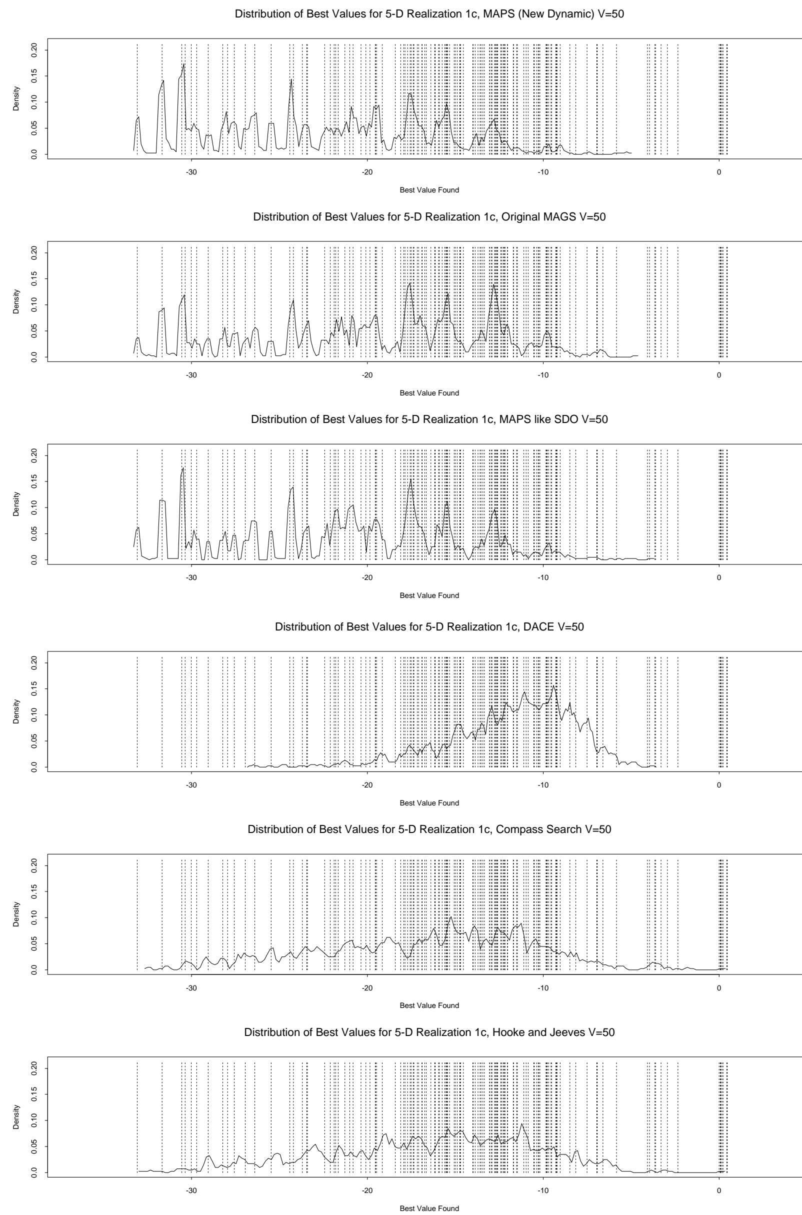

Figure A.2: 5-D Krigifier(1C) density plots, 50 evaluations.

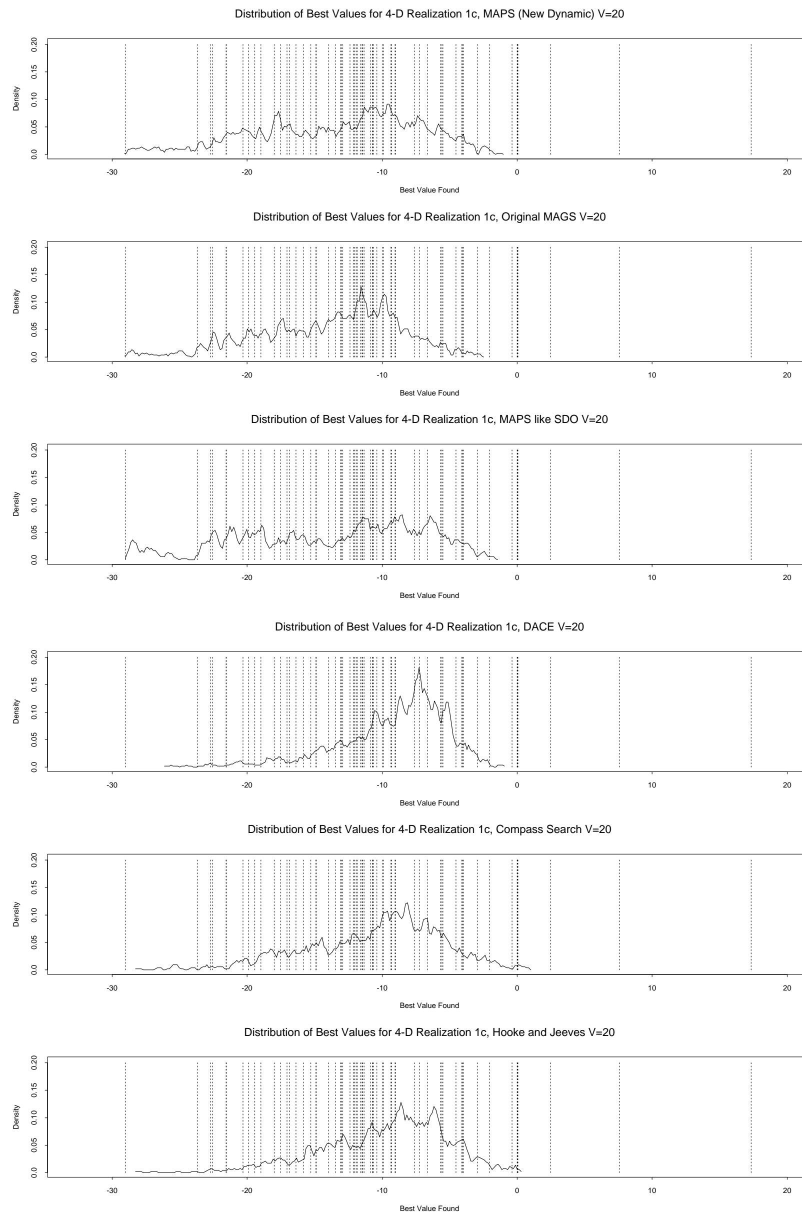

Figure A.3: 4-D Krigifier(1C) density plots, 20 evaluations.

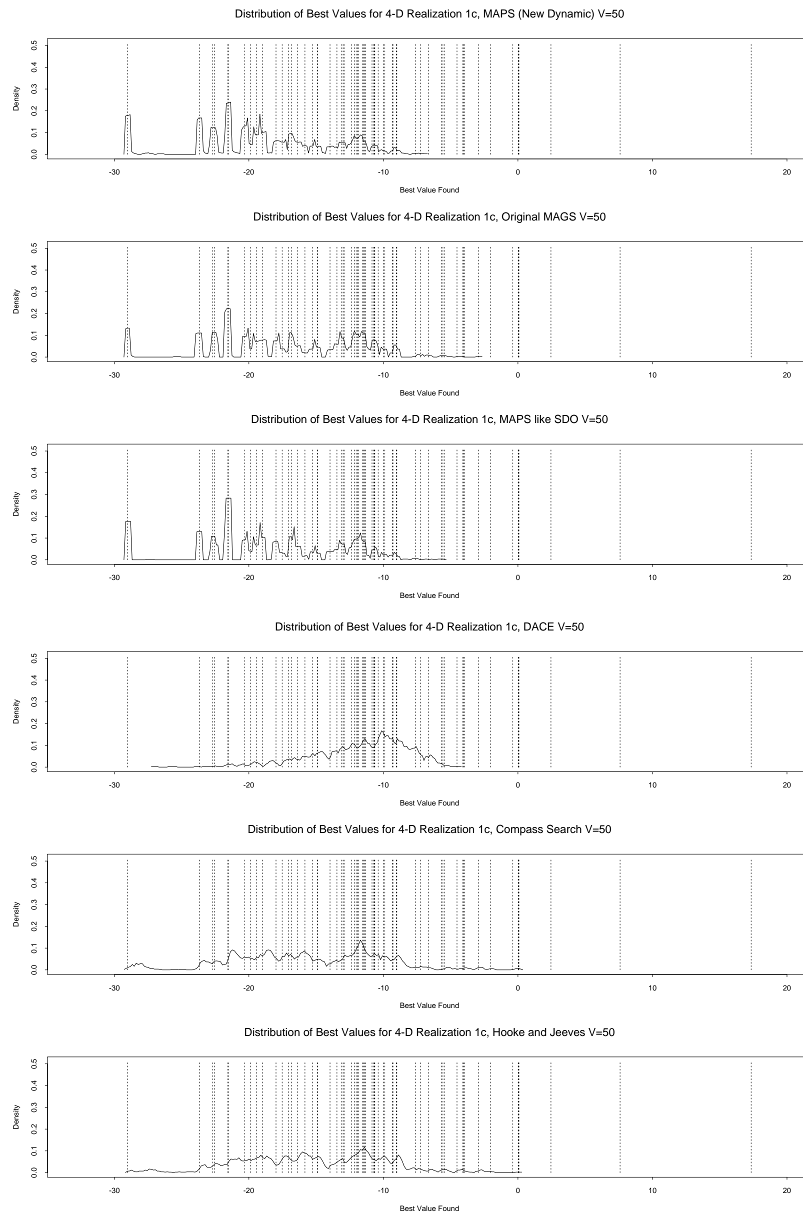

Figure A.4: 4-D Krigifier(1C) density plots, 50 evaluations.

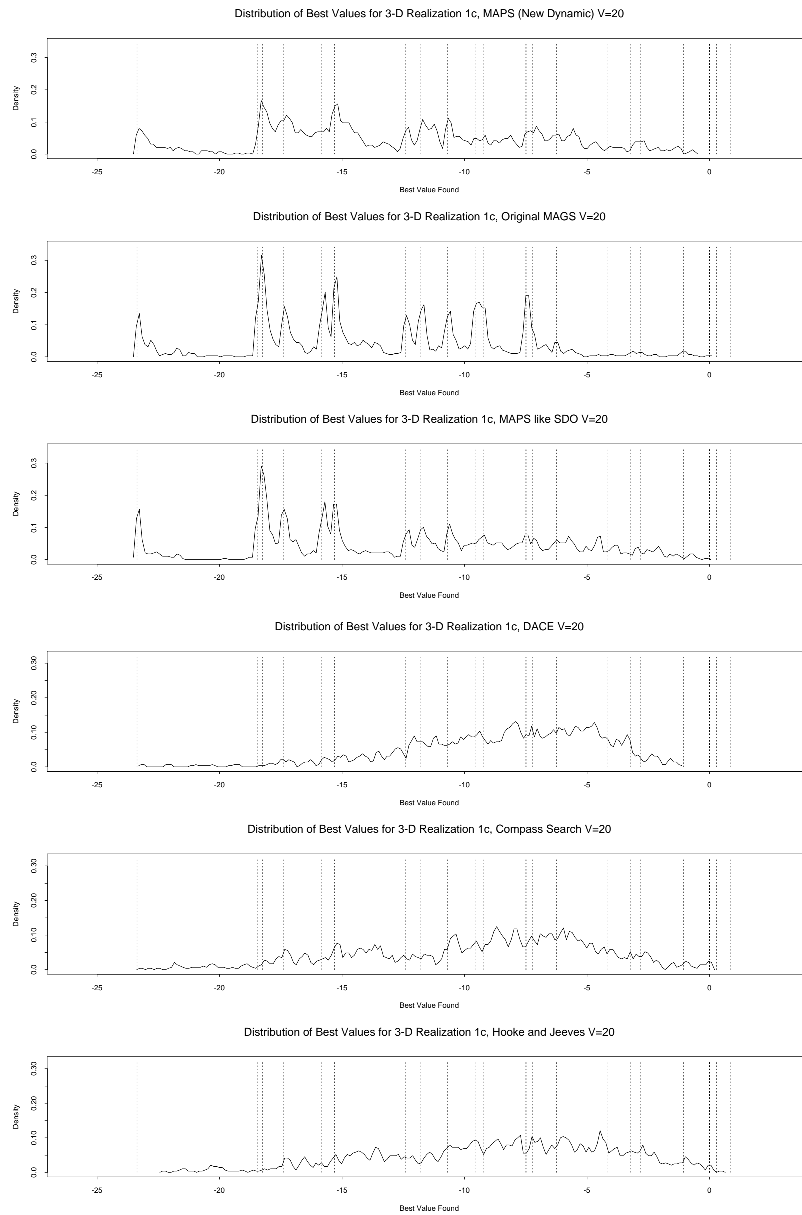

Figure A.5: 3-D Krigifier(1C) density plots, 20 evaluations.

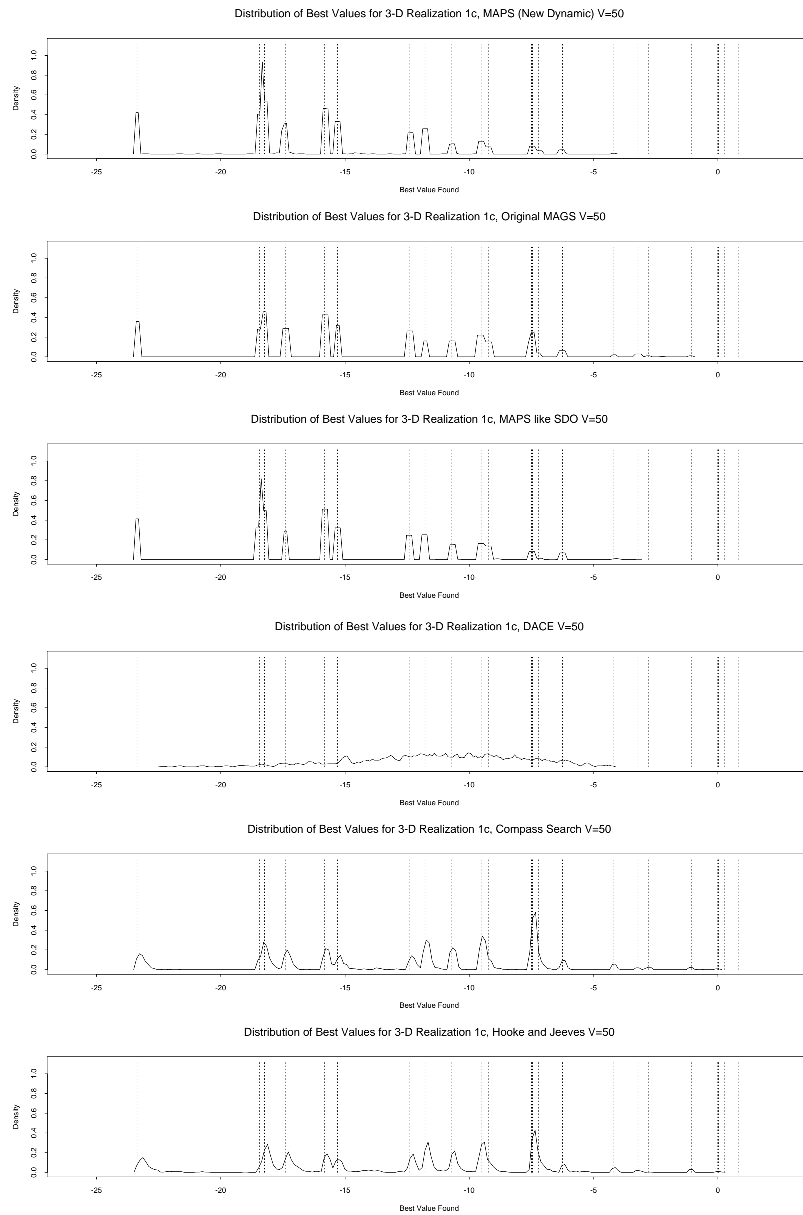

Figure A.6: 3-D Krigifier(1C) density plots, 50 evaluations.

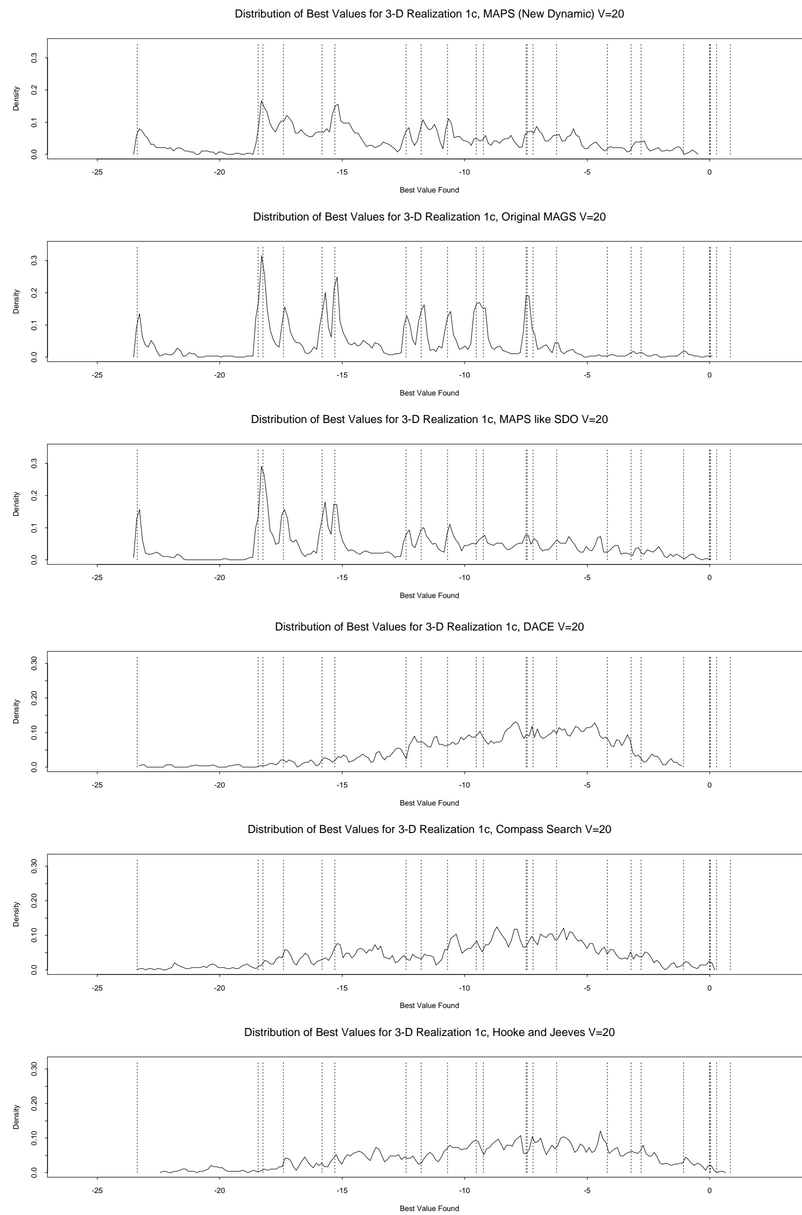

Figure A.7: 3-D Krigifier(1Q) density plots, 20 evaluations.

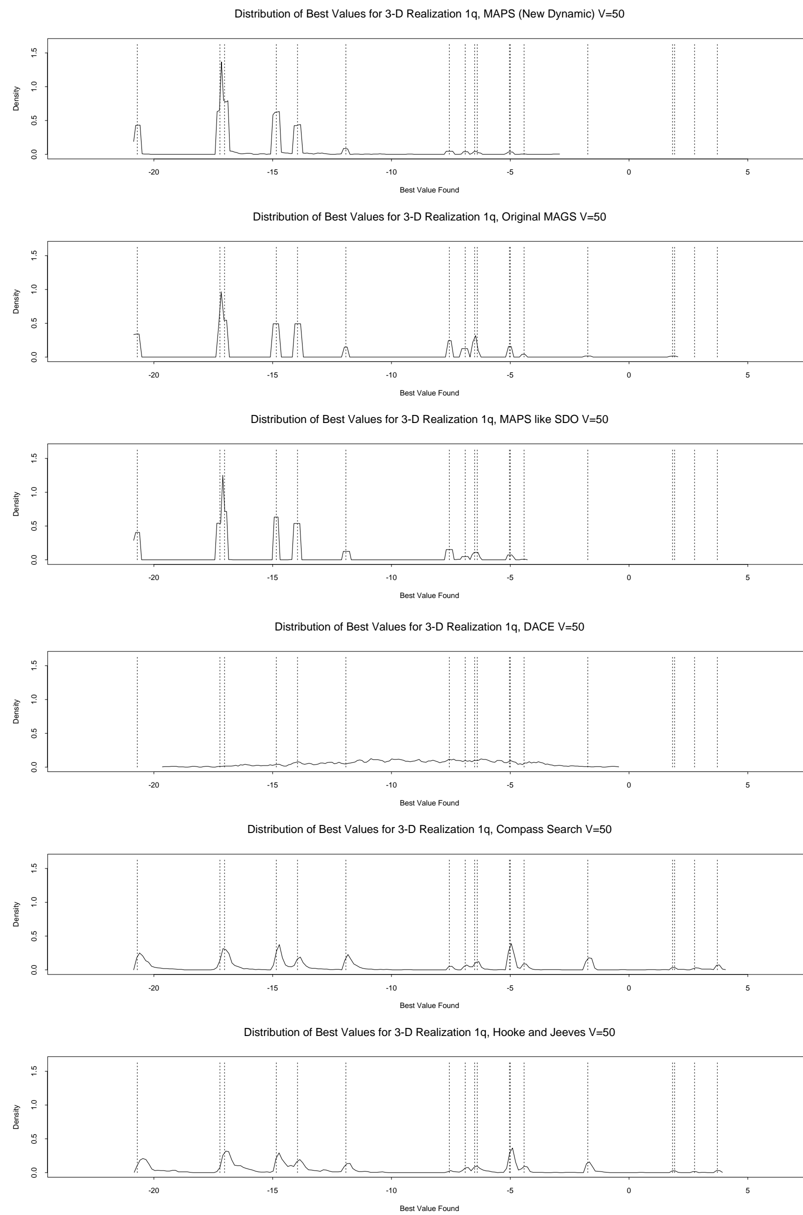

Figure A.8: 3-D Krigifier(1Q) density plots, 50 evaluations.

Appendix B

Global Comparisons with DACE

This appendix is composed of tables detailing the results of each of the seven optimization algorithms run. The baseline for the comparison shown is the 25th percentile of results returned by the DACE algorithm, for a budget of $V = 50$ evaluations. The number listed in the columns is the number of runs which returned a value better than the baseline value, for a budget of $V = 50$ evaluations. The results are shown for each of the ten krigifier realizations for all dimensions tested. The realizations denoted with a "c" possess a constant trend, and the realizations denoted with a "q" possess a quadratic trend. Results are also shown for the Shekel and Hartman families of test functions.

Figure B.1: Krigifier 2-D — Runs better than DACE's 25th percentile.

Algorithm/Realization	R1c	R1q	R2c	R2q	R3c	R3q	R4c	R4q	R5c	R5q
MAPS(New Dynamic)	891	927	514	466	503	760	656	909	389	421
MAGS	685	501	199	158	109	202	219	301	147	152
MAPS like SDO	855	806	372	245	432	639	496	739	347	315
Hooke & Jeeves	426	283	118	61	181	274	167	228	75	97
Compass Search	437	191	94	60	191	251	172	215	57	62
Latin Compass Search	717	588	229	143	372	499	448	584	169	86
DACE	250	250	250	250	250	250	250	250	250	250

Figure B.2: Krigifier 3-D — Runs better than DACE's 25th percentile.

Algorithm/Realization	R1c	R1q	R2c	R2q	R3c	R3q	R4c	R4q	R5c	R5q
MAPS(New Dynamic)	724	947	839	764	505	343	825	851	782	900
MAGS	606	728	711	628	396	244	744	681	616	719
MAPS like SDO	673	882	804	735	468	286	810	811	725	874
Hooke & Jeeves	420	680	643	668	398	329	578	628	464	682
Compass Search	391	634	626	618	357	201	591	653	444	639
Latin Compass Search	619	813	698	643	464	301	715	687	611	676
DACE	250	249	250	250	250	250	250	250	250	250

Figure B.3: Krigifier 4-D — Runs better than DACE's 25th percentile.

Algorithm/Realization	R1c	R1q	R2c	R2q	R3c	R3q	R4c	R4q	R5c	R5q
MAPS(New Dynamic)	804	843	816	862	860	875	812	896	814	853
MAGS	673	777	633	721	741	801	660	790	691	727
MAPS like SDO	740	863	758	833	807	891	757	906	766	858
Hooke & Jeeves	580	657	477	521	613	687	493	558	536	586
Compass Search	591	654	492	507	596	667	485	528	536	566
Latin Compass Search	643	724	603	636	654	725	637	696	630	697
DACE	250	250	250	250	250	250	250	250	250	250

Figure B.4: Krigifier 5-D — Runs better than DACE's 25th percentile.

Algorithm/Realization	R1c	R1q	R2c	R2q	R3c	R3q	R4c	R4q	R5c	R5q
MAPS(New Dynamic)	900	896	939	870	892	872	889	859	919	891
MAGS	778	871	848	891	795	855	813	841	792	843
MAPS like SDO	851	933	932	938	863	915	888	915	875	920
Hooke & Jeeves	646	642	650	625	620	637	604	629	612	654
Compass Search	663	627	687	661	670	643	631	647	626	659
Latin Compass Search	722	726	721	752	712	768	683	734	736	776
DACE	250	250	250	250	250	250	250	250	250	250

Figure B.5: Shekel Family — Runs better than DACE's 25th percentile.

Algorithm/Realization	R5	R7	R10
MAPS(New Dynamic)	1000	1000	999
MAGS	1000	1000	1000
MAPS like SDO	1000	1000	1000
Hooke & Jeeves	999	995	993
Compass Search	989	997	995
Latin Compass Search	987	991	985
DACE	250	250	250

Figure B.6: Hartman Family — Runs better than DACE's 25th percentile.

Algorithm/Realization	R3	R6
MAPS(New Dynamic)	981	999
MAGS	897	960
MAPS like SDO	954	1000
Hooke & Jeeves	404	233
Compass Search	385	172
Latin Compass Search	467	169
DACE	249	250

Appendix C

"Nearness" to the Global Minimizer

This appendix is composed of tables detailing the performance at global optimization for each of the algorithms tested. For each algorithm, for all of the 1,000 runs with evaluation budget $V = 50$, the number of runs returning results "near" to the function value corresponding to the global minimizer is listed. The "nearness" criterion used for a given result is that the point must be less than half the distance to the function value of the next best stationary point. The results are shown for each of the ten krigifier realizations for all dimensions tested. The realizations denoted with a "c" possess a constant trend, and the realizations denoted with a "q" possess a quadratic trend. Results are also shown for the Shekel and Hartman families of test functions.

Figure C.1: Krigifier 2-D — Runs "near" the function value of the global minimizer.

Algorithm/Realization	R1c	R1q	R2c	R2q	R3c	R3q	R4c	R4q	R5c	R5q
MAPS(New Dynamic)	897	928	515	467	515	768	660	909	401	427
MAGS	685	501	201	160	160	274	270	396	198	192
MAPS like SDO	855	806	372	245	432	645	496	748	347	323
Hooke & Jeeves	426	283	118	61	181	274	167	228	75	97
Compass Search	437	191	94	60	191	251	172	215	57	62
Latin Compass Search	722	592	229	144	378	504	456	592	171	88
DACE	566	583	259	264	411	463	386	451	381	374

Figure C.2: Krigifier 3-D — Runs “near” the function value of the global minimizer.

Algorithm/Realization	R1c	R1q	R2c	R2q	R3c	R3q	R4c	R4q	R5c	R5q
MAPS(New Dynamic)	123	130	73	118	218	213	181	421	211	156
MAGS	102	102	62	95	155	129	126	257	157	120
MAPS like SDO	118	121	64	92	198	169	149	367	180	149
Hooke & Jeeves	79	143	39	98	210	247	60	140	9	43
Compass Search	72	144	44	84	184	151	75	154	10	29
Latin Compass Search	68	78	62	117	218	203	119	166	7	66
DACE	6	5	3	4	43	39	9	1	0	8

Figure C.3: Krigifier 4-D — Runs “near” the function value of the global minimizer.

Algorithm/Realization	R1c	R1q	R2c	R2q	R3c	R3q	R4c	R4q	R5c	R5q
MAPS(New Dynamic)	101	49	24	216	37	18	163	354	28	116
MAGS	70	44	13	151	25	19	126	211	23	113
MAPS like SDO	94	49	16	201	29	17	155	315	27	112
Hooke & Jeeves	23	20	1	50	5	0	25	48	7	25
Compass Search	38	31	1	59	5	0	40	61	6	30
Latin Compass Search	45	34	0	44	2	0	40	61	2	82
DACE	1	0	0	1	1	0	0	0	0	0

Figure C.4: Krigifier 5-D — Runs “near” the function value of the global minimizer.

Algorithm/Realization	R1c	R1q	R2c	R2q	R3c	R3q	R4c	R4q	R5c	R5q
MAPS(New Dynamic)	30	114	30	38	55	47	21	58	67	16
MAGS	17	60	25	29	33	33	12	39	49	11
MAPS like SDO	26	114	38	49	46	47	29	73	70	14
Hooke & Jeeves	2	2	0	2	10	3	0	4	2	0
Compass Search	2	4	0	1	9	6	0	8	2	0
Latin Compass Search	0	0	0	1	3	1	0	7	0	0
DACE	0	0	0	0	0	0	0	0	0	0

Figure C.5: Shekel Family — Runs “near” the function value of the global minimizer.

Algorithm/Realization	R5	R7	R10
MAPS(New Dynamic)	276	374	369
MAGS	261	213	181
MAPS like SDO	260	293	280
Hooke & Jeeves	58	39	32
Compass Search	28	25	19
Latin Compass Search	13	17	12
DACE	0	0	0

Figure C.6: Hartman Family — Runs “near” the function value of the global minimizer.

Algorithm/Realization	R3	R6
MAPS(New Dynamic)	983	621
MAGS	897	566
MAPS like SDO	954	664
Hooke & Jeeves	560	0
Compass Search	525	0
Latin Compass Search	812	0
DACE	629	0

Figure C.7: T-WAFER — Runs “near” the function value of the global minimizer.

Algorithm/Realization	R
MAPS(New Dynamic)	1000
Hooke & Jeeves	972
Compass Search	935

Appendix D

Source Code for Approximation

D.1 Headers

D.1.1 Approximate

```
/****
Approximate Class - 5/24/99 cmsief
This is an abstract base class for approximations
****/
#ifndef _MAPS_APPROX_
#define _MAPS_APPROX_

#include "maps_general.h"

#define A_KRIG_APPROX 1
#define A_NULL_APPROX 2

class Approximate {

public:
  Approximate();
  Approximate(Approximate &A);
  Approximate(long dim);
  /*call AutoGenerate*/
  virtual ~Approximate();
  virtual Approximate& operator=(Approximate &A)=0;
  //  virtual void copy(Approximate &A);

  /*****QUERY FUNCTIONS*****/

  virtual long GetDim() const =0;
  /****
  OUTPUT: Returns the dimension of the space.
  ****/
```

```
  virtual long GetType() const=0;
  /****
  OUTPUT: returns the proper code for type of approximation
  ****/

  virtual void debug(char *code) const=0;
  /****
  EFFECT: Outputs a state dump.
  ****/

  /*****ACTION FUNCTIONS*****/

  virtual bool EvaluateApprox(const mp_vector &x, const pt_collect &Eval_c,
      double* Approx, double* D_c)=0;
  /****
  INPUT: The location to evaluate at, the location of evaluated sites, and two
  double pointers for return values.
  OUTPUT: Returns true if the operation was sucessful.
  EFFECT: It Approixmates, and returns the Approximation and Design Criterion
  Values in the two pointers
  ****/

  virtual bool UpdateApprox(pt_collect &Eval_c, mp_vector &f_values, long
                            numpoints, double delta,const Bounds &b)=0;
  /****
  INPUT: The collection of evaluated sites, their values, the number of points
  evaluated, delta and the bounds.
  OUTPUT: Returns true if the operation was sucessful.
  EFFECT: This updates the approximation.
  ****/

};/*end class*/

#endif
```

D.1.2 KrigApprox

```
/****
KrigApprox Class - 6/7/99 cmsief
Derived from Approximate.  This class is a kriging-based function approximation.
NOTE: You must call Update before Evaluate the first time.
****/
#ifndef _MAPS_KRIG_APPROX_
#define _MAPS_KRIG_APPROX_

#include "maps_general.h"
#include "ParamEstimate.h"
```

```
#include "Approximate.h"

/*estimation trend codes*/
#define TREND_CONSTANT 0
#define TREND_QUADRATIC 1
#define TREND_CUSTOM 2

class KrigApprox: public Approximate {

public:
  KrigApprox();
  KrigApprox(long dim);
  /*Call SetDefault*/
  KrigApprox(KrigApprox &K);
  KrigApprox(ParamEstimate &PE1, long dim);
  virtual ~KrigApprox();
  KrigApprox& operator= (Approximate &A)
    {if(A.GetType()==A_KRIG_APPROX) *this=(KrigApprox&)A;
     return *this;}

  KrigApprox& operator= (KrigApprox &K);

  /*****RESET FUNCTIONS*****/

  void ResetKrigApproxState(ParamEstimate &PE1);
  /****
  INPUT: ParamEstimate object.
  EFFECT: Resets this object's state.
  ****/

  bool ResetEstimateConstantTrend();
  /****
  OUTPUT: Always returns true.
  EFFECT: Chooses Estimation with a constant trend.  Since the trend is a
  constant, no parameter is needed.
  ****/

  bool ResetEstimateQuadraticTrend(mp_vector& a);
  /****
  INPUT: A vector signifying quadratic trend.
  OUTPUT: True if the vector is reasonable.
  EFFECT: Chooses Estimation with that quadratic trend.
  NOTE: IMPLEMENTATION DEFERRED!
  ****/

  bool ResetEstimateCustomTrend(void(*fcn)(const mp_vector &, mp_vector&),
long fcn_dim);
```

```
/****
INPUT: A function pointer, the vector-valued trend function and its dimension.
OUTPUT: Always returns true.
EFFECT: Chooses Custom trend Estimation.
****/

bool ResetCorrelationFamily(long cor);
/****
INPUT: Correlation family number (see ParamEstimate.h).
OUTPUT: true if correlation family number is valid.
EFFECT: return PE.ResetCorrelationFamily(cor);
****/

/*****QUERY FUNCTIONS*****/

long GetDim() const;
/****
OUTPUT: Returns the dimension of the space.
****/

long GetType() const {return A_KRIG_APPROX;}
/****
OUTPUT: returns the proper code for type of approximation
****/

void debug(char *code) const;
/****
EFFECT: Outputs a state dump to stdout.
****/

/*****ACTION FUNCTIONS*****/

bool EvaluateApprox(const mp_vector &x, const pt_collect &Eval_c,
    double* Approx, double* D_c);
/****
INPUT: The location to evaluate at, the location of the previously
evaluated sites, and two double pointers for return values.
OUTPUT: Returns true if the operation was successful.
EFFECT: Using kriging to evaluate the approximation at this point.  It also
evaluates the design criterion.
****/

bool UpdateApprox(pt_collect &Eval_c, mp_vector &f_values, long
                  numpoints, double delta,const Bounds &b);
/****
INPUT: The collection of evaluated sites, their values, the number of points
evaluated, the iteration number, the current delta and the bounds;
```

```
  OUTPUT: Returns true if the operation was successful.
  EFFECT: This updates the weighting scheme and then rekrigs the function,
  calling for parameter estimation as needed.  Also, it updates the Design
  Criterion.
  ****/

private:

  void SetDefault(long dim);
  /****
  INPUT: Dimension of space.  Used for parameter estimation.
  EFFECT: Constant Trend Estimation, and ParamEstimate's default family.
  ****/

  double EvaluateTrend(const mp_vector& x);
  /****
  INPUT: A mp_vector, x
  OUTPUT: The value of the trend at that particular point.  This is assembled
  from the betas, as appropriate.
  ****/

  long P;               /*Dimension of Space*/
  mp_vector* v;         /*from estimation*/
  mp_vector* theta;     /*Estimated*/
  mp_vector* beta;      /*Estimated*/
  double sigma2;        /*Estimated*/
  ParamEstimate* PE;
  mp_vector* quad;      /*For quadratic estimate*/
  void(*custom_f)(const mp_vector &, mp_vector&);/*For custom function estimate*/
  int estimation_trend;/*see top for codes*/
  pt_collect* MLE;      /*the MLE matrix... from estimation*/
  long fdim;            /*dimension of the vector-valued function fcn*/
  bool first_time;
};/*end class*/

#endif
```

D.1.3 ParamEstimate

```
/****
ParamEstimate Class - 6/1/99 cmsief
This wonderfully useful class should take care of all of your parameter
estimation needs.
****/

#ifndef _MAPS_PARAM_
#define _MAPS_PARAM_
```

```
#include "maps_general.h"
#include "PatternSearch.h"
#include "f2c.h"

/*CLAPACK Double-Precision SVD routine*/
extern "C" {
int dgesvd_(char *, char *, long*, long*, double *, long *, double*,
double *, long *, double *, long *, double *, long *, long *);
};

#define START_THETA_STEP (0.8 * GetThetaUpperBound(curr_delta))
#define STOP_THETA_STEP (0.00001 * GetThetaUpperBound(curr_delta))

/*correlation family numbers*/
/*There are to be used to pick your correlation function type.
 See the private member functions for more details.*/
#define COR_EXP_ISOTROPIC 0
#define COR_GAUSS_ISOTROPIC 1
#define COR_EXP_PRODUCT 2
#define COR_GAUSS_PRODUCT 3
#define COR_CUBIC_ISOTROPIC 4

class ParamEstimate {

public:
  ParamEstimate();
  ParamEstimate(ParamEstimate &PE);
  ParamEstimate(long dim);
  /*Prep work and call SetDefault. This is the default.*/
  ParamEstimate(int correlate, long dim, PatternSearch &PS1);
  /*WARNING: REQUIRES A FULLY INITIALIZED PATTERNSEARCH*/
  ~ParamEstimate();
  ParamEstimate& operator=(ParamEstimate &PE);


  /*****RESET FUNCTIONS*****/

  void ResetParamEstimateState(int correlate, long dim, PatternSearch &PS1);
  /****
  INPUT: Correlation family number (see above), dimension of space, and
  a PatternSearch.
  EFFECT: Resets all the major state variables.
  ****/

  void ResetPatternSearch(PatternSearch &PS1);
  /****
```

```
INPUT: PatternSearch Object.
EFFECT: Sets PS to be this PatternSearch.
****/

bool ResetCorrelationFamily(int correlate);
/****
INPUT: Correlation family number (see above).
OUTPUT: True if number is valid.
EFFECT: Sets function pointer r to correct function.
****/

bool ResetDim(long dim);
/****
INPUT: Dimension of the space.
OUTPUT: True if dimension is valid.
EFFECT: P=dim;
****/

/****QUERY FUNCTIONS****/

long GetDim() const;
/****
OUTPUT: Returns the dimension of the space.
****/

long GetThetaSize() const;
/****
OUTPUT: Returns the size of the theta vector
****/

void debug(char *code) const;
/****
EFFECT: Does a complete state dump to stdout.
****/

static double GetThetaUpperBound(double delta);
/****
INPUT: delta.
OUTPUT: Returns a double representing the upper bound for
theta = -log(0.8) / curr_delta^2.
****/

/*****ESTIMATION FUNCTIONS*****/

bool EstimateConstantTrend(pt_collect &pts, mp_vector &fvals, double delta,
   double* beta, double* sigma2, mp_vector* theta,
   pt_collect* MLE_Matrix, mp_vector* v);
```

```
/****
INPUT: Collection of evaluated points, their values, delta , and pointers for
the parameters to be returned.  theta should come in with a 'guess' value for
the optimizer to start with.
OUTPUT: Returns true if the operation was successful.
EFFECT: This estimates the parameters, assuming a constant trend, and returns
them.
****/

bool EstimateQuadraticTrend(pt_collect &pts, mp_vector &fvals, double delta,
                            mp_vector &a, mp_vector* beta,
                            double* sigma2, mp_vector* theta, pt_collect*
                            MLE_Matrix, mp_vector* v);
/****
INPUT: Collection of evaluated points, their values, and pointers for the
parameters to be returned. theta should come in with a 'guess' value for the
optimizer to start with.
OUTPUT: Returns true if the operation was successful.
EFFECT: This constructs the appropriate function vector and calls
EstimateCustom, using the Correlation function specified as the function
pointer parameter.
NOTE: IMPLEMENTATION DEFERRED TO A LATER DATE.  DON'T CALL THIS.
****/

bool EstimateCustomTrend(pt_collect &pts, mp_vector &fvals, double delta, void
                         (*fcn2)(const mp_vector &, mp_vector&), long
                         dimf, mp_vector* beta, double* sigma2, mp_vector* theta,
                         pt_collect* MLE_Matrix, mp_vector* v);
/****
INPUT: Collection of evaluated points, their values, and pointers for the
parameters to be returned.  There is also a function pointer for the
'a' function, and the dimension of the vector it returns.  theta should come
in with a 'guess' value for the optimizer to chew on.
OUTPUT: Returns true if the operation was successful.
EFFECT: This estimates the function.
****/

friend void OptimizeMLEConstant(void* PE,long dimtheta, double *x,
double &f, int &success)
  {(*(ParamEstimate*)PE).MLEConstant(dimtheta,x,f,success);}
/****
INPUT: Modified PatternSearch calling scheme.  See PatternSearch.h
EFFECT: Indirection to get around C++'s inability to let function pointer
point to class member functions.  This gets optimized by PatternSearch.
****/

friend void OptimizeMLECustom(void * PE, long dimtheta, double *x,
```

```
double &f, int &success)
  {(*(ParamEstimate*)PE).MLECustom(dimtheta,x,f,success);}
  /****
  INPUT: Modified PatternSearch calling scheme.  See PatternSearch.h
  EFFECT: Indirection to get around C++'s inability to let function pointer
  point to class member functions.  This gets optimized by PatternSearch.
  ****/

  double EvaluateMyCorrelation(long dim, const mp_vector &x1,
       const mp_vector &x2, const mp_vector &theta);
  /****
  INPUT: Dimesion of space, two points, and a 'vector' theta.  For the
  Isotropic functions, this theta is a one dimensional vector, for the
  Product functions, this is a P-dimensional vector.
  OUTPUT: Correlation between the two points.
  ****/

private:

  void SetDefault(long dim);
  /****
  INPUT: Dimension of the space.
  EFFECTS: This will use a happy default set of values to set up the parameter
  estimation.
  ****/

  /*For the Optimization*/

  void MLEConstant(long dimtheta, double *x, double &f, int &success);
  /****
  INPUT: Modified PatternSearch calling scheme.  See PatternSearch.h
  EFFECT: Evaluates the MLE for a given theta.  Uses the constant trend.  This
  is to be optimized by the Pattern Search, via a little bit of indirection.
  ****/

  void MLECustom(long dimtheta, double *x, double &f, int &success);
  /****
  INPUT: Modified PatternSearch calling scheme.  See PatternSearch.h
  EFFECT: Evaluates the MLE for a given theta.  Uses the custom trend.  This is
  to be optimized by the Pattern Search, via a little bit of indirection.
  ****/

  bool pseudoinvert(pt_collect &M, mp_vector* d);
  /****
  INPUT: A matrix M, and a pointer to a mp_vector.
  OUTPUT: true if svd can be computed, else false.
  EFFECT: Replaces M with the pseudoinverse of M.  If d!=NULL, an mp_vector with
```

```
  the one over the singular values, with those singular values below TOLERENCE
  instead to zero, is returned.  The matrix has to be *SYMMETRIC AND SQUARE* for
  this to work.
  ****/

  inline void GenerateCorrelationMatrix(pt_collect &R, const mp_vector &theta);
  /****
  INPUT: A np x np pt_collect, and the correlation family parameter vector, theta.
  EFFECT: Replaces R with the appropriate correlation matrix.
  ****/

  bool ComputeSVD(const pt_collect &A, pt_collect* U, pt_collect* D,
                  pt_collect* VT);
  /****
  INPUT: Matrix A, to take the SVD of, and pointers to three matrices that will
  contain what is returned by the SVD routine.
  OUTPUT: true if CLAPACK's dgesvd returns ok.
  EFFFECT: Calls CLAPACK's degsvd routine to compute the Singular Value
  Decomposition(SVD) of A.  Recall that a SVD of A yields U, D, and V',
  s.t. A=U*D* V', where U and V are orthogonal matrices, and D is a diagonal
  matrix with the singular values on its main diagonal.
  ****/

/*
  Correlation Functions
  INPUT: Dimesion of space, two points, and a 'vector' theta.  For the
  Isotropic functions, this theta is a one dimensional vector, for the
  Product functions, this is a P-dimensional vector.
  NOTE: You specify these by use of the integer #defined correlation family
  numbers, and ResetCorrelationFamily().  These are static class member functions.
*/

static double Cor_Exp_Isotropic(long dim, const mp_vector &x1,
const mp_vector &x2,const mp_vector &theta);
  /*Exponential Isotropic Correlation Family
    r_theta(s,t) = exp(-theta ||s-t||)
  */
static double Cor_Gauss_Isotropic(long dim, const mp_vector &x1,
  const mp_vector &x2,const mp_vector &theta);
  /*Gaussian Isotropic Correlation Family
    r_theta(s,t) = exp(-theta ||s-t||)^2
  */
static double Cor_Exp_Product(long dim, const mp_vector &x1,
      const mp_vector &x2, const mp_vector &theta);
  /*Exponential Product Correlation Family
    r_theta(s,t) = \prod_{j=1}^{P} exp(-theta_j * |s_j - t_j|)
  */
```

```
static double Cor_Gauss_Product(long dim, const mp_vector &x1,
const mp_vector &x2,const mp_vector &theta);
  /*Gaussian Product Correlation Family
    r_theta(s,t) = \prod_{j=1}^{P} exp(-theta_j * |s_j - t_j|^2)
  */
static double Cor_Cubic_Isotropic(long dim, const mp_vector &x1, const mp_vector &x2,
                         const mp_vector &theta);
  /*Cubic Product Correlation Family
    r_theta(s,t)={ 1-1.5*(||s-t||)/theta + 0.5 ((||s-t||)/theta)^3, for ||s-t|| < theta }
                 { 0 otherwise }
  */

private:

  pt_collect* points;   /*pointer to the evaluated points*/
  mp_vector* values;    /*pointer to function values*/
  long np;              /*number of point evaluated*/

  PatternSearch* PS;    /*some kind of optimizer*/
  long P;               /*dimension of the space*/
  double (*r)(long dim,const mp_vector&, const mp_vector&, const mp_vector& );
  /*for pointing to the correlation function*/
  long fdim;            /*dimension of the vector returned by fcn*/
  void (*fcn)(const mp_vector &, mp_vector&);
  /*for pointing to the 'A' function in custom estimation*/
  double curr_delta;    /*Current delta. Used for bounds on theta*/
  double MACH_EPS;      /*Machine Epsilon*/

};/*end class*/

double GetMachineEpsilon();
/****
OUTPUT: An estimate of Machine Epsilon.  See Heath's
Scientific Computing, chapter 1.
****/
#endif
```

D.2 Code

D.2.1 Approximate

```
/****
Approximate Implementation - 6/2/99 cmsief
This is an abstract base class for approximations
****/
```

```
#include "Approximate.h"
Approximate::Approximate(){;}
Approximate::Approximate(Approximate &A){;}
Approximate::Approximate(long dim){;}
Approximate::~Approximate(){;}
```

D.2.2 KrigApprox

```
/****
KrigApprox Implementation - 6/3/99 cmsief
****/
#include "KrigApprox.h"

/**********************CONSTRUCTORS + DESTRUCTOR**********************/

KrigApprox::KrigApprox() {
  P=0; v=NULL;
  theta=NULL; beta=NULL;
  sigma2=0; PE=NULL;
  quad=NULL;
  estimation_trend=TREND_CONSTANT;
  custom_f=NULL;MLE=NULL;fdim=1;first_time=true;
}/*end KrigApprox*/

KrigApprox::KrigApprox(long dim) {
/*Call SetDefault*/
  v=NULL; theta=NULL;beta=NULL;PE=NULL;quad=NULL;MLE=NULL;first_time=true;
  SetDefault(dim);
}/*end special constructor*/

KrigApprox::KrigApprox(KrigApprox &K) {
  P=K.P;sigma2=K.sigma2;estimation_trend=K.estimation_trend;
  if(K.v!=NULL) v=new mp_vector(*K.v);
  else v=NULL;
  if(K.theta!=NULL) theta=new mp_vector(*K.theta);
  else theta=NULL;
  if(K.beta!=NULL) beta=new mp_vector(*K.beta);
  else beta=NULL;
  if(K.PE!=NULL) PE=new ParamEstimate(*K.PE);
  else PE=NULL;
  if(K.quad!=NULL) quad=new mp_vector (*K.quad);
  else quad=NULL;
  if(K.MLE!=NULL) MLE=new pt_collect(*K.MLE);
  else MLE=NULL;
  custom_f=(K.custom_f);
  fdim=K.fdim;first_time=K.first_time;
}/*end copy constructor*/
```

```
KrigApprox::KrigApprox(ParamEstimate &PE1, long dim) {
  SetDefault(dim);first_time=true;
  (*PE)=PE1;
  (*theta).newsize((*PE).GetThetaSize());
}/*end special constructor*/

KrigApprox::~KrigApprox() {
  if(PE!=NULL) delete PE;
  if(v!=NULL) delete v;
  if(theta!=NULL) delete theta;
  if(beta!=NULL) delete beta;
  if(quad!=NULL) delete quad;
  if(MLE!=NULL) delete MLE;
}/*end destructor*/

KrigApprox& KrigApprox::operator= (KrigApprox &K) {
  if(this!=&K) {

    if(PE!=NULL) delete PE;
    if(v!=NULL) delete v;
    if(theta!=NULL) delete theta;
    if(beta!=NULL) delete beta;
    if(quad!=NULL)delete quad;

    P=K.P;sigma2=K.sigma2;estimation_trend=K.estimation_trend;
    custom_f=K.custom_f;fdim=K.fdim;first_time=K.first_time;

    if(K.PE!=NULL) PE=new ParamEstimate(*K.PE);
    else PE=NULL;
    if(K.v!=NULL) v=new mp_vector(*K.v);
    else v=NULL;
    if(K.theta!=NULL) theta=new mp_vector(*K.theta);
    else theta=NULL;
    if(K.beta!=NULL) beta=new mp_vector(*K.beta);
    else beta=NULL;
    if(K.quad!=NULL) quad=new mp_vector(*K.quad);
    else quad=NULL;
    if(K.MLE!=NULL) MLE=new pt_collect(*K.MLE);
    else MLE=NULL;

  }/*end if*/
  return (*this);
}/*end overloaded =*/

/****************************RESET FUNCTIONS***************************/
```

```
void KrigApprox::ResetKrigApproxState(ParamEstimate &PE1) {
/****
INPUT: ParamEstimate object.
EFFECT: Resets this object's state.
****/
  (*PE)=PE1;
}/*end ResetKrigApproxState*/

bool KrigApprox::ResetEstimateConstantTrend() {
/****
OUTPUT: Always returns true.
EFFECT: Chooses Estimation with a constant trend.  Since the trend is a
constant, no parameter is needed.
****/
  estimation_trend=TREND_CONSTANT;fdim=1;
  return true;
}/*end ResetEstimateConstantTrend*/

bool KrigApprox::ResetEstimateQuadraticTrend(mp_vector& a) {return false;}
  /****
  INPUT: A vector signifying quadratic trend.
  OUTPUT: True if the vector is reasonable.
  EFFECT: Chooses Estimation with that quadratic trend.
  NOTE: IMPLEMENTATION DEFERRED!
  ****/

bool KrigApprox::ResetEstimateCustomTrend(void(*fcn)(const mp_vector &,
  mp_vector&), long fcn_dim){
/****
INPUT: A function pointer, the the vector-valued trend function.
OUTPUT: Always returns true.
EFFECT: Chooses Custom trend Estimation.
****/
  custom_f=fcn;
  estimation_trend=TREND_CUSTOM;
  fdim=fcn_dim;
  return true;
}/*end ResetEstimateConstantTrend*/

bool KrigApprox::ResetCorrelationFamily(long cor) {
/****
INPUT: Correlation family number (see ParamEstimate.h).
OUTPUT: true if correlation family number is valid.
EFFECT: return PE.ResetCorrelationFamily(cor);
****/
  bool s;
  if(PE!=NULL) {
```

```
    s=(*PE).ResetCorrelationFamily(cor);
    if (s) (*theta).newsize((*PE).GetThetaSize());
    return s;
  }/*end if*/
  else return false;
}/*end ResetCorrelationFamily*/

/****************************QUERY FUNCTIONS****************************/

long KrigApprox::GetDim() const {
/****
OUTPUT: Returns the dimension of the space.
****/
  return P;
}/*end GetDim*/

void KrigApprox::debug(char *code) const{
/****
EFFECT: Outputs a state dump to stdout.
****/
  cout<<code<<":***KrigApprox::debug()***\n";
  cout<<code<<":P="<<P<<" sigma2="<<sigma2<<" fdim="<<fdim<<endl;
  if(estimation_trend==TREND_CONSTANT)
    cout<<code<<":Constant Trend Estimation\n";
  else if(estimation_trend==TREND_QUADRATIC)
    cout<<code<<":Quadratic Trend Estimation, quad="<<quad;
  else if(estimation_trend==TREND_CUSTOM)
    cout<<code<<":Custom Trend Estimation\n";
  else cout<<code<<":Trend unknown\n";
  if(PE!=NULL) (*PE).debug(code);
  else cout<<code<<":PE=NULL\n";
  if(v!=NULL) cout<<code<<":v="<<(*v);
  else cout<<code<<":v=NULL\n";
  if(beta!=NULL) cout<<code<<":beta="<<(*beta);
  else cout<<code<<":beta=NULL\n";
  if(theta!=NULL) cout<<code<<":theta="<<(*theta);
  else cout<<code<<":theta=NULL\n";
  if(MLE!=NULL) cout<<code<<":MLE="<<(*MLE);
  else cout<<code<<":MLE=NULL\n";
  cout<<code<<":***end KA::debug()***\n";
}/*end debug*/

/****************************ACTION FUNCTIONS****************************/

bool KrigApprox::EvaluateApprox(const mp_vector &x, const pt_collect &Eval_c,
                                double* Approx, double* D_c){
/****
```

```
INPUT: The location to evaluate at, the location of the previously
evaluated sites, and two double pointers for return values.
OUTPUT: Returns true if the operation was successful.
EFFECT: Using kriging to evaluate the approximation at this point.  It also
evaluates the design criterion.
****/
#if DEBUG>=3
  cout<<"KAEA   :function called\n";
  cout<<"KAEA   :X="<<x;
#endif
  if(first_time) return false;/*Update not run yet*/
  long points=(*v).dim();
  long i;
  mp_vector r(points);
  mp_vector MLE_r(points+fdim);

  /*Create the correlation vector between the points and the design sites*/
  for(i=0;i<fdim;i++) MLE_r[i]=1;
  for(i=0;i<points;i++) {
    r[i]=(*PE).EvaluateMyCorrelation(P,x,Eval_c.row(i), (*theta));
    MLE_r[i+fdim]=r[i];
  }/*end for*/
#if DEBUG>=4
  cout<<"KAEA   : r="<<r;
  cout<<"KAEA   : MLE_r="<<MLE_r;
#endif
  /*evaluate the value of the approximation here*/
  (*Approx)=EvaluateTrend(x) + (*v) * r;

  /*evaluate the sqrt of the mean squared error for the design criterion*/
  double MLE_X = MLE_r * ((*MLE) * MLE_r);
  /*Uses special multiply operator to use 1st MLE_r as row vector*/
#if DEBUG>=3
  cout<<"KAEA   :MLE_X="<<MLE_X<<endl;
#endif

  /*Deal with a common numerical error*/
  if(sigma2<0) {
#if DEBUG>=2
    cout<<"KAEA : ERROR(non-fatal) sigma2="<<sigma2
<<" Zeroing D_C..."<<endl; (*D_c)=0;
#endif
  }/*end if*/
  else if((1-MLE_X)<0)
    {
#if DEBUG>=2
      cout<<"KAEA   : ERROR(non-fatal) 1-MLE_X="<<1-MLE_X
```

```
  <<" Zeroing D_C..."<<endl
  <<"KAEA   : ERROR(non-fatal) x="<<x;
#endif
      (*D_c)=0;
    }/*end if*/
  else
  (*D_c) = sqrt(sigma2 * ( 1.0 - MLE_X));

#if DEBUG>=3
  cout<<"KAEA   :function completed\n";
#endif
  return true;

}/*end EvaluateApprox*/

bool KrigApprox::UpdateApprox(pt_collect &Eval_c, mp_vector&
                              f_values, long numpoints, double delta,
      const Bounds &b){
/****
INPUT: The collection of evaluated sites, their values, the number of points
evaluated, the iteration number, delta and the bounds.
OUTPUT: Returns true if the operation was successful.
EFFECT: This updates the weighting scheme and then rekrigs the function,
calling for parameter estimation as needed.  Also, it updates the Design
Criterion.
****/
  if(MLE==NULL) MLE=new pt_collect(1,1);
  (*MLE).newsize(fdim+numpoints, fdim+numpoints);
  if(v==NULL) v=new mp_vector(1,1);
  (*v).newsize(numpoints);

  bool s=false;
  if(first_time) {
    first_time=false;
    if(theta==NULL) theta=new mp_vector((*PE).GetThetaSize());

    /*New Initial Theta Scheme*/
    double L=b.upper[0]-b.lower[0];
    for(long i=1;i<P;i++)
      if(b.upper[i]-b.lower[i] > L) L=b.upper[i]-b.lower[i];

    (*theta)=-100*log(0.5) / sqr(L);
  }/*end if*/

  switch(estimation_trend) {
  case TREND_CONSTANT:
    s=(*PE).EstimateConstantTrend(Eval_c, f_values, delta, (*beta).begin(),
```

```
  &sigma2, theta, MLE, v);
    break;
  case TREND_QUADRATIC:
    break;
  case TREND_CUSTOM:
    s=(*PE).EstimateCustomTrend(Eval_c, f_values, delta, custom_f, fdim,
beta, &sigma2, theta, MLE, v);
    break;
  default:
    s=false;
  };/*end switch*/
  return s;
}/*end UpdateApprox*/

void KrigApprox::SetDefault(long dim) {
/****
INPUT: Dimension of space.  Used for parameter estimation.
EFFECT: Constant Trend Estimation, and ParamEstimate's default family.
****/
  PE=new ParamEstimate(dim);
  P=dim;
  v=new mp_vector(1);/*it needs to be numpoints later*/
  theta=new mp_vector((*PE).GetThetaSize());

  (*theta)=0.0001; /*assigns a guess to all elements of theta*/
  estimation_trend=TREND_CONSTANT;
  beta=new mp_vector(1);/*Assumes Trend Constant*/
  sigma2=0;
  quad=NULL;fdim=1;
  custom_f=NULL;
  MLE=new pt_collect(1,1);/*will be resized later*/
}/*end SetDefault*/


double KrigApprox::EvaluateTrend(const mp_vector& x){
/****
INPUT: A mp_vector, x
OUTPUT: The value of the trend at that particular point.  This is assembled
from the betas, as appropriate.
****/
  switch(estimation_trend) {
  case TREND_CONSTANT:
    return (*beta)[0];/*for constant trend, value is in beta[0]*/
    break;
  case TREND_QUADRATIC:
    return ERROR;
    break;
```

```
  case TREND_CUSTOM:
    return ERROR;
    break;
  default:
    return ERROR;
  };/*end switch*/

}/*end EvaluateTrend*/
```

D.2.3 ParamEstimate

```
/****
ParamEstimate Implementation - 6/3/99 cmsief
It slices, it dices, it does singular value decomposition!
****/
#include "ParamEstimate.h"
#include "CompassSearch.h"
#include <math.h>          /*e^whatever and sqrt*/
#include <malloc.h>
#include <stdlib.h>
#include <iostream.h>

// #define COR_EXP_ISOTROPIC 0
// #define COR_GAUSS_ISOTROPIC 1
// #define COR_EXP_PRODUCT 2
// #define COR_GAUSS_PRODUCT 3
// #define COR_CUBIC_ISOTROPIC 4


/***********************CONSTRUCTORS + DESTRUCTOR************************/

ParamEstimate::ParamEstimate() {
  PS=NULL;
  P=0;
  np=0;
  r=NULL;
  fdim=0;
  fcn=NULL;
  curr_delta=0;
  points=NULL;
  values=NULL;
  MACH_EPS=GetMachineEpsilon();
}/*end default constructor*/
```

```
ParamEstimate::ParamEstimate(ParamEstimate &PE){
  PS=new CompassSearch;
  (*PS)=(*PE.PS);
  P=PE.P;
  r=PE.r;
  np=PE.np;
  fcn=PE.fcn;
  fdim=PE.fdim;
  curr_delta=PE.curr_delta;
  MACH_EPS=PE.MACH_EPS;
  /*Pointers to these are copied*/
  points=PE.points;
  values=PE.values;
}/*end copy constructor*/

ParamEstimate::ParamEstimate(long dim) {
/*Prep work and call AutoGenerate*/
  PS=NULL;
  points=NULL;
  values=NULL;
  MACH_EPS=GetMachineEpsilon();
  SetDefault(dim);
}/*end special constructor*/

ParamEstimate::ParamEstimate(int correlate, long dim, PatternSearch &PS1) {
  P=dim;
  points=NULL;
  values=NULL;
  np=0;
  fcn=NULL;
  fdim=0;
  curr_delta=0;
  PS=NULL;
  r=NULL;
  ResetCorrelationFamily(correlate);
  ResetPatternSearch(PS1);
  MACH_EPS=GetMachineEpsilon();
}/*end special constructor*/

ParamEstimate::~ParamEstimate(){
  points=NULL;/*I know that I'm doing this*/
  values=NULL;
  if(PS!=NULL) {delete PS;PS=NULL;}
}/*end destructor*/

ParamEstimate& ParamEstimate::operator=(ParamEstimate &PE){
```

```
  if (this!=&PE) {
    if(PE.PS!=NULL) ResetPatternSearch(*PE.PS);
    else if(PS!=NULL) {delete PS;PS=NULL;}
    P=PE.P;
    r=PE.r;
    fcn=PE.fcn;
    fdim=PE.fdim;
    np=PE.np;
    curr_delta=PE.curr_delta;
    MACH_EPS=PE.MACH_EPS;
    /*Pointers to these are copied*/
    points=PE.points;
    values=PE.values;
  }/*end if*/
  return (*this);
}/*end overloaded=*/

/***************************RESET FUNCTIONS****************************/

void ParamEstimate::ResetParamEstimateState(int correlate, long dim,
    PatternSearch &PS1){
/****
INPUT: Correlation family number, dimension of space, and a PatternSearch.
EFFECT: Resets all the major state variables.
****/
  ResetCorrelationFamily(correlate);
  ResetDim(dim);
  ResetPatternSearch(PS1);
}/*end ResetParamEstimateState*/

void ParamEstimate::ResetPatternSearch(PatternSearch &PS1) {
/****
INPUT: PatternSearch Object
EFFECT: Sets PS to be this PatternSearch
****/
  if(PS==NULL) PS=new CompassSearch;
  (*PS)=(PS1);

}/*end ResetPatternSearch*/

bool ParamEstimate::ResetCorrelationFamily(int correlate){
/****
INPUT: Correlation family number
OUTPUT: True if number is valid
EFFECT: Sets function pointer r to correct function
****/
  switch(correlate) {
```

```
  case COR_EXP_ISOTROPIC:
    r=Cor_Exp_Isotropic;
    return true;
  case COR_GAUSS_ISOTROPIC:
    r=Cor_Gauss_Isotropic;
    return true;
  case COR_EXP_PRODUCT:
    r=Cor_Exp_Product;
    return true;
  case COR_GAUSS_PRODUCT:
    r=Cor_Gauss_Product;
    return true;
  case COR_CUBIC_ISOTROPIC:
    r=Cor_Cubic_Isotropic;
    return true;
  default:
    return false;
  };/*end switch*/
}/*end ResetCorrelationFamily*/

bool ParamEstimate::ResetDim(long dim) {
/****
INPUT: Dimension of the space.
OUTPUT: True if dimension is valid.
EFFECT: P=dim;
****/
  if(dim>0) {P=dim;return true;}
  else return false;
}/* ResetDim*/

/***************************QUERY FUNCTIONS****************************/

long ParamEstimate::GetDim() const {
/****
OUTPUT: Returns the dimension of the space.
****/
  return P;
}/*end GetDim*/


long ParamEstimate::GetThetaSize() const {
/****
OUTPUT: Returns the size of the theta vector
****/
  if(r==Cor_Exp_Isotropic || r==Cor_Gauss_Isotropic
     ||r==Cor_Cubic_Isotropic) return 1;
  else if(r==Cor_Exp_Product || r==Cor_Gauss_Product) return P;
```

```
  else return ERROR;
}/*end GetThetaSize*/

double ParamEstimate::GetThetaUpperBound(double delta){
/****
INPUT: delta.
OUTPUT: Returns a double representing the upper bound for
theta = -log(0.8) / delta^2.
****/
  return (-log(.8) / sqr(delta));
}/*end GetThetaUpperBound*/

void ParamEstimate::debug(char *code) const{
/****
EFFECT: Does a complete state dump to stdout.
****/
  cout<<code<<":***ParamEstimate::debug()***\n";
  cout<<code<<":P="<<P<<" curr_delta="<<curr_delta<<" fdim="<<fdim<<" np="<<np<<endl;
  if(r==Cor_Exp_Isotropic)
    cout<<code<<":Exponential Isotropic Correlation\n";
  else if(r==Cor_Gauss_Isotropic)
    cout<<code<<":Gaussian Isotropic Correlation\n";
  else if(r==Cor_Exp_Product)
    cout<<code<<":Exponential Product Correlation\n";
  else if(r==Cor_Gauss_Product)
    cout<<code<<":Gaussian Product Correlation\n";
  else if(r==Cor_Cubic_Isotropic)
    cout<<code<<":Cubic Isotropic Correlation\n";
  else cout<<code<<":Correlation unknown\n";
  if(points!=NULL) cout<<code<<":points="<<(*points);
  else cout<<code<<":points=NULL\n";
  if(values!=NULL) cout<<code<<":values="<<(*values);
  else cout<<code<<":values=NULL\n";
  cout<<code<<":***end PE::debug()***\n";
}/*end debug*/

/****************************ACTION FUNCTIONS****************************/

bool ParamEstimate::EstimateConstantTrend(pt_collect &pts, mp_vector &fvals,
                                          double delta, double* beta,
                                          double* sigma2, mp_vector* theta, pt_collect*
                                          MLE_Matrix, mp_vector* v) {
/****
INPUT: Collection of evaluated points, their values, and pointers for the
parameters to be returned.  theta should come in with a 'guess' value for the
optimizer to chew on.
OUTPUT: Returns true if the operation was sucessful.
```

```
EFFECT: This estimates the parameters, assuming a constant trend, and returns them.
****/
#if DEBUG>=2
  cout<<"PEECT  :Started\n";
#endif
  bool success;
  curr_delta=delta;
  np=pts.num_rows();
  pt_collect R(np,np);
  mp_vector A(np);
  A=1;/*fills A with 1's*/
  long i,j;
  /*Pointers for these to save memory*/
  points=&pts;
  values=&fvals;

  /*Pattern Search w/bounds on OptimizeMLEConstant*/
  double* pstheta=(double*)malloc((*theta).dim()*sizeof(double));
  double* ret_theta=(double*)malloc((*theta).dim()*sizeof(double));
  for(i=0;i<(*theta).dim();i++) pstheta[i]=(*theta)[i];
  Bounds bds;
  bds.lower.newsize((*theta).dim());bds.lower=0;
  bds.upper.newsize((*theta).dim());bds.upper=GetThetaUpperBound(curr_delta);

  (*PS).CleanSlate((*theta).dim(), pstheta,START_THETA_STEP, STOP_THETA_STEP,
                  this, bds,OptimizeMLEConstant);

#if DEBUG >=3
  cout<<"PEECT  :Initialized PS\n";
#endif
  (*PS).ExploratoryMoves();
#if DEBUG >=3
  cout<<"PEECT  :PS.ExploritoryMoves() complete\n";
#endif
  (*PS).GetMinPointFLOP(ret_theta);

  for(i=0;i<(*theta).dim();i++) (*theta)[i]=ret_theta[i];
  free(ret_theta);
  free(pstheta);

  for(i=0;i<(*theta).dim();i++)
    /*Make sure that theta is non-zero*/
    if((*theta)[i]<TOLERANCE) (*theta)[i]=TOLERANCE;

  /*Generate the corration matrix for theta*/
  GenerateCorrelationMatrix(R,(*theta));
```

```
  /*Generate MLE_Matrix - This is a block matrix as follows:
   | 0  A |
   | AT R |
  */
  (*MLE_Matrix)[0][0]=0.0;
  for(i=1;i<np+1;i++) {
    (*MLE_Matrix)[0][i]=A[i-1];
    (*MLE_Matrix)[i][0]=A[i-1];
    for(j=1;j<np+1;j++)
      (*MLE_Matrix)[i][j]=R[i-1][j-1];
  }/*end for*/
#if DEBUG >=3
  cout<<"PEECT  :MLE Matrix Constructed:"<<(*MLE_Matrix);
#endif

  success=pseudoinvert(R,NULL);
  if(!success) return (success); /*break out!*/
  success=pseudoinvert(*MLE_Matrix,NULL);
  if(!success) return (success); /*break out!*/
#if DEBUG >=3
  cout<<"PEECT  :Pseudoinversions complete\n";
#endif
  (*v)=A*R;/*this multiply assumes that A is actually AT */
  (*beta)=((*v) * (*values)) / ((*v)*A); /*a special form for the constant trend case*/
  (*v)=((*values) - A*(*beta));
  (*sigma2)=((*v) * R * (*v))/np; /*again, the special multiply assumes that the first v
                                    is vT*/
  (*v)=(*v)*R;/*special multiply which assumes v=vT*/

#if DEBUG >=2
  cout<<"PEECT  :terminating success="<<success<<"\n";
  cout<<"PEECT  :theta ="<<(*theta)[0]<<endl;
  cout<<"PEECT  :beta ="<<(*beta)<<endl;
#endif

  return (success);
}/*end EstimateConstantTrend*/


  bool ParamEstimate::EstimateQuadraticTrend(pt_collect &pts, mp_vector &fvals,
                                             double delta, mp_vector &a, mp_vector* beta,
                                             double* sigma2, mp_vector* theta,
                                             pt_collect* MLE_Matrix, mp_vector*
                                             v){return false;}
  /****
  INPUT: Collection of evaluated points, their values, and pointers for the
  parameters to be returned. theta should come in with a 'guess' value for the
```

```
  optimizer to chew on.
  OUTPUT: Returns true if the operation was sucessful.
  EFFECT: This constructs the appropriate function vector and calls
  EstimateCustom, using the Corellation function specified as the function
  pointer parameter.
  NOTE: IMPLEMENTATION DEFERRED TO A LATER DATE.  DON'T CALL THIS.
  ****/

bool ParamEstimate::EstimateCustomTrend(pt_collect &pts, mp_vector &fvals,
                                        double delta, void
                                        (*fcn2)(const mp_vector &, mp_vector&), long
                                        dimf, mp_vector* beta, double* sigma2,
mp_vector* theta, pt_collect* MLE_Matrix,
mp_vector* v){
  /****
  INPUT: Collection of evaluated points, their values, and pointers for the
  parameters to be returned.  There is also a function pointer for the
  'a' function, and the dimension of the vector it returns.  theta should come
  in with a 'guess' value for the optimizer to chew on.
  OUTPUT: Returns true if the operation was successful.
  EFFECT: This estimates the function.
  ****/
#if DEBUG >=2
  cout<<"PEECUT :Started\n";
#endif
  bool success;
  curr_delta=delta;
  np=pts.num_rows();
  fcn=fcn2;
  points=&pts;
  values=&fvals;
  fdim=dimf;

  pt_collect R(np,np);
  pt_collect A(np,fdim);
  pt_collect AT(fdim,np); /*A transpose*/
  mp_vector ai(fdim);
  pt_collect temp(fdim,fdim);
  long i,j;/*counters*/

 /*Pattern Search w/bounds on MLECustom*/
  double* pstheta=(double*)malloc((*theta).dim()*sizeof(double));
  double* ret_theta=(double*)malloc((*theta).dim()*sizeof(double));
  Bounds bds;
  bds.lower.newsize((*theta).dim());bds.lower=0;
  bds.upper.newsize((*theta).dim());bds.upper=GetThetaUpperBound(curr_delta);
```

```
  for(i=0;i<(*theta).dim();i++) pstheta[i]=(*theta)[i];

  (*PS).CleanSlate((*theta).dim(), pstheta,START_THETA_STEP,STOP_THETA_STEP,
   this, bds,OptimizeMLECustom);
#if DEBUG >=3
  cout<<"PEECUT :Initialized PS\n";
#endif
  (*PS).ExploratoryMoves();
#if DEBUG >=3
  cout<<"PEECUT :PS.ExploritoryMoves() complete\n";
#endif
  (*PS).GetMinPointFLOP(ret_theta);

  for(i=0;i<(*theta).dim();i++) (*theta)[i]=pstheta[i];
  free(pstheta);
  free(ret_theta);

  for(i=0;i<(*theta).dim();i++)
    /*Make sure that theta is non-zero*/
    if((*theta)[i]<TOLERANCE) (*theta)[i]=TOLERANCE;

  /*Generate the correlation matrix for theta*/
  GenerateCorrelationMatrix(R,(*theta));

  /*Generate A and AT*/
  for(i=0;i<np;i++) {
    (*v)=(*points).row(i);
    (*fcn)(*v,ai);
    for(j=0;j<fdim;j++) {
      A[i][j]=ai[j];
      AT[j][i]=ai[j];
    }/*end for*/
  }/*end for*/


  /*Generate MLE_Matrix - This is a block matrix as follows:
   | 0  A |
   | AT R |
  */
  for(i=0;i<fdim+np;i++)
    for(j=0;j<fdim+np;j++)
      if(i<fdim && j<fdim) (*MLE_Matrix)[i][j]=0;
      else if(i<fdim) (*MLE_Matrix)[i][j]=AT[i][j-fdim]; /*and j>fdim*/
      else if(j<fdim) (*MLE_Matrix)[i][j]=A[i-fdim][j]; /*and i>fdim*/
      else (*MLE_Matrix)[i][j]=R[i-fdim][j-fdim];
#if DEBUG >=3
  cout<<"PEECUT :A, AT and MLE Matrix Constructed:"<<(*MLE_Matrix);
```

```
#endif
  success=pseudoinvert(R,NULL);
  if(!success) return (success); /*break out!*/
  success=pseudoinvert(*MLE_Matrix,NULL);
  if(!success) return (success); /*break out!*/
#if DEBUG >=3
  cout<<"PEECUT :Pseudoinversions complete\n";
#endif
  /*Get the Params*/
  temp=AT*R*A;
  success=(int)pseudoinvert(temp,NULL);
  if(!success) return (success); /*break out!*/
  (*beta)=temp*AT*R*fvals;
  (*v)=fvals- A*(*beta);
  (*sigma2)=((*v)*R*(*v))/np; /*the special multiply assumes that the first v is vT*/
  (*v)=(*v) * R;
#if DEBUG >=2
  cout<<"PEECUT :terminating success="<<success<<"\n";
#endif
  return(success);
}/*end EstimateCustomTrend*/


void ParamEstimate::SetDefault(long dim){
/****
INPUT: Dimension of the space.
EFFECTS: This will use a happy default set of values to set up the
parameter estimation.  Specifically, it chooses Gaussian Isotropic
Correlation and uses a Compass Search.
****/
  P=dim;
  r=Cor_Gauss_Isotropic;
  PS=new CompassSearch();
  np=0;
  fdim=0;
  curr_delta=0;
  fcn=NULL;
}/*end SetDefault*/

void ParamEstimate::MLEConstant(long dimtheta, double *x, double &f, int &success){
/****
INPUT: Modified PatternSearch calling sequence
EFFECT: Evaluates the MLE for a given theta.  Uses the constant trend.
NOTES: 5/31/99 - tested and functions well.
****/
  long i;
  double temp=GetThetaUpperBound(curr_delta);
```

```
#if DEBUG>=3
  cout<<"PEMC   :MLEConstant Called\n";
  cout<<"PEMC   :trial_theta=(";
  for(i=0;i<dimtheta;i++) cout<<x[i]<<" ";
  cout<<")\n";
#endif
  double beta, sigma2, sum=0.0;

  for(i=0;i<dimtheta;i++)
    if((x[i]>temp)||(x[i]<TOLERANCE)){
#if DEBUG>=3
      cout<<"PEMC   :Theta value invalid bound="<<temp<<"\n";
#endif
      success=(int)false;return;
    }
  mp_vector d(np);
  mp_vector v(np);
  mp_vector theta(dimtheta,x);/*creates the mp_vector out of a bunch of doubles*/
  pt_collect R(np,np);
  mp_vector A(np);
  A=1; /*fills A with 1's*/

  GenerateCorrelationMatrix(R,theta);
  success=(int)pseudoinvert(R,&d);
#if DEBUG>=4
  cout<<"PEMC   :Pseudoinverse Complete\n";
#endif
  if(success) {
    temp=log(TOLERANCE); /*temporary for extra execution speed*/
    /*Compute log determinant of matrix via singular values = eigenvalues*/
    for(i=0;i<np; i++)
      if (fabs(d[i])<TOLERANCE) sum+=temp;
      else sum+=log(d[i]);

    v=A*R; /*this multiply assumes that A is actually AT */
    beta=(v * (*values)) / (v*A); /*a special version for the constant trend case*/
    v=((*values) - A*beta);
    sigma2=(v * R * v)/np; /*again, the special multiply assumes that the first v
                             is vT*/
#if DEBUG>=4
    cout<<"PEMC   :beta="<<beta<<endl;
    cout<<"PEMC   :sigma2="<<sigma2<<endl;
#endif
    if(sigma2<TOLERANCE) {
#if DEBUG>=1
      cout<<"PEMC   :ERROR(non-fatal) sigma2="<<sigma2<<" resetting to TOLERANCE..\n";
#endif
```

```
      sigma2=TOLERANCE;
    }/*end if*/
    f=np*log(sigma2) +sum;/*returned value*/
#if DEBUG>=3
    cout<<"PEMC   :Call Complete\n";
#endif
  }/*end if*/
}/*end MLEConstant*/


void ParamEstimate::MLECustom(long dimtheta, double *x, double &f, int &success){
/****
INPUT: Modified PatternSearch calling sequence
EFFECT: Evaluates the MLE for a given theta.  Uses the custom trend.
****/
  long i;
  double dtemp=GetThetaUpperBound(curr_delta);
#if DEBUG>=2
  cout<<"PEMCU  :MLECustom Called\n";
  cout<<"PEMCU  :trial_theta=(";
  for(i=0;i<dimtheta;i++) cout<<x[i]<<" ";
  cout<<")\n";
#endif
  double sigma2, sum=0.0;
  for(i=0;i<dimtheta;i++)
    if(x[i]>dtemp||x[i]<TOLERANCE){
#if DEBUG>=3
      cout<<"PEMCU  :Theta value invalid bound="<<temp<<"\n";
#endif
      success=(int)false;return;
    }

  pt_collect R(np,np);
  mp_vector d(np);
  mp_vector v(np);
  mp_vector theta(dimtheta,x);/*creates an mp_vector out of a bunch of doubles*/
  mp_vector beta(fdim);
  pt_collect temp(fdim,fdim);
  mp_vector ai(fdim);
  pt_collect A(np,fdim);
  pt_collect AT(fdim,np); /*A transpose*/

  /*Generate the correlation matrix and pseudoinvert it*/
  GenerateCorrelationMatrix(R,theta);
  success=(int)pseudoinvert(R,&d);
#if DEBUG>=3
  cout<<"PEMCU  :Pseudoinverse Complete\n";
```

```
#endif

  if(success) {
    dtemp=log(TOLERANCE); /*temporary for extra execution speed*/
    /*Compute log determinant of matrix via singular values = eigenvalues*/
    for(i=0;i<np; i++)
      if (fabs(d[i])<TOLERANCE) sum+=dtemp;
      else sum+=log(d[i]);

    /*Generate A and AT*/
    for(i=0;i<np;i++) {
      v=(*points).row(i);
      (*fcn)(v,ai);
      for(long j=0;j<fdim;j++) {
        A[i][j]=ai[j];
        AT[j][i]=ai[j];
      }/*end for*/
    }/*end for*/
#if DEBUG>=3
  cout<<"PEMCU  :A,AT computed\n";
#endif
    temp=AT*R*A;
    success=(int)pseudoinvert(temp,NULL);
    if(success) {
      beta=temp*AT*R*(*values);
      v=(*values)- A*beta;
      sigma2=(v*R*v)/np; /*the special multiply assumes that the first v is vT*/
#if DEBUG>=3
      cout<<"PEMCU  :beta="<<beta<<endl;
      cout<<"PEMCU  :sigma2="<<sigma2<<endl;
#endif
      if(sigma2<TOLERANCE) {
#if DEBUG>=1
cerr<<"PEMCU  :ERROR(non-fatal) sigma2="<<sigma2<<" resetting to TOLERANCE..\n";
#endif
sigma2=TOLERANCE;
      }/*end if*/

      f=np*log(sigma2)+sum;
    }/*end if*/
  }/*end if*/
#if DEBUG>=2
  cout<<"PEMCU   :Call Complete\n";
#endif
}/*end MLECustom*/

inline void ParamEstimate::GenerateCorrelationMatrix(pt_collect &R, const
```

```
     mp_vector &theta) {
/****
INPUT: A np x np pt_collect, and the correlation family parameter vector, theta.
EFFECT: Replaces R with the appropriate correlation matrix.
****/
  mp_vector temp(P);
  for(long i=0;i<np;i++) {
    R[i][i]=1.0;
    temp=(*points).row(i);
    for(long j=i+1;j<np;j++) {
      R[i][j]=(*r)(P,temp, (*points).row(j), theta);
      R[j][i]=R[i][j];
    }/*end for*/
  }/*end for*/
}/*end GenerateCorrelationMatrix*/


bool ParamEstimate::pseudoinvert(pt_collect &M, mp_vector* d) {
/****
INPUT: A matrix M, and a pointer to a mp_vector.
OUTPUT: true if svd can be computed, else false.
EFFECT: Replaces M with the pseudoinverse of M.  If d!=NULL, an
mp_vector with the one over the singular values, with those singular
values below TOLERENCE instead to zero, is returned.  The matrix has
to be *SYMMETRIC AND SQUARE* for this to work.
****/
  if(M.num_rows()!=M.num_cols()) return false;
  long md = M.num_rows();
#if DEBUG >=3
  cout<<"PEPI   :Pseudoinverse Commencing md="<<md<<"\n";
#endif
  bool ret_val;
  double t_tolerance;
  pt_collect U(md,md);
  pt_collect VT(md,md);
  pt_collect D(md,md);
#if DEBUG >=3
  cout<<"PEPI   :Alloc complete\n";
#endif
  ret_val=ComputeSVD(M,&U,&D,&VT);
#if DEBUG >=3
  cout<<"PEPI   :SVD complete - returning "<<ret_val<<endl;
#endif
  if (ret_val) {
    if(d!=NULL)
      for(long j=0;j<md;j++) (*d)[j]=D[j][j];
```

```
    t_tolerance=MACH_EPS * M.num_rows() * D[0][0];
    /*Tolerance as per Trosset*/

    for(long i=0;i<md;i++) {
      if (fabs(D[i][i])<t_tolerance) D[i][i]=0;
      else D[i][i]=1/D[i][i];
    }
#if DEBUG >=3
    cout<<"PEPI   :D has been reconstructed, reallocs complete\n";
#endif
    //M=U*D*VT should work for symmetric matrices, but alas, numerical
    //error seems to have bested this timesaver.
    M=transpose(VT) * D * transpose(U);

#if DEBUG >=3
    cout<<"PEPI   :M is computed\n";
#endif
  }/*end if*/
#if DEBUG >=3
  cout<<"PEPI   :terminating\n";
#endif
  return ret_val;
}/*end pseudoinvert*/


bool ParamEstimate::ComputeSVD(const pt_collect &A, pt_collect* U,
       pt_collect* D, pt_collect* VT){
/****
INPUT: Matrix A, to take the SVD of, and pointers to three matrices that will
contain what is returned from dgesvd_.
OUTPUT: true if CLAPACK's dgesvd_ returns ok.
EFFFECT: Calls CLAPACK's degsvd routine to compute the Singular Value
Decomposition(SVD) of A.  Recall that a SVD of A yields U, D, and V',
s.t. A=U*D* V', where U and V are orthogonal matrices, and D is a diagonal
matrix with the singular values on its main diagonal.
****/
  if(A.num_rows()!=A.num_cols()) return false;

  long md=A.num_rows();
  long idx, LWORK=md*md+5*md;
  double* MA=(double*)malloc(sizeof(double)*md*md);
  double* MU=(double*)malloc(sizeof(double)*md*md);
  double* MVT=(double*)malloc(sizeof(double)*md*md);
  double* MS=(double*)malloc(sizeof(double)*md);
  double* WORK=(double*)malloc(sizeof(double)*LWORK);
  char ch='A';
  long i,j;/*counters*/
```

```
  for(i=0;i<md;i++)
    for(j=0;j<md;j++)
      MA[md*j+i]=A[i][j]; /*This should flip the Matrix into column-major order
                            ala Fortran.*/

  dgesvd_(&ch,&ch,&md,&md,MA,&md,MS,MU,&md,MVT,&md,WORK,&LWORK,&idx);

  if(idx<0) {free(MA);free(MU);free(MVT);free(MS);free(WORK);return false;}
#if DEBUG>0
  else if (idx>0) cerr<<"PECSVD : WARNING - SVD has "<<idx
      <<" unconverged superdiagonal values\n";
#endif

  /*Copy data to all the outgoing matrices*/
  for(j=0;j<md;j++) {/*cols*/
    idx=md*j;/*Partial computation of index... for speed*/
    for(i=0;i<md;i++) {/*rows*/
      (*U)[i][j]=MU[idx+i];
      (*VT)[i][j]=MVT[idx+i];
      if(i==j) (*D)[i][j]=MS[i];
      else (*D)[i][j]=0.0;
    }/*end for*/
  }/*end for*/

#if DEBUG>=4
  cout<<"PECSVD :A="<<(A);
  cout<<"PECSVD :U="<<(*U);
  cout<<"PECSVD :D="<<(*D);
  cout<<"PECSVD :VT="<<(*VT);
  cout<<"PECSVD :U*D*VT="<<((*U)*(*D)*(*VT));
#endif
  /*Free memory!*/
  free(MA);free(MU);free(MVT);free(MS);free(WORK);
  return true;
}/*end ComputeSVD*/

/***********************CORRELATION FUNCTIONS**************************/

double ParamEstimate::Cor_Exp_Isotropic(long dim, const mp_vector &x1,
const mp_vector &x2,const mp_vector &theta) {
  /*Exponential Isotropic Correlation Family
    r_theta(s,t) = exp(-theta ||s-t||)
  */
  /*This assumes that the theta of value is theta[0]*/
  mp_vector temp(dim);
  temp=x2-x1;
```

```
  return exp(-theta[0] * temp.l2norm());
}/*end Cor_Exp_Isotropic*/

double ParamEstimate::Cor_Gauss_Isotropic(long dim, const mp_vector &x1,
const mp_vector &x2,const mp_vector &theta) {
  /*Gaussian Isotropic Correlation Family
    r_theta(s,t) = exp(-theta ||s-t||)^2
  */
  /*This assumes that the theta of value is theta[0]*/
  mp_vector temp(dim);
  temp=x2-x1;
  return exp(-theta[0] * temp.l2norm_sqr());
}/*end Cor_Gauss_Isotropic*/

double ParamEstimate::Cor_Exp_Product(long dim, const mp_vector &x1,
      const mp_vector &x2, const mp_vector &theta) {
  /*Exponential Product Correlation Family
    r_theta(s,t) = \prod_{j=1}^{P} exp(-theta_j * |s_j - t_j|)
  */
  double sum=1.0;
  for(long i=0;i<dim;i++) sum*=exp(-theta[i]*fabs(x2[i]-x1[i]));
  return sum;
}/*end Cor_Exp_Product*/

double ParamEstimate::Cor_Gauss_Product(long dim,const mp_vector &x1,
const mp_vector &x2,const mp_vector &theta) {
  /*Gaussian Product Correlation Family
    r_theta(s,t) = \prod_{j=1}^{P} exp(-theta_j * |s_j - t_j|^2)
  */
  double sum=1.0;
  for(long i=0;i<dim;i++) sum*=exp(-theta[i]*sqr(x2[i]-x1[i]));
  return sum;
}/*end Cor_Gauss_Product*/

double ParamEstimate::Cor_Cubic_Isotropic(long dim, const mp_vector &x1,
                                const mp_vector &x2,const mp_vector &theta) {
  /*Cubic Product Correlation Family
    r_theta(s,t)={ 1-1.5*(||s-t||)/theta + 0.5 ((||s-t||)/theta)^3, for ||s-t|| < theta }
                 { 0 otherwise }
  */
 /*This assumes that the theta of value is theta[0]*/
  mp_vector temp(dim);
  temp=x2-x1;
  double norm=temp.l2norm();
  double norm3theta= (norm*norm*norm)/(theta[0]*theta[0]*theta[0]);
  if(norm < theta[0])
    return 1.0 - 1.5*(norm / theta[0]) + 0.5 * norm3theta;
```

```
  else
    return 0.0;

}/*end Cor_Cubic_Isotropic*/

double ParamEstimate::EvaluateMyCorrelation(long dim, const mp_vector &x1,
      const mp_vector &x2, const mp_vector &theta) {
  /****
  INPUT: Dimesion of space, two points, and a 'vector' theta.
  For the Isotropic functions, this theta is a one dimensional vector,
  for the Product functions, this is a P-dimensional vector.
  OUTPUT: Correlation between the two points.
  ****/
  return (*r)(dim,x1,x2,theta);
}/*end EvaluateMyCorrelation*/

double GetMachineEpsilon() {
  /*From Heath*/

  return fabs( 3.0 * (4.0/3.0 - 1.0) - 1.0 );
}/*end GetMachineEpsilon*/
```

Appendix E

Source Code for Optimization

E.1 Headers

E.1.1 MAPS

```
/****
maps class - 6/17/99 cmsief
This IS the Model-Assisted Pattern Search.  To run without any sort of
difficulty:
call the  maps(long V1, long P1, const Bounds &B1, char*fn) constructor,
call RunSearch,
call RetrieveBestSeen.

NOTE : Make sure the Grid and Pattern Search Core Patterns are compatible!
I am not responsible for what this might do if you don't.


FLAGS:
define ITERATE_DEBUG to output statistics iteration by iteration.

****/
#ifndef _MAPS_
#define _MAPS_
#include "maps_general.h"
#include "PSGrid.h"
#include "PatternSearch.h"
#include "InitialDesign.h"
#include "LatinHCDesign.h"
#include "SearchCriterion.h"
#include "rngs.h"

class maps {

public:
```

```
maps();
/*status=NOTINIT*/

maps(long V1, long N1, long P1, const Bounds &B1, char* fn, const PSGrid &G1,
     const InitialDesign &I, const SearchCriterion &S1, const PatternSearch &PS1);
/*Prep work and call ResetMAPSState*/
maps(long V1, long P1, Bounds &B1, char*fn);
/*Prep work and call SetDefault*/
maps(maps& old);
~maps();

/*****RESET FUNCTIONS****/


bool ResetMAPSState(long V1, long N1, long P1, const Bounds &B1, char* fn,
    const PSGrid &G1, const InitialDesign &I1,
    const SearchCriterion &S1, const PatternSearch &PS1);
/****
INPUT: A partial set of state parameters for MAPS
OUTPUT: Returns true if the operation was successful.
EFFECT: Sets all the parameters, and sets status=READY, clears out Eval_c and
f_values.  This wipes out everything else entirely.
*****/

void NewMAPSRun();
/****
EFFECT: If status=MAPS_DONE, clears the Eval_c and F_values, resets C=0
and sets status=READY.
This allows you to reset MAPS between runs on a given function.
****/

bool ResetGrid(const PSGrid &G1);
/****
INPUT: A PSGrid
OUTPUT: If it copies G1 it returns true, else it returns false.
EFFECT: If status==READY, it copies G1 to G, else no effect.
****/

bool ResetDesign(const InitialDesign &I);
/****
INPUT: An InitialDesign.
OUTPUT: If it copies I it returns true, else it returns false.
EFFECT: If status==READY and I.IsDesignInitialized()==true, it copies
I to ID, else no effect.
****/

bool ResetSearchCriterion(const SearchCriterion &S);
```

```
/****
INPUT: A SearchCriterion.
OUTPUT: If it copies S it returns true, else it returns false.
EFFECT: If status==READY, it copies S to SC, else no effect.
****/

bool ResetPatternSearch(const PatternSearch &PS1);
/****
INPUT: A PatternSearch object.
OUTPUT: If it copies S it returns true, else it returns false.
EFFECT: If status==READY, it copies PS1 to PS, else no effect.
****/

bool ResetFunction(char* fn);
/****
INPUT: String containing the name of the function to be optimized.
OUTPUT: Returns true if file exists and is executable.
EFFECT: Fname gets fn.
****/

/*****QUERY FUNCTIONS*****/

bool RetrieveBestSeen(mp_vector* bpoint, double* bvalue) const;
/****
INPUT: Two Pointers, one for a mp_vector, one for a value
OUTPUT: True if status==DONE, false if otherwise.
EFFECT: If status==DONE, it finds the best point and its value.
Malloc's memory for a copy, and assigns it to the pointers.  This
will be done with a linear search.
****/

void debug(char* code) const;
/****
EFFECT: Does a complete state dump to stdout.  This should do
state dumps of all components.
*****/

void GenerateMAPSGraphics(char* fpref) const;
/****
INPUT: Filename prefix.
EFFECT: Outputs to fpref.MAPS.dat stuff that would be nice in
producing pictures via GNUplot.  Due to limits in visualization
technology, this feature is only available for functions from R or R^2.
To produce said pictures in GNUPlot, do the following for f:R^2 -> R:
gnuplot> set parametric
gnuplot> splot "fpref.MAPS.NAME.dat" with lines
To produce said pictures in GNUPlot, do the following for f:R -> R:
```

```
  gnuplot> plot "fpref.MAPS.NAME.dat" with lines

  GenerateMAPSGraphics will also produce a file with the evaluation
  sites in it, that can be graphed with the above plots as such:
  gnuplot>splot "fpref.MAPS.GRAPH.dat" with lines, "fpref.MAPS.PTS.dat" with points
  or:
  gnuplot>plot "fpref.MAPS.GRAPH.dat" with lines, "fpref.MAPS.PTS.dat" with points
  ****/

  void GenerateIterationHistory() const;
  /****
  EFFECT: Outputs the chosen points, their values and the best point
  and values returned to stdout.  To convert these data files into LaTeX
  tables with ease, use the included mdat2tex.awk script with the
  following syntax :
  awk -f mdat2tex.awk <mapsdatafile>
  awk will output the LaTeX to stdout.
  ****/

  /*****ACTION FUNCTIONS*****/
  bool RunMAPS(){return RunMAPS(NULL);}

  bool RunMAPS(char * fpref);
  /****
  INPUT: a pointer that specifies the filename prefix for
  iteration-by-iteration graphics.  Set this to NULL for no graphics, or
  call the parameter-free version above.
  OUTPUT: Returns true if nothing crashed.
  EFFECT: Assume status=READY.  Then it evaluates the function at the
  initial design sites. Then it begins the iterative process of looping
  until we use up the evaluation budget (or converge). When it finishes,
  it sets status=MAPS_DONE.  This will call EvalF.  If fpref!=NULL, then
  this will output graphics to files named :
  <fpref><C>.MAPS.AP.dat and <fpref><C>.MAPS.SC.dat .
  ****/

private:

  void SetDefault(long V1, long P1, Bounds &B1, char*fn);
  /****
  INPUT: The evaluation budget, dimension of space, bounds and
  function name.
  EFFECT: This will automagically take care of generating an
  InitialDesign, PSGrid, Search Criterion and a PatternSearch.
  Life couldn't be easier. Sets status=READY.  This gives you
  SearchCriterion's default, Latin Hypercube initial design and
  a CompassSearch.
```

```
  ****/

  static bool PCSubset(const pt_collect &A, const pt_collect &B);
  /****
  INPUT: Two collections of points, with each point forming a row vector
  OUTPUT: Returns true if A is a subset of B.
  ****/

  static bool IsElement(const mp_vector &A, const pt_collect &B);
  /****
  INPUT: A point, and a collection of points.
  OUTPUT: Returns true if A is in B.
  ****/

  double EvalF(const mp_vector& point, bool &success);
  /****
  INPUT: The point at which to evaluate the function, a boolean flag.
  OUTPUT: The function value.
  EFFECT: This function will do the fork/exec dance, sending the
  program the parameters on its stdin, then after waiting, it will get a
  double from the program's stdout.  If the program exited with an error,
  it will return ERROR and set success to false.  Otherwise success will
  be true.
  ****/

  long V;                 /*Evaluation Budget*/
  long N;                 /*Initial Design Allotment*/
  long C;                 /*Current Point Number*/
  long P;                 /*Dimension of the Space*/
  Bounds b;               /*Problem Bounds*/
  PSGrid *g;              /*The Grid - a real matrix*/
  pt_collect* Eval_c;     /*List of Evaluated Points - stored by row not column*/
  mp_vector* f_values;    /*List of Function Values*/
  char* Fname;            /*File Name of function to be optimized*/
  int status;             /*See above #defines*/
  long x_c;               /*The x_c'th element of Eval_c contains the current point*/
  InitialDesign* ID;      /*Initial Design*/
  SearchCriterion* SC;    /*Search Criterion*/
  PatternSearch* PS;      /*This will take care of optimizing S_c*/
};/*end class*/

#endif
```

E.1.2 SearchCriterion

```
/****
```

```
SearchCriterion Class - 7/15/99 cmsief
This is the search criterion class.
****/
#ifndef _MAPS_SC_
#define _MAPS_SC_

#include "maps_general.h"
#include "Approximate.h"
#include "NullApprox.h"
#include "KrigApprox.h"

/*weighting schemes*/
#define SC_WEIGHT_MULTIPLE 0
#define SC_WEIGHT_TARGET_VALUE 1
#define SC_WEIGHT_DELTA_MULTIPLE 2
#define SC_WEIGHT_DYNAMIC_GUESS 3
#define SC_WEIGHT_DYNAMIC_BEST 4

class SearchCriterion {

public:
  SearchCriterion();
  SearchCriterion(long dim,Bounds &b);
  /*prep work and invokes SetDefault*/
  SearchCriterion(const SearchCriterion &S);
  SearchCriterion(Approximate &A, Bounds &b);
  ~SearchCriterion();
  SearchCriterion& operator=(SearchCriterion &S);

  /*****RESET FUNCTIONS*****/

  bool ResetSearchCriterionState(Approximate &A, int scheme_number,
 mp_vector &params);
  /****
  INPUT: An Approximation, a weight scheme number and its parameters.
  OUTPUT: returns true iff ResetWeightScheme returns true.
  EFFECT: Calls ResetApproximateion, and ResetWeightScheme.
  ****/

  void ResetApproximation(Approximate &A);
  /****
  INPUT: An Approximation.
  EFFECT: (*APX) = A; P=A.GetDim();
  ****/

  bool ResetWeightScheme(int scheme_number, mp_vector &params);
  /****
```

```
INPUT: The scheme number and its parameters.
OUTPUT: Returns true if everything is valid.
EFFECT: This sets the w_scheme integer, and then sets w_params.
This will also set w_c to the correct value, based on the current
evaluation number.
****/

/*****QUERY FUNCTIONS*****/

void debug(char *code) const;
/****
EFFECT: does a complete state dump to stdout.
****/

/*****ACTION FUNCTIONS*****/

void EvaluateSearchCriterion(long P, double *x, double &f, int &success);
/****
INPUT: The dimension of the space, a list of values, a reference
passed value of the function, and a reference-passed flag.  The input
syntax is this convoluted to ensure that it works with Liz Dolan's
PatternSearch.
EFFECT: This evaluates the search criterion at a given point,
calling Approximate::EvaluateApprox to do so, and takes the weighting
schedule into account.  If the evaluation was a success, then success is
1, else it is set to 0.
****/

bool UpdateSearchCriterion(pt_collect &points, mp_vector& values, long
                           numpoints, long n, double delta);
/****
INPUT: The list of evaluated sites, the values, and the number of
points listed, the number of points in the initial design, and the
curent delta.
OUTPUT: Returns true if the operation was sucessful.
EFFECT: This will call Approximate::UpdateApprox to update the
approximation, then will update the weighting scheme by calling the
appropriate UpdateWeight function.
****/

friend void OptimizeSearchCriterion(void* SC,long dimx, double *x,
    double &f, int &success)
  {(*(SearchCriterion*)SC).EvaluateSearchCriterion(dimx,x,f,success);}
/****
INPUT: Modified PatternSearch calling scheme.  See PatternSearch.h
EFFECT: Indirection to get around C++'s inability to let function pointer
point to class member functions.  This gets optimized by PatternSearch.
```

```
  ****/

  void GetApproximationGraphics(long P, double *x, double &f, int &success);
  /***
  INPUT: Modified PatternSearch Interface
  OUTPUT: Value of the Approximation only at that point.  Meant for use by
  GenerateMAPSGraphics().
  ****/

private:

  void SetDefault(long dim);
  /****
  INPUT: Dimension of space.
  EFFECT: Automagically generates one of these objects, uses (2,0.8)
  WeightMultiple scheme and Kriging Approximation.
  ****/

  /*Weighting Scheme functions*/

  void CreateWeightTargetValue(mp_vector &params);
  /*This does the computation of w_params[3]
    m = (p[1] / p[0])^(1/p[2])*/
  void UpdateWeightTargetValue(long c);
  /****
  INPUT: The iteration number.
  EFFECT: Multiply w_c by the correct multiplier in order to reach
  the level of insignificance by the specified iteration.
  PARAMS FORMAT: [0] - w_0
                 [1] - target value of insignificance
                 [2] - iteration number to reach insignificance
  Though w_params[3] is not specified by the user, when you install the
  Target Value weighting scheme, the multiple is computed and stored here.
  ****/

  void UpdateWeightMultiple(long c);
  /****
  INPUT: The iteration number.
  EFFECT: This will multiply w_c by w_params[1]
  PARAMS FORMAT: [0] - w_0
                 [1] - m, s.t. w_c = m * w_{c-1}
  ****/

  void UpdateWeightDeltaMultiple(double curr_delta);
  /****
  INPUT: current delta
  EFFECT: If delta has been refined, the weight is multiplied by w_1
```

```
  PARAMS FORMAT: [0] - w_0, the weight before delta refinement
                 [1] - m, s.t. w_c = m*w_{c-1}, if delta was refined last iteration
                                  = w_{c-1}, if delta wasn't refined last iteration
  ****/

  void UpdateWeightDynamicGuess(long lastsite, mp_vector& values);
  /****
  INPUT: number of last site eval'd and vector of function values
  EFFECT: If the approximation was 'close' to correct at the last site, then we
  reduce the weight, less we increase it.
  PARAMS FORMAT: [0] - w_0
                 [1] - t, the threshhold percentage
                 [2] - m_b, multiplier if approximation is 'bad'
                 [3] - m_g, multiplier if approximation is 'good'

                 w_c=m_g * w_{c-1}, if |f(x_{c-1}) -\hat{f_{c-1}(x_c-1)}| <= t*frange
                    =m_b * w_{c-1}, otherwise
  ****/

  void UpdateWeightDynamicBest(long lastsite, mp_vector& values);
  /****
  INPUT: number of last site eval'd and vector of function values
  EFFECT: If the approximation was 'close' to correct at the last site, then we
  reduce the weight, less we increase it.  Similar to DynamicGuess, except it uses
  |expval - bestpoint| / 2 as the threshold.
  PARAMS FORMAT: [0] - w_0
                 [1] - m_b, multiplier if approximation is 'bad'
                 [2] - m_g, multiplier if approximation is 'good'

                 w_c=m_g * w_{c-1}, if |f(x_{c-1}) -\hat{f_{c-1}(x_c-1)}| <= |expval-best|/2
                    =m_b * w_{c-1}, otherwise
  ****/


  Bounds *bds;             /*the bounds*/
  long P;                  /*dimension of space*/
  int w_scheme;            /*Weighting scheme number*/
  mp_vector* w_params;     /*Parameters for that scheme.  In general w_params[0]
                            will always be w_0*/
  long np;                 /*The number of points evaluated so far*/
  Approximate* APX;        /*Some kind of Approximation*/
  pt_collect* Eval_c;      /*A pointer to the evaluated points list*/
  mp_vector* f_values;     /*A pointer to the function vals*/
  double w_c;              /*w_c for the current iteration*/
  double original_delta;   /*original delta - for Delta Based Weight Scheme*/

};/*end class*/
```

```
#endif
```

E.1.3 InitialDesign

```
/****
InitialDesign Class - 5/24/99 cmsief
This is the parent class from which all InitialDesign derived classes
shall be derived.  Derived classes will include: Latin Hypercubes.
****/

#ifndef _MAPS_ID_
#define _MAPS_ID_

#include "maps_general.h"
#include "PSGrid.h"

class InitialDesign {

public:
  InitialDesign(); /*default constructor*/
  /*init_done=false*/
  InitialDesign(long dim);  /*sets the dimension of the space*/
  /*init_done=false*/
  InitialDesign(const InitialDesign &I); /*cc*/
  virtual ~InitialDesign();
  virtual InitialDesign& operator=(const InitialDesign &I);

  void SetDimension(long dim);
  /****
  INPUT: The dimension of the problem
  EFFECTS: P=dim;
  ****/

  virtual bool GenerateDesign(const Bounds &b, long num_points, const PSGrid
                              &grid, long stream);
  /****
   INPUT: The problem's bounds, the number of points to put in the design,
   the grid and a random stream number.  This assumes, of course that you've set
   up rngs right.
   OUTPUT: Returns true if the operation was sucessful.
   EFFECTS: Sets init_done=true, and calls MoveToGrid(b,grid)
   Derived classes must call this..
  ****/

  void GetDesign(pt_collect* outptr);
  /****
```

```
  INPUT: Pointer of type pt_collect*.
  EFFECTS: Copies points to outptr if init_done==true.
  ****/

  long GetDesignSize() {return N;}
  /****
  OUTPUT: returns N.
  ****/

  bool IsDesignInitialized()const {return init_done;}
  /****
  OUTPUT: Returns true iff design is done.
  ****/

  virtual void debug(char *code);
  /****
  EFFECT: Outputs a state dump of the object.
  ****/

protected:
  long P; /*dimension*/
  long N; /*number of points for design*/
  pt_collect* points;

private:
  void MoveToGrid(const Bounds &b, const PSGrid &grid);
  /****
   INPUT: The problem's bounds and grid.
   EFFECTS: Moves all the points in 'points' to grid position.
   Calls grid.SnapToGrid, and uses that to do the snapping.
   Then it just fixes the point list.
  ****/

  bool init_done;

};/*end class*/

#endif
```

E.1.4 LatinHCDesign

```
/****
LatinHCDesign Class - 5/18/99 cmsief
Derived from InitialDesign
****/
```

```
#ifndef _MAPS_ID_LHC_
#define _MAPS_ID_LHC_

#include "maps_general.h"
#include "InitialDesign.h"
#include "rngs.h"
#include "rvgs.h"

class LatinHCDesign: public InitialDesign {

public:
  LatinHCDesign(); /*default constructor*/
  LatinHCDesign(long dim);  /*sets the dimension of the space*/
  LatinHCDesign(const LatinHCDesign &I);
  ~LatinHCDesign();

  bool GenerateDesign(const Bounds &b, long num_points,
      const PSGrid &grid, long stream);
  /****
   INPUT: The problem's bounds, the number of points to put in
   the design, and the grid.
   OUTPUT: Returns true if the operation was sucessful.
   EFFECTS: Generates the initial design via latin hypercubes.
   Then calls InitialDesign's GenerateDesign.
  ****/

};/*end class*/

#endif
```

E.1.5 PSGrid

```
/****
PSGrid class - 6/4/99 cmsief
This deals with the grid.  It assumes Compass Search for the
Core Pattern.  In terms of the grid object, view it as if it
were a real honest to God matrix, unlike its counterpart in maps.h.

NOTE: When choosing your grid or delta, it is probably a good
idea that you pick them s.t. delta*grid[i][i] divides evenly
into the amount of space in the ith dimension.  This allows the
program to be guaranteed to pick up on minima on the boundaries.
SetDefault should do this automatically.

ANOTHER NOTE: The implementation of SnapToGrid only deals with
grids parallel to the bounds.  You will have to re-enable the LU
```

```
factorization in order to deal with these other grids.  Sorry.
****/

#ifndef _MAPS_PSGRID_
#define _MAPS_PSGRID_
#include "maps_general.h"
#include "lu.h"

#define PSG_DEFAULT_RESOLUTION 0.01
/*aka the default grid resolution is 1/100th of the space*/
#define PSG_DEFAULT_REFINE 0.5
/*aka halve delta by default*/


class PSGrid {
public:
  PSGrid();
  PSGrid(const PSGrid &g);
  PSGrid(long P1, const mp_vector& o, const pt_collect& g,
 double ref, double d);
  PSGrid(long P1, const Bounds &b);
  /*if you aren't sure what to use, use this.*/
  PSGrid(long P1, const Bounds &b, const mp_vector &resolution);
  /*Resolution is in actual units, not percent of the space*/

  ~PSGrid();
  PSGrid & operator=(PSGrid &g);

  /*****RESET FUNCTIONS*****/

  void ResetPSGridState(long P1, const mp_vector& o, const pt_collect& g,
double ref, double delta);
  /****
  INPUT: A complete set of state values for the grid.
  EFFECT: It copies these into the object's state.  As for your
  choice of delta and the grid, see the note above.
  ****/

  bool ReadGridFromFile(char* fn);
  /****
  INPUT: File name string.
  OUTPUT: Returns true if read was sucessful.
  EFFECT: Reads the grid data from a file.
  FILE FORMAT: dimension<cr>
               delta<cr>
               origin(P)<cr>
               refine(P)<cr>
```

```
             grid(PxP)
NOTES: Watch VERY carefully what you do with directions,
when you write your grid to a file, assume that it is written
like a real matrix. Also, keeping the grid vectors positive
would be a good idea.  As for your choice of delta and the
grid, see the note above.
****/

/*****QUERY FUNCTIONS*****/

void debug(char *code) const;
/****
EFFECT: does a complete state dump to stdout.
****/

double GetDelta() const;
/****
OUTPUT: returns delta.
****/

bool IsFeasible(const Bounds &b, const mp_vector &x) const;
/****
INPUT: The bounds and an mp_vector.
OUTPUT: True if the point is within the founds, false if otherwise
****/

long FindBestPSMultiple(const Bounds &b);
/****
INPUT: The bounds of the space.
OUTPUT: Finds the best multiple for delta for use in the PatternSearch.
It does so with the following methodology:
1) Find i s.t. i maximizes delta*grid[i] / b.upper[i]-b.lower[i].
2) Returns half the biggest multiple s.t.
   mult * delta*grid[i] < b.upper[i]-b.lower[i].
****/

/*****ACTION FUNCTIONS*****/

void SnapToGrid(const Bounds &b, const mp_vector &old_pt,
mp_vector* new_pt) const;
/****
INPUT: The bounds and two mp_vectors - one reference and one pointer.
OUTPUT: The nearest grid point to old_pt will be returned in new_pt
****/

void GetCore(const Bounds &b, const mp_vector &x,
     pt_collect& core, long &pts);
```

```
  /****
  INPUT: The bounds, a point, a reference to point collection
  to be returned, and a reference passed number of points.
  EFFECT: Returns a the list of points in the Core Pattern of x,
  and the number of points in this pattern. This should also call
  IsFeasible, and return only those points within the bounds. Note
  that the returned values are stored on the rows (aka each row
  represents a point in P-space), much as Eval_c is stored in maps.h.
  ****/

  void RefineGrid();
  /****
  EFFECT: Actually refines the grid. delta *= refine;
  ****/

private:

  void SetDefault(long P1, const Bounds &b, const
  mp_vector &resolution);
  /****
  INPUT: Dimension, bounds and a grid resolution.
  EFFECT: This will automagically generate a simple grid
  that is parallel to the axes.  It will be similar to the
  default settings you give it, but not exactly that.
  ****/

  long P;               /*dimension of space*/
  mp_vector* origin;    /*The origin for the purpose of the grid (P)*/
  pt_collect* gridvect; /*The grid matrix.  This is in terms of
  multiples of delta, hence it does not
  take scaling into account in and of itself.*/
  double refine;        /*Grid refinement constant - aka what delta is
                          multiplied by when the grid is refined*/
  double delta;         /*delta from PatternSearch - it is the
  current 'step size'*/

};/*end class*/

#endif
```

E.2 Code

E.2.1 MAPS

```
/*MAPS IMPLEMENTATION FILE - 6/17/99*/

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include "maps.h"
#include "CompassSearch.h"

/*Iteration History Field Widths*/
#define _GFN 4
#define _GFP 10
#define _GFV 10

maps::maps() {
/*status=NOTINIT*/
  V=N=C=P=x_c=0;
  status=NOTINIT;g=NULL;
  Eval_c=NULL;f_values=NULL;Fname=NULL;
  ID=NULL;SC=NULL;PS=NULL;
}/*end default constructor*/

maps::maps(long V1, long N1, long P1, const Bounds &B1, char* fn,
   const PSGrid &G1, const InitialDesign &I,
   const SearchCriterion &S1, const PatternSearch &PS1) {
/*Prep work and call ResetMAPSState*/
  ID=NULL; SC=NULL; PS=NULL;g=NULL;
  ResetMAPSState(V1,N1,P1,B1,fn,G1,I,S1,PS1);
}/*end special constructor*/

maps::maps(long V1, long P1, Bounds &B1, char*fn) {
  /*Prep work and call SetDefault*/
  ID=NULL; SC=NULL; PS=NULL;g=NULL;
  SetDefault(V1,P1,B1,fn);
}/*end special constructor*/
```

```
maps::maps(maps& old){
  ID=NULL;SC=NULL;PS=NULL;g=NULL;
  bool success=true;
  V=old.V;N=old.N;P=old.P;b=old.b;x_c=old.x_c;
  status=READY;/*just temporary*/
  Fname=NULL;
  if(old.Fname!=NULL) success=ResetFunction(old.Fname);
  g=NULL;
  if(old.g!=NULL) success=success&&ResetGrid(*old.g);
  ID=NULL;
  if(old.ID!=NULL) {
    ID = new InitialDesign;
    (*ID) =(*old.ID);
  }/*end if*/
  SC=NULL;
  if(old.SC!=NULL) success=success&&ResetSearchCriterion(*old.SC);
  PS=NULL;
  if(old.PS!=NULL) success=success&&ResetPatternSearch(*old.PS);
  if(old.Eval_c!=NULL) Eval_c = new pt_collect(*old.Eval_c);
  else Eval_c=NULL;
  if(old.f_values!=NULL) f_values = new mp_vector(*old.f_values);
  else f_values=NULL;

  status=old.status;
#if DEBUG>=1
  if(!success) cerr<<"MPCC   : One items in the copy failed.\n";
#endif
}/*end Copy Constructor*/

maps::~maps() {
  if(ID!=NULL) delete ID;
  if(SC!=NULL) delete SC;
  if(PS!=NULL) delete PS;
  if(Eval_c!=NULL) delete Eval_c;
  if(f_values!=NULL) delete f_values;
  if(Fname!=NULL) delete Fname;
  if(g!=NULL) delete g;
}/*end destructor*/

/*****RESET FUNCTIONS****/


bool maps::ResetMAPSState(long V1, long N1, long P1, const Bounds &B1,
  char* fn, const PSGrid &G1,
  const InitialDesign &I1,
  const SearchCriterion &S1,
                        const PatternSearch &PS1) {
```

```
/****
INPUT: A partial set of state parameters for MAPS
OUTPUT: Returns true if the operation was successful.
EFFECT: Sets all the parameters, and sets status=READY,
clears out Eval_c and f_values.  This wipes out everything
else entirely.
*****/
  bool success;
  NewMAPSRun();/*wipes out Eval_c and f_values*/
  V=V1;N=N1;P=P1;b=B1;
  success=ResetFunction(fn)&&ResetGrid(G1)&&ResetDesign(I1)&&
    ResetSearchCriterion(S1)&&ResetPatternSearch(PS1);
#if DEBUG>=1
  if(!success) cerr<<"MPRMS  : One or more resets failed.\n";
#endif
  return success;
}/*end ResetMAPSState*/

void maps::NewMAPSRun(){
/****
EFFECT: If status=MAPS_DONE, clears the Eval_c and F_values,
resets C=0 and sets status=READY. This allows you to reset MAPS
between runs on a given function.
****/
  if(Eval_c!=NULL) {delete Eval_c;Eval_c=NULL;}
  if(f_values!=NULL) {delete f_values;f_values=NULL;}
  C=0;x_c=0;
  status=READY;
}/*end NewMAPSRun*/


bool maps::ResetGrid(const PSGrid &G1) {
/****
INPUT: A PSGrid
OUTPUT: If it copies G1 it returns true, else it returns false.
EFFECT: If status==READY, it copies G1 to G, else no effect.
****/
  if(status==READY) {
    if(g!=NULL) delete g;
     g = new PSGrid(G1);
    return true;
  }/*end if*/
  else return false;
}/*end ResetGrid*/

bool maps::ResetDesign(const InitialDesign &I) {
/****
```

```
INPUT: An InitialDesign.
OUTPUT: If it copies I it returns true, else it returns false.
EFFECT: If status==READY and I.IsDesignInitialized()==true,
it copies I to ID, else no effect.
****/
 if(status==READY&&I.IsDesignInitialized()) {
    if(ID!=NULL) delete ID;
    ID = new InitialDesign;
    (*ID) =I;
    N=(*ID).GetDesignSize();
    return true;
  }/*end if*/
  else return false;
}/*end ResetDesign*/

bool maps::ResetSearchCriterion(const SearchCriterion &S){
/****
INPUT: A SearchCriterion.
OUTPUT: If it copies S it returns true, else it returns false.
EFFECT: If status==READY, it copies S to SC, else no effect.
****/
  if(status==READY) {
    if(SC!=NULL) delete SC;
    SC = new SearchCriterion(S);
    return true;
  }/*end if*/
  else return false;
}/*end ResetSearchCriterion*/

bool maps::ResetPatternSearch(const PatternSearch &PS1) {
/****
INPUT: A PatternSearch object.
OUTPUT: If it copies S it returns true, else it returns false.
EFFECT: If status==READY, it copies PS1 to PS, else no effect.
****/
  if(status==READY) {
    if(PS!=NULL) delete PS;
    PS = new CompassSearch;
    (*PS) = PS1;
    return true;
  }/*end if*/
  else return false;
}/*end ResetPatternSearch*/

bool maps::ResetFunction(char* fn){
/****
INPUT: String containing the name of the function to be optimized.
```

```
OUTPUT: Returns true if file exists and is executable.
EFFECT: Fname gets fn.
****/
  struct stat m;

  if((stat(fn,&m)==0)&&(m.st_mode&&S_IXUSR) ) {
    /*if it exists and the owner can execute it.
      This doesn't check ownership and permissions.
    */
    if(Fname!=NULL) free(Fname);
    Fname=(char*)malloc(strlen(fn)+1);
    strcpy(Fname,fn);
    return true;
  }
  return false;
}/*end ResetFunction*/

  /*****QUERY FUNCTIONS*****/

bool maps::RetrieveBestSeen(mp_vector* bpoint, double* bvalue) const{
/****
INPUT: Two Pointers, one for a mp_vector, one for a value
OUTPUT: True if status==DONE, false if otherwise.
EFFECT: If status=DONE, it finds the best point and its value.
Malloc's memory for a copy, and assigns it to the pointers.  This
is done using a linear search.
****/
  long best;
  double bval;

  if(status==MAPS_DONE) {
    best=0;bval=(*f_values)[0];
    for(long i=0;i<V;i++)
      if((*f_values)[i]<bval) {
        bval=(*f_values)[i];
        best=i;
      }/*end if*/

    (*bpoint)=(*Eval_c).row(best);
    (*bvalue)=bval;
    return true;
  }/*end if*/
  else return false;
}/*end RetrieveBestSeen*/

void maps::debug(char* code) const{
/****
```

```
EFFECT: Does a complete state dump to stdout.  This should do state dumps of
all components.
*****/
  cout<<code<<":***maps::debug()***\n";
  cout<<code<<":V="<<V<<" N="<<N<<" C="<<C<<" P="<<P<<" x_c="<<x_c<<endl;
  cout<<code<<":status=";
  switch(status) {
  case NOTINIT:
    cout<<"NOTINIT";break;
  case READY:
    cout<<"READY";break;
  case MAPS_DONE:
    cout<<"MAPS_DONE";break;
  default:
    cout<<"ERROR!";
  };/*end switch*/
  cout<<" Fname="<<Fname<<endl;

  cout<<code<<":b.lower="<<b.lower;
  cout<<code<<":b.upper="<<b.upper;
  if(Eval_c!=NULL) cout<<code<<":Eval_c="<<(*Eval_c);
  else cout<<code<<":Eval_c=NULL!\n";
  if(f_values!=NULL) cout<<code<<":f_values="<<(*f_values);
  else cout<<code<<":f_values=NULL!\n";
  if(g!=NULL) (*g).debug(code);
  else cout<<code<<":g=NULL!\n";
  if(SC!=NULL) (*SC).debug(code);
  else cout<<code<<":SC=NULL!\n";
  cout<<code<<":***end MP:debug()***\n";
}/*end debug*/

void maps::GenerateMAPSGraphics(char* fpref) const{
/****
INPUT: Filename prefix.
EFFECT: Outputs to fpref.MAPS.dat stuff that would be nice
in producing pictures via GNUplot.  Due to limits in
visualization technology, this feature is only available for
functions from R or R^2.
To produce said pictures in GNUPlot, do the following for f:R^2 -> R:
gnuplot> set parametric
gnuplot> splot "fpref.MAPS.NAME.dat" with lines
To produce said pictures in GNUPlot, do the following for f:R -> R:
gnuplot> plot "fpref.MAPS.NAME.dat" with lines

GenerateMAPSGraphics will also produce a file with the evaluation sites
in it, that can be graphed with the above plots as such:
gnuplot> splot "fpref.MAPS.GRAPH.dat" with lines, "fpref.MAPS.PTS.dat" with points
```

```
or:
gnuplot> plot "fpref.MAPS.HRAPH.dat" with lines, "fpref.MAPS.PTS.dat" with points
****/

  if(status!=MAPS_DONE && status!=MAPS_RUNNING) return;

  if(status==MAPS_DONE) {
    (*SC).UpdateSearchCriterion(*Eval_c,*f_values,V,N,(*g).GetDelta());
  }/*end if*/
  /*this is to update the SC for the last point.*/

  if(P==1) {
    ofstream ofs_sc,ofs_ap,ofs_gf;
    char* fn_sc = (char*)malloc(strlen(fpref) +13);
    char* fn_ap = (char*)malloc(strlen(fpref) +13);
    char* fn_gf = (char*)malloc(strlen(fpref) +14);
    double* y=(double*)malloc(sizeof(double)*P);
    double f;
    int s;
    mp_vector diff=(b.upper-b.lower) * (1.0/500.0);

    sprintf(fn_sc,"%s.MAPS.SC.dat",fpref);
    sprintf(fn_ap,"%s.MAPS.AP.dat",fpref);
    sprintf(fn_gf,"%s.MAPS.PTS.dat",fpref);
    ofs_sc.open(fn_sc,ios::out);
    ofs_ap.open(fn_ap,ios::out);
    ofs_gf.open(fn_gf,ios::out);
    ofs_sc<<"# This Graphics File has been brought to you by MAPS\n"
          <<"# implemented by: Chris Siefert, College of William and Mary\n"
          <<"# for parameters V="<<V<<" N="<<N<<" P="<<P;
    if(g!=NULL) ofs_sc<<" delta="<<(*g).GetDelta();
    ofs_sc<<endl
          <<"# on function: "<<Fname<<endl;
    ofs_sc<<"# Current Search Criterion Graph:\n";
    ofs_ap<<"# This Graphics File has been brought to you by MAPS\n"
          <<"# implemented by: Chris Siefert, College of William and Mary\n"
          <<"# for parameters V="<<V<<" N="<<N<<" P="<<P;
    if(g!=NULL) ofs_ap<<" delta="<<(*g).GetDelta();
    ofs_ap<<endl
          <<"# on function: "<<Fname<<endl;
    ofs_ap<<"# Current Approximation Graph:\n";
    ofs_gf<<"# This Graphics File has been brought to you by MAPS\n"
          <<"# implemented by: Chris Siefert, College of William and Mary\n"
          <<"# for parameters V="<<V<<" N="<<N<<" P="<<P;
    if(g!=NULL) ofs_gf<<" delta="<<(*g).GetDelta();
    ofs_gf<<endl
          <<"# on function: "<<Fname<<endl;
```

```
  ofs_gf<<"# Current Evaluated sites:\n";

  for(long m=0;m<C;m++)
    ofs_gf<<setprecision(4)<<(*Eval_c)[m][0]<<" "<<(*f_values)[m]<<endl;

  for(double k=b.lower[0];k<b.upper[0];k+=diff[0]) {
      y[0]=k;
      (*SC).EvaluateSearchCriterion(P,y,f,s);
      ofs_sc<<setprecision(4)<<y[0]<<" "<<setprecision(8)<<f<<"\n";
      (*SC).GetApproximationGraphics(P,y,f,s);
      ofs_ap<<setprecision(4)<<y[0]<<" "<<setprecision(8)<<f<<"\n";
    }/*end for*/

  free(y);
  free(fn_sc);
  free(fn_ap);
  free(fn_gf);
  ofs_gf.close();
  ofs_sc.close();
  ofs_ap.close();
}/*end if P==1*/
else if(P==2) {
  ofstream ofs_sc,ofs_ap, ofs_gf;
  char* fn_sc = (char*)malloc(strlen(fpref) +13);
  char* fn_ap = (char*)malloc(strlen(fpref) +13);
  char* fn_gf = (char*)malloc(strlen(fpref) +14);
  double* y=(double*)malloc(sizeof(double)*P);
  double f;
  int s;
  mp_vector diff=(b.upper-b.lower) * (1.0/50.0);

  sprintf(fn_sc,"%s.MAPS.SC.dat",fpref);
  sprintf(fn_ap,"%s.MAPS.AP.dat",fpref);
  sprintf(fn_gf,"%s.MAPS.PTS.dat",fpref);
  ofs_sc.open(fn_sc,ios::out);
  ofs_ap.open(fn_ap,ios::out);
  ofs_gf.open(fn_gf,ios::out);

  ofs_sc<<"# This Graphics File has been brought to you by MAPS\n"
        <<"# implemented by: Chris Siefert, College of William and Mary\n"
        <<"# for parameters V="<<V<<" N="<<N<<" P="<<P;
  if(g!=NULL) ofs_sc<<" delta="<<(*g).GetDelta();
  ofs_sc<<endl
        <<"# on function: "<<Fname<<endl;
  ofs_sc<<"# Current Search Criterion Graph:\n";
  ofs_ap<<"# This Graphics File has been brought to you by MAPS\n"
        <<"# implemented by: Chris Siefert, College of William and Mary\n"
```

```
          <<"# for parameters V="<<V<<" N="<<N<<" P="<<P;
    if(g!=NULL) ofs_ap<<" delta="<<(*g).GetDelta();
    ofs_ap<<endl
          <<"# on function: "<<Fname<<endl;
    ofs_ap<<"# Current Approximation Graph:\n";

    ofs_gf<<"# This Graphics File has been brought to you by MAPS\n"
          <<"# implemented by: Chris Siefert, College of William and Mary\n"
          <<"# for parameters V="<<V<<" N="<<N<<" P="<<P;
    if(g!=NULL) ofs_gf<<" delta="<<(*g).GetDelta();
    ofs_gf<<endl
          <<"# on function: "<<Fname<<endl;
    ofs_gf<<"# Current Evaluated sites:\n";

    for(long m=0;m<C;m++)
      ofs_gf<<setprecision(4)<<(*Eval_c)[m][0]<<" "
    <<(*Eval_c)[m][1]<<" "<<(*f_values)[m]<<endl;

    for(double i=b.lower[0];i<b.upper[0];i+=diff[0]) {
      y[0]=i;
      for(double j=b.lower[1];j<b.upper[1];j+=diff[1]) {
        y[1]=j;
        (*SC).EvaluateSearchCriterion(P,y,f,s);
        ofs_sc<<setprecision(4)<<y[0]<<" "<<y[1]<<" "<<setprecision(8)<<f<<"\n";
        (*SC).GetApproximationGraphics(P,y,f,s);
        ofs_ap<<setprecision(4)<<y[0]<<" "<<y[1]<<" "<<setprecision(8)<<f<<"\n";
      }/*end for*/
      ofs_sc<<"\n";
      ofs_ap<<"\n";
    }/*end for*/

    free(y);
    free(fn_sc);
    free(fn_ap);
    free(fn_gf);
    ofs_sc.close();
    ofs_ap.close();
    ofs_gf.close();
  }/*end ifP==2*/
  else return;
}/*end GenerateMAPSGraphics*/

void maps::GenerateIterationHistory() const{
/****
EFFECT: Outputs the chosen points, their values and the best point and values
returned to stdout.  To convert these data files into LaTeX tables with ease,
use the included mdat2tex.awk script with the following syntax :
```

```
awk -f mdat2tex.awk <mapsdatafile>
awk will output the LaTeX to stdout.
****/
  if(status!=MAPS_DONE) return;

  cout<<"******MAPS ITERATION HISTORY******\n"
      <<"# for parameters V="<<V<<" N="<<N<<" P="<<P;
  if(g!=NULL) cout<<" delta="<<(*g).GetDelta();
  cout<<endl
      <<"# on function: "<<Fname<<endl;

  cout<<setw(_GFN)<<"NUM ";
  for(long k=0;k<P;k++) cout<<setw(_GFP)<<k+1<<" ";/*+1 notation as per Trosset*/
  cout<<setw(_GFV)<<"VAL\n";

  for(long i=0;i<V;i++) {
    cout<<setw(_GFN)<<i+1<<" ";/*+1 notation as per Trosset*/
    for(long j=0;j<P;j++) cout<<setw(_GFP)<<setprecision(_GFP-3)<<(*Eval_c)[i][j]<<" ";
    cout<<setw(_GFV)<<setprecision(_GFV-2)<<(*f_values)[i]<<endl;
  }/*end if*/
  cout<<"\nBest Point found is #"<<x_c+1<<", value="<<setw(_GFV)<<(*f_values)[x_c]<<endl;
  /*+1 notation as per Trosset*/

  cout<<"**********************************\n";
}/*end GenerateIterationHistory*/

  /*****ACTION FUNCTIONS*****/

bool maps::RunMAPS(char * fpref){
/****
INPUT: a pointer that specifies the filename prefix for iteration-by-iteration
graphics.  Set this to NULL for no graphics, or call the parameter-free
version above.
OUTPUT: Returns true if nothing crashed.
EFFECT: Assume status=READY.  Then it evaluates the function at the initial
design sites. Then it begins the iterative process of looping until we use up
the evaluation budget. When it finishes, it sets status=MAPS_DONE.  This will
call EvalF.  If fpref!=NULL, then this will output graphics to files named :
<fpref><C>.MAPS.AP.dat and <fpref><C>.MAPS.SC.dat .
****/

  /*Sanity checks*/
  if(status!=READY||g==NULL||ID==NULL||SC==NULL||PS==NULL) {
#if DEBUG>=1
    cerr<<"MPRM   :MAPS not initialized correctly\n";
#endif
    return false;
```

```
  }/*end if*/

  if(Eval_c!=NULL) delete Eval_c;
  if(f_values!=NULL) delete f_values;
  status=MAPS_RUNNING;

  /*Variable Declarations*/
  pt_collect temp_eval(V,P);/*a size VxP array, for resizing Eval_c*/
  pt_collect temp_core(1,P);/*an array for the core pattern... this will be resized*/
  mp_vector prosp_pt(P);    /*a prospective point*/
  mp_vector prosp_pt2(P);   /*a prospective point... for the SnapToGrid Checks*/
  mp_vector temp_fval(V);   /*a size V array for resizing f_values*/
  long temp_core_pts;       /*number of points in the core pattern*/
  double temp_f;            /*value from a function evaluation*/
  bool success;             /*you just always need a bool:) */
  double *temp_pspoint;     /*pointer to feed values to PatternSearch's CleanSlate*/
  double *temp_retpoint;    /*pointer to get values back from PS*/
  int s;                    /*for calling EvaluateSearchCriterion*/
  long i,j;                 /*loop counters to the stars*/
  char *gfx_fp=NULL;        /*augmented fname for graphics*/
  bool converged=false;     /*stopping due to convergence*/
  if(fpref!=NULL) gfx_fp=(char*)malloc(1*sizeof(char));//just for 1st iteration


  /*Sanity Check - design size*/
  if(N!=(*ID).GetDesignSize()) {
#if DEBUG>=1
    cerr<<"MPRM   :WARNING - Initial Design Size(N) set incorrectly, resetting...\n";
#endif
    N=(*ID).GetDesignSize();
  }/*end if*/

  /*Memory Allocs*/
  Eval_c=new pt_collect(N,P);
  f_values=new mp_vector(N);

  /*Get initial design sites, and evaluate the function at them*/
  (*ID).GetDesign(Eval_c);
#if DEBUG>=3
  cout<<"MPRM   :Before evals...Eval_c="<<(*Eval_c);
#endif

  for(i=0;i<N;i++) {
    temp_f=EvalF((*Eval_c).row(i),success);

    if(success) {
      (*f_values)[i]=temp_f;
```

```
      temp_fval[i]=temp_f;
      for(j=0;j<P;j++) temp_eval[i][j]=(*Eval_c)[i][j];
    }/*end if*/
    else {
#if DEBUG>=1
      cerr<<"MPRM   :Evaluation failed at point="<<(*Eval_c).row(i);
#endif
      /*SUPPORT FOR EVALUATION FAILURE SHOULD BE ADDED HERE....*/
      abort();
    }/*end else*/
  }/*end for*/

  /*Find point with minimal value and make it x_c*/
  x_c=0;
  temp_f=(*f_values)[0];
  for(i=0;i<N;i++)
    if((*f_values)[i]<temp_f) {
      temp_f=(*f_values)[i];
      x_c=i;
    }/*end if*/


#ifdef ITERATE_DEBUG
  /*For outputting information iteration by iteration....*/
  for(long w=0;w<N;w++) {
    cout<<w<<" ";
    for(long r=0;r<P;r++) cout<<setprecision(10)<<(*Eval_c)[w][r]<<" ";
    cout<<(*f_values)[w]<<" delta="<<(*g).GetDelta()<<endl;
  }/*end for*/
#endif

  /*ENTER THE MAIN LOOP*/
  for(C=N;!converged&&C<V;C++) {

    /*Main block*/
#if DEBUG>=2
    cout<<"MPRM    :******BEGIN ITERATION C="<<C<<"******\n";
    cout<<"MPRM    :x_c="<<x_c<<endl;
    cout<<"MPRM    :Eval_c="<<(*Eval_c);
    cout<<"MPRM    :Best Value ="<<(*f_values)[x_c]<<endl;
    cout<<"MPRM    :Best Point ="<<(*Eval_c).row(x_c);
    (*g).debug("MPRM    ");
#endif


#ifdef ITERATE_DEBUG
    /*For outputting information iteration by iteration....*/
```

```
    if(C!=N) {
      cout<<C-1<<" ";
      for(long q=0;q<P;q++) cout<<setprecision(15)<<(*Eval_c)[C-1][q]<<" ";
      cout<<(*f_values)[C-1]<<" delta="<<(*g).GetDelta()<<endl;
    }
#endif

    /*Get the Core Pattern of x_c*/
    (*g).GetCore(b,(*Eval_c).row(x_c), temp_core, temp_core_pts);

    /*if all pts in core have been evaluated, refine the grid*/
    if(PCSubset(temp_core, *Eval_c)) {
      (*g).RefineGrid();
#if DEBUG>=2
      cout<<"MPRM   :Grid Refined to delta="<<(*g).GetDelta()<<endl;
#endif
      if((*g).GetDelta() < 10*TOLERANCE) {
        converged=true;
        break;
      }/*end if*/
    }/*end if*/

    /*Update the Search Criterion*/
    success=(*SC).UpdateSearchCriterion(*Eval_c,*f_values,C,N,(*g).GetDelta());
#if DEBUG>=2
    cout<<"MPRM  :Eval_c="<<*Eval_c;
    cout<<"MPRM  :f_values="<<*f_values;
    cout<<"MPRM  :Search Criterion Updated\n";
#endif

#if DEBUG>=1
    if(!success) cerr<<"MPRM   :Search Criterion Update FAILED!\n";
#endif

    /*Iteration-by-Iteration Graphics code*/
    if(fpref!=NULL) {
      i=C+1;j=1;
      while(i>=10) {i/=10;j++;} /*count # of digits in C*/
      realloc(gfx_fp,strlen(fpref) + j + 1);
      sprintf(gfx_fp,"%s%ld",fpref,C+1);
      GenerateMAPSGraphics(gfx_fp);
    }/*end if*/

    /*We might want to pick a better starting point than *Eval_c.row(x_c).  If
      so, that would be done here.*/

    /*OPTIMIZE THE SEARCH CRITERION*/
```

```
    /*Reset the pattern search*/
    temp_pspoint=(double*)malloc(P*sizeof(double));
    temp_retpoint=(double*)malloc(P*sizeof(double));
    for(i=0;i<P;i++) temp_pspoint[i]=(*Eval_c)[x_c][i];
    /*This reference here passes out the row's pointer, and ref's like an array*/

    (*PS).CleanSlate(P,temp_pspoint, (*g).FindBestPSMultiple(b)* (*g).GetDelta(),
                     (*g).GetDelta(), SC,b, OptimizeSearchCriterion);

#if DEBUG>=3
    cout<<"MPRM   :PS::Clean Slate Complete\n";
#endif
    /*Run the PS*/
    (*PS).ExploratoryMoves();
    (*PS).GetMinPointFLOP(temp_retpoint);

    for(i=0;i<P;i++) prosp_pt[i]=temp_retpoint[i];
    delete(temp_retpoint);
    delete(temp_pspoint);
#if DEBUG>=2
    cout<<"MPRM   :Search Criterion Optimized, point="<<prosp_pt;
#endif

    /*HERE WE NEED TO MAKE SURE THE POINT IS ON THE GRID (WHICH IT SHOULD BE, BUT
      IT NEVER HURTS TO CHECK) AND THAT THE POINT HAS NOT ALREADY BEEN EVALUATED
      AT.  IF EITHER CONDITION DOESN'T HOLD, WE HAVE TO DO SOMETHING HERE.*/

    (*g).SnapToGrid(b,prosp_pt,&prosp_pt2);

    if(prosp_pt!=prosp_pt2) {
#if DEBUG>=2
      cout<<"MPRM   :ERROR(non-fatal) Pattern Search Point not on Grid:"<<prosp_pt
          <<"MPRM   :Replaced with:"<<prosp_pt2;
#endif
      prosp_pt=prosp_pt2;
    }/*end if*/

    /*Check for previously evaluated site*/
    if(IsElement(prosp_pt, *Eval_c)) {
      /*this site has already been evaluated*/
      double best_core_val=ERROR;     /*Best S_c value in the core*/
      long best_core_num=ERROR;       /*Best S_C point in the core*/
#if DEBUG>=2
      cout<<"MPRM   :Site Already Eval'd, site="<<prosp_pt;
#endif
```

```
      /*Get the Core Pattern of this new point*/
      (*g).GetCore(b,prosp_pt, temp_core, temp_core_pts);

      /*NEED TO CHECK TRUTH VALUES HERE!!!!*/
      for(i=0;i<temp_core_pts;i++) {
        if(!IsElement(temp_core.row(i),*Eval_c)) {
          (*SC).EvaluateSearchCriterion(P,temp_core[i],temp_f, s);
          if(best_core_num==ERROR || temp_f<best_core_val){
            best_core_val=temp_f;
            best_core_num=i;
          }/*end if*/
        }/*end if*/
      }/*end for*/

      if(best_core_num==ERROR) {
#if DEBUG>=1
          cerr<<"MPRM   :No valid point found in core.\n";
          cerr<<"MPRM   :Reval'd point="<<prosp_pt;
#endif
  /*Not only have we evaluated the point our search criterion
    returns, but we've also evaluated its core pattern!
    Instead, we get the Core Pattern of x_c again, and eval
    one of those!*/
  (*g).GetCore(b,(*Eval_c).row(x_c), temp_core, temp_core_pts);
  success=false;
  for(i=0;!success&&i<temp_core_pts;i++) {
    if(IsElement(temp_core.row(i),(*Eval_c)))
      success=true;prosp_pt=temp_core.row(i);
      }/*end if*/
#if DEBUG>=2
  if(success)
    cerr<<"MPRM   :Using alternate point="<<prosp_pt;
  else{
    cerr<<"MPRM   :No available re-eval pt.  Dumping core.\n";
    abort();
  }/*end else*/
#endif
        }/*end if*/
      else
prosp_pt=temp_core.row(best_core_num);
    }/*end if*/

#if DEBUG>=2
    cout<<"MPRM   :Prospective Point chosen, point="<<prosp_pt;
#endif

    /*Evaluate the Function*/
```

```
    temp_f=EvalF(prosp_pt,success);

    if(success) {
      /*copy to temp_eval and temp_fval*/
      temp_fval[C] = temp_f;
      for(i=0;i<P;i++) temp_eval[C][i] = prosp_pt[i];

      /*realloc Eval_c and f_values*/
      (*Eval_c).newsize(C+1,P);
      (*f_values).newsize(C+1);

      for(i=0;i<C+1;i++) {
        (*f_values)[i]=temp_fval[i];
        for(j=0;j<P;j++) (*Eval_c)[i][j] = temp_eval[i][j];
      }/*end for*/

    }/*end if*/
    else {
#if DEBUG>=1
      cerr<<"MPRM   :Evaluation failed at point="<<prosp_pt;
#endif
      /*Evaluation failure has to be dealt with here*/
      abort();
    }/*end else*/

    /*Update x_c if needed*/
    if(temp_f < (*f_values)[x_c]) x_c=C;

  }/*end the main for loop*/

#if DEBUG>=1
  if(converged) cout<<"MPRM   :MAPS Terminating due to convergence"<<endl;
#endif

  if(fpref!=NULL) free(gfx_fp);
  status=MAPS_DONE;
  return true;
}/*end RunMAPS*/

void maps::SetDefault(long V1, long P1, Bounds &B1, char* fn) {
/****
INPUT: The evaluation budget, dimension of space, bounds and function name.
EFFECT: This will automagically take care of generating an InitialDesign,
PSGrid, Search Criterion and a PatternSearch.  Life couldn't be easier. Sets
status=READY.  This gives you SearchCriterion's default, Latin hypercube
initial design and a CompassSearch.
****/
```

```
  V=V1;P=P1;b=B1;
  Fname=NULL;
  if (ResetFunction(fn)) status=READY;
  else {
#if DEBUG>=1
    cerr<<"MPSD   :Executable File is not valid\n";
#endif
    status=NOTINIT;
  }/*end else*/

  C=0;x_c=0;
  N=V1/3; /*This is an arbitrary choice of N*/
  Eval_c=NULL;
  f_values=NULL;
  g=new PSGrid(P,b);
  ID=new LatinHCDesign(P);
  SC=new SearchCriterion(P,B1);
  PS=new CompassSearch;
  (*ID).GenerateDesign(b,N,*g,0);
}/*end SetDefault*/

bool maps::PCSubset(const pt_collect &A, const pt_collect &B) {
/****
INPUT: Two collections of points, with each point forming a row vector
OUTPUT: Returns true if A is a subset of B.
****/
  long AD=A.num_rows();
  long BD=B.num_rows();
  bool s=true;

  if(BD<AD) return false;

  for(long i=0;s&&i<AD;i++) {
    s=false;
    s=IsElement(A.row(i),B);
  }/*end for*/
  return s;
}/*end PCSubset*/


bool maps::IsElement(const mp_vector &A, const pt_collect &B) {
/****
INPUT: A point, and a collection of points.
OUTPUT: Returns true if A is in B.
****/
  long BD=B.num_rows();
  bool s=false;
```

```
  for(long j=0;!s&&j<BD;j++) {
    if(isnear(A,B.row(j),2*TOLERANCE)) s=true;
  }/*end for*/
  return s;
}/*end IsElement*/


double maps::EvalF(const mp_vector &point, bool &success) {
/****
INPUT: The point at which to evaluate the function, a boolean flag.
OUTPUT: The function value.
EFFECT: This function will do the fork/exec dance, sending the
program the parameters on its stdin, then after waiting, it will
get a double from the program's stdout.  If the program exited with
an error, it will return ERROR and set success to false.  Otherwise
success will be true.
****/
  int childpid, pipe1[2], pipe2[2];
  long i;
  double ret_val;
  FILE *readpipe;
  FILE *writepipe;

  int state;

  if ((pipe(pipe1) < 0) || (pipe(pipe2) < 0) ) {
    perror("pipe");
    exit(-1);
  }/*end if*/

  if ((childpid = fork()) < 0) {
    perror("fork");
    exit(-1);
  } else if (childpid > 0) {   /*Parent*/
    close(pipe1[0]); close(pipe2[1]);
    /* Write to child on pipe1[1], read from child on pipe2[0]. */
    readpipe=fdopen(pipe2[0],"r");
    writepipe=fdopen(pipe1[1],"w");
    if(writepipe==NULL || readpipe==NULL)
      {perror("fdopen");cerr<<"MPEF   : Cannot open pipes... exiting\n";abort();}
    setlinebuf(writepipe);/*important!  This way it won't just hang*/

    /*Output the vector x*/
    for(i=0;i<P;i++) fprintf(writepipe,"%1.8f ", point[i]);
    fprintf(writepipe,"\n");

    /*Read in the returned value, and WAIT*/
```

```
    fscanf(readpipe,"%lf", &ret_val);
    wait(&state);

    /*Close outstanding pipes*/
    fclose(writepipe);fclose(readpipe);

    if(WIFEXITED(state)&&(WEXITSTATUS(state)==0)) {
      /*normal exit - no abort, return code 0*/
      success=true;
      return ret_val;
    }/*end if*/
    else {
      success=false;
      return ERROR;
    }/*end else*/

  } else { /*Child*/
    close(pipe1[1]); close(pipe2[0]);
    /* Read from parent on pipe1[0], write to parent on pipe2[1]. */
    dup2(pipe1[0],0); dup2(pipe2[1],1);
    close(pipe1[0]); close(pipe2[1]);

    if (execlp(Fname, Fname, NULL) < 0) {
      perror("execlp");
      abort();
    }
    return ERROR;
    /* Never returns */
  }/*end else*/

  return ERROR;  /*never returns*/
}/*end EvalF*/
```

E.2.2 SearchCriterion

```
/****
SearchCriterion implementation - 7/13/99 cmsief
****/
#include <iostream.h>
#include "SearchCriterion.h"

/*weighting schemes*/
```

```
//#define SC_WEIGHT_MULTIPLE 0
//#define SC_WEIGHT_TARGET_VALUE 1
//#define SC_WEIGHT_DELTA_MULTIPLE 2
//#define SC_WEIGHT_DYNAMIC_GUESS 3
//#define SC_WEIGHT_DYNAMIC_BEST 4

SearchCriterion::SearchCriterion() {
  w_scheme=ERROR;
  w_params=NULL;
  np=0;
  APX=NULL;
  Eval_c=NULL;f_values=NULL;
  w_c=ERROR;
  P=0;
  original_delta=ERROR;
}/*end default constructor*/

SearchCriterion::SearchCriterion(long dim,Bounds &b) {
  bds=&b;
  SetDefault(dim);
}/*end default constructor*/

SearchCriterion::SearchCriterion(const SearchCriterion &S) {
  w_scheme=S.w_scheme;
  w_params=new mp_vector(*S.w_params);
  np=S.np;
  APX=NULL;
  ResetApproximation(*S.APX);
  w_c=S.w_c;
  P=S.P;
  bds=S.bds;
  original_delta=S.original_delta;
  /*Pointers to these are copied to save memory*/
  Eval_c=S.Eval_c;
  f_values=S.f_values;
}/*end copy constructor*/

SearchCriterion::SearchCriterion(Approximate &A,Bounds &b) {
  P=A.GetDim();
  w_scheme=SC_WEIGHT_MULTIPLE;
  w_params=new mp_vector(2);
  (*w_params)[0]=2;/*default value*/
  (*w_params)[1]=0.8;/*default value*/
  bds=&b;
  np=0;w_c=ERROR;
  Eval_c=NULL;f_values=NULL;
  APX=NULL;
```

```
  original_delta=ERROR;
  ResetApproximation(A);
}/*end special constructor*/

SearchCriterion::~SearchCriterion(){
  if(w_params!=NULL) delete w_params;
  /*Don't delete Eval_c.  It's deletion is controlled by maps*/
  if(APX!=NULL) delete APX;
}/*end destructor*/

SearchCriterion& SearchCriterion::operator=(SearchCriterion &S){
  if(this !=&S) {
    bds=S.bds;
    w_scheme=S.w_scheme;
    (*w_params)=(*S.w_params);
    np=S.np;
    ResetApproximation(*S.APX);
    (*Eval_c)=(*S.Eval_c);
    (*f_values)=(*S.f_values);
    w_c=S.w_c;
    P=S.P;
    original_delta=S.original_delta;
  }/*end if*/
  return (*this);
}/*end overloaded =*/

bool SearchCriterion::ResetSearchCriterionState(Approximate &A,
   int scheme_number, mp_vector &params){
/****
INPUT: An Approximation, a weight scheme number and its parameters.
OUTPUT: returns true iff ResetWeightScheme returns true.
EFFECT: Calls ResetApproximateion, and ResetWeightScheme.
****/
  ResetApproximation(A);
  return ResetWeightScheme(scheme_number, params);
}/*end ResetWeightScheme*/


void SearchCriterion::ResetApproximation(Approximate &A) {
/****
INPUT: An Approximation.
EFFECTS: (*APX) = A
****/
  if(APX!=&A) {
    if(APX!=NULL) delete APX;
    switch(A.GetType()) {
    case A_NULL_APPROX:
```

```
      APX=new NullApprox();
      (*APX)=(NullApprox&)A;
      break;
    case A_KRIG_APPROX:
      APX=new KrigApprox();
      (*APX)=(KrigApprox&)A;
      break;
    default:
#if DEBUG>=1
      cerr<<"SCRA   :Approximation Type Incorrect\n";
#endif
      break;
    };/*esac*/
  }/*end if*/
}/*end ResetApproximation*/

bool SearchCriterion::ResetWeightScheme(int scheme_number, mp_vector &params) {
/****
INPUT: The scheme number and its parameters.
OUTPUT: Returns true if everything is valid.
EFFECTS: This sets the w_scheme integer, and then sets w_params.
This will also set w_c to the correct initial value
****/
  bool success;

  switch(scheme_number){
  case SC_WEIGHT_MULTIPLE:
    if(params.dim() !=2) return false;/*sanity check*/
    w_scheme=SC_WEIGHT_MULTIPLE;
    if(w_params==NULL) w_params=new mp_vector(params);
    else {(*w_params).newsize(2); (*w_params)=params;}
    success=true;
    break;
  case SC_WEIGHT_TARGET_VALUE:
    if(params.dim()!=3) return false;/*sanity check*/
    w_scheme=SC_WEIGHT_TARGET_VALUE;
    if(w_params==NULL) w_params=new mp_vector(4);
    else (*w_params).newsize(4);
    CreateWeightTargetValue(params);
    success=true;
    break;
  case SC_WEIGHT_DELTA_MULTIPLE:
    if(params.dim()!=2) return false;/*sanity check*/
    w_scheme=SC_WEIGHT_DELTA_MULTIPLE;
    if(w_params==NULL) w_params=new mp_vector(2);
    else (*w_params).newsize(2);
    (*w_params)=params;
```

```
    w_c=(*w_params)[0];
    success=true;
    break;
  case SC_WEIGHT_DYNAMIC_GUESS:
    if(params.dim() !=4) return false;/*sanity check*/
    w_scheme=SC_WEIGHT_DYNAMIC_GUESS;
    if(w_params==NULL) w_params=new mp_vector(params);
    else {(*w_params).newsize(4); (*w_params)=params;}
    w_c=(*w_params)[0];
    success=true;
    break;
  case SC_WEIGHT_DYNAMIC_BEST:
    if(params.dim() !=3) return false;/*sanity check*/
    w_scheme=SC_WEIGHT_DYNAMIC_BEST;
    if(w_params==NULL) w_params=new mp_vector(params);
    else {(*w_params).newsize(3); (*w_params)=params;}
    w_c=(*w_params)[0];
    success=true;
    break;
  default:
    success=false;
  };/*end case*/
  return success;
}/*end SetWeightScheme*/


void SearchCriterion::EvaluateSearchCriterion(long P, double *x,
      double &f, int &success) {
/****
INPUT: The dimension of the space, a list of values, a reference passed value
of the function, and a reference-passed flag.  The input syntax is this
convoluted to ensure that it works with Liz Dolan's PatternSearch.
EFFECT: This evaluates the search criterion at a given point, calling
Approximate::Evaluate to do so, and takes the weighting schedule into
account.  If the evaluation was a success, then success is 1, else it is
set to 0.
****/
  for(long i=0;i<P;i++)
    if(x[i]<(*bds).lower[i] || x[i]> (*bds).upper[i]) {
          success=0;
#if DEBUG>=3
  cout<<"SCESC  :POINT OUT OF BOUNDS\n";
#endif
          return;
    }

  double f_true=ERROR,fhat_c, d_c;
```

```
  bool realpoint=false;
  mp_vector temp(P,x);

  /*This check should prevent degradation at previously eval'd sites*/
  for(long j=0;!realpoint&&j<(*Eval_c).num_rows();j++) {
    if(isnear((*Eval_c).row(j),temp,TOLERANCE)) {
      f_true=(*f_values)[j];/*set approx to true value*/
      realpoint=true;
    }/*end if*/
  }/*end for*/

  success=(*APX).EvaluateApprox(temp, (*Eval_c), &fhat_c, &d_c);

  if(realpoint) fhat_c=f_true; /*for degradation check*/

#if DEBUG>=3
  cout<<"SCESC  :hat{f_c}="<<fhat_c<<" w_c="<<w_c<<" d_c="<<d_c<<endl;
#endif
  if(success) f=(fhat_c - w_c * d_c);
  else f=ERROR;
}/*end EvaluateSearchCriterion*/

void SearchCriterion::GetApproximationGraphics(long P, double *x,
       double &f, int &success){
/****
INPUT: Modified PatternSearch Interface.
OUTPUT: Value of the Approximation only at that point.  Meant for use by
GenerateMAPSGraphics.
****/
  for(long i=0;i<P;i++)
    if(x[i]<(*bds).lower[i] || x[i]> (*bds).upper[i]) {
         success=0;
         return;
    }/*end if*/

  double fhat_c, d_c;
  mp_vector temp(P,x);
  success=(*APX).EvaluateApprox(temp, (*Eval_c), &fhat_c, &d_c);

  /*This check should prevent degradation at previously eval'd sites*/
  for(long j=0;j<(*Eval_c).num_rows();j++) {
    if(isnear((*Eval_c).row(j),temp,TOLERANCE)) {
      f=(*f_values)[j];/*set approx to true value*/
      success=1; return;
    }/*end if*/
  }/*end for*/
```

```
  if(success) f=fhat_c;
  else f=ERROR;
}/*end GetApproximationGraphics*/

bool SearchCriterion::UpdateSearchCriterion(pt_collect &points,
    mp_vector& values, long numpoints,
    long n, double delta){
/****
INPUT: The list of evaluated sites, the values, and the number of
points listed, the number of points in the initial design, and the
curent delta.
OUTPUT: Returns true if the operation was sucessful.
EFFECT: This will call (*APX).Update to update the approximation,
then will update the weighting scheme by calling the appropriate
UpdateWeight function.
****/
  if((points.num_rows() != numpoints)||(values.dim()!=numpoints)) {
    cerr<<"SCUSC  :ERROR!  Number of Points doesn't match parameters"<<endl;
  }/*end if*/

  /*Copy pointers to conserve memory*/
  Eval_c=&points;
  f_values=&values;
  np=numpoints;
  if(original_delta==ERROR) original_delta=delta;

#if DEBUG>=3
  cout<<"SCUSC  :Old weight="<<w_c;
#endif
  switch(w_scheme) {
  case SC_WEIGHT_MULTIPLE:
    UpdateWeightMultiple(numpoints-n);
    break;
  case SC_WEIGHT_TARGET_VALUE:
    UpdateWeightTargetValue(numpoints-n);
    break;
  case SC_WEIGHT_DELTA_MULTIPLE:
    UpdateWeightDeltaMultiple(delta);
    break;
  case SC_WEIGHT_DYNAMIC_GUESS:
    UpdateWeightDynamicGuess(numpoints,values);
    break;
  case SC_WEIGHT_DYNAMIC_BEST:
    UpdateWeightDynamicBest(numpoints,values);
    break;
  default:
    break;
```

```
  };/*esac*/
#if DEBUG>=3
  cout<<", New weight="<<w_c<<endl;
#endif

  bool success=(*APX).UpdateApprox(points, values, numpoints,delta, *bds);

  return success;
}/*end UpdateSearchCriterion*/


void SearchCriterion::debug(char *code) const {
/****
EFFECT: does a complete state dump to stdout.
****/
  cout<<code<<":***SearchCriterion::debug()***\n";
  cout<<code<<":P="<<P<<" np="<<np<<endl;
  cout<<code<<":w_scheme="<<w_scheme<<" w_c="<<w_c<<" w_params="<<(*w_params);
  //  cout<<code<<"Eval_c="<<Eval_c;
  (*APX).debug(code);
  cout<<code<<":***end SC:debug()***\n";
}/*end debug*/


void SearchCriterion::SetDefault(long dim){
/****
INPUT: Dimension of space
EFFECT: Automagically generates one of these objects, uses (2,0.8)
WeightMultiple scheme and Kriging Approximation.
****/
  P=dim;
  w_scheme=SC_WEIGHT_MULTIPLE;
  w_params=new mp_vector(2);
  (*w_params)[0]=2;/*default value*/
  (*w_params)[1]=0.8;/*default value*/
  np=0;w_c=0;
  Eval_c=NULL;
  APX=new KrigApprox(dim);/*defaults to Kriging*/
}/*end SetDefault*/


  /*Weighting Scheme functions*/
void SearchCriterion::CreateWeightTargetValue(mp_vector &params){
/*This does the computation of w_params[3]
  m = (p[1] / p[0])^(1/(p[2] * (V-N)))
*/
  for(long i=0;i<3;i++) (*w_params)[i]=params[i];
```

```
  (*w_params)[3]= pow( params[1] / params[0], 1.0 / params[2]);
}/*end CreateWeightTargetValue*/

void SearchCriterion::UpdateWeightTargetValue(long c){
/****
INPUT: The iteration number.
EFFECT: Multiply w_c by the correct multiplier in order to
reach the level of insignificance by the specified iteration.
PARAMS FORMAT: [0] - w_0
               [1] - target value of insignificance
               [2] - iteration number at which to reach insignificance
Though w_params[3] is not specified by the user, when you install the
Target Value weighting scheme, the multiple is computed and stored here.
****/
  w_c=(*w_params)[0];
  for(long i=0;i<c;i++) w_c *= (*w_params)[3];
}/*end UpdateWeightTargetValue*/


void SearchCriterion::UpdateWeightMultiple(long c) {
/****
EFFECTS: This will multiply w_c by w_params[1]
PARAMS FORMAT: [0] - w_0
               [1] - c, s.t. w_c = c * w_{c-1}
****/
  w_c=(*w_params)[0];
  for(long i=0;i<c;i++) w_c *= (*w_params)[1];
}/*end UpdateWeightMultiple*/


void SearchCriterion::UpdateWeightDeltaMultiple(double curr_delta) {
/****
INPUT: current delta
EFFECT: If delta has been refined, the weight is multiplied by w_1
PARAMS FORMAT: [0] - w_0, the weight before delta refinement
               [1] - m, s.t. w_c = m*w_{c-1}, if delta was refined last iteration
                                 = w_{c-1}, if delta wasn't refined last iteration
****/
  if(original_delta-curr_delta>=TOLERANCE) {
    original_delta=curr_delta;
    w_c *= (*w_params)[1];
  }/*end if*/
}/*end UpdateDeltaBased*/


void SearchCriterion::UpdateWeightDynamicGuess(long lastsite, mp_vector& values){
/****
```

```
INPUT: number of last site eval'd and vector of function values
EFFECT: If the approximation was 'close' to correct at the last site, then we
reduce the weight, less we increase it.
PARAMS FORMAT: [0] - w_0
               [1] - t, the threshhold percentage
               [2] - m_b, multiplier if approximation is 'bad'
               [3] - m_g, multiplier if approximation is 'good'

               w_c=m_g * w_{c-1}, if |f(x_{c-1}) -\hat{f_{c-1}(x_c-1)}| <= t*frange
                  =m_b * w_{c-1}, otherwise
****/
  double v_min,v_max;
  double f_hat, d_c;
  bool s;
#if DEBUG>=2
  cout<<"LASTSITE="<<lastsite;
#endif
  mp_vector x(1);
  x=(*Eval_c).row(lastsite-1);

  /*Compute the Function range*/
  v_min=v_max=values[0];
  for(long i=1;i<lastsite;i++) {
    if(values[i]<v_min) v_min=values[i];
    if(values[i]>v_max) v_max=values[i];
  }/*end for*/

  /*Eval Old Approximation*/
  s=(*APX).EvaluateApprox(x, *Eval_c, &f_hat,&d_c);

  if(fabs(values[lastsite-1] - f_hat) <= (*w_params)[1] *(v_max - v_min)){
    w_c *= (*w_params)[3];/*the approx is good*/
#if DEBUG>=2
    cout<<"... approx is good ";
#endif
  }
  else {
    w_c *= (*w_params)[2];/*approx is not so good*/
#if DEBUG>=2
    cout<<"... approx is not good ";
#endif
  }
#if DEBUG>=2
  cout<<"w_c="<<w_c<<endl;
#endif

}/*end UpdateWeightDynamicGuess*/
```

```
void SearchCriterion::UpdateWeightDynamicBest(long lastsite, mp_vector& values){
/****
INPUT: number of last site eval'd and vector of function values
EFFECT: If the approximation was 'close' to correct at the last site, then we
reduce the weight, less we increase it.  Similar to DynamicGuess, except it uses
|expval - bestpoint| / 2 as the threshold.
PARAMS FORMAT: [0] - w_0
               [1] - m_b, multiplier if approximation is 'bad'
               [2] - m_g, multiplier if approximation is 'good'

               w_c=m_g * w_{c-1}, if |f(x_{c-1}) -\hat{f_{c-1}(x_c-1)}| <= |expval-best|/2
                  =m_b * w_{c-1}, otherwise
****/
  double v_min;
  double f_hat, d_c;
  bool s;
#if DEBUG>=2
  cout<<"LASTSITE="<<lastsite;
#endif
  mp_vector x(1);
  x=(*Eval_c).row(lastsite-1);

  /*Compute the Function range*/
  v_min=values[0];
  for(long i=1;i<lastsite-1;i++) {
    /*Avoid the last site for this calulation - get old minimizer*/
    if(values[i]<v_min) v_min=values[i];
  }/*end for*/

  /*Eval Old Approximation*/
  s=(*APX).EvaluateApprox(x, *Eval_c, &f_hat,&d_c);

  if(fabs(values[lastsite-1] - f_hat) <= fabs((v_min - f_hat)/2.0)){
    w_c *= (*w_params)[2];/*the approx is good*/
#if DEBUG>=2
    cout<<"... approx is good ";
#endif
  }
  else {
    w_c *= (*w_params)[1];/*approx is not so good*/
#if DEBUG>=2
    cout<<"... approx is not good ";
#endif
  }
#if DEBUG>=2
```

```
  cout<<"w_c="<<w_c<<endl;
#endif
}/*end WeightDynamicBest*/
```

E.2.3 InitialDesign

```
/****
InitialDesign Class Implementation- 5/25/99 cmsief
This is the parent class for Initial Designs.  Just inherit this, and you're
almost all set.
****/

#include "maps_general.h"
#include "PSGrid.h"
#include "InitialDesign.h"

InitialDesign::InitialDesign() {
  init_done=false;
  P=0;
  N=0;
  points=NULL;
}/*end default constructor*/

InitialDesign::InitialDesign(long dim){
  init_done=false;
  P=dim;
  N=0;
  points=NULL;
}/*end special constructor*/


InitialDesign::InitialDesign(const InitialDesign &I) {
  init_done=I.init_done;
  P=I.P;
  N=I.N;
  if(I.points!=NULL) points=new pt_collect(*I.points);
  else points=NULL;
}/*end copy constructor*/


InitialDesign::~InitialDesign() {
  if(points!=NULL) delete points;
}/*end destructor*/

InitialDesign& InitialDesign::operator=(const InitialDesign &I) {
```

```
  if(this!=&I) {
    init_done=I.init_done;
    P=I.P;
    N=I.N;
    if(points!=NULL) delete points;
    if(I.points!=NULL) points=new pt_collect(*I.points);
    else points=NULL;
  }/*end if*/
  return (*this);
}/*end overloaded =*/

void InitialDesign::SetDimension(long dim) {
  P=dim;
}/*end SetDimension*/


bool InitialDesign::GenerateDesign(const Bounds &b, long num_points,
                                         const PSGrid &grid, long stream){
/****
INPUT: The problem's bounds, the number of points to put in the design,
the grid and a random stream number.  This assumes, of course that you've set
up rngs right.
OUTPUT: Returns true.
EFFECTS: Sets init_done=true, and calls MoveToGrid(b,grid)
Derived classes must call this function.
****/
#ifndef DACE_ONLY_MODE
    MoveToGrid(b,grid);
#endif
  init_done=true;
  return true;
}/*end GenerateDesign*/


void InitialDesign::GetDesign(pt_collect* outptr){
/****
INPUT: Pointer of type pt_collect*.
EFFECTS: Copies points to outptr if init_done==true.
****/
  (*outptr)=(*points);
}/*end GetDesign*/


void InitialDesign::debug(char *code) {
  cout<<code<<" InitialDesign::debug()\n";
  cout<<code<<" P="<<P<<" N="<<N<<endl;
  cout<<code<<(*points);
```

```
}/*end debug*/

void InitialDesign::MoveToGrid(const Bounds &b, const PSGrid &grid) {
/****
INPUT: The problem's bounds and grid.
EFFECTS: Moves all the points in 'points' to grid position.  Calls
grid.SnapToGrid, and uses that to do the snapping.  Then it just fixes the
point list.
****/
  mp_vector temp(P);

  for(long i=0; i<N;i++){
    grid.SnapToGrid(b,(*points).row(i),&temp);
    for(long j=0;j<P;j++)
      ((*points)[i])[j]=temp[j];
  }/*end for*/
}/*end MovetoGrid*/
```

E.2.4 LatinHCDesign

```
/****
LatinHCDesign Class Implementation - 6/7/99 cmsief
Derived from InitialDesign
****/
#include "LatinHCDesign.h"


LatinHCDesign::LatinHCDesign():InitialDesign(){;}
LatinHCDesign::LatinHCDesign(long dim):InitialDesign(dim){;}
LatinHCDesign::LatinHCDesign(const LatinHCDesign &I):InitialDesign(I){;}
LatinHCDesign::~LatinHCDesign(){;}

bool LatinHCDesign::GenerateDesign(const Bounds &b, long num_points,
   const PSGrid &grid, long stream) {
/****
INPUT: The problem's bounds, the number of points to put in the
design, the grid, and the random stream.
EFFECTS: Generates the initial design via latin hypercubes.
Then calls InitialDesign's GenerateDesign.
****/
  N=num_points;
```

```
  ml_vector a(num_points);/*Partition of a given dimension*/
  long s,temp;
  double int_width;/*width of interval*/
  long i,j; /*counters*/

  if(points!=NULL) delete points;
  points=new pt_collect(N,P);

  SelectStream(stream);

  for(i=0;i<P;i++) { /*loop on dimension*/
  int_width = (b.upper[i]-b.lower[i])/N;
    for(j=0;j<N;j++) a[j]=j;/*clear array*/

    /*Shuffle the array, as per Park and Leemis*/
    for(j=0;j<N;j++) {
      s=Equilikely(j,N-1);
      temp=a[s];
      a[s]=a[j];
      a[j]=temp;
    }/*end for*/
    /*END SHUFFLE*/

    for(j=0;j<N;j++)
      ((*points)[j])[i]=b.lower[i]+Uniform(a[j]*int_width,(a[j]+1)*int_width);
  }/*end for dimension*/
  InitialDesign::GenerateDesign(b,num_points,grid,stream);
  return true;
}/*end GenerateDesign*/
```

E.2.5 PSGrid

```
/****
PSGrid implementation - 6/17/99 cmsief
This is downright exciting.
****/

#include <fstream.h>
#include <iostream.h>
#include "PSGrid.h"
/**********************CONSTRUCTORS + DESTRUCTOR***********************/
PSGrid::PSGrid(){
  P=0;
  origin=NULL;
  gridvect=NULL;
```

```
  refine=0.0;
}/*end default construtor*/

PSGrid::PSGrid(const PSGrid &g){
  origin=NULL;
  gridvect=NULL;
  ResetPSGridState(g.P,*g.origin,*g.gridvect, g.refine, g.delta);
}/*end copy constructor*/

PSGrid::PSGrid(long P1, const mp_vector& o, const pt_collect& g,
               double ref, double d){
  origin=NULL;
  gridvect=NULL;
  ResetPSGridState(P1,o,g,ref,d);
}/*end special constructor*/

PSGrid::PSGrid(long P1, const Bounds &b){
/*this assumes a certain relative grid thickness,
  and it just goes with it*/
  double minres;

  mp_vector resolution(P1);
  resolution=PSG_DEFAULT_RESOLUTION * (b.upper-b.lower);

  /*Makes sure resolution is constant across dimensions*/
  minres=resolution[1];
  for(long i=0;i<P1;i++)
    if(resolution[i]<minres) minres=resolution[i];

  resolution=minres;

  origin=NULL;
  gridvect=NULL;
  SetDefault(P1,b,resolution);
}/*end special constructor*/

PSGrid::PSGrid(long P1, const Bounds &b, const mp_vector &resolution) {
  origin=NULL;
  gridvect=NULL;
  SetDefault(P1, b, resolution);
}/*end special constructor*/

PSGrid::~PSGrid(){
  if(origin !=NULL) delete origin;
  if(gridvect!=NULL) delete gridvect;
}/*end destructor*/
```

```
PSGrid& PSGrid::operator=(PSGrid &g) {
  if(this!=&g) {
    P=g.P;refine=g.refine;delta=g.delta;
    if(origin !=NULL) delete origin;
    if(g.origin==NULL) origin=NULL;
    else origin=new mp_vector(*g.origin);

    if(gridvect !=NULL) delete gridvect;
    if(g.gridvect==NULL) gridvect=NULL;
    else gridvect=new pt_collect(*g.gridvect);
  }
  return (*this);
}/*end overloaded =*/

/***************************RESET FUNCTIONS****************************/

void PSGrid::ResetPSGridState(long P1, const mp_vector& o, const
      pt_collect& g, double ref, double d){
/****
INPUT: A complete set of state values for the grid
EFFECTS: It copies these into the object's state
****/
  P=P1;
  refine=ref;
  delta=d;
  /*Memory Allocation+Deallocation*/
  if(origin!=NULL) delete origin;
  if(gridvect!=NULL)  delete gridvect;

  origin=new mp_vector(P);
  gridvect=new pt_collect(P,P);
  /*this makes sure there is enough memory*/

  (*origin)=o;
  (*gridvect)=g;
}/*end ResetState*/

bool PSGrid::ReadGridFromFile(char* fn){
/****
INPUT: File name string
EFFECTS: Reads the grid data from a file
FILE FORMAT: dimension<cr>
             delta<cr>
             refine<cr>
             origin(P)<cr>
             grid(PxP)
****/
```

```
  ifstream ifs;
  long dim,i,j;
  double ref,d;
  ifs.open(fn,ios::in);

  ifs>>dim;
  if(ifs.eof()) return false;
  ifs>>d;
  if(ifs.eof()) return false;
  ifs>>ref;
  if(ifs.eof()) return false;

  /*Memory allocation*/
  mp_vector* or=new mp_vector(dim);
  pt_collect* grid=new pt_collect(dim,dim);

  /*origin*/
  for(i=0;i<dim&&!ifs.eof();i++) ifs>>(*or)[i];
  if(ifs.eof())
    {ifs.close(); delete or; delete grid; return false;}

  /*gridvect*/
  for(i=0;i<dim&&!ifs.eof();i++)
    for(j=0;j<dim&&!ifs.eof();j++)
      ifs>>((*grid)[i])[j];

  /*final checks*/
  if(ifs.eof()) {
    ifs.close();
    delete or; delete grid;
    return false;
  }/*end if*/
  else {
    ifs.close();
    P=dim;
    refine=ref;
    delta=d;
    delete origin;
    delete gridvect;
    origin=or;
    gridvect=grid;
    return true;
  }/*end else*/
}/*end ReadGridFromFile*/

/****************************QUERY FUNCTIONS*****************************/
```

```
void PSGrid::debug(char *code) const{
/****
EFFECTS: does a complete state dump to stdout.
****/
  cout<<code<<":***PSGrid::debug()***\n";
  cout<<code<<":P="<<P<<", refine="<<refine<<", delta="
      <<delta<<"\n"<<code<<"ORIGIN:\n"<<(*origin);
  cout<<code<<":GRID:\n"<<(*gridvect);
  cout<<code<<":******\n";
}/*end debug*/

double PSGrid::GetDelta() const {
/****
OUTPUT: returns delta
****/
  return delta;
}/*end GetDelta*/

bool PSGrid::IsFeasible(const Bounds &b, const mp_vector &x) const {
/****
INPUT: The bounds and an mp_vector.
OUTPUT: True if the point is within the founds, false if otherwise
****/
  bool ret=true;
  for(long i=0;i<P&&ret;i++) {
    if ((b.lower[i]>x[i])||(b.upper[i]<x[i])) ret=false;
  }/*end for*/
  return ret;
}/*end IsFeasible*/

long PSGrid::FindBestPSMultiple(const Bounds &b) {
/****
INPUT: The bounds of the space.
OUTPUT: Finds the best multiple for delta for use in the PatternSearch.
It does so with the following methodology:
1) Find i s.t. i maximizes delta*grid[i] / b.upper[i]-b.lower[i].
2) Returns half the biggest multiple s.t.
   mult * delta*grid[i] < b.upper[i]-b.lower[i].
****/
  long i=0;
  double ratio=0;
  long mult=1;
  mp_vector d = b.upper - b.lower;

  /*Find the optimal i*/
  for(long j=0; j<P;j++)
    if((*gridvect)[j][j] / d[j] > ratio)
```

```
      {i=j; ratio=(*gridvect)[j][j]/d[j];}

  ratio*=delta;

  /*Find the maximal multiple*/
  while(ratio*mult < d[i]) mult*=2;

  mult/=4; /*find the largest less than d[i]*/

  return mult;
}/*end FindBestPSMultiple*/


/**************************ACTION FUNCTIONS****************************/

void PSGrid::SnapToGrid(const Bounds &b, const
mp_vector &old_pt, mp_vector* new_pt) const {
/****
INPUT: The bounds and two mp_vectors - one reference and one pointer.
OUTPUT: The nearest grid point to old_pt will be returned in new_pt
NOTES: Its amazing what a little bit of linear algebra can do!  Note
that this will not necessarily work for grids that aren't parallel
to the bounds.
****/

  ml_vector ipiv(P);
  pt_collect T(P,P);
  long i;
  mp_vector x=old_pt-(*origin);
#if DEBUG>=1
  if(gridvect==NULL) cerr<<"PSGSTG :Grid not initialized\n";
#endif

  T=(*gridvect)*delta;

  //This is a more general method, that will work for
  //grids not parallel to the axes.
  //LU_Factor(T,ipiv);
  //LU_Solve(T,ipiv,x);
  //  for(long i=0;i<P;i++) x[i]=round(x[i]);
  for(i=0;i<P;i++) x[i]=round(x[i]/T[i][i]);

  (*new_pt)= (*origin)+T*x;
  /*BOUNDS CHECKS*/
  for(i=0;i<P;i++) {
    if((*new_pt)[i]<b.lower[i]) (*new_pt)[i] = b.lower[i];
    if((*new_pt)[i]>b.upper[i]) (*new_pt)[i] = b.upper[i];
```

```
  }/*end for*/
}/*end SnapToGrid*/


void PSGrid::RefineGrid(){
/****
EFFECT: Actually refines the grid. delta *= refine;
****/
  delta*=refine;
}/*end RefineGrid*/


void PSGrid::GetCore(const Bounds &b, const mp_vector &x,
     pt_collect&core, long &pts) {
/****
INPUT: The bounds, a point, a pointer to the point collection
to be returned, and a reference passed number of points.
EFFECT: Returns the list of points in the Core Pattern of x,
and the number of points in this pattern. This should also call
IsFeasible, and return only those points within the bounds. Note
that the returned values are stored on the rows (aka each row
represents a point in P-space), much as Eval_c is stored in maps.h.
****/
  pt_collect temp(P,P);
  mp_vector test(P);
  pts=0;
  pt_collect t2(2*P,P);/*allocate more memory than we need*/
  long i,j; /*counters*/

  temp=(*gridvect)*delta;

  for(i=0;i<P;i++) {
    test=x+temp.col(i);/*create the potential core pattern vector*/
    if(IsFeasible(b,test)) {
      /*if feasible, copy it into a ROW of the outgoing matrix*/
      for(j=0;j<P;j++) t2[pts][j]=test[j];
      pts++;
    }/*end if*/

    test=x-temp.col(i);/*create the potential core pattern vector*/
    if(IsFeasible(b,test)) {
      /*if feasible, copy it into a ROW of the outgoing matrix*/
      for(j=0;j<P;j++) t2[pts][j]=test[j];
      pts++;
    }/*end if*/
  }/*end for*/
```

```
  /*Now play the memory dance, and reclaim what was once ours!*/
  core.newsize(pts,P);
  for(i=0;i<pts;i++)
    for(j=0;j<P;j++)
      core[i][j]=t2[i][j];
}/*end GetCore*/

void PSGrid::SetDefault(long P1, const Bounds &b,
const mp_vector &resolution){
/****
INPUT: Dimension, bounds and a grid resolution.
EFFECTS: This will automagically generate a simple grid
that is parallel to the axes.
****/
  long i;

#if DEBUG>=3
  cerr<<"PSGSD  :Called. resolution="<<resolution;
#endif
  P=P1;
  gridvect=new pt_collect(P,P);
  origin=new mp_vector(b.lower);
  refine=PSG_DEFAULT_REFINE;
  delta=resolution[0];

  /*Finds smallest suggested resolution to make it delta*/
  for(i=0;i<P;i++)
    if(resolution[i]<delta)
      delta=resolution[i];

  for(i=0;i<P;i++)
    for(long j=0;j<P;j++)
      if(i!=j) (*gridvect)[i][j]=0;
      else (*gridvect)[i][j]=round(resolution[i]/delta);

#if DEBUG>=3
  cerr<<"PSGSD  :Done\n";
#endif
}/*end SetDefault*/
```