MULTIFIDELITY GLOBAL DESIGN OPTIMIZATION INCLUDING PARALLELIZATION POTENTIAL

By
STEVEN E. COX

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2002

For my wife, Amanda and my parents who have always supported me.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr Raphael Haftka who provide me with the opportunity and funding to pursue a PhD. He has been very helpful in exploring new avenues of research and in securing opportunities and contacts to broaden my exposure to the state of the art in the optimization field. He has been a patient instructor and a wealth of knowledge and experience to draw from.

My thanks go especially to my parents who provided me with the education and encouragement which made this dissertation possible. They sacrificed to send me to good schools and took an active interest in my education. They are responsible for my strong work ethic and instilled a deep desire to never stop learning new things. Together with the rest of my family, they provided the support which has allowed me to get where I am now.

I would like to thank my wife Amanda who encouraged me to complete my research and never complained about the long hours and travel required. She has been supportive and enthusiastic about my work which has made my life much easier and all of the effort worthwile.

I would also like to thank the past and present members of the Structural and Multidisciplinary Optimization Group for their friendship and support over the last five years. SatchiVenkataramam, Melih Papila and Raluca Rosca were wonderful sources of information on the various requirements for graduate students including publications, classes, examinations and this dissertation. Laurent Grosset, Jaco Schutte, Amit Kale and Xueyong Qu were very helpful for discussing ideas and provided additional viewpoints on research as well as different cultures.

I would like to thank my committee members for their guidance and support on my research. Their comments and advice made this dissertation a much more useful document. I would also like to thank Dr B. J. Fregly for standing in at my defense when Dr George was unable to attend.

I am grateful for the financial support provided in NASA grant NAG-2-1179 which covered most of my graduate career and from William Hart through Sandia National Laboratories which provided the funding to allow me to finish my graduate work. I am also grateful for the use of the Ross parallel cluster of processors at Sandia where some of the parallel experiments were run. William Hart was also very helpful in the formulation of the DIRECT-BP algorithm and the conversion of my code into more robust C++ libraries.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS iii

LIST OF TABLES vii

LIST OF FIGURES ix

ABSTRACT xi

1 INTRODUCTION AND LITERATURE SURVEY 1

Global Optimization 5
Genetic Algorithms 6
Simulated Annealing 8
Particle Swarm 10
Multistart Methods 11
Pattern Search Methods 14
Lipschitzian Optimization 16
Approximation Methods for Optimization and Reduced Cost 18
Multifidelity Optimization 21
Multidisciplinary Optimization 24
Parallel Computing 26
Overview of Research 28

2 PERFORMANCE OF GLOBAL OPTIMIZATION ALGORITHMS NOISE VERSUS WIDELY SEPARATED OPTIMA 30

Optimizers 31
Sequential Quadratic Programming 31
Dynamic Search 31
The DIRECT Algorithm 33
Constraints 38
Test Functions 40
Griewank Function 40
Quartic Function 44
HSCT Design Problem 47
Comparison Concluding Remarks 53

3 DIRECT-BP 55

Original DIRECT Algorithm 56
Convergence Behavior 56
Demonstration of Local Convergence Difficulty 58
The DIRECT-BP Algorithm 61
Implementation 66
Identifying a Positive Spanning Set 67
Maintaining a Positive Spanning Set 69
Limiting Λ 70
Local Search 71
Experimental Comparisons 72
Test problems 72
Experimental methods 74
Test results 75

4 MULTIFIDELITY DIRECT 80

Correction Methods 80
Constant Correction Ratio 81
Linear Correction Response Surface 83
Test Problems 88
Test Results 90
Sixth Order Test Problem 90
Griewank Function 94
Algorithm Status 98

5 PARALLEL DIRECT 100

Previous Parallel DIRECT 100
DIRECT Modifications 102
Accelerated Start 102
Continuous DIRECT 104
Message Traffic 108
Performance 115
Alternate Convex Hull Calculation 118
Algorithm Status 121

6 CONCLUSIONS 123

A Sequential DIRECT C++ Library 126

Example Problem Code 128

B DIRECT and DIRECT-BP 129

Sequential DIRECT and DIRECT-BP code 129

C Parallel DIRECT Code 168

Parallel DIRECT Code 168

D Multifidelity DIRECT Code 204

CCF Multifidelity DIRECT code 204
LRS Multifidelity DIRECT Code 229

LIST OF REFERENCES 256

BIOGRAPHICAL SKETCH 262

LIST OF TABLES

1. Comparisons for 5, 10, and 20 DV Griewank functions.....42

2. Effect of Variation of *d* in 10-dimensional Griewank function.....43

3. Comparisons for 5, 10, and 20 DV quartic functions. 45

4. Comparison of 5, 10, 15, and 20 DV HSCT problem.....50

5. Fifteen DV optima for DOT and DIRECT. 51

6. Comparison of DIRECT and DIRECT-BP on f_1 of Eq. 15, n = 10. 76

7. Comparison of the average performance (30 runs) of DIRECT and DIRECT-BP on the ten-dimensional Griewank function.....77

8. Comparison of the performance of DIRECT and DIRECT-BP on the first problem used from Floudas et al.(1999), Eq. 19.....78

9. Comparison of the performance of DIRECT and DIRECT-BP on the design of a CSTR, Eq. 21. 78

10. Multifidelity versions of DIRECT vs. single fidelity DIRECT for the sixth order test problem, Eq. 25.....91

11. Multifidelity DIRECT comparison for modified sixth order problem, Eq. 28, 29.....92

12. Multifidelity constant correction factor DIRECT with modified potentially optimal box selection 93

13. Multifidelity DIRECT Comparison for Griewank Function 94

14. Multifidelity DIRECT comparison for modified Griewank function.....96

15. Performance of the parallel DIRECT algorithms on a 20DV Griewank function.....116

14. Parameters in Object Oriented DIRECT library code. 127

LIST OF FIGURES

1. False local optima due to noise 2
2. Function with true local optima 2
3. Illustration of single point crossover in Genetic algorithm 7
4. Example of a single point mutation. 7
5. Tunneling to locate a different objective function valley. 13
6. Nine steps of a pattern search with 4 possible step directions. 15
7. Lipschitzian Optimization 35
8. One-dimensional example of DIRECT 35
9. Point selection in DIRECT. 37
10. DIRECT box division in 2D. 37
11. Griewank function in one dimension. 40
12. Quartic function in one dimension(e= 0.2). 44
13. Definitions of design variables. 48
14. Distribution of Design Variables for the best 25 designs found by DOT for the 10 DV HSCT Case 53
15. Instance where the current best box does not contain the local optimum 58
16. Possible distribution of points for potentially optimal box selection. 58
17. A one dimensional plot of $f_1(x)$ (Eq. 15) in the range x = [0,3] 59
18. Distance between c*(t) and the global optimum for equation 15 in ten dimensions… 60

19. Value of f(c*(t)) for f_1 (Eq. 15) in ten dimensions as the optimizations progress. 60

20. An illustration of the sets of vectors Tt used for X*(t) at various positions in the feasible domain. 63

21. Two-dimensional example of vectors failing to form a positive spanning set. 65

22. One dimensional Griewank function with d = 100. 73

23. Constant correction scheme for multifidelity DIRECT 81

24. High-fidelity Function in One Dimension 88

25. Low-fidelity Function in One Dimension 89

26. Modified Griewank function and low fidelity approximation 96

27. Initial Box Division for Parallel DIRECT 103

28. Alternate Initial Box Division for Parallel DIRECT 104

29. Continuous parallel DIRECT organization 105

30. Test Problem in One Dimension for Parallel DIRECT (e = 0.2) 108

31. Initial message pattern for 4 processors using Prob. B. 110

32. Final message pattern for 4 processors using Prob. B. 111

33. Final message pattern for 4 processors using Prob. A. 111

34. Initial message pattern for 16 processors using Prob. B. 113

35. Final message pattern for 16 processors using Prob. B. 113

36. Initial message pattern for 25 processors using Prob. B. 114

37. Final message pattern for 25 processors using Prob. B. 114

38. Comparison of computational efficiency of the three parallel DIRECT algorithms. 116

39. Outline of existing parallel DIRECT code arrangement. 119

40. Modification to the DIRECT code to reduce idle time. 120

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

MULTIFIDELITY GLOBAL DESIGN OPTIMIZATION
INCLUDING PARALLELIZATION POTENTIAL
By

Steven E. Cox

December 2002

Chair: Raphael Haftka
Department: Mechanical and Aerospace Engineering

The DIRECT global optimization algorithm is a relatively new space partitioning algorithm designed to determine the globally optimal design within a designated design space. This dissertation examines the applicability of the DIRECT algorithm to two classes of design problems: unimodal functions where small amplitude, high frequency fluctuations in the objective function make optimization difficult; and multimodal functions where multiple local optima are formed by the underlying physics of the problem (as opposed to minor fluctuations in the analysis code). DIRECT is compared with two other multistart local optimization techniques on two polynomial test problems and one engineering conceptual design problem.

Three modifications to the DIRECT algorithm are proposed to increase the effectiveness of the algorithm. The DIRECT-BP algorithm is presented which alters the way DIRECT searches the neighborhood of the current best point as optimization progresses. The algorithm reprioritizes which points to analyze at each

iteration. This is to encourage analysis of points that surround the best point but that are farther away than the points selected by the DIRECT algorithm. This increases the robustness of the DIRECT search and provides more information on the characteristics of the neighborhood of the point selected as the global optimum.

A multifidelity version of the DIRECT algorithm is proposed to reduce the cost of optimization using DIRECT. By augmenting expensive high-fidelity analysis with cheap low-fidelity analysis, the optimization can be performed with fewer high-fidelity analyses. Two correction schemes are examined using high- and low-fidelity results at one point to correct the low-fidelity result at a nearby point. This corrected value is then used in place of a high-fidelity analysis by the DIRECT algorithm. In this way the number of high-fidelity analyses required is reduced and the optimization became less expensive.

Finally the DIRECT algorithm is parallelized to allow its implementation on multiple processors. Two master-slave implementations are proposed using an arbitrary number of processors to speed the analysis of points for the optimization. The two methods are compared on a heterogeneous collection of processors with special attention to computational and algorithmic efficiency.

CHAPTER 1
INTRODUCTION AND LITERATURE SURVEY

Most of the weight and performance characteristics of a vehicle are set in the conceptual level of a design. Designers would like to use the most accurate models available and a large number of design variables at the conceptual level to produce a globally optimal design for the finished vehicle. Consequently, as computational power has increased and more accurate physical models become available, the design of complex machines has become more involved at the earliest stages of development. As the number of variables increases, the volume of the design space increases exponentially and gradient calculations become more expensive. This increases the cost of optimizing the design over even a small range for each variable.

Often the feasible design space or the objective functions are nonconvex and the resulting multiple local optima can trap local optimizers and prevent them from locating the best design. To solve this problem, either multiple starting points or a global optimization algorithm may be used. Both of these methods will increase the cost of the optimization.

Discretization errors, round-off errors, and less than fully converged iterative calculations within analysis codes can result in noisy constraints and objective functions. This can create additional spurious optima. A distinction is made between these noise-generated numerical optima and genuine optima due to physical nonconvexity. Designers are primarily concerned with locating the physical local optima modeled by the

analysis functions and with finding a way to bypass the numerical noise. Our research has used approximations, dynamic search techniques, and space partitioning methods to try to overcome numerical noise.

As the design space becomes larger, complex constraint boundaries can lead to many local optima and disjoint feasible regions for even simple objective functions. This requires an optimization method which can jump from one feasible region to another and that can search for other local optima away from explored regions. Numerical noise can also hinder local optimizers by corrupting gradient calculations and creating false local optima. Different optimizers handle one of these challenges and not the other. One objective of this dissertation is to select an approach that is able to deal with both numerical noise (Figure 1) and widely separated local optima (Figure 2).

In the past, designers relied on simple algebraic estimates for such things as weight and load calculations for structural components, lift and drag for wings and fuselages, or magnetic and electrical effects for electronic components (Raymer 1992). This worked well for designing conventional aircraft or spacecraft, but did not necessarily produce the best designs for the mission for which they were needed. Experts

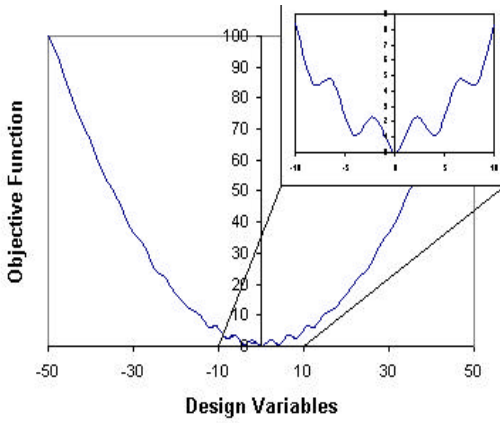


Figure 1. False local optima due to noise

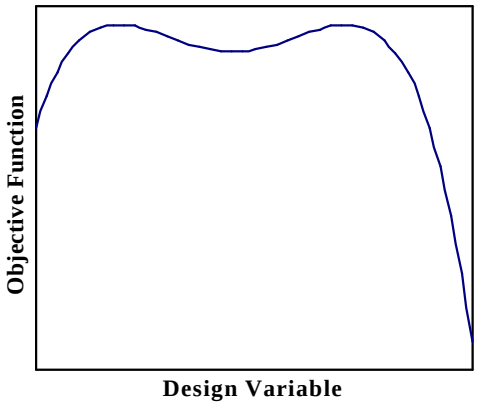


Figure 2. Function with true local optima

typically narrow the potential designs to a manageable range based on existing designs. This reduces the possibility of developing new designs that could be useful for new mission requirements. Local optimization methods are applied to subsets of the design variables as each part of the craft is optimized in turn. Multiple iterations through the design variables leads to a locally optimal design according to the analysis equations used. Many of the algebraic estimates are based on empirical data from a limited number of designs and cannot predict the behavior of a design with a substantially different configuration. These methods can lead to designs whose physical behavior is not truly optimal.

Lamberti et al. (2001) demonstrated the use of approximate analysis equations in the design of a reusable launch vehicle. By using simple shell and plate theories to optimize the design of an integrated fuel tank, they were able to examine 14 different design concepts with about 10 variables each in less time than it took to perform a single nonlinear finite element analysis. To use these simple models, however, they were forced to constrain the design to regions were the models were accurate. This precluded consideration of potentially optimum designs.

As computer power increased, new analysis codes were developed that can accurately predict the behavior of unusual and complicated structures. However, these codes can take anywhere from hours to days for a single high-fidelity analysis. For this reason they have been used extensively in the final stages of the design cycle but they are generally considered too expensive for the conceptual stage of the design (where as many as tens of thousands of analyses must be performed). To search more of the design space in the conceptual stage, designers are forced to use simpler versions of these codes. A

prominent area of research today is designed to find ways to combine the simple and complex analysis codes so that the designer gains the accuracy of the high-fidelity codes while retaining the lower computational cost of the low-fidelity codes

This dissertation examines ways to combine the high- and low-fidelity models with the DIRECT algorithm for efficient global optimization. Multifidelity local optimization has been used for over a decade on various engineering problems with good results. This dissertation attempts to convert these methods to operate efficiently for global optimization. It examines two correction methods to improve the results from approximate models: constant correction factors (over a limited region of the design space) and linear correction response surface models (to improve the correction away from the high-fidelity solutions).

As analysis programs become more expensive and optimizers require more evaluations to search ever larger design domains, the computing power of a single processor has become a major limiting factor to design optimization. Parallel systems have become available with thousands of processors, which can perform trillions of operations per second. With the profusion of powerful PCs in the average company and the improvement of building-wide networks, simple parallel arrangements are now possible for very little additional investment (Groenwold et al. 2000). New optimization methods that distribute the work evenly over multiple processors or computers would allow engineers to incorporate more expensive analysis programs and examine more design concepts with more variables.

With this in mind, the DIRECT algorithm used here was examined for its potential for use on parallel computers. An optimization algorithm that generates

function evaluations in large batches is inherently suited for running on a parallel machine. Baker et al. (2001) showed this with the DIRECT optimizer, but room still exists for improvement. This research examines ways of reducing the idle time of the parallel code by continuously generating work for the processors and working with the strengths and weaknesses of different parallel systems.

Global Optimization

One reason so many types of global optimizers exist is that no one optimizer performs well on every type of problem. Much research has been done to find computationally efficient methods to perform global optimization for high dimensional, nonconvex design spaces. Many global optimization codes have been developed and tested for use with different classes of problems (Floudas and Pardalos 1996). However, most of these global optimization algorithms are specialized to a narrow class of problems. The optimization methods examined here are those commonly used in engineering design applications.

Groenwold et al. (2000) applied eight global optimization algorithms to twelve test problems and found that no optimizer was the fastest on more than four of the problems. Different optimizers are designed to take advantage of different characteristics of various classes of problems. Because of this, Groenwold et al. proposed using multiple optimizers running in parallel on each problem and stopping the process once the first optimizer finds the global optimum. They found that this increased the speed of the optimization provided there were idle computers available.

Unfortunately, for most classes of nonconvex problems there is no way to guarantee that you have reached the global optimum with a feasible number of analyses except for very small design spaces (Haftka and Gürdal 1992). Some popular methods,

such as evolution strategies and the multistart method, can provide statistical estimates of the probability that the global optimum has been reached.

Evolution strategies include methods such as genetic algorithms, simulated annealing, and other stochastic methods. These methods rely on random steps to slowly move toward a better value. Statistical measures are used to determine whether a global optimum has been reached. Deterministic methods do not rely on random variables and their behavior is predictable based on the characteristics of the problem they are optimizing. These methods include optimizers such as Lipschitzian optimization (including DIRECT) and some pattern search methods.

Genetic Algorithms

Genetic algorithm strategies model the natural process of reproduction and survival of the fittest. They start with a random population of initial designs. Each design is coded into a chromosome or list of values representing all of the design variables together. Each variable is limited to discrete values and is usually coded in binary form or integer numbers. For binary coding, the range of each continuous variable can be divided into 2^n intervals and the value is coded as a binary string *n* digits long. For other divisions or discrete variables not all of the binary values will necessarily correspond to a possible variable value, in which case they must be removed from the population.

Each design is analyzed and the designs are ranked in order of best to worst. The parent designs are then randomly crossed depending on their relative performance to produce child designs. The best parent is the most likely to be selected and the worst parent is least likely. The two parent designs are combined in various ways ranging from simple single or multiple point crossover to more complicated selection of the order of

points based on some of the points from each design. This is useful for problems such as the traveling salesman problem where all of the points must be used once. Crossover involves taking two designs and selecting some number of points at random in the chromosome. The chromosomes are split at those points and two new chromosomes are formed using segments from both of the parent chromosomes. This continues until the population size of the new points equals the number of parent designs.

Mutation is also used to increase the randomness of the search and prevent the designs from converging to a small range too quickly. Each value in the chromosome has a small chance of randomly changing to a different value. After the population is set, they are checked to see that they are all possible designs and any illegal designs are regenerated (Haftka and Gürdal 1992).

Crossover

Parent 1 00110110|1100
Parent 2 01101110|0011
??
Offspring 1 00110110**0011**
Offspring 2 01101110**1100**

Figure 3. Illustration of single point crossover in Genetic algorithm

Mutation

Original Gene 01101**1**001101

Mutated Gene 01101**0**001101

Figure 4. Example of a single point mutation.

Ramírez-Rosado and Bernal-Agustín (1998) used genetic algorithms in the design of power distribution networks with over 300 variables. They used an integer coding for simplicity and showed that it was possible to optimize this type of system using genetic algorithms. Similarly, Savic and Walters (1997) used genetic algorithms to design water distribution networks. Designs having up to 34 discrete variables with 16 possible values were coded using gray coding. This method is a variation of binary coding such that the

integer values of the binary numbers are rearranged such that adjacent values differ by only one bit. Their results were comparable to previous studies using other optimization methods.

Soh and Yang (1996) developed a fuzzy-logic version of genetic algorithms as a way to soften the constraint boundaries and handle imprecise performance data. Their method allows the designer to include fuzzy rules as a way of including expert knowledge and experience. This was shown to speed up the convergence to a lowest weight design on a variety of structural problems.

Vasconcelos et al. (1997) combined the global search capabilities of the genetic algorithm with the faster convergence of a local optimizer to optimize electromagnets. They tried several different criteria for switching from the genetic algorithm to an augmented Lagrangian method for the final convergence. They found that this generally speeded up the convergence rate but it hurt the optimizer's ability to locate the global optimum.

Simulated Annealing

Simulated annealing optimization is based on the physical process of slowly cooling a piece of metal so that it settles into a minimum energy state. While the metal is very hot the atoms can move freely into different energy configurations but as it cools, the atoms tend towards the lowest energy state provided the cooling schedule gives the atoms time to reach a stable configuration. For simulated annealing, the objective function value is treated as the energy state of the material and the changes in the design variables is analogous to the movement of the atoms.

The optimizer starts at a random point in the design space and takes a random step in any allowed direction. If the objective function decreases in value (for minimization)

then the new point is accepted and another random step is taken from the new point. If the new point has a higher function value then the new point can be rejected or accepted based on a random probabilistic decision. As the optimization progresses, the chance of an inferior point being accepted decreases slowly from almost 100% to 0%. This is analogous to the metal being cooled. This allows the optimizer to wander freely at first while still prejudicing it to move towards a better design. As the optimization progresses the probability that the optimizer can move to a higher function value decreases and it is forced to a local minimum (Haftka and Gürdal 1992).

Lombardi et al. (1992) examined the use of SA in the optimization of composite plates. They compared cooling strategies and the effect that the increase in the number of plies had on the cost of the optimization. They found that as the size of the design space increased, a smaller fraction of the designs needed to be explored to reach an optimum. Their results indicated that SA performed well on bucking and contiguous ply constrained problems but had difficulties with strain constraints.

Swift and Batill (1992) used SA for discrete optimization of several structural designs. They used finite element analyses of truss and wing structures and used the data to train a neural network to allow rapid calculation of additional designs. The design spaces contained from 410 to 4113 discrete designs and would have been too expensive to search completely. They produced improvements of 6% to 13% for all three test cases using less than 5000 iterations per optimization.

Tzan and Pantelides (1996) proposed a modified SA algorithm for optimizing structural designs. As better feasible designs were located, the search space was reduced. They also incorporated sensitivity analyses to determine which design variables needed

to be changed when the design violated constraints to return it to the feasible region. These changes were found to increase the convergence rate of the optimizer but violated the symmetry found in conventional simulated annealing, as the optimization progresses it becomes impossible to get back to some of the points in the design space. They demonstrated this method on the design of a ten bar truss and a ten story building and found it worked well for problems with dynamic constraints.

Particle Swarm

A relatively new stochastic optimization technique proposed by Kennedy and Eberhart (1995) is based on the swarming behavior of creatures such as insects, birds and fish. It is composed of multiple search 'bots' run in parallel through the design space. Each bot adjusts its path to attempt to move back towards the best point it had located while also attempting to move towards the best point located by any bot in the swarm. This is accomplished by assigning a position and step length or velocity at each iteration. The position at subsequent iterations is simply

$$x_{j,i+1} = x_{j,i} + v_{j,i} \,, \tag{1}$$

while the velocity at each iteration is augmented to redirect the bot back towards its individual best point and the swarm best point located.

$$v_{j,i+1} = v_{j,i} + c_1 r_1 \left(x_{j,b_l} - x_{j,i} \right) + c_2 r_2 \left(x_{b_g} - x_{j,i} \right), \tag{2}$$

where r_1 and r_2 are continuously updated random numbers, x_{j,b_l} is the best point located by bot *j* and x_{b_g} is the best point located by the swarm. When the paths of all of the bots are examined at once it mimics the flight of a swarm of insects about a light or

food source. The swarm of bots move about the best points located to converge towards an optimal solution.

There have been a number of variations to the particle swarm algorithms to attempt to maximize the performance over different classes of problems. Clerc (1999) suggested adding a constriction factor to the velocity term to limit its growth while Shi and Eberhart (1998) proposed adding an inertia term to reweight the velocity from the previous iteration and Fourie and Groenwold (2000) allowed for dynamic reduction of the weighting term based on the performance of the optimization. Schutte and Groenwold (2002) compared these different weighting methods for two simple truss problems and proposed a dynamic penalty function method for increasing the need for feasibility as the optimization progresses to deal with constraints. Carlisle and Dozier (2001) attempted to establish standard parameter settings for an off the shelf particle swarm optimizer. Ranges of values for the number of bots, values for c_1 and c_2, limits on the magnitude of v_i and other parameters were explored to determine general guidelines for using particle swarm optimization.

Multistart Methods

Multistart methods use local optimizers, which have been used in engineering for many years, to increase the chances of locating the global optimum. Standard gradient-based optimizers have been developed which are robust and efficient at locating a local optimum. By starting the local optimizer at multiple starting points scattered throughout the design space it is hoped that at least one of the optimizations will result in the global optimum. Statistical methods exist to determine the probability that the optimum you have located is the global optimum (Boender and Kan 1983 and Snyman and Fatti 1987).

One statistical stopping criterion for multistart optimization is based on the assumption that the region of convergence to the global optimum is at least as large as the region of convergence of all other local optima. After *n* local optimizations from random starting points, let *r* be the number of points that led the optimizer to the best function value located by any of the optimizations, $\tilde{f}$. This criterion states that the probability that the global optimum, f^*, has been found is given by;

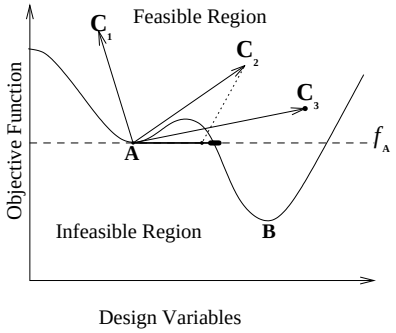
$$P\left(\tilde{f} = f^*\right) = 1 - \frac{(n+1)!(2n-r)!}{(2n+1)!(n-r)!} \tag{3}$$

The optimization stops once *P* exceeds a desired confidence level provided by the user. This is known as a Bayesian global stopping criteria (Groenwold et al. 1995, 2000).

Baker et al. (1998) optimized the design of an HSCT using multistart local optimization to examine substantially different types of designs. Relatively few starting points and no statistical measure of the probability of finding the global optimum were used. The different starting points were able to illustrate the separate local optima in their analysis models. Kam et al. (1996) used multistart local optimization to optimize the weight of a composite plate. The layers were represented as continuous variables in the local optimizations in order to make use of gradient-based optimizers. Instead of rounding the answers to whole numbers of layers, a branch-and-bound method was used to find the global optimum design in the neighborhood of the continuous solution. Groenwold et al. (1995) used a gradient-based optimization method known as dynamic search trajectory (Snyman 1983) to perform multistart optimization. This gradient method has some ability to move through weak local minima in order to reach a lower one nearby. It does this by modeling the optimization as a ball rolling down hill. As the

optimizer moves down the gradient it builds momentum which can carry it over small bumps in the objective function. It still functions primarily as a local optimizer however, and requires multiple starting points to indicate that it has located the global optimum and it is more expensive than other gradient methods due to the need to continuously calculate gradients.

Chan et al. (1999) presented a global optimization strategy using multistart optimization that uses tunneling to locate a starting point for the next local optimization. After the first local optimum is found, random directions are searched to see if a new point can be found with the same objective function as the local optimum. If such a point can be found, then it must lie in another valley than the previous optimum. If the search direction remains feasible, then the end point is extrapolated towards the origin to find a design with the same weight as the local optimum. The line connecting the local optimum to this point is searched to try to find a feasible point of similar weight. Additionally, if the points along the search direction change from infeasible to feasible as you move away from the local optimum then the feasible points most likely lie in a different valley. The new point is accepted as a starting point if the function value is less than the second best local optimum found to prevent it from tunneling back into a previous local optimum.



Possible Randon search results

1. Search does not leave current valley
2. Search direction is feasible
   Extrapolation towards the origin leads to a point of the same function value
   Extrapolation from local optimum leads to a point in a different valey
3. Search passes from infeasible to feasible
   The end of the search vector becomes a new starting point

Figure 5. Tunneling to locate a different objective function valley. Taken from Chan et al. (1999)

Pattern Search Methods

Generalized pattern search (GPS) theory defines a semi-global method for stepping through the design space in search of a good local optimum. It is a well established class of optimizers that has been in use for over 50 years on many classes of problems. At each iteration, a set of search directions and lengths are used to select a finite set of points surrounding the current point to analyze. This set of search directions must positively span $\Re^n$ and is selected from a finite pool of possible search directions for each algorithm. A set of directions, $\boldsymbol{V}$, is defined as positively spanning $\Re^n$ if it is possible to move from any point in $\Re^n$ to any point in $\Re^n$ through a finite number of steps in the directions of $\boldsymbol{V}$. If a better point is located in this way, the next iteration starts at the new best point and selects another set of surrounding points to analyze using the same (or different) set of search directions and the same step length or longer. If no improvement is made in an iteration, then the step lengths of the search are reduced and a new set of points are analyzed (Torczon 1997, Dennis and Torczon 1991, Audet and Dennis 2000A,B). In this way, the optimizer takes a series of finite steps through the design space on a finite set of search directions until it reaches a local optimum (Figure 5).

Torczon (1997) provided for a local convergence proof for a general pattern search algorithm; for any continuously differentiable problem, $\liminf_{k\to+\infty}\|\nabla f(x_k)\| = 0$. However, while initial steps may step from one local basin to another, there is no guarantee of convergence to a global optimum. The algorithm is capable of stepping over noise and narrow local optima and is a popular non-gradient based method. This class of algorithm is also well suited to mixed integer and other discrete problems due to

the fact that all of the points selected will lie on a mesh determined by the choice of the search pattern and the minimum step size.

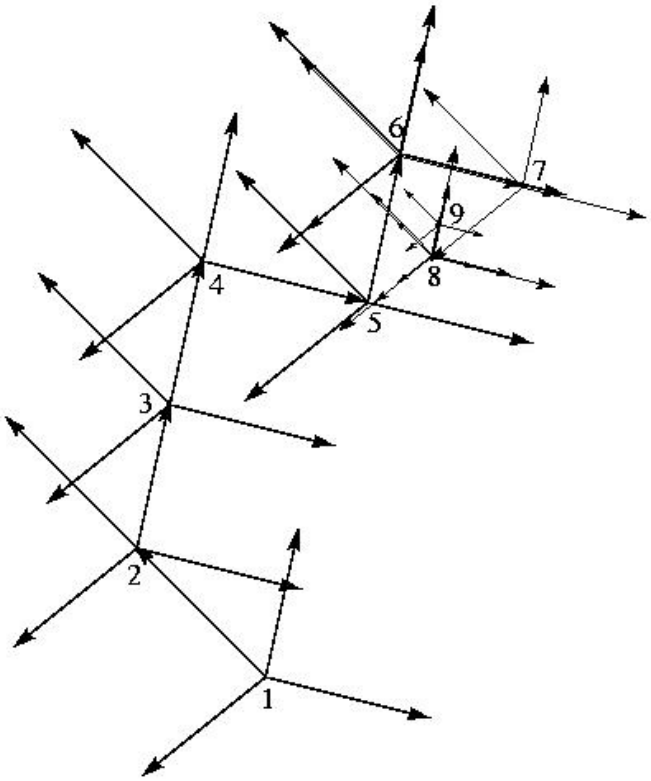


Figure 6. Nine steps of a pattern search with 4 possible step directions. The algorithm samples the function at the end of each search vector and moves to a new point with a better function value. If no improvement is found with the current step lengths (points 6 and 8), the step length is contracted and the pattern (or possibly a different pattern) is sampled about the current point.

Pattern search theory has been combined with filtering to handle constraints (Audet and Dennis 2000B), and with stochastic searches such as genetic algorithms to improve the convergence properties (Hart 2001). The algorithms may move to the first better point they locate at each iteration or can wait until all of the points have been analyzed before moving to the best point. For the first case, adaptive selection of the

order of the search directions to sample can be applied to speed the movement of the algorithm through the design space, reducing the number of points sampled at some iterations. Additional points can also be added to the ones selected by the search steps to take advantage of knowledge of the local behavior of the design space before performing a rigorous search of the pattern about the best point.

Lipschitzian Optimization

Lipschitzian optimization can be guaranteed to converge to the global optimum of the design space for an unconstrained problem with a continuous objective function that has a known Lipschitz constant. The Lipschitz constant is an upper bound on the absolute value of the slope of the function at any point in the design space. Since it is known that the function cannot change any faster than this it is possible to eliminate portions of the design space that are near analyzed designs with poor performance. In practical use it is rare that the Lipschitz constant is known for a physical optimization problem, but this method is used by estimating a value for the constant based on expert knowledge. If the constant is chosen too large, then the optimizer will be slow and spend more time searching near regions of poor performance. If the constant is chosen smaller than it should be, then the optimizer will search more near the best points it finds and ignore larger unexplored regions. This can allow it to settle on a local optimum instead of the global optimum. Commonly used Lipschitzian optimization also requires that all of the vertices of the design box be analyzed at the start. This requires 2^n evaluations, where n is the number of dimensions. This makes this method impractical for more than a few dimensions (Jones et al. 1993).

Jones et al. (1993) developed a variation of Lipschitzian optimization that addresses the problem of higher dimensions and an unknown Lipschitz constant. Jones'

DIRECT algorithm, (for *DI*viding *RECT*angles) progressively divides the design space into smaller boxes and analyses the center of each box instead of the vertices. This allows the optimizer to work on problems of higher dimension with fewer points at the beginning. The algorithm also selects points for sampling based on the assumption that the Lipschitz constant could be any number from zero to infinity. This allows a simultaneous global and local search of the design space where large unexplored regions are examined at the same time as the regions with good function values are explored in more detail. A complete description of this algorithm is found in Chapter 2.

Nelson and Papalambros (1998) modified this method by including a local optimizer when the design space is divided. When DIRECT selects rectangles to divide, it first performs a local optimization starting at the best function value found. It then divides the design space to place the local optimum found in its own box. This can speed up the convergence of the optimizer but it results in boxes with irregular aspect ratios and evaluation points that are not at the center of their respective boxes in the design space.

Gablonsky and Kelley (2001) proposed a locally biased version of the DIRECT algorithm. They modified the box size calculation to reduce the number of different box sizes at each iteration. This reduces the number of boxes which are divided at each iteration while still dividing the box which containes the best point located. Their formulation is intended for problems with only a few local optima. Baker et al. (2000 & 2001) examined several parallel versions of the DIRECT algorithm. They demonstrated that the algorithm can maintain a reasonable efficiency on up to 128 processors with a problem that takes about 1.7 seconds to analyze. More information on their work is found on page 28 and in Chapter 5. He et al. (2002) have examined data management for

DIRECT codes to reduce the excess memory used by some implementations of DIRECT. They developed dynamic data structures for a Fortran implementation of DIRECT to allow the code to adapt to arbitrary problem sizes while maintaining code efficiency.

Bartholomew-Biggs et al. (2002) combined DIRECT with a restart to modify the design space after DIRECT has been run for a number of iterations. In their version, after DIRECT has been run for a set number of iterations (usually 50-100) the optimizer is stopped and then restarted with a (possibly) smaller design space centered at the best point located. They found that this can reduce the cost of DIRECT for an aircraft routing problem. However, this removes a large portion of the design space from consideration and can potentially cause DIRECT to converge to a poorer local optimum.

Approximation Methods for Optimization and Reduced Cost

Response surface techniques were originally used to fit an approximation to physical experiments to reduce the noise in the observations and provide an inexpensive method for calculating the performance at other points. They are often used in optimization to replace expensive numerical analyses both for efficiency and to remove local numerical optima that are caused by finite resolution and incomplete convergence. Ishikawa et al. (1999) proposed using a radial basis method combined with sequential quadratic programming (SQP) to optimize engineering problems. The radial basis method is a way of interpolating the function based on a scattering of sample points. The data is fit to a basis function to allow efficient prediction of the function at other points and provide an approximate solution. From there SQP is used for the final convergence.

Jones et al. (1998) used a variation of response surface modeling known as a DACE model to perform global optimization. They combined high fidelity analyses to

create a simple response surface and added a correction factor based on how close the approximation was to a high fidelity point used to construct the surface. When a response surface is generated, it does not pass through the data points exactly. There is a known bias error in the response surface at each of these points. Since the error in the response surface near these points is correlated to the known error, a correction factor is calculated to find a closer approximation to the true function, similar to Kriging modeling. This allows the model to capture multimodal functions with even linear or quadratic response surfaces. They use this model for their efficient global optimization (EGO) algorithm. This algorithm uses statistical measures based on the potential error of the DACE model combined with the predicted values to find points where the expected improvement is maximized. It continues evaluating points and refining the model until convergence is satisfied. This optimizer compared favorably to other global optimizers with the added benefit that it provided a simple visualization tool for multiobjective optimization after the DACE models are generated. This can be very useful in dealing with problems where design requirements must be rewritten due to constraint violations.

Giunta and Watson (1998) compared the performance of a polynomial response surface to the kriging model for a simple test function of 1, 5 or 10 dimensions. The test problem was a combination of sinusoidal terms that could be shifted from a noisy quasi-quadratic form to a noisy quasi-sinusoidal form. For the one dimensional case kriging was more accurate for the sinusoidal form while the quadratic response surface modeled the quadratic form better, as expected. For the higher dimensional cases, however, the response surface approach was better for both forms. Simpson et al. (1998) compared these two models for use in multidisciplinary design of an aerospike propulsion system.

response surface and kriging models were fit to the structural weight and the aerodynamic performance of the engine and the results were compared for their accuracy. They were then used to optimize the engine for four different criteria. Their results indicated that the errors were similar for both methods. They demonstrated that either of these models could be used for approximating the high-fidelity results for optimization.

Knill et al. (1999) turned to response surfaces to reduce noise and eliminate spurious local optima in the design of the HSCT. By identifying the major sources of noise in their analysis and replacing them with response surface they were able to transform a five design variable space with many local optima from noise into a convex region with one optimum. For the higher dimensional design spaces they were able to smooth the objective function and constraints and reduce the number of local optima found by local optimizers.

Liu et al. (1999) used response surfaces to combine panel level optimizations with wing level optimizations for a composite wing box structure. This method works well when the design variables can be divided into sets of local variables with a small number of connecting global variables, in this case three in-plane loads and the number of 0^{o}, $\pm 45^{o}$ and 90^{o} plies. Genetic optimizations were used to optimize ply arrangements for composite plates based on given numbers of plies and loads to maximize the buckling load multiplier. These optimum values were fitted to a response surface to provide a cheap approximation to the buckling load multiplier for any combination of plies and loads within the design space. This allowed them to perform global optimization of the ply arrangements followed by a local optimization of the wing thicknesses. Performing a global optimization of the entire wing at once would have been much too expensive.

Multifidelity Optimization

As computer power has increased, new analysis codes have been developed which can accurately predict the behavior of unusual and complicated structures. However, these codes can take anywhere from hours to days to generate high fidelity results. For this reason they have been used extensively in the final stages of the design cycle but they are generally considered too expensive for use in the conceptual stage of the design where as many as tens of thousands of analyses must be performed. In order to search more of the design space in the conceptual stage, designers are forced to use less expensive, lower-fidelity versions of these codes. A prominent area of research today is designed to find ways to combine the low- and high-fidelity analysis codes so that the designer gains the accuracy of the high-fidelity codes while retaining the lower computational cost of the low-fidelity codes.

One disadvantage of using response surfaces for approximating a high-fidelity function is that they are not based on the physical properties of the behavior they are trying to represent. Because they are based on a small number of points and are low order polynomials, they are best at modeling small regions of design space. For larger regions, they can fail to capture local fluctuations on the performance of the system and generally have large errors wherever extrapolation is required. For most engineering applications, low-fidelity models based on the physical behavior of the system components are available. These models match the trends in the design space better than response surfaces over the entire design space. In order to obtain the accuracy of the high-fidelity models, however, you must have some way of including the high-fidelity analyses in the optimization.

Haftka (1991) proposed a method for scaling simple approximations to improve their accuracy over large regions of design space known as a Global Local Approximation (GLA). He proposed using a linear scaling factor to preserve the accuracy of the corrected approximate function over a larger region than a constant scaling factor or a Taylor approximation to the high-fidelity function. At an initial starting point, the function value and first derivatives of the high and low-fidelity analyses are calculated and used to create a correction function for the low-fidelity values such that:

$$f_h(x_0) = b(x_0) f_l(x_0) \tag{4}$$

and

$$\nabla f_h(x_0) = b(x_0) \nabla f_l(x_0) \tag{5}$$

where f_h is the high-fidelity value and f_l is the low-fidelity value. $\beta(x)$ is a linear function, which means that as the design gets further away from $\mathbf{x_0}$, the correction function will no longer correct the low-fidelity value to the high-fidelity value. He demonstrated the accuracy of this model as the design moves away from the starting point for a simple beam analysis. Chang et al. (1993) continued the comparison with other approximation models for the design of an HSCT wing box. They showed that GLA was able to increase the accuracy over that of several other approximation models as the design variables moved further from the starting point.

McQuade et al. (1995) has worked on combining high and low-fidelity analyses by using GLA on the design of a 2-D scramjet engine. They compared this method with the older method of using Taylor series approximations as the low-fidelity model. A trust region approach similar to traditional nonlinear programming is used to limit the optimizer from moving too far from the starting point. Once the move limit is reached, a

new set of high-fidelity analyses is used to update the correction function and the optimization is restarted with new move limits. This method is guaranteed to converge to the same design as an optimization on the high-fidelity function for appropriate move limits. GLA worked well for the optimization of the nozzle shape but failed to capture the forebody performance as well as the Taylor series. This was due to the movement of shock impingement points and showed the sensitivity of GLA to discontinuities in the design space. Alexandrov et al. (1998, 2000) used GLA to construct an approximation management framework for the optimization of 3-D wings. This method was shown to be effective for several local optimizers and is guaranteed to converge to the same design as an optimization on the high fidelity function for appropriate move limits.

Giunta et al. (1995) and Kaufman et al. (1996) used simple models to help determine which variables had the largest effect on wing structural weight and locate the regions of the design space where reasonable designs could occur. This reduced the volume of the design space that the optimizers had to examine with the more expensive evaluations. Response surfaces were fit to detailed analyses scattered throughout this reasonable design space and local optimizers were applied to these surfaces to find the optimum designs. Balabanov et al., (1998) used thousands of structural optimizations of designs using a coarse finite-element model to make a quadratic response surface for calculating wing bending material weights. They then used about a hundred refined finite-element optimizations to calculate a correction factor for the coarser model and generated a linear response surface model for the correction factor over the entire design space. This was found to reduce the errors in the low fidelity response surface by more than half.

Vitali et al. (1998) used correction response surfaces to calculate stress intensity factors for a cracked composite plate. They fit a quadratic response surface to the results from a high fidelity analysis of the cracked plate and compared the results to using a quadratic response surface of the ratio of the high fidelity model to a simpler model to correct the simple model. They found that it was more accurate to use a correction factor on a simple model than to use traditional response surfaces directly on the high fidelity model. Vitali and Sankar (1999) then used this method to optimize the weight of a stiffened composite panel constrained by crack propagation limits. The optimum weight found using the correction response surface satisfied the design constraints when checked by the high fidelity analysis.

Multidisciplinary Optimization

The design of complex structures usually involves many design variables from several different disciplines. The requirements for the structure may entail structural, mechanical, aerodynamic, electrical or magnetic considerations, each of which can require complex formulas and programs to evaluate. Traditional design methods consider each of these disciplines separately. Teams of specialists are used to optimize separate portions of the design and then the groups come together to compromise on design variables affecting multiple aspects of the design. This can lead to sub-optimal designs with no clear understanding of the tradeoffs between different disciplines (Korngold and Gabriele 1997). Integrating the analysis of each discipline into a single optimization can provide an explicit way to deal with their interactions and handle constraints. MacMillin et al. (1997) investigated the interaction of conflicting requirements of different disciplines on the gross takeoff weight for an HSCT design. They showed that the addition of conflicting constraints from different disciplines resulted in a design that was

as much as 35% heavier than a design that was optimized without these interactions. This shows the large changes that can occur when interdisciplinary effects are included in the optimization.

Kroo et al. (1994) presented a way of decomposing the multidisciplinary optimization of an aircraft into separate disciplines and sub-optimizations and using constraints to handle their interactions. The design is divided into separate disciplines with the constraints delegated to their respective disciplines and auxiliary variables are inserted to represent the cross dependencies of different disciplines. The system wide optimization attempts to minimize the objective function by providing the sub optimizers with design variables and target values for the auxiliary variables. Each sub optimization attempts to minimize the difference between the target values for the auxiliary variables and the values used to satisfy the local constraints. The system optimization is constrained so that the target values equal the values used in each sub-optimization. This method worked well for problems with few auxiliary variables and allows each sup-optimizer to use local expertise in optimizing each discipline.

Another popular way of combining multiple disciplines in to a single optimization is the use of response surfaces. Korngold and Gabriele (1997) used localized response surfaces to create an approximation for each discipline model involved in a design and then performed a global sensitivity analysis to approximate the global design space in a small region. This global approximation is then used to perform a discrete optimization for a design and manufacturing example program. After each optimization, the local response surfaces are regenerated and a new global approximation is calculated until convergence is satisfied.

Kaufman et al. (1996) used response surfaces to integrate the results from commercial codes from different disciplines. An empirical weight equation was used to locate the design variables with the largest effect on structural weight to reduce the number of dimensions that the response surface has to model. Structural optimizations were performed over the resulting design space and the results were used to fit a response surface for the wing structural weight of an HSCT design. This also served to reduce the noise found when trying to calculate gradients directly from the structural optimization results.

Parallel Computing

As analysis programs become more expensive and optimizers require more evaluations to search ever-larger design domains, the computing power of a single processor has become a major limiting factor to design optimization. Parallel systems have become available with thousands of processors that can perform trillions of operations per second. New optimization methods that distribute the work evenly over multiple processors hold much promise for allowing engineers to incorporate more expensive analysis programs and examine more design concepts with more variables.

Optimization offers many avenues for parallelization from coarse-grained division of large jobs to each processor to fine grained parallel codes that divide each job into small tasks for multiple processors. The optimizer can generate multiple evaluation requests at a time, each of which can run on a different machine. This allows the designer to use commercially available analysis codes that were written for serial machines without having to modify them. This is the simplest form of parallelization but it is limited in the number of processors that it can efficiently utilize. Krasteva et al. (1999) explored different methods of distributing these jobs to multiple processors.

Distributed dynamic load balancing techniques and termination criteria were examined to help determine efficient ways of utilizing multiple processors with a minimum amount of processor idle time.

Vendors have also started to modify some of their codes to run on parallel machines. By dividing each analysis into smaller parts and spreading them over a large number of processors, the optimization can utilize more processors and reduce latency where processors sit idle waiting for other large jobs to finish. This creates other problems with communication and synchronization between processors and requires a lot of work to efficiently divide the analysis into equal parts that do not rely extensively on one another (Culler et al. 1999).

Burgee et al. (1996) used parallel computing to evaluate multiple designs at once to create response surface models for each discipline. Simple models were used to locate good regions in the design space where more complex models were used to generate a local response surface. Response surfaces allowed the optimizer to use black box programs for separate disciplines and combine them for inexpensive genetic optimization. In this way the analysis within each discipline was done in parallel as opposed to parallelism across disciplines. Weston et al. (1994) described a Framework for Interdisciplinary Design Optimization (FIDO), which is designed to break up each analysis into separate disciplines and evaluating them in parallel. They optimized a simplified HSCT design by running structural, aerodynamic, performance and propulsion codes on separate workstations and using FIDO to coordinate the interactions between them. The FIDO program allows the user to utilize any stand-alone analysis code which

can be parallel or sequential. It also allows the user to replace specific codes if higher fidelity is needed.

Bhardwaj et al. (2000) described a parallel structural dynamics code for static and transient response of structures. The code was tested on as many as 1000 processors on problems with as many as 8 million degrees of freedom. This program demonstrates the benefits that can be obtained from using a parallel implementation by reducing the solution time for one problem from 5 days to 30 minutes.

Baker et al. (2000) examined the use of the DIRECT algorithm on a parallel architecture. They demonstrated the variation of evaluations needed per iteration for the DIRECT optimizer for the optimization of an HSCT using 28 variables. They examined different load balancing strategies which provided up to 86% efficiency for 64 processors on a problem where each evaluation took about 1.75 seconds. However, as written for sequential operation, DIRECT requested at most 400 evaluations per iteration and the large variation in the number of evaluations per iteration could make the use of more processors inefficient. Modifications to the DIRECT algorithm to generate more evaluation requests per iteration would be needed to take advantage of more processors.

Overview of Research

The main thrust of this dissertation is efficient ways to deal with global optimization of complex structures. An efficient global optimization method is selected as a starting point for combining with alternate concepts for reducing the time required to perform an optimization. The dissertation then examines ways of combining multifidelity information and multiple processors with the chosen algorithm to improve the utility of an already efficient algorithm. The next section of this dissertation presents the work done comparing three single fidelity global optimization algorithms. They are

compared to see how they deal with noise and multimodal functions for high dimensional problems. The results of this work have been published in the Journal of Global Optimization. Chapter 3 describes the DIRECT-BP algorithm, a modification to DIRECT intended to provide for a more robust search about the best local optimum located by DIRECT as the optimization progresses. Chapter 4 describes the multifidelity versions of the DIRECT optimizer used in this research and the results obtained on several test problems. Chapter 5 presents a parallel version of the DIRECT optimizer designed for use on large clusters of heterogeneous processors.

CHAPTER 2
PERFORMANCE OF GLOBAL OPTIMIZATION ALGORITHMS: NOISE VERSUS WIDELY SEPARATED OPTIMA

The need for global optimization is driven by a combination of multiple locally optimal designs for complex structures and noise in analysis functions due to the use of discrete and iterative analysis methods. The purpose of this chapter is to examine some general algorithms to see how they perform with respect to each of these two conditions. Three global optimization methods were compared on how well they optimized these two types of problems. The first method uses multiple starting points to optimize the design using sequential quadratic programming (SQP) as implemented in DOT, (Vanderplaats 1995). The second method, Snyman's dynamic search algorithm, (Snyman 1983), is capable of passing through shallow local minima to locate a better optimum but still requires multiple starting points. The third method is DIRECT, which is run only once.

Two test functions were examined to explore the performance of the optimizers on these two types of problems, the so-called Quartic and Griewank functions. The comparisons help to show the strengths of each optimizer and suggest the types of problems each is best suited for. The optimizers were then run on a more complicated problem with both numerical noise and widely separated local optima to see how they handled a more realistic engineering problem.

The configuration design of the High Speed Civil Transport (HSCT) is an example of a high dimensional design problem with a nonconvex feasible design space due to nonlinear constraints and numerical noise (Knill et al. 1999). This is typical of

many problems in industry that require global optimization. The analysis code uses a combination of simple functions from various disciplines to evaluate general planforms for the HSCT. The combination of iterative range calculations, structural formulas which are pieced together from multiple programs and minimum gauge and spacing requirements lead to irregular design spaces and numerical noise. The relative impact of noise and multiple local optima can also be adjusted by including more variables.

Optimizers

Sequential Quadratic Programming

The commercial program *D*esign *O*ptimization *T*ools (DOT), (Vanderplaats 1995), was used to optimize the HSCT design using SQP. SQP forms quadratic approximations of the objective function and a linear approximation for the constraints and moves towards the optimum within given move limits. It then forms a new approximation and repeats the process until it reaches a local optimum. Due to the use of approximations, DOT is relatively quick to perform a single optimization and will handle a limited amount of noise without becoming stuck in spurious local minima. DOT has been successfully used in the past with the HSCT problem, and compared favorably to other local optimizers (Haim et. al. 1999). In order to perform a global search of the design space, DOT was started at 100 random initial designs for the HSCT problem and up to 20,000 starting points for the test problems. The lowest objective function value found for each case was taken as the global minimum.

Dynamic Search

The second optimizer tested was Snyman's dynamic search method, Leap Frog Optimization Procedure with Constraints, Version 3 (LFOPCV3) (Snyman 1983). LFOPCV3 is a semi-global optimization method that is capable of moving through

shallow local minima. The method is based on the physical situation of a particle rolling down hill. As the particle moves down, it builds momentum, which carries it out of small dips in its path. At each step, LFOPCV3 subtracts the gradient of the objective function from the velocity of the particle from the previous step. The new velocity determines the step size and direction for the current step. When the velocity vector makes an angle of more than 90^{o} with the gradient vector, i.e., when it is descending, the velocity increases and the particle builds up momentum to go through areas where it is ascending. When ascent occurs, a damping strategy is used to extract energy from the particle to prevent endless oscillation about a minimum. The dampening strategy used in the version of LFOPCV3 used here has been modified to increase its ability to search further away from a local optimum. Whenever the optimizer is at a point where the function value is higher than the best function value located so far, it creates a shallow quadratic function centered at the best location found and uses the slope of the quadratic function plus a small percentage of the objective function slope to calculate the next step. This reduces the dampening when it is moving uphill while preventing it from becoming trapped in a local optimum that is worse than the best point found so far.

LFOPCV3 handles constraints with a standard quadratic penalty function approach. This causes the gradient of the penalty function to increase as the constraint violation increases which gives a smooth gradient at the constraint boundaries. In our initial experiments with this algorithm it demonstrated the ability to move further than DOT when searching for the global minimum and has been successfully used on functions with hundreds of local minima. However, it still requires multiple starts to sample the entire design box. For this comparison, we used the same 100 initial designs

to compare the performance with DOT for the HSCT problem and up to 20,000 starting points for the test problems.

The DIRECT Algorithm

The DIRECT algorithm (Jones et al. 1993) is a variation of Lipschitzian optimization that uses all values for the Lipschitz constant. The basic problem that DIRECT can solve is defined as

$$\begin{aligned} \min \quad & f(x) \\ & x \in \Re^n \\ & x_L \le x \le x_U \end{aligned} \tag{6}$$

where $x_L, x_U \in R^n$ are simple bounds on the vector **x**. Implementations of DIRECT commonly renormalize these bounds such that $x_L = 0$ and $x_U = e = (1,\ldots,1)$.

Classical Lipschitzian optimization requires the user to specify the Lipschitz constant *K*, which is used as a prediction of the maximum possible slope of the objective function over the global domain. Lipschitzian optimization uses the value of the objective function at the corners of each box and *K* to find the box with potentially the lowest objective function value. The design with the predicted minimum possible function value is evaluated and the process is repeated for a set number of iterations. This method is guaranteed to asymptotically approach the global optimum for a Lipschitz continuous function provided a large enough value of *K* is used.

DIRECT does not require an estimate of the Lipschitz constant. Instead of evaluating every vertex of each box, DIRECT uses the function value at the center of each box and the box size to find the boxes which potentially contain the optimum. This reduces the number of points that need to be analyzed at the first iteration and allows the use of DIRECT on higher dimensional problems than are usually feasible for Lipschitzian

optimization. DIRECT selects a box *j* to divide if using some Lipschitz constant K_i, that box could contain a lower function value than any other box. DIRECT is guaranteed to come within γ of the global optimum once there are no boxes left larger than γ.

The DIRECT algorithm is given below. The parameter $\gamma \geq 0$ defines the minimum box diameter that is divided by DIRECT. If γ is nonzero, then it can implicitly define a stopping condition for DIRECT. The boxes generated by DIRECT can be uniquely labeled with an integer index, *j*. We denote the center of box *j* as c_j, and let $X_j(t)$ denote box *j* in iteration *t*. This reflects the fact that the size of box *j* can vary over time, but the center remains the same. The set B_t is the set of all boxes considered in iteration *t*. Note that boxes are never removed from B_t; dividing a box $X_j(t)$ simply shrinks it and adds new boxes, so $X_j(t+1) \in B\{t+1\}$ after $X_j(t)$ is divided. We denote the best box in iteration *t* as X*(t), which has center c*(t). The function diam(X_j (t)) computes the diameter of $X_j(t)$, twice the distance from one vertex of $X_j(t)$ to center point of that box. Finally, recall that e_i is the unit vector in the *i*-th dimension.

1. Normalize the search space to the unit box. Evaluate $f(c_1)$ and set $B_t = \{X_1\}$.
2. For t =1, …
    a. Identify the set $S_t \subseteq B_t$ of potentially optimal boxes.
    b. For each box $X_j(t) \in S_t$ where diam($X_j(t)$) > γ:
        i. Identify *I*, the dimensions of $X_j(t)$ with the longest side length. Let δ equal one-third of this length.
        ii. Sample the function at the points $c_j \pm \delta e_i$ for all $i \in I$, where c_j is the center of box $X_j(t)$ and e_i is the *i*th unit vector.
        iii. While *I* is not empty:

1. Divide $X_j(t)$ into thirds along the dimension $i \in I$ with the lowest value of $f(c_j \pm \delta e_i)$ to create two new boxes and reduce the size of box $X_j(t)$.
2. Remove i from I and add the new boxes to B_t.

c. Terminate if an appropriate stopping rule is satisfied.

Since DIRECT divides boxes of all scales simultaneously, it can perform both a localized search and a global search (using both low and high values for the Lipschitz constant K). However, to ensure sufficient local progress, K is required to be large enough that there is a minimum possible improvement, ε, over the current best point based on a slope of K. This is to prevent a small box from being divided when there is not much predicted possible improvement within that box. This lower bound on K is a function of box size; smaller boxes will have a higher lower bound than larger boxes. For our work we have used $\varepsilon = 10^{-8}$.

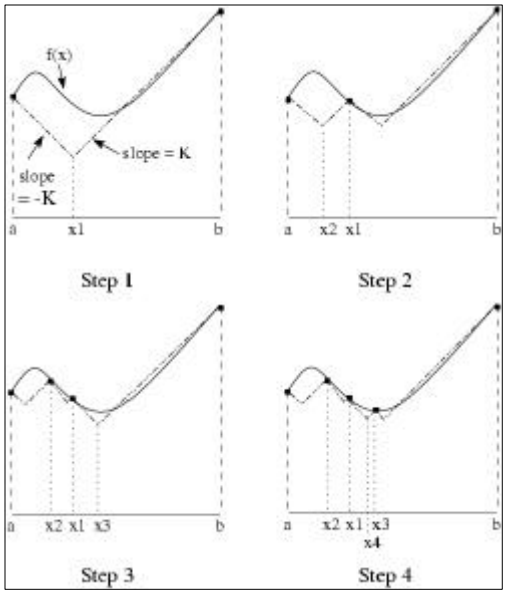


Figure 7. Lipschitzian Optimization

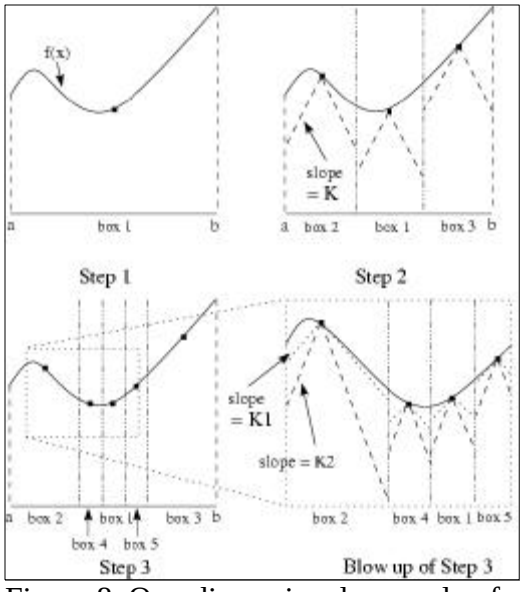


Figure 8. One-dimensional example of DIRECT

Figures 7 and 8 contrasts the behavior of DIRECT and Lipschitzian optimization on a simple one-dimensional problem. In Step 2 of Figure 8, box #1 is selected for dividing because, for any value of the Lipschitz constant *K*, that box can potentially contain a lower function value than any other box. In Step 3, both boxes 2 and 4 are selected for division because there are some values of *K*, *K1* and *K2*, for which those boxes potentially contain a lower value than any other box. It can be shown (Jones et al. 1993) that this requires these two boxes to lie on the bottom, right hand part of the convex hull of the set of boxes in a graph such as Figure 9.

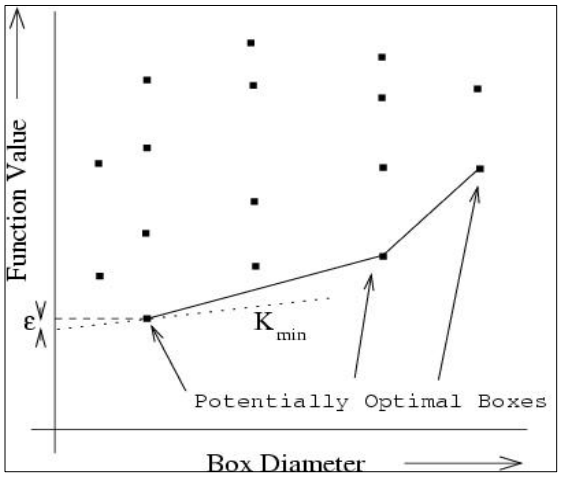


Figure 9. Point selection in DIRECT.

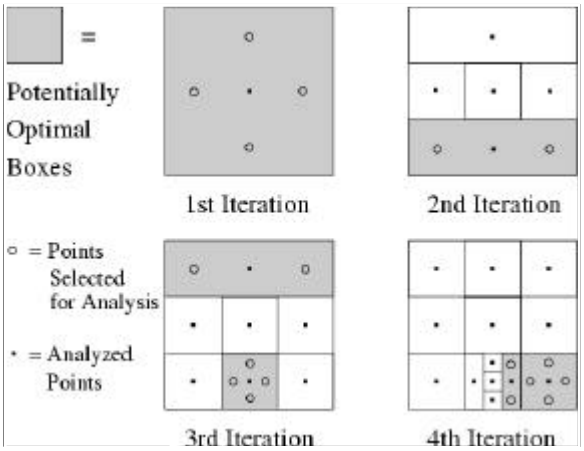


Figure 10. DIRECT box division in 2D.

In Figure 9, each of the boxes is one of five sizes. The best designs from the second smallest box size and the largest and next to largest box sizes are potentially optimal because, for each of these, there is some value of *K* where that box could contain a better design than any other box. The box with the lowest function value is not potentially optimal because it is not a large improvement over the function value of the next larger box on the convex hull. Any value of *K* which would make this box potentially optimal would not provide for sufficient improvement over the current best value.

Graham's scan routine is used to identify the potentially optimal boxes (Jones et al. 1993). Boxes that were not previously potentially optimal can become potentially optimal in later iterations as the other boxes are divided. (Figure 10) As the boxes on the convex hull of Figure 9 are divided, new boxes are added and the box that was divided becomes smaller. This may put a new set of boxes on the convex hull, some of which were interior points in previous iterations. DIRECT will usually select more than one box to divide at each iteration. This allows for a wider search than Lipschitzian optimization and makes DIRECT more suitable for parallelization (Baker et al. 2000).

Potentially optimal boxes are divided into thirds on their long sides (Figure 10). Each dimension is examined separately, with two points added in each coordinate direction corresponding to a long side of the potentially optimal box. Thus the number of points from each box division increases linearly with the number of dimensions. These new points are ranked and the box containing the center point is divided into thirds in the dimension which will put the best new point into a box of its own. This is continued with the rest of the long dimensions, dividing off the best remaining point each time until the original box has been divided on all of the long sides.

More recent work by Jones has suggested dividing each potentially optimal box in only one dimension per iteration. The selection of which long dimension to divide is based on which dimensions have been divided the most so far over the entire design space. He has found that this can reduce the cost of the optimization while only slightly reducing the robustness of the algorithm. The comparison given in this chapter was completed before the modified algorithm was proposed, however, the original algorithm was more useful for this work for several reasons. The extra robustness of the original

algorithm is helpful for some of the difficult test problems used where there are a large number of local optima and the algorithm needs to explore more of the design space before concentrating on the best optimum it has found. There is more work using the original algorithm in the literature to examine. Finally, the conversion of DIRECT to a parallel implementation needs to increase the number of points analyzed at each iteration to allow the use of more processors and improve the load balancing across them.

One of the shortcomings of the DIRECT algorithm is the lack of an obvious stopping criterion. Different authors have advocated stopping after a set number of function evaluations or a limit on the number of iterations (Jones et al. 1993, Nelson II 1998 and Baker et al. 2000). However, this does not take into account the behavior of DIRECT and can lead to continuing the search long after the minimum is located or stopping it while improvements are still being made. This implementation of DIRECT has employed a stopping criteria based on the smallest current box. The search is stopped once the size of the smallest box has reached a set limit, and a local optimizer is then used to refine the solution in the immediate vicinity. This is motivated by the recognized characteristic of DIRECT that it gets close to the optimum relatively quickly but the final convergence is slow compared to gradient based methods. DOT was used to perform SQP optimizations from up to 15 different starting points after DIRECT was stopped. These points were the 15 best points separated by at least 5% of the width of the design space. Additional stopping conditions are contained in recent work by He et al. (2002).

Constraints

The implementation of DIRECT used here uses a linear penalty function to handle constraints, $g = 0$. Initial work with this code was based on a quadratic penalty function.

This tended to over penalize design points with moderate constraint violations. Due to the sparse sampling of the design space in the first few iterations, this penalty function could eliminate large regions from consideration. A linear penalty function was better able to sample more of the design space while still concentrating on the most promising regions. The penalty function is given as follows.

$$F = f + \sum_{i=1}^{n} g_i P_i \,, \tag{7}$$

where f is the objective function, $\boldsymbol{g}$ is the constraint vector, and P_i is a penalty parameter, which is zero for satisfied constraints and a small positive number for violated constraints.

The performance of DIRECT depends on the magnitude of the penalty parameter, P, used. The parameter should be as small as possible while still preventing the optimizer from selecting an infeasible design. Otherwise, boxes which cover the constraint boundaries can be over penalized by an infeasible center. This can make it unlikely that DIRECT will analyze points along the constraint boundaries. Choosing a value that is approximately twice the magnitude of the largest Lagrange multipliers will push the optimizer out of the infeasible region without over penalizing boxes with an infeasible center.

For the HSCT problem the range constraint had the largest effect on the objective function. On average, the HSCT requires approximately 90 lbs. of fuel per mile of range deficiency. Therefore, the penalty constant was chosen to increase the objective function by twice this much per mile of violation. The range constraint is given as:

$$g = 20\left(\frac{R}{5500} - 1\right) \tag{8}$$

where *R* is the range. The objective function, f, is the gross weight normalized by 700,000 lbs. This gives a value of 0.071 for the penalty multiplier. For our study we rounded this up to 0.1.

Test Functions

The optimizers were compared for two algebraic test functions in addition to the HSCT problems. The Griewank function was used as an example of a problem with one true optimum superimposed with noise. A simple quartic function was optimized in a hypercube to examine the optimizer's ability to locate the global optimum from a large number of widely separated local optima. Each optimizer was run multiple times for the 5, 10, and 20-design variable (DV) cases and the results are given below.

Griewank Function

The Griewank function is a quadratic function with noise added by including a cosine function. Without the noise, the function is convex with one optimum. This function was used to test how well the optimizers could move through the noise to locate the actual optimum. The one-dimensional Griewank function is given in Figure 11.

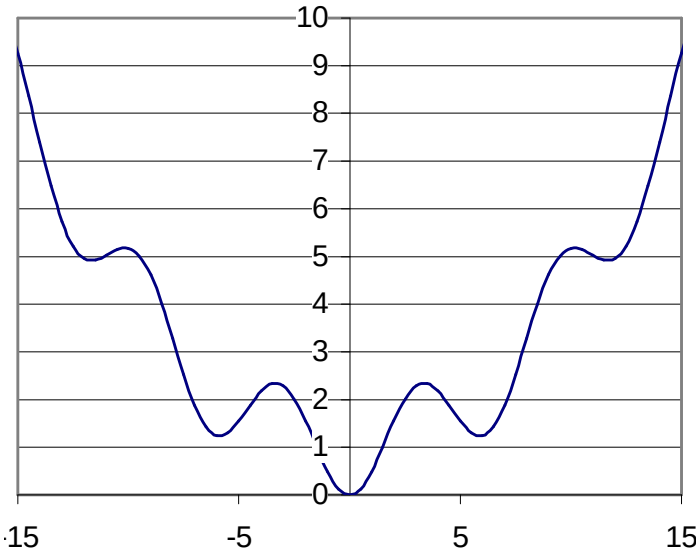


Figure 11. Griewank function in one dimension.

The *n*-dimensional Griewank function is defined as:

$$F(x) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{d} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right), \qquad (9)$$

The relative strength of the noise can be controlled by adjusting *d*. When *d* is small, the Griewank function is primarily a quadratic function with a noise component from the cosine terms. As *d* is increased, the objective function becomes flatter and the cosine portion becomes dominant. The function changes from a noisy function with one global optimum, to an almost flat surface with hundreds of nearly equal local optima.

The constant *d* is taken to be 200, 1000 and 20,000 for the 5, 10, and 20 DV cases, respectively. The design domain was $[-400,600]^n$ for the DOT and LFOPCV3 optimizers with random starting points. For DIRECT, the box had edges of length 1000 with the upper limit randomly set between 100 and 900 to randomize the results for the different runs. This value of *d* and the size of the design space is much larger than that used by He et al. (2002), which makes the problem more difficult to solve. Each optimizer was run enough times to compute the average number of evaluations for each optimum found. The global optimum is at $\mathbf{x} = \mathbf{0}$. For this comparison, the optimizer was considered to have reached the global optimum if all of the design variables were in the range ± 0.1. The optimizers were compared based on the number of function evaluations required and the number of times each optimizer located the global optimum. The results are given in Table 1.

We use two measures of efficiency of the optimizers. The first is the average number of function evaluations per optimum. The second measure is the number of function evaluations needed to achieve 90% confidence that the global optimum was found. The first measure favors the random search procedure because it is based on

Table 1. Comparisons for 5, 10, and 20 DV Griewank functions. (d = 200, 1000 and 20000 respectively)

5 DV	# of optima located / # of runs	Function evaluations per optimum	90% confidence
DIRECT	87/100	3400	3335
DOT	5/300	10260	23416
LFOPCV3	60/400	9050	19235
10 DV			
DIRECT	56 / 100	11810	18550
DOT	96 / 500	1260	2604
LFOPCV3	136 / 500	32240	63598
20 DV			
DIRECT	8 / 200	102650	231593
DOT	16 / 2000	19740	45265
LFOPCV3	128 / 2000	7220	16084

average performance that does not take into account that occasionally random search can give poor results purely due to chance. The second measure includes some cost of "insurance" against chance. To calculate the number of function evaluations for 90% confidence, we start by observing that the probability of not locating the optimum with one optimization run is

$$p_1=(n-1)/n \tag{10}$$

where *n* is the total number of optimization runs per optimum located. The probability of not locating the optimum in *r* optimizations is

$$p_r= ((n-1)/n)^r \tag{11}$$

We set $p_r=0.1$, and solve for *r*, which gives the number of runs needed for 90% confidence that the global optimum was found. Multiplying this number by the average number of function evaluations per run, *a*, gives the desired number of function evaluations for 90% confidence, f_{90}.

$$f_{90} = a\frac{\ln(0.1)}{\ln\left(1-\frac{1}{n}\right)} \tag{12}$$

When p_1 is close to one, it is easy to check that the number of function evaluations for 90% confidence is approximately 2.3 (ln10) times the average number of function evaluations per optimum.

The results in Table 1 do not give a clear advantage to any one optimizer but they do indicate that DIRECT is not the most efficient optimizer in higher dimensional space. The Griewank function is smooth with an underlying quadratic shape. This is suitable for gradient based optimization if the algorithm is capable of moving past the weak local optima. LFOPCV3 does this by design while DOT is able to do this by using approximations based on widely separated points.

The lack of a clear trend in Table 1 may be due to the effect of *d*, which was different for each case. The relative strength of the noise changed as the number of design variables increased. To investigate the effect of the noise on the different optimizers, the 10-variable case was repeated for different values of *d*. The results for this comparison are given in Table 2.

Table 2. Effect of Variation of *d* in 10-dimensional Griewank function.

DIRECT - 100 runs	# of times optimum located	function evaluations per optimum	90% Confidence
d = 4000	19/100	39480	81958
d = 1000	56/100	11810	18550
d = 200	83/100	6990	7539
DOT – 500 runs			
d = 4000	37/500	3970	8801
d = 1000	96/500	1260	2604
d = 200	160/500	620	1188
LFOPCV - 500 runs			
d = 4000	25/500	193410	434117
d = 1000	136/500	32240	63598
d = 200	390/500	830	989

As expected, the optimization is easier for small values of *d*, where the global optimum is more distinct from other local optima. DOT appears to be the least sensitive to this parameter while LFOPCV3 was the most sensitive to *d*. The approximations used by DOT appear to allow it to move past the noise and capture the underlying quadratic function. LFOPCV3, however, is slowed down as it passes through the noise and is not able to locate small improvements in the local optima for the large value of *d*.

Quartic Function

The one-dimensional quartic function is given in Figure 12.

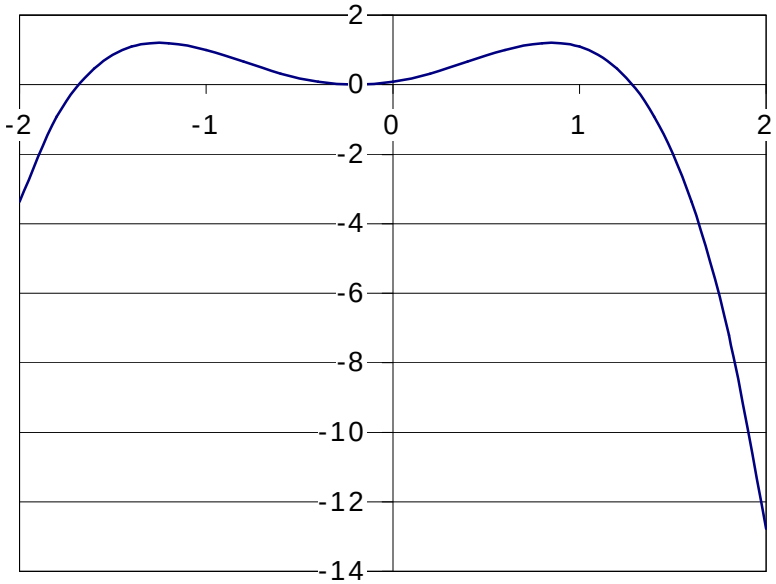


Figure 12. Quartic function in one dimension(*e*= 0.2).

The *n*-dimensional quartic function is defined as

$$F(x) = \sum_{i=1}^{n} \left[2.2(x_i + e_i)^2 - (x_i + e_i)^4 \right], \qquad (13)$$

where *n* is the number of dimensions and e_i is a random number uniformly distributed in the range [0.2, 0.4]. At the start of each optimization, the *n* values of **e** are selected and frozen until the optimization is complete. The values are then randomly selected again

for the next run. The design space was the hypercube $[-2,2]^n$. The global optimum is at $\mathbf{x} = \mathbf{2}$. This problem contains 3^n local optima located at the constraint boundaries and close to the center of each variable. This problem was used to examine the optimizer's abilities to locate the global optimum for a nonconvex objective function with a large number of widely separated local optima. The optimizer was considered to have located the optimum if all of the design variables were greater than 1.9.

DIRECT was run for 200 different random combinations of **e** for each case. LFOPCV3 and DOT used 20,000 starting points in order to get a statistically meaningful average number of function evaluations per optimum located. The results are given in Table 3.

Table 3. Comparisons for 5, 10, and 20 DV quartic functions.

5 DV	# of optima located / # of runs	function evaluations per optimum	90% confidence
DIRECT	200 / 200	1025	1025
DOT	410 / 20000	3397	7741
LFOPCV3	8 / 20000	2581418	5942746
10 DV			
DIRECT	200 / 200	2192	2192
DOT	16 / 20000	174189	400925
LFOPCV3	0 / 20000	-	-
20 DV			
DIRECT	200 / 200	11266	11266
DOT	0 / 20000	-	-
LFOPCV3	0 / 20000	-	-

For the Griewank function, none of the optimizers were 100% effective in locating the optimum. For the quartic function, DIRECT was 100% efficient at locating the optimum. This means that a single run of DIRECT more than satisfies the 90%

requirement. For this reason the 90% confidence column equals the function evaluations per optimum column for DIRECT.

DIRECT is clearly better suited for this problem than either DOT or LFOPCV3. Examination of the order of the points analyzed by DIRECT indicates that it is able to detect the slight global trend towards the global optimum from the first iteration. At each iteration, DIRECT divided the box which contains the global optimum. Due to the extreme change in function value near each local optimum, once DIRECT has sampled a point in the basin of a local optimum, it quickly refines the solution in that region. The large difference in function values also has the effect of masking the intermediate sized boxes when the convex hull is calculated for the potentially optimal box selection. DIRECT divided the best box and some of the largest boxes at each iteration while ignoring the intermediate sized boxes with average function values. The fact that DIRECT found the global optimum at the beginning of the search allowed it to quickly move to the best optimum but limited the amount of the design space that was searched. If DIRECT initially locates a different deep local optimum, then the optimization will converge to that local optimum as our experiments with the parallel DIRECT found (Chapter 5). While DIRECT does locate the basins of other local optima, its ability to locate the global trend in this problem allowed it to immediately move to the basin of the global optimum.

DOT is unable to adequately capture the shape of the entire design space with the quadratic approximation. The quadratic approximation is only good for describing small portions of the design space, which prevents it from accurately extrapolating to the regions of other local optima.

LFOPCV3 is strictly a local optimizer for this problem. It is designed to move through noise and weak local minima, not the deep basins found in this problem. In order for this optimizer to find the global optimum, it must start in the basin that contains the global optimum. For $e = 0.3$, the odds of starting in this region for the 5, 10, and 20 DV cases are 1 in 333, 1 in 111,000 and 1 in 123×10^8 respectively. This makes multistart local optimization methods a poor choice for this type of problem. LFOPCV3's performance in the 5 DV case indicates that it is not even this good at locating the optimum. In many cases LFOPCV3 stopped at saddle points and some points where the slope was not zero.

HSCT Design Problem

The design problem used to compare the three optimization methods is the configuration design of a HSCT. The HSCT design code uses up to 29 design variables and up to 68 constraints (MacMillin et al. 1997). The design goal is to minimize the gross take off weight (GTOW) while satisfying 68 range, performance, and geometric constraints. The HSCT code uses simple structural and aerodynamic models to analyze a conceptual design of the aircraft. In more recent versions of the code the aerodynamic analysis for the drag is replaced by a response surface constructed on the basis of a large number of simulations.

The version of the HSCT code employed in this study employs an aerodynamic drag response surface based on solutions of the Euler equations (McGrory et al. 1993). Once the design variables were selected using design of experiments theory, the wing camber for each design was found using Carlson's modified linear theory optimization method WINGDES (Carlson and Miller 1974 and Carlson and Walkley 1984). This design was then analyzed using the Euler equations. The Euler analysis was made at two

angles of attack and a response surface was constructed for the drag polar in terms of the intervening variables C_{Do} and K:

$$C_D = C_{Do} + KC_L^2. \tag{14}$$

The viscous contribution to the drag C_{Do} is obtained from standard algebraic estimates of the skin friction assuming turbulent flow (Hopkins 1972).

Previous work with the HSCT code has demonstrated the nonconvexity of the feasible domain and the existence of multiple local minima (Knill et al. 1999). In addition, several analyses and sub optimizations, including range calculations and structural weights, result in noisy performance functions. This noise adds additional local minima and makes accurate gradient calculations difficult. Optimization methods must be able to deal with this noise and move from one region of design space to another without becoming trapped in local minima.

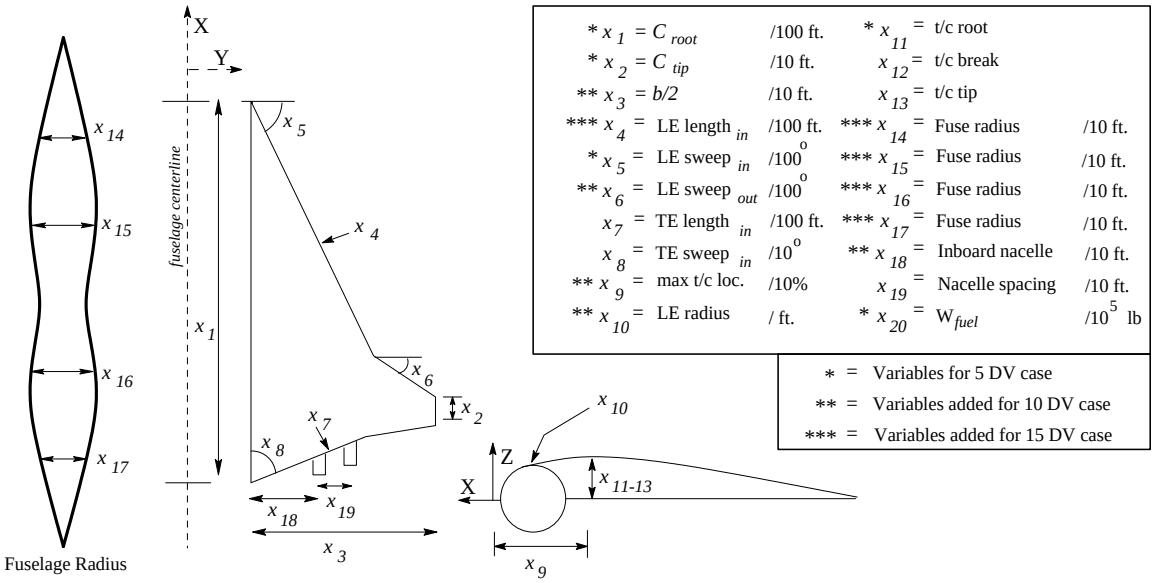


Figure13. Definitions of design variables.

The HSCT code optimization can work with subsets of 5, 10, 15, or 20 of the design variables as defined in Figure 13. The variables designated with one asterisk are used in the 5 DV case. The 10, 15 and 20 DV cases add the variables designated with two, three and four asterisks respectively.

These design cases were used to compare the performance of the optimizers. The five DV cases have only one optimum, while the higher dimensional designs contain many local optima and nonconvex feasible regions. This allowed us to compare the optimizers on problems of different complexities.

For each design case, the HSCT code was run for 100 starting points for DOT and LFOPCV3, and once for DIRECT. Table 4 shows the results from these runs. The 90% confidence column here refers to the number of function evaluations needed to find a design within 1000 lb of the best optimum design out of all 3 optimizers. In order to ensure that DIRECT will locate the same optimum with slightly different starting conditions, it was run a second time for each case with the design box perturbed by 1%. In each case, DIRECT located an optimum that was within 300 lb of the original optimum. For a small number of design variables DOT has a large advantage over DIRECT, but this advantage diminishes with dimension, and for the 20 design variable case, DIRECT is slightly better. LFOPCV3 did very poorly, and was not able to find the global optimum except for the five variable case. LFOPCV3 was slowed down by the need to continuously calculate gradients by finite differences while DOT was able to construct an approximation to the problem which reduced the number of gradient calculations it required.

Table 4. Comparison of 5, 10, 15, and 20 DV HSCT problem.

5 DV	Best GTOW Located	Function evaluations	# of Opt*	90% Confidence
DIRECT	638238	2345	1	2345
DOT	638231	10870	94	89
LFOPCV3	638813	33234	5	14919
10 DV				
DIRECT	624751	9067	1	9067
DOT	624731	29791	12	5366
LFOPCV3	631300	137929	0	-
15 DV				
DIRECT	603127	40662	1	40662
DOT	602685	47440	12	8545
LFOPCV3	620538	1437312	0	-
20 DV				
DIRECT	588404	51147	1	51147
DOT	588586	57428	2	65453
LFOPCV3	620092	418315	0	-

* 1 run for DIRECT, 100 runs each for DOT and LFOPCV3

For the 5 DV case, each optimizer found approximately the same global optimum. This was expected due to earlier experiments that showed the 5 DV constraint set was convex with only one local optimum. The difference in the design variables is less than 1%, which can be attributed to noise in the objective function causing premature convergence for LFOPCV3. DIRECT uses DOT for the final convergence, which should cause both to reach the same design when optimizing in the same local region. Here DOT was the most efficient optimizer by far since it did not require more than a few starting points to locate the optimum.

For the 10 DV case, DIRECT and DOT located a design of the same weight, with DOT being more efficient by a factor of two. The 10 DV case has distinct local optima, (Knill et al. 1999), and LFOPCV3 could not locate the global optimum.

For the 15 DV case DOT found a slightly lower weight than DIRECT and was also much more efficient. However, this difference is less than 0.08% of the total weight

of the HSCT. This case illustrates the unpredictable performance of a multistart method. Out of the initial 100 random starting points, DOT was only able to locate the global optimum once but it found seven designs with weights as good as DIRECT's result. To evaluate the likelihood of finding the optimum with DOT, another 100 random starting points were chosen for DOT. DOT was unable to locate the best optimum for this set of starting points. Examination of the design space near the best optimum indicates that it lies in a small feasible region, which explains the difficulty locating it. The designs found by DOT and DIRECT are shown in Table 5. They differ only slightly in most of the design variables. This indicates that the difference between the two is primarily due to noise preventing the optimizer from moving through a narrow region of similar weights to find the best optimum.

Table 5. Fifteen DV optima for DOT and DIRECT.[1]

Design Variables	DOT	DIRECT	% difference
Root chord	1.7306	1.7116	4.2
Tip chord	0.8425	0.8563	2.3
Semispan	6.9164	6.9336	1.1
LE length in.	1.2052	1.2074	1.0
LE sweep in.	0.7128	0.7121	1.2
LE sweep out.	0.1205	0.1300	4.8
Max t/c loc.	4.8400	4.8502	0.6
LE radius	3.0660	2.4353	33.2
t/c ratio	1.9872	2.0304	5.4
Fuse. Radius 1	0.5176	0.5144	2.1
Fuse. Radius 2	0.5669	0.5699	2.0
Fuse. Radius 3	0.5659	0.5699	2.7
Fuse. Radius 4	0.4966	0.4928	2.5
Inboard nacelle	2.8293	2.8519	0.9
Fuel wt.	3.0938	3.1005	1.1

[1] The design variables in Table 5 are defined in Figure 11.

For the 20 DV case, DIRECT performed slightly better than DOT. The weights they found differed by only 200 lb, but the design variables differed by as much as 35% of their respective ranges. This case showed that DIRECT is still able to perform well for a 20 dimensional design space. The number of function evaluations increased rapidly for more than 10 dimensions, but it was still less expensive than the two multistart methods for the 20 DV case. Without prior knowledge of the design space, it is difficult to decide on the number of starting points to choose for the multistart methods.

In an effort to determine the number of local optima scattered throughout the design space, the optimum designs from the 10 DV DOT optimizations were compared. Figure 10 shows the distribution of the design variables for the 25 best optimizations found by DOT over the range of each design variable. The range of the GTOW was only 16,000 lb. but some of the variables differ by their entire range. This plot indicates that there are some variables, such as the root chord, t/c ratio, the inboard nacelle location, fuel weight, and the outboard sweep angle, which tend to a small range of values for the optimized design. The other variables fluctuate widely without affecting the GTOW much. This indicates that the design space contains narrow channels with slowly varying weights. DOT easily locates these regions with similar weights but cannot always move through them without becoming trapped by minor fluctuations in the objective function.

The HSCT problem has both physical local optima and numerical noise. The good performance of DIRECT and DOT for this problem is consistent with the fact that DOT handled well low amplitude noise for the Griewank test function, and DIRECT handled well widely separated local optima for the quartic test function. As the dimensionality of the problem increases, the number of substantially separated optima can increase

exponentially, and the effectiveness of multistart methods appears to deteriorate faster than that of DIRECT. The poor performance of LFOPCV3 compared to the Griewank problem may have to do with the fact that the noise for the HSCT problem is not differentiable.

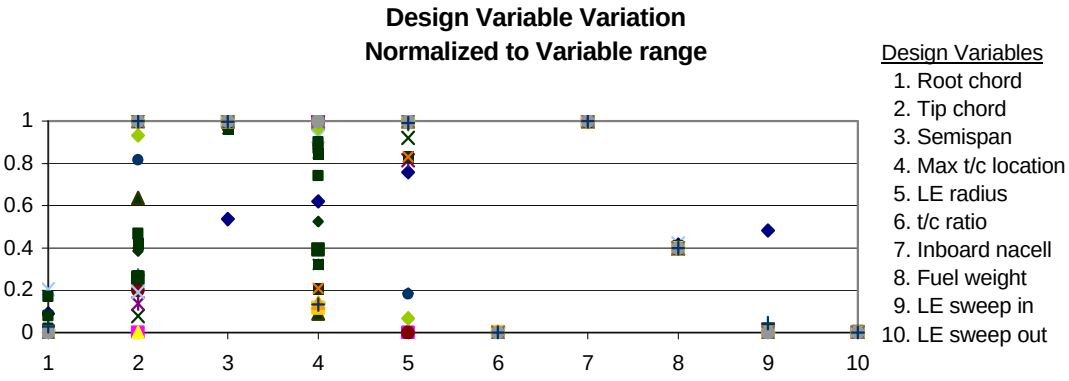


Figure 14. Distribution of Design Variables for the best 25 designs found by DOT for the 10 DV HSCT Case.

Earlier work on optimizing the HSCT found that the aerodynamic and structural evaluations were noisy which hindered local optimization algorithms (Baker et al. 1998 and Balabanov et al. 1998). Response surfaces were used to reduce the noise and improve gradient calculations for the local optimizers. However, there still exist some sources of noise in the constraints and objective function which could account for the poor performance of LFOPCV3 for the HSCT problem. The use of additional response surfaces to smooth the analysis may improve the performance of LFOPCV3 for this case.

Comparison Concluding Remarks

Three optimization procedures were tested for the global optimization of a HSCT. DOT-- a local optimizer used with sequential quadratic programming--and LFOPCV3--a

semi-global optimizer--were applied with random multistarts. DIRECT--a global Lipschitzian optimizer--was applied in a single run.

The three optimization procedures were first tested on two simple algebraic problems. The Griewank function is a convex function superimposed with low amplitude noise from a trigonometric function. The quartic test function does not have any noise but displays a large number of widely separated local optima. These two functions revealed that the two multistart procedures, DOT and LFOPCV3, dealt well with trigonometric noise, while DIRECT was much more effective in finding the global minimum among widely separated local optima.

The HSCT problem presents a combination of numerical noise and multiple physical optima, and thus has elements from both test functions. For a small number of design variables DOT had a large advantage over DIRECT, but the advantage decreased with increasing dimensionality, with the two having similar performance for the 20 variable case. LFOPCV3 did poorly for the HSCT problem, possibly because of the effect of the noise on the derivatives that have to be calculated by its search procedure more frequently than by DOT.

CHAPTER 3
DIRECT-BP

Based on the results from the optimization comparison, DIRECT was selected as a promising global optimization method for high dimensional designs. However, while DIRECT does have a guarantee of global convergence to the global optimum, this result does not necessarily imply its practical utility in most applications. DIRECT searches globally for promising regions of the design space while locally refining the solution about the best point located so far. Because of this, the behavior of DIRECT about the best points it finds quickly is of more practical importance than its performance in the limit as t? ∞.

Other than the global guarantee of convergence, there is no guarantee that DIRECT will converge towards a locally optimal point at any intermediate time. By dividing the design space into separate regions or boxes, DIRECT may have difficulty extending its local search past the bounds of the current best box. Thus DIRECT only moves its local search between boxes when the global search identifies a better box. This can lead to DIRECT dividing boxes arbitrarily small on the border of a much larger box and indicating that no improvement can be made about that point when the local slope is clearly not zero.

The incorporation of a local optimizer into DIRECT, either at the end of the optimization as in the previous chapter or periodically during the DIRECT optimization (Cox et al. 2001 and Nelson II 1998), can partially solved this problem. This has worked well for smooth functions with widely separated local optima where a local optimizer

can quickly move to the only optimum in the immediate vicinity. However, for noisy functions with large numbers of weak local minima in the region of a true minimum, a local optimizer might easily become trapped and fail to locate the best point. It would be preferable in this case to modify the DIRECT search to allow it to refine boxes in the neighborhood of the best point without being confined by the bounds of the current best box. The modified version of DIRECT, DIRECT-BP algorithm for *B*ox *P*enetration was designed to address this issue.

Original DIRECT Algorithm

Convergence Behavior

The DIRECT algorithm is a space partitioning heuristic for selecting the order of the boxes to divide, so as to bias the divisions towards boxes with known good function values while not ignoring sparsely sampled areas. Dividing the boxes in any order will guarantee that you locate a point within λ of the global optimum by the time the design space has been completely sampled on a mesh of λ spacing. The important distinguishing feature of different heuristics is how likely the method is to locate such a point when far fewer points have been analyzed.

In this measure, DIRECT has proven to be a very efficient method. The global search using a high value of K is what gives DIRECT an asymptotic guarantee of convergence to the global optimum. However, this search is too expensive as a practical matter to depend on for locating a good value. In practice, DIRECT is run long enough to give the global portion of the search a fair chance of locating a basin which contains a very good optimum, which the local portion of DIRECT searches while the global part continues to look for a better basin. In this way, DIRECT only has to sample on a fine

mesh near each good local optima and can ignore most of the design space where a sparse sampling indicates poor function values.

One problem with this method is that the local portion of the search is confined to the potentially optimal boxes which contain the best points located. As these boxes get small, the local search is unable to make appreciable progress within that box towards the optimum. It may be useful to enable the local search to move past the bounds of the best box if there is a large unexplored region nearby.

Let X*(t) be the box containing the best function value at iteration *t*, and c*(t) be the center point of box X*(t). Consider a situation such as Figures 15 and 16 where X*(t) is next to a much larger box with a poor function value at the center. When the local optimum near c*(t) is outside of X*(t), DIRECT is unable to move to the local optimum using the local portion of its search. The local portion of the DIRECT search will continue to divide X*(t) and sample points which asymptotically approach the box boundary near the optimum. That is, the progress of the local portion of DIRECT is limited by the boundary of X*(t). The local portion of the DIRECT search will stall, even if larger improvements are possible in the neighborhood of X*(t).

Asymptotically, DIRECT deals with this situation through its global search. By continuing to divide the large boxes in the design space, the algorithm will eventually divide box 2 in Figure 15. This would allow DIRECT to shift the local search to the box that contains the optimum and eventually reach the correct solution. However, Figure 16 suggests that almost all of the boxes that are larger than box 2 will have to be divided before box 2 will be located on the convex hull and divided. Thus DIRECT may terminate before box 2 is identified and its sub-boxes are divided enough to locate a point

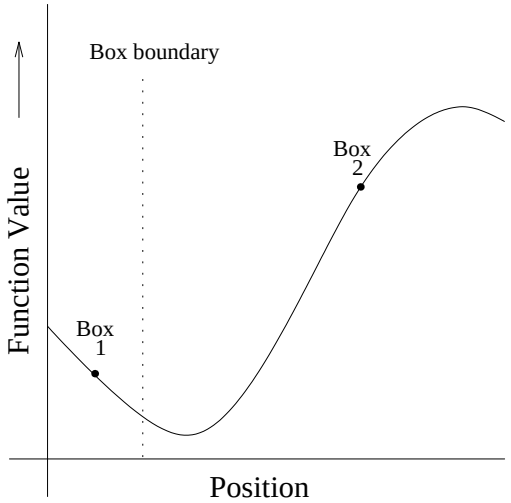


Figure 15. Instance where the current best box does not contain the local optimum

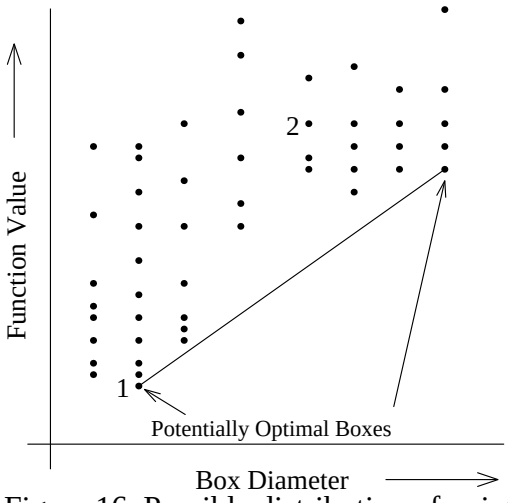


Figure 16. Possible distribution of points for potentially optimal box selection.

that is better than any other located. This problem is aggravated in higher dimensional problems. The number of boxes will increase rapidly with the number of dimensions, and each box will generate more new boxes when it is divided. This results in more large boxes which may need to be divided before the one containing the global optimum is identified. Previous work has attempted to deal with problems like this by stopping the optimization early and performing additional DIRECT optimizations with a smaller design space centered at the current best point (Bartholomew-Biggs et al. 2003). However, this runs the risk of excluding regions of the design space which have unidentified good local optima.

Demonstration of Local Convergence Difficulty

The following example shows how DIRECT can stall in its progress towards a global minimum. Equation 15 defines f_1, a combined quadratic and sinusoidal function.

$$f_1(x) = \sum_{i=1}^{n} \left\{ \frac{(x_i + 1)}{1.7} \sin\left((x_1 - 0.1)1.5\pi\right) + \left(\frac{(x_i - 0.4)}{1.2} \right)^2 + 0.05\cos(77x_i) + \frac{ix_i}{99} \right\} \quad (15)$$

The last two terms are used to add `noise' and make the contribution from each dimension different. The one dimensional graph of this function is given in Figure 17.

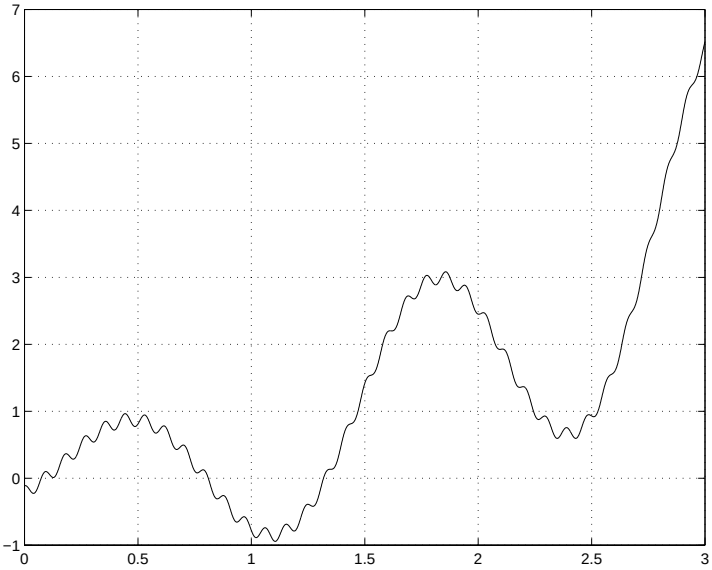


Figure 17. A one dimensional plot of $f_1(x)$ (Eq. 15) in the range x = [0,3].

If the design space is $0 \leq x_i \leq 2.5$, for i=1…,n, then DIRECT has no difficulty locating the global optimum at $\mathbf{x} = \mathbf{1.1}$. For a ten dimensional case, DIRECT locates the optimum in about 100 iterations and 1500 function evaluations. However, setting the upper bound to 3 causes the search to move to the wrong portion of the design space initially. For the same ten dimensional case, DIRECT can be run for almost 19,000 iterations and over 60,000 function evaluations without locating the global optimum.

Figures 18 and 19 show the progress of DIRECT towards the global optimum for the ten dimensional case of Equation 15 and shows how the progress of the best design stalls as it approaches $\mathbf{x} = \mathbf{1}$. The convergence of DIRECT to a local optimum depends on locating the box that contains the optimum and then refining the solution within that box. For this problem, the function value at the center of the box that contains the global optimum is higher than the function value at any other box of that size or larger. This

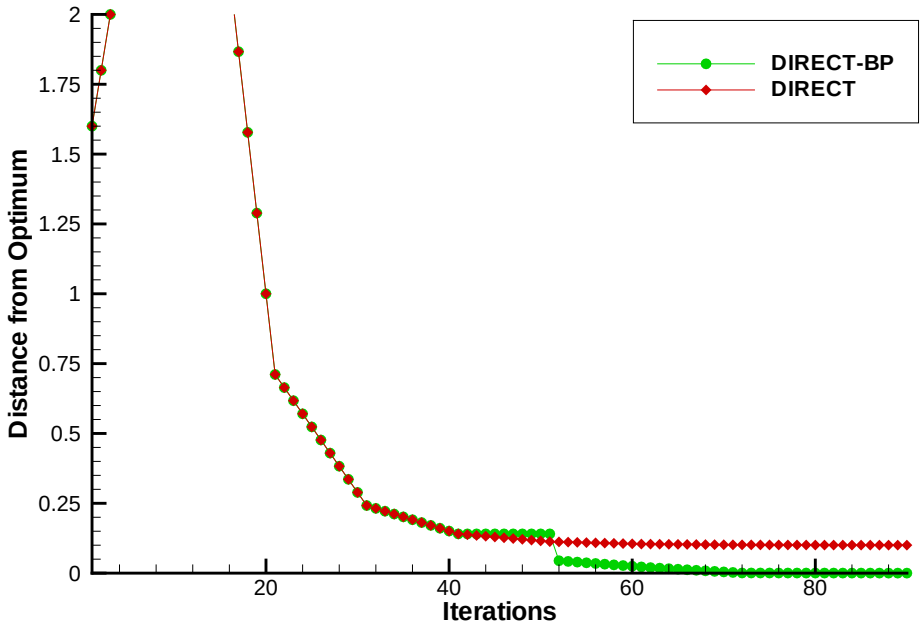


Figure 18. Distance between c*(t) and the global optimum for equation 15 in ten dimensions.

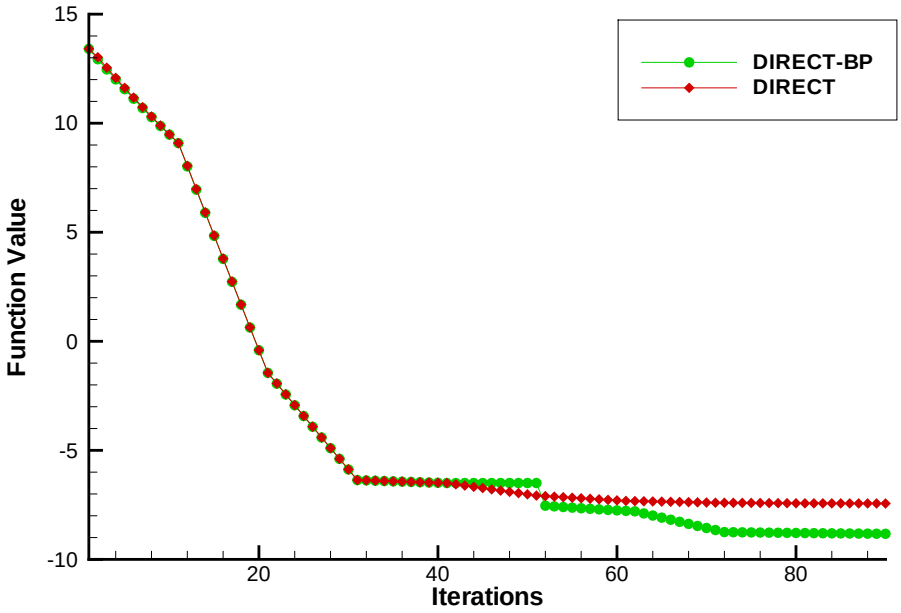


Figure 19. Value of f(c*(t)) for f_1 (Eq. 15) in ten dimensions as the optimizations progress.

means that every other box with side lengths of ⅓ or larger will be divided before the one containing the global optimum. For the 10 dimensional case there are 59,049 separate boxes this size that would have to be divided. For most practical implementations of DIRECT, the optimizer would be stopped before the middle box is divided for problems of as few as 5 or 6 dimensions. This prevents DIRECT from identifying the box that contains the global optimum as potentially optimal before it is stopped.

The DIRECT-BP Algorithm

The DIRECT-BP algorithm was proposed as a way to help with this shortcoming of the DIRECT algorithm. The DIRECT-BP algorithm forces the neighboring boxes of the best box to be divided at a similar rate as the best box. This will allow DIRECT-BP to extend its search for a local optimum beyond the bounds of the best box. This local search is inspired by generalized pattern search theory (Dennis and Torczon 1991, and Torczon 1997). By forcing the neighboring boxes to be divided as the best box is shrunk, DIRECT-BP will be able to detect better regions near the best box that were masked by large neighbors with poor function values at the center. This will provide a more thorough search of the design space surrounding the best point located and add a measure of confidence that DIRECT-BP has found the best point in the immediate vicinity in exchange for a moderate increase in the cost of the opotimization.

The DIRECT-BP algorithm differs from the original DIRECT algorithm by reprioritizing whether boxes are divided. Specifically, DIRECT-BP may (a) prevent the best potentially optimal box from being divided and (b) add additional boxes in each iteration to the list of boxes that are divided. This reprioritization of the boxes in DIRECT-BP ensures that the neighbors of the box containing the best point are refined and divided. This prevents the boxes surrounding the best point from becoming too large

relative to the size of the best box being divided. This also allows DIRECT-BP to balance its global search with effective local search about the best point without having the local search confined to a single box.

Let $\sigma_j(t)$ be the smallest box edge length of the j^{th} box at iteration *t*, $X_j(t)$, and let $\Delta^*t = \sigma^*_j(t)$, where $X^*(t) = X_j^*(t)$. Further, let $\mathbf{v}_j(t)$ be the vectors from the center of the best box, $c^*(t)$, to the center of box $X_j(t)$, $\rho_j(t) = |\mathbf{v}_j(t)|$ and $\lambda_j(t) = \rho_j(t)/\Delta^*t$. We say that $X_j(t)$ is a *neighboring box* of $X^*(t)$ if the box touches at least one point of $X^*(t)$. A *face neighbor* is any neighboring box that shares an $n-1$ dimensional face with the best box.

We say that there is a balanced neighborhood about the best box if (some) neighboring boxes have centers at similar distances $|\mathbf{v}_j|$ on all sides of the best box. The balance in distances is controlled by a parameter Λ which defines the largest ratio of distances to the size of the smallest box (λ_i) that still allows $|\mathbf{v}_j|$ to be considered a "similar" distance to the best box. The formal definition of "on all sides" comes through the concept of a positive spanning set defined below.

A set of vectors *V* is a *positive spanning set* for $A \subseteq \Re^n$ if each point in *A* is a nonnegative linear combination of finitely many elements of *V*. Consider

$$V_t = \{ \mathbf{v}_j(t) \mid X_j(t) \text{ is a neighbor of } X^*(t) \}. \qquad (16)$$

Let $T_t \subseteq \{\pm \mathbf{e}_1, \ldots, \pm \mathbf{e}_n\}$, such that the vectors in T_t indicate the faces of $X^*(t)$ *not* on the boundary of the design space. Figure 20 illustrates the values of T_t for boxes at different positions in the feasible domain. We say that V_t is a *balanced neighborhood* of $X^*(t)$ if $V'_t \subseteq V_t$ is a positive spanning set for T_t and

$$\max_{v \in V_t'} |v| \le \Lambda \Delta_t^* \qquad (17)$$

where $\Lambda \geq 1$ is a user specified constant which defines the maximum allowed ratio of $\rho_j(t)/\Delta^*_t$ for the boxes used to form a positive spanning set in the neighborhood of $X^*(t)$.

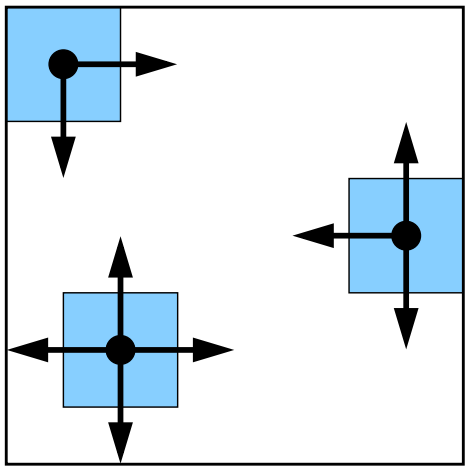


Figure 20. An illustration of the sets of vectors Tt used for X*(t) at various positions in the feasible domain.

The DIRECT-BP algorithm is described below, with modifications to the original DIRECT algorithm on page 33 in boldface.

1. Normalize the search space to the unit box. Evaluate $f(c_1)$, and set $B_t = \{X_1\}$
2. For t=1,…
    a. Identify the set $S_t \subseteq B_t$ of potentially optimal boxes.
    b. If t > 1, examine the neighborhood of X*(t) and modify S_t.
        i. Form $V_t = \{v_j(t)\}$, from the neighbors of X*(t), and let $V'_t = \{ v_j(t) \in V_t \mid \lambda_j(t) \leq \Lambda \}$
        ii. If V'_t is not a positive spanning set for T_t, then
            1. remove X*(t) from S_t if X*(t) is a cube;
            2. add the face neighbors $X_j(t)$ of X*(t) to S_t, where the face neighbor has the property that $\sigma_j(t) \geq \Delta^*_t$;
            3. **optionally, add additional neighbors of X*(t) into S_t.**
    c. For each box $X_j(t) \in S_t$ where $\text{diam}(X_j(t)) > \gamma$:

i. Identify *I*, the dimensions of $X_j(t)$ with the longest side length. Let δ equal ⅓ of this length.

ii. Sample the function at the points $(c_j \pm \delta e_i)$ for all $i \in I$.

iii. While *I* is not empty:

1. Divide $X_j(t)$ into thirds along the dimension $i \in I$ with the lowest value of $f(c_j \pm \delta e_i)$ to create two new boxes and reduce the size of box $X_j(t)$.
2. Remove i from *I* and add the new boxes to B_{t+1}.

d. Terminate if an appropriate stopping rule is satisfied.

In Step 2.b, DIRECT-BP ensures that the best box is only divided if (a) it is not a cube or (b) it is a cube and the neighborhood is balanced. This allows DIRECT's local search to progress locally beyond the bounds of the current best box without relying on the global portion of the DIRECT search. The description above provides the basic requirements of this step, but we defer the details of our implementation until the next section. For example, the method for selecting boxes to add to S_t in Step 2.b can significantly impact the efficiency of the search in DIRECT-BP. The most important consideration for efficiency is to rapidly form a balanced neighborhood with the fewest number of extra function evaluations. Thus, it is clear that an effective design for DIRECT-BP will probably not include all of the neighbors that define V_t - V'_t in S_t. However, including only one box per iteration is probably not sufficient to ensure rapid local search about $X^*(t)$.

Figure 21 illustrates that it is possible for V_t to *not* form a positive spanning set for T_t. In this case, we need to select a neighboring box to divide in order to improve the

span of V'_t. Regardless how the neighboring boxes are selected, V'_t will form a positive spanning set for T_t within a finite number of iterations.

When $X^*(t)$ is on a boundary of the domain, then the requirement that V'_t positively spans T_t to form a balanced neighborhood has the following effect. Consider some $d \in T_t$ that is parallel to a domain boundary. V'_t will positively span the direction d if there exists some face neighbor $X_j(t)$ for which c_j is located along d and $\lambda_j(t) \leq \Lambda$. Further, if V'_t does not positively span d, the face neighbor indicated by d will be divided. This requires only finitely many iterations of DIRECT-BP, since it suffices to divide this face neighbor until it is the same size as the best box. This puts the center of the face neighbor at $c^*(t) + e_i \Delta^*(t)$, e_i has the same direction as d.

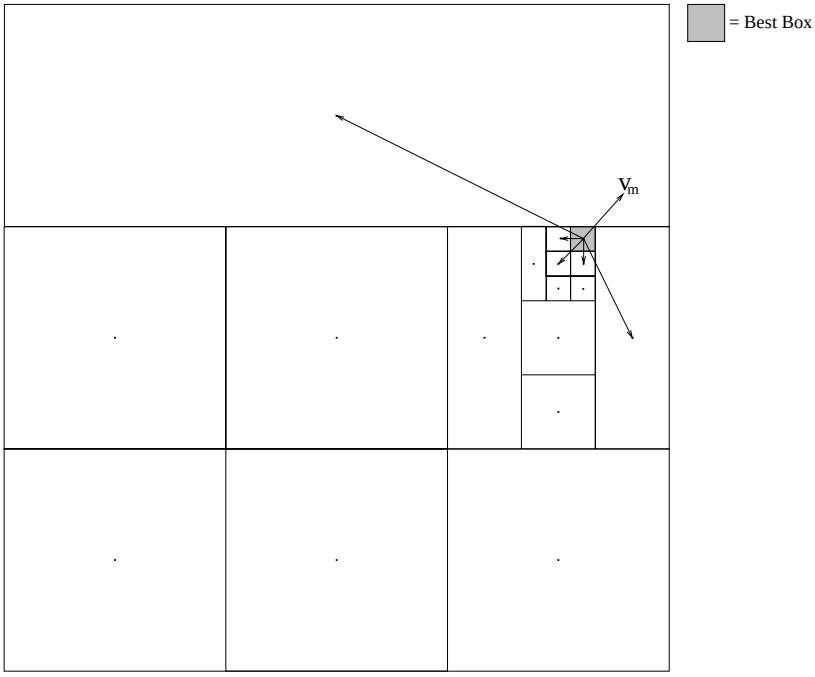


Figure 21. Two-dimensional example of vectors failing to form a positive spanning set. The vectors from the center of the best box to the centers of the neighboring boxes do not form a positive spanning set for $T_t = \{\pm e_1, \pm e_2\}$. v_m points towards the center of the design space not positively spanned by V'_t.

Finally, note that these changes to DIRECT will not affect the search far from the best point. Instead, the more balanced search about the best point adds an element of robustness to DIRECT by widening the search in the area likely to contain the global optimum. Consequently, DIRECT-BP effectively converges both locally and globally.

Implementation

The performance of DIRECT-BP may be quite dependent on the manner in which it is implemented. In particular, the number of points to be analyzed in each iteration will be significantly influenced by the method used by DIRECT-BP to select the neighbors of X*(t) to divide. Further, V_t can become quite large, so the method of identifying V'_t that forms a positive spanning set for T_t may impact the overhead of the program. The code used here attemptes to remain as close as possible to the original version of DIRECT to avoid hurting the performance on problems where DIRECT already performs well.

DIRECT-BP requires that V'_t forms a positive spanning set for T_t before a cubic X*(t) can be divided. Algorithmically, we break this into two separate subproblems. First, we identify a positive spanning subset of V_t for T_t. If such a subset does not exist, we select neighboring boxes to divide to ensure that such a subset exists in a subsequent iteration. Otherwise, we confirm that for all vectors v in the subset, $|v|/\Delta^*_t \leq \Lambda$. If this is not true, then we select neighboring boxes to divide to reduce the lengths of the long vectors in this subset. The details of this implementation are outlined below with a more detailed discussion following.

1. Examine the neighborhood of X*(t) and modify S_t.
    a. Form $V_t = \{\mathbf{v}_j(t)\}$, from the neighbors of X*(t).

b. Find the set Ω of neighboring boxes with shortest vectors $\mathbf{v}_j(t)$ that form a positive spanning set for T_t, if it exists. Include all of the face neighbors of X*(t) in Ω if V'_t is not a positive spanning set.

c. If Ω does not exist,

   i. Add to the set S_t one or more of the neighboring boxes corresponding to the vectors in the set V_t that are considered likely to increase the balance of V_t when divided, if they are not already there,

   ii. Remove the best box X*(t) from S_t if X*(t) is cubic.

d. Else if $\lambda_j \geq \Lambda$ for any $X_j(t) \in \Omega$ or diam(X*(t)) is smaller than γ, then

   i. Add the boxes corresponding to all of the $\mathbf{v}_j(t)$ where $\lambda_j \geq \Lambda$,

   ii. If diam(X*(t)) < γ, then add the boxes $X_j(t) \in \Omega$ where $\rho_j(t) > c\gamma$ to S_t where $\Lambda > c > 1$,

   iii. Remove the best box from S_t if X*(t) is cubic.

Identifying a Positive Spanning Set

An important subroutine of DIRECT-BP performes the test of whether a set of vectors forms a positive spanning set for T_t. The following theorem provides a general mechanism for performing this test efficiently.

Theorem 1. *Let $V = \{\mathbf{v}_1,\ldots,\mathbf{v}_k\}$ span $\Re^n$, then if $\exists \alpha_i \geq 0$ such that* $-\sum_{i=1}^{k} v_i = \sum_{i=1}^{k} \alpha_i v_i$ *then V forms a positive spanning set for $\Re^n$.*

Proof Let $\mathbf{w} = -\sum_{j=1}^{k} v_i$. Since V spans $\Re^n$, it follows that $V \cup -V$ forms a

positive spanning set for $\Re^n$. Also $-\mathbf{v}_i = \sum_{j=1, j\neq i}^{k} v_j + w \ \forall i$. Therefore, $V \cup \mathbf{w}$ positively spans $V \cup -V$. Now suppose that $\exists\ \alpha_i \geq 0$ s.t. $\mathbf{w} = \Sigma\alpha_i \mathbf{v}_i$. Then V positively spans $V \cup \mathbf{w}$ and in turn forms a positive spanning set for $\Re^n$.

Based on previous experiments, most practical implementations of DIRECT-BP will use $\Lambda > \sqrt{n}$. Using such a value of Λ, the vectors $\mathbf{v}_i(t)$ shorter than $\Lambda\Delta^*_t$ are guaranteed to span $\Re^n$ because of the way that DIRECT generates boxes. In this case, the check for a positive spanning set for interior $X^*(t)$ only requires solving a single set of constrained linear equations to calculate the α_i from Theorem 1. Thus, as each box's $\mathbf{v}_j(t)$ is added to V'_t to help form a positive spanning set, only solving a single set of constrained linear equations is needed to determine if a positive spanning set for T_t has been generated. If $X^*(t)$ is on the boundary, either a linear solve is required for each vector in T_t, or an intricate scheme to ensure that all directions *d* parallel to the boundary are positively spanned is used, in which case Theorem 1 can be used by artificially adding to V_t all the outer normals to $X^*(t)$ on the boundary. In this case, the requirement that the face neighbors of $X^*(t)$ on the boundary of the design space be divided to the same size as $X^*(t)$ is explicitly included to ensure that the added vectors do not artificially allow V_t to span the directions *d* parallel to the boundary.

In the implementation of DIRECT-BP used in this research, instead of solving a set of linear equations to determine if a positive spanning set has been formed, an iterative heuristic was used to solve for $\mathbf{v}_m$ by maximizing the minimum angle between v_m and all $v_i(t)$ (Figure 21). Starting with $\mathbf{v}_m$ equal to the negative average of $\mathbf{v}_i(t)$, $\mathbf{v}_m$ is moved away from the closest $\mathbf{v}_i(t)$ until the smallest angle with the closest $\mathbf{v}_i(t)$ can not be increased. If V_t does not form a positive spanning set, then $\mathbf{v}_m$ is guaranteed to be located

in the space not covered by V_t and the angles between $\mathbf{v}_m$ and all of the $\mathbf{v}_i(t)$ will be greater than 90^o. If V_t does form a positive spanning set, then the maximum smallest angle will be less than 90^o. This method was used because of a lack of suitable code for solving a constrained set of linear equations when the code was first written and because it generates and approximation to $\mathbf{v}_m$ which is used in the next section.

Maintaining a Positive Spanning Set

Whenever the set V_t does not form a positive spanning set for T_t, some of the neighboring boxes and X*(t) (only if it is not square) must be divided until a balanced neighborhood is formed. There is a set $\underline{N}$ of neighbors that intersect the space not covered by the positive cone of the vectors in V_t. Dividing some of the boxes associated with $\underline{N}$ and perhaps X*(t) itself will eventually generate a positive spanning set for T_t about the best box. If all of the face neighbors in $\underline{N}$ are divided until they are the same size or smaller than the best box, then all of the coordinate directions in T_t not positively spanned by the set V_t will be added to V_t and it will be guaranteed to form a positive spanning set (see below). If some neighbors which are in $\underline{N}$ but are not face neighbors are also divided, the number of iterations needed to form a positive spanning set may decrease (though at the cost of additional boxes being divided at each iteration).

In this implementation of the DIRECT-BP algorithm, when no positive spanning set for T_t is formed from V_t we select other neighboring boxes to divide by locating the vector v_m which maximizes the minimum angle $\min_i \cos^{-1} \frac{v_m \bullet v_i(t)}{\|v_m\| \|v_i(t)\|}$ (Figure 21). In addition to the face neighbors of X*(t), the boxes, $X_i(t)$, corresponding to the $\mathbf{v}_i(t)$ which form the smallest angles with $\mathbf{v}_m$ are added to the set of boxes to be divided. These boxes make up a subset of $\underline{N}$ and potentially have a large portion of their space not spanned by

V_t. The use of all of the face neighbors is guaranteed to eventually form a positive spanning set and experiments have suggested that dividing the other boxes which were added can help generate a positive spanning set in fewer iterations.

Limiting Λ

DIRECT-BP requires that some subset of neighboring points must define a positive spanning set and the ratio $\Lambda = \rho_{max}(t)/\Delta^*_t$ must be bounded above as $\Delta^*_t \rightarrow 0$ where $\rho_{\max}(t) = \max_i \rho_i(t)$. If the neighboring boxes do form a positive spanning set but some $\lambda_i \geq \Lambda$, then the best box is not divided if it is cubic and instead the neighbors needed for a positive spanning set where $\rho_i(t) \geq \Lambda\Delta^*_t$ are divided.

In order to eliminate the need to divide every neighboring box where $\rho_i(t) \geq \Lambda\Delta^*_t$, it is necessary to select the smallest set of $\rho_i(t)$ that are larger than $\Lambda\Delta^*_t$ and that are needed to form a positive spanning set. In order to select a near minimal set of $\rho_i(t)$ without adding excessive overhead, DIRECT-BP starts with all of the boxes where $\rho_i(t)$ is less than a designated size. If the best box is larger than γ, DIRECT-BP starts with all of the boxes where $\rho_i(t) \leq \Lambda\Delta^*_t$. If the initial set of vectors does not form a positive spanning set, then DIRECT-BP adds the box corresponding to the vector $\mathbf{v}_i(t)$ closest to the direction $\mathbf{v}_m$ to this set of boxes (Figure 21). It continues to add boxes in this way until a positive spanning set is formed or all of the neighbors have been added. If vectors were added to form the positive spanning set, then the best box is not divided at that iteration if it is a cube and the neighboring boxes used to form the positive spanning set where $\rho_i(t)$ is larger than $\Lambda\Delta^*_t$ are divided.

If the best box is too small to be divided any further, the optimizer reduces the maximum allowable length of the vectors used to form a positive spanning set. It

replaces Λ by c and progresses as in the previous paragraph. This parameter, c, is used to force the optimizer to further refine the neighborhood of the best point at the end of the optimization. For this research the value of c was set to 10 while Λ ranged from 15 to 35. The value of c is linked to one possible stopping criterion for DIRECT-BP based on the longest vector needed to form a positive spanning set. The fourth stopping rule on page 73 stops the optimizer when the longest vector needed to form a positive spanning set is shorter than 10 times the minimum divisible box size. Without changing Λ to c, the neighborhood would not be refined sufficiently for this stopping rule.

The choice of Λ can greatly affect the performance of DIRECT-BP. Choosing a value that is too small can increase the number of neighbors that are divided and degrade the efficiency of the search. Choosing a number that is too large can cause the optimizer to fail to search the surrounding boxes until after the best point has been refined to a very small box. This can lead to excessive sampling about a suboptimal box before examining its neighbors or DIRECT-BP could reach its limit on function evaluations or iterations before dividing any of the neighbors.

Local Search

Figures 18 and 19 demonstrate the difference between the DIRECT-BP search and the DIRECT search for the 10 dimensional case of the example problem given in equation 15. DIRECT progresses rapidly towards the optimum design in the first 40 iterations but then the local portion of the optimization becomes stuck at $\mathbf{x} = \mathbf{1}$. DIRECT-BP makes the same progress as DIRECT in the first 40 iterations but, for about the next 10 iterations, it stops dividing the best box while it examines the neighboring boxes for a better point. At this point DIRECT continues to divide the best box progressively smaller

as it asymptotically approaches the corner of the best box. However, at about the 50th iteration DIRECT-BP locates a better point in one of the neighboring boxes. This creates the jump in the function values and position seen in Figures 18 and 19. A new box next to the previous best box contains the new best point which allows the local search to step past the box boundary into the neighboring box and progress to the global optimum.

The progress of the local portion of DIRECT-BP may be slower than DIRECT while it examines the surrounding boxes, but DIRECT-BP has the advantage of performing a more robust local search around X*(t). By forcing the neighbors of the best box to divide at the same rate as the best box, DIRECT-BP can move from box to box to follow trends in the objective function that may not have been apparent to DIRECT due to the placement of the analyzed points. The local search does not stall at the edge of the box bordering **x** = **1**. DIRECT-BP is able to step past this point and identify the actual optimum in the neighboring box.

Experimental Comparisons

Test problems

DIRECT-BP was compared with DIRECT on five test problems: the simple sinusoidal function, f_1, and Griewank's function given earlier and two engineering problems from Floudas et al. (1999). The *n*-dimensional Griewank function is defined again as

$$f_2(x) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{d} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right), \qquad (18)$$

where d is a constant selected to control the relative magnitude of the quadratic and the trigonometric portions. For the 10 dimensional problem, d = 1000, each element in x_U was randomly set in the range [100,400], and the elements of $\mathbf{x}_L$ were set to give a span

of 500 in each dimension. Each instance of $\mathbf{x}_U$ and $\mathbf{x}_L$ was run on both versions of DIRECT in order to compare their performance on the same problem. The one-dimensional Griewank function is depicted again in Figure 22.

The Griewank function is a quadratic function with noise in the form of a cosine function. Without the noise, the function is convex with one optimum. With the noise, the function adds thousands of local optima in the region where the derivative of the sum term has a magnitude ≤ 1. This function was used to test how well the optimizers could move through the noise to locate the actual optimum.

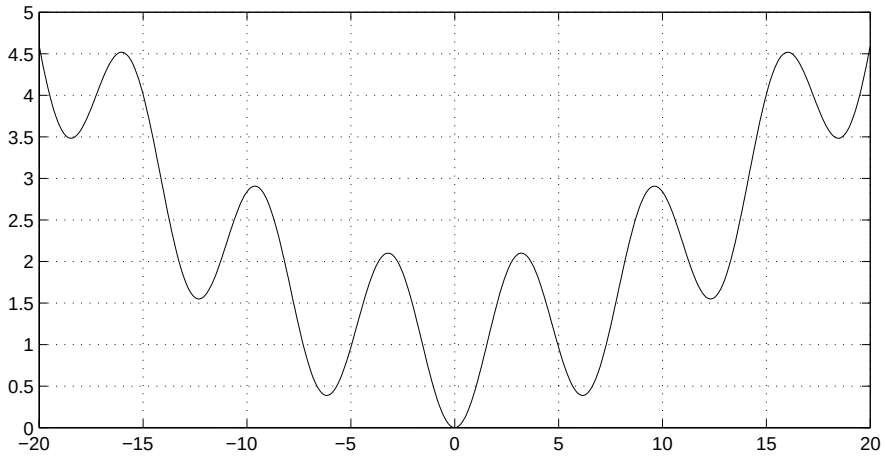


Figure 22. One dimensional Griewank function with d = 100.

Two test problems from Floudas et al. (1999) were also used as problems more representative of engineering applications. The first problem is given in equations 19.

$$\min \quad 5.3578x_3^2 + 0.8357x_1x_5 + 37.2392x_1 \tag{19}$$

$$\text{s.t.} \quad \begin{aligned} 0.00002584x_3x_5 - 0.00006663x_2x_5 - 0.0000734x_1x_4 &\leq 1 \\ 0.000853007x_2x_5 + 0.00009395x_1x_4 - 0.00033085x_3x_5 &\leq 1 \\ 1330.3294x_2^{-1}x_5^{-1} - 0.42x_1x_5^{-1} - 0.30586x_2^{-1}x_3^2x_5^{-1} &\leq 1 \\ 0.00024186x_2x_5 + 0.00010159x_1x_2 + 0.00007379x_3^2 &\leq 1 \\ 2275.1327x_3^{-1}x_5^{-1} - 0.2668x_xx_5^{-1} - 0.40584x_4x_5^{-1} &\leq 1 \\ 0.00029955x_3x_5 + 0.00007992x_1x_3 + 0.00012157x_3x_4 &\leq 1 \end{aligned}$$

The bounds of the design space are given in equations 20.

$$\begin{aligned} 78 &\le x_1 \le 102 \\ 33 &\le x_2 \le 45 \\ 27 &\le x_3 \le 45 \\ 27 &\le x_4 \le 45 \\ 27 &\le x_5 \le 45 \end{aligned} \tag{20}$$

The second engineering problem from Flodas et al. (1999) is the optimal design of a Continuous Stirred Tank Reactor (CSTR) used in chemical manufacturing. This problem has eight variables with four nonlinear inequality constraints and is given in equations 21.

$$\min \quad 0.4x_1^{0.67}x_7^{-0.67} + 0.4x_2^{0.67}x_8^{-0.67} + 10.0 - x_1 - x_2 \tag{21}$$

$$\text{s.t.} \quad \begin{aligned} 0.0588x_5x_7 + 0.1x_1 &\le 1 \\ 0.0588x_6x_8 + 0.1x_1 + 0.1x_2 &\le 1 \\ 4x_3x_5^{-1} + 2x_3^{-0.71}x_5^{-1} + 0.0588x_3^{-1.3}x_7 &\le 1 \\ 4x_4x_6^{-1} + 2x_4^{-0.71}x_6^{-1} + 0.0588x_4^{-1.3}x_8 &\le 1 \end{aligned}$$

The bounds of the design space are given in equation 24.

$$0.1 \le x_i \le 10, \qquad i = 1,...,8 \tag{22}$$

For both of these problems from Flodas et al. (1999), a penalty function was used to handle the constraints. A penalty multiplier of 100 was found to perform adequately.

Experimental methods

For this comparison, we assume that the local optimization is available for free and each optimizer will be considered to have located the global optimum if the best point it locates is within the basin containing the global optimum. Four different stopping conditions were used:

1. Stop when the diameter of the smallest box is = 0.0001.
2. Stop after a set number of iterations.

3. Stop after a set number of function evaluations.
4. (DIRECT-BP only) Stop when a balanced neighborhood is formed with $c\Delta^*_t = 0.001$.

The Griewank function has a random component, the bounds of the design space. Therefore, it was run 30 times using the same sets of values of $\mathbf{x}_U$ and $\mathbf{x}_L$ for each stopping condition with each optimizer and the results were averaged.

The DIRECT-BP algorithm has one additional parameter not used in DIRECT, namely Λ. Λ designates how long a vector from c*(t) to the center of a neighboring box can be before the neighbor must be divided and Δ^*_t is held constant. This parameter can have a large effect on the performance of DIRECT-BP so three values were used for each problem; 15, 25 and 35.

Test results

The first problem examined was the bound constrained problem in Equation 15. Table 6 gives the results of the two optimizers on this problem. The original version of DIRECT was unable to locate the global optimum but instead stopped at the point $\mathbf{x} = \mathbf{1.0}$ for all three stopping conditions while DIRECT-BP was able to locate the global optimum at $\mathbf{x} = \mathbf{1.1}$ in all but two cases. In the two instances where the DIRECT-BP was unable to locate the global optimum, it still performed better than the original DIRECT. However, in these cases the value of Λ was too large for DIRECT-BP to successfully refine its search before the stopping rule was applied. This problem shows how exploring the neighborhood of the best point makes the search more robust and will lead the optimizer to a good local minimum when the global search is insufficient.

Table 6. Comparison of DIRECT and DIRECT-BP on f_1 of Eq. 15, n = 10.

Method	Λ	Stop Cond	Func Eval	Iterations	Opt Found
Original DIRECT	NA	1	2613	92	No
	NA	2	15967	2000	No
	NA	3	59981	18913	No
DIRECT-BP	15	1	9121	136	Yes
	25	1	9019	133	Yes
	35	1	28919	258	Yes
	15	2	5525	100	Yes
	25	2	5479	100	Yes
	35	2	5799	100	No
	15	3	8173	132	Yes
	25	3	8711	132	Yes
	35	3	8515	133	No
	15	4	11789	148	Yes
	25	4	12069	146	Yes
	35	4	29777	264	Yes

Table 7 summarizes the performance of DIRECT and DIRECT-BP on the Griewank function. Both versions of DIRECT are able to get near the global optimum but locating the actual global minimum is more difficult. For both optimizers, and for any value of Λ, the global optimum was located nine out of the thirty optimizations. However, the best local optimum located in each optimization was different. The average value of the result returned by DIRECT was about 30% higher than that returned by DIRECT-BP. This was primarily due to one run where DIRECT selected a point 27.5 units away from the global optimum with a function value of 1.17 while DIRECT-BP found a better point 12.1 units away from the global optimum with a function value of 0.147. The Griewank function has all of the local optima clustered in one area of the design space around the global optimum. In this case, DIRECT naturally examines many of the boxes surrounding the best box, so the modifications for DIRECT-BP do not have as large of an effect. However, this case illustrates how the extra robustness of the search around the best box can make an impact on multimodal problems in general.

Table 7. Comparison of the average performance (30 runs) of DIRECT and DIRECT-BP on the ten-dimensional Griewank function[2]

Method	Λ	Stop Cond	Func Eval	Iterations	Ave Func Val
Original DIRECT	NA	1	5678	69	.142
	NA	2	5859	70	.143
	NA	3	10211	104	.142
DIRECT-BP	15	1	8017	71	.106
	25	1	7987	71	.106
	35	1	7431	69	.107
	15	2	7948	70	.107
	25	2	7941	70	.107
	35	2	7636	70	.107
	15	3	10235	87	.105
	25	3	10252	86	.106
	35	3	10232	88	.105
	15	4	9625	81	.105
	25	4	9263	78	.106
	35	4	8950	78	.105

The comparison of the first test problem from Floudas et al. (1999), Eq. 19, is given in Table 8. For this problem both optimizers located the global optimum of 10123 no matter which stopping condition was used. This is an example of a problem where DIRECT already searches the neighborhood of the best point without requiring any additional boxes to be divided. Here, DIRECT identifies the region containing the global optimum without any difficulty and divides the sequence of boxes containing the global optimum. The neighborhood of the best box is always balanced and there is no need to divide any of the neighboring boxes during the optimization. In a problem such as this where DIRECT is particularly efficient, the DIRECT-BP algorithm only increases the overhead of the optimization but not the number of analyses.

[2] For both optimizers and any value of Λ, the global optimum was located nine out of thirty runs.

Table 8. Comparison of the performance of DIRECT and DIRECT-BP on the first problem used from Floudas et al.(1999), Eq. 19.

Method	Λ	Stop Cond	Func Eval	Iterations	Func Val
Original DIRECT	NA	1	1033	47	10123
	NA	2	2859	100	10123
	NA	3	993	47	10123
DIRECT-BP	15	1	1033	47	10123
	25	1	1033	47	10123
	35	1	1033	47	10123
	15	2	2859	100	10123
	25	2	2859	100	10123
	35	2	2859	100	10123
	15	3	993	47	10123
	25	3	993	47	10123
	35	3	993	47	10123
	15	4	1035	48	10123
	25	4	1035	48	10123
	35	4	1035	48	10123

The comparison of the results of the optimization of the CSTR, Eq. 21, is given in Table 9. The total number of function evaluations was limited to 80000 for all cases.

Table 9. Comparison of the performance of DIRECT and DIRECT-BP on the design of a CSTR, Eq. 21.

Method	Λ	Stop Cond	Func Eval	Iterations	Func Val
Original DIRECT	NA	1	47357	97	3.9580
	NA	2	7087	100	3.9511
	NA	3	5999	88	3.9516
DIRECT-BP	15	1	80000	653	3.9525
	25	1	79999	705	3.9576
	35	1	61745	274	3.9526
	15	2	7263	100	3.9511
	25	2	7147	100	3.9511
	35	2	7139	100	3.9511
	15	3	5985	87	3.9524
	25	3	5999	88	3.9524
	35	3	5999	88	3.9519
	15	4	79999	705	3.9579
	25	4	79999	705	3.9579
	35	4	79997	731	3.9547

For this problem the biggest penalty for DIRECT-BP came when trying to use the first or fourth stopping criteria. Here, DIRECT-BP was very slow at dividing the smallest boxes and in four cases reached the alternate limit of 80000 function evaluations before shrinking the best box sufficiently. The performance of DIRECT and DIRECT-BP using the other stopping criteria however, showed that DIRECT-BP was as quick at getting close to the global optimum as DIRECT and did not divide many more boxes per iteration than DIRECT, as illustrated by the second stopping rule. Both versions of DIRECT came close to the global optimum at 3.9511 and were efficient at locating the global optimum for the second and third stopping rules.

These problems illustrate the improved robustness of the DIRECT-BP algorithm over the standard DIRECT algorithm. By examining the neighborhood of the best box at each iteration, the DIRECT-BP search can overcome one shortcoming of the DIRECT algorithm which can occasionally affect its performance on strongly multimodal problems. As the last three problems illustrate, this shortcoming of DIRECT does not affect the search on most problems but is instead a robustness issue. The modification is intended to improve the local search in the case where the global search is unable to move it to a neighboring box where a better function value can be found. The DIRECT-BP algorithm is intended to increase the confidence that the result is at least locally optimal while the global portion of the DIRECT search suggests that the optimum found is a relatively good local optima within the design space.

CHAPTER 4
MULTIFIDELITY DIRECT

Based on the results from the optimization comparison, DIRECT was selected as a promising global optimization method for high dimensional designs. In order to increase the applicability of DIRECT to problems requiring expensive analysis, a multifidelity version is desired. Although DIRECT is relatively efficient and robust, it still requires tens of thousands of evaluations to locate the global optimum for more than about 10 design variables for nonconvex functions. This would make it too expensive to use high-fidelity analysis to perform an optimization using the standard form of DIRECT. For this reason we have begun to explore different ways to incorporate multifidelity data into the DIRECT optimizer.

Correction Methods

In order to incorporate multifidelity data in the DIRECT algorithm, a correction ratio is used to improve the accuracy of a low fidelity analysis. The goal of the correction scheme is to improve the accuracy over the entire design space while putting the most emphasis on points near the optima found by DIRECT. The DIRECT algorithm provides a ready-made method for selecting points to perform high-fidelity analyses without substantially adding complexity to the algorithm. DIRECT selects boxes for division that are either large, which puts their centers far away from other analyzed points, or have low function values, which implies that they are near an optimum. This is also the desired distribution of high-fidelity points. For the multifidelity implementation of the DIRECT algorithm, whenever DIRECT selects a box for division, a high-fidelity analysis is

performed at the center of the box before it is divided. The correction factor at this point would then be used to calculate a correction for the low-fidelity analyses generated in that box in one of two ways.

Constant Correction Ratio

The first correction factor, multifidelity version of DIRECT uses a constant correction factor (CCF) for each box. Figure 23 shows how this correction is applied. The correction factor is based on the center point of the last box that the low-fidelity point was in before the box was divided. The correction factor for the box to be divided is calculated immediately before the new points are analyzed and the correction is applied before the box is split into separate boxes. This avoids the need for response surfaces or other surrogates. If one of the corrected boxes is divided at a later iteration, a new high-fidelity value is calculated at the center of that box to correct the new low fidelity points that are generated and the center point.

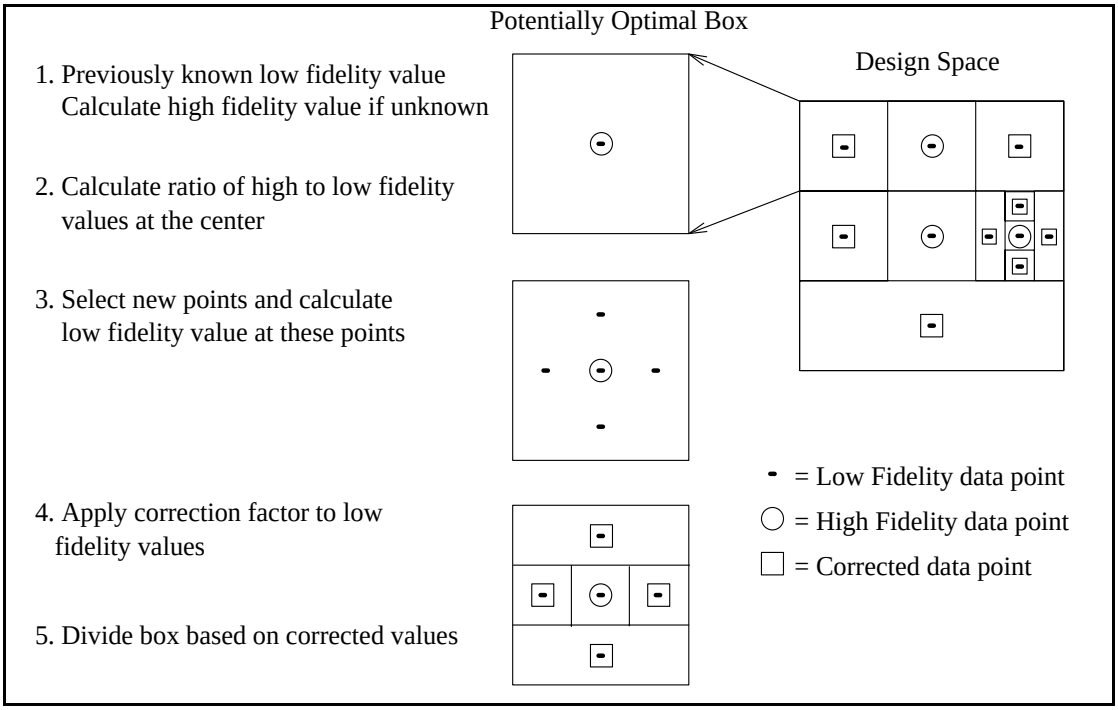


Figure 23. Constant correction scheme for multifidelity DIRECT

Equation 19 shows how the low-fidelity values are corrected in the CCF multifidelity DIRECT. a is the ratio of $f_{high_{center}}$ and $f_{low_{center}}$, the high and low-fidelity function values at the center of the box being divided, while f_{low} and f_{corr} are the low-fidelity and corrected function values at the new points that are generated in the box.

$$f_{corr} = a \quad f_{low} \tag{23}$$

The CCF multifidelity DIRECT algorithm is outlined as follows. The components added for the multifidelity version are in boldface.

1. Normalize the search space to the unit hypercube. Evaluate $f(c_1)$ and set $B_t = \{X_1(1)\}$.
2. For t =1, …
    a. Identify the set $S_t \subseteq B_t$ of potentially optimal boxes.
    b. For each box $X_j(t) \in S_t$ where $\text{diam}(X_j(t)) > \gamma$:
        i. **Perform a high-fidelity analysis at the center point of the box and calculate f_h**
        ii. **Calculate the ratio $a = f_h/f_l$ at the center**
        iii. Identify I, the dimensions of $X_j(t)$ with the longest side length. Let δ equal one-third of this length.
        iv. Sample the function at the points $c_j \pm \delta e_i$ for all $i \in I$, where c_j is the center of box $X_j(t)$ and e_i is the ith unit vector.
        v. **Apply the correction ratio to the low-fidelity values within the box $f_{cor} = f_l a$**
        vi. While I is not empty:

1. Divide $X_j(t)$ into thirds along the dimension $i \in I$ with the lowest value of $f(c_j \pm \delta e_i)$ to create two new boxes and reduce the size of box $X_j(t)$.
2. Remove i from I and add the new boxes to B_t.

c. Terminate if an appropriate stopping rule is satisfied.

Linear Correction Response Surface

The second correction method that we have studied uses a linear response surface (LRS) to predict the correction factor for the low-fidelity points. A linear fit to the correction factor at two points per dimension is used to predict the slope of the correction factor in part of the design space as described next.

For each potentially optimal box, a high-fidelity analysis is performed at the center, the correction factor at that point is calculated, and this correction factor is then compared with the correction factor used originally at that point. If the correction factor was accurate enough, then the slopes used in that box will not be updated. For this work the allowable error in the correction factor was set to 20%. The correction factor at that point will be combined with the slopes of the correction factor response surface that had been used to calculate the original correction factor at that point and used to calculate the correction factor for the low fidelity points generated in that box. If the correction factor was not accurate enough, then a high and low-fidelity analysis is performed at each new point that is generated to divide the potentially optimal box. The corrected value of each of the new points will then be set to the high-fidelity value. The correction factors at these points will generate new slopes to be used in the boxes that result from dividing the potentially optimal box. For any dimensions that were not divided at this time, the slope

of the response surface used to calculate the corrected value at the center of the potentially optimal box is copied over to the new response surface.

This correction method refines the correction factor used within each box by incorporating a first order approximation to the correction factor. By checking the accuracy of the correction when each box is divided, the algorithm can identify where the correction response surface is still accurate and where it is in need of refinement. As each correction response surface is generated, it is numbered and the slopes are saved. For each box in the design space a variable is used to indicate which response surface it should use to calculate the correction factor for the points generated when it is divided.

This version of the multifidelity DIRECT took much longer to converge than the original form of DIRECT. The convergence criterion that was used is based on the size of the smallest box in the design space just as in the original version of DIRECT. For this version, the size of the smallest box size shrunk rapidly at the start of the optimization but took a large number of iterations to get any smaller. DIRECT considers the points on the lower, right side of the convex hull of points in Figure 9 to be potentially optimal. Points on the left side of the convex hull are not potentially optimal because one can find a competing point that is both lower and is in a larger box. For the LRS version of DIRECT, after a few iterations, the smallest box generally had a corrected function value that was slightly higher than the corrected function value of a slightly larger box. This continued for a large number of iterations before a box with the smallest size would get divided. The difference between the function values of the smallest box and the smallest one selected for division was generally very small and comparable to the size of the error in the corrected value, but it resulted in an order of

magnitude increase in the number of function evaluations used by DIRECT. For this reason, the points on the lower, *left* side of the convex hull were included provided their value was less than 0.1% higher than the next largest box on the convex hull. This was an arbitrary value based on limited experimentation, but it worked well for these test problems.

Equation 26 shows how the low-fidelity values are corrected in the LRS multifidelity DIRECT for a box using the jth response surface. $f_{high_{center}}$ and $f_{low_{center}}$ are the high and low-fidelity function values at the center of the box being divided while f_{low} and f_{corr} are the low-fidelity and corrected function values at the new points that are generated. $RS_j(i)$ is the coefficient for the slope in the e_i direction of response surface j and $\left(x_i - x_{i_{center}}\right)$ is the distance of the new point from the center of the box being divided.

$$f_{corr} = \left(\frac{f_{high_{center}}}{f_{low_{center}}} + \sum_{i=1}^{n} \left(RS_j(i)(x_i - x_{i_{center}}) \right) \right) f_{low} \qquad (24)$$

The LRS version of the multifidelity DIRECT algorithm is described as follows. The components added for this version are in boldface.

1. Normalize the search space to the unit hypercube.
    a. Divide Box 1
        i. Perform **high-** and low-fidelity analysis at the points c_1 & $c_1 \pm \delta e_i$ for all $i \in I$, where c is the center of the design space and e_i is the ith unit vector.
        ii. $\boldsymbol{f_{cor} = f_{high}}$ **for each point analyzed**
        iii. Create correction response surface (RS) from initial points

iv. Store coefficients in RS array and set 'RS to use' to 1 for these boxes

v. While *I* is not empty:

1. Divide $X_j(t)$ into thirds along the dimension $i \in I$ with the lowest value of $\boldsymbol{f_{cor}(c_j \pm d e_i)}$ to create two new boxes and reduce the size of box $X_j(t)$.
2. Remove *i* from *I* and add the new boxes to B_t.

2. For t =1, …

a. Identify the set $S_t \subseteq B_t$ of potentially optimal boxes.

b. For each box $X_j(t) \in S_t$ where $\text{diam}(X_j(t)) > \gamma$:

i. If no high fidelity analysis at center

1. **Perform high fidelity analysis at center**
2. **Compare actual correction factor (CF) to CF used previously in that box**
3. **If correction is inaccurate**

a. Identify *I*, the dimensions of $X_j(t)$ with the longest side length. Let δ equal one-third of this length.

b. Perform **high** and low fidelity analysis at the points $c \pm \delta e_i$ for all $i \in I$, where *c* is the center of the potentially optimal box and e_i is the *i*th unit vector.

c. $\boldsymbol{f_{cor} = f_{high}}$ **for each point analyzed**

d. **Create correction RS from new points. Use terms from old RS for undivided dimensions**

e. Store RS in next row of RS array

f. Update 'RS to use' in new boxes and divided box to indicate new RS

4. **If correction is OK**

a. Get 'RS to use' from potentially optimal box

b. Get RS from RS array

c. Identify *I,* the dimensions of $X_j(t)$ with the longest side length. Let δ equal one-third of this length.

d. Perform low-fidelity analysis at the points $c \pm \delta e_i$ for all $i \in I$, where c is the center of the box and e_i is the ith unit vector.

e. Calculate correction factor (CF) for the center of the box

f. Apply RS and CF at the center of the box to low fidelity values

ii. If previous high fidelity analysis at center

1. Identify *I,* the dimensions of $X_j(t)$ with the longest side length. Let δ equal one-third of this length.
2. Perform low-fidelity analysis at the points $c \pm \delta e_i$ for all $i \in I$, where c is the center of the box and e_i is the ith unit vector.
3. **Get 'RS to use' from potentially optimal box and the appropriate RS from the RS array.**

4. **Calculate CF for the center of the box.**
5. **Use RS and CF at the center of the box to calculate the CF for the low fidelity values.**
6. While *I* is not empty:
    a. Divide $X_j(t)$ into thirds along the dimension $i \in I$ with the lowest value of $\boldsymbol{f_{cor}}\ \boldsymbol{(c_j \pm d e_i)}$ to create two new boxes and reduce the size of box $X_j(t)$.
    b. Remove *i* from *I* and add the new boxes to B_t.

c. Terminate if an appropriate stopping rule is satisfied.

Test Problems

The first test problem is a 6^{th} order function confined to the box $[-2,2]^n$. The high-fidelity function is defined as:

$$y = 25 + \sum_{i=1}^{n} \left\{ 2(x_i + e_i)^2 - 1.5(x_i + e_i)^4 + .2(x_i + e_i)^6 \right\} \qquad (25)$$

The one-dimensional graph of this function is given in Figure 24.

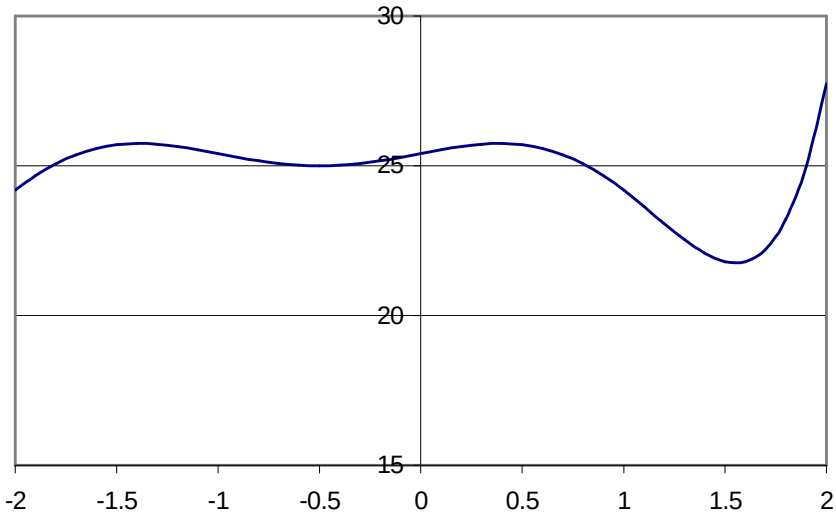


Figure 24. High-fidelity Function in One Dimension

The low-fidelity function is a truncated Chebyshev expansion of the high-fidelity function. It drops the 6^{th} order term of the Chebyshev expansion of the high-fidelity equation. This represents a problem where the simple model fails to capture the higher order behavior of the true function. The low-fidelity function is given by:

$$y = 25 + \sum_{i=1}^{n} -1 - 2\left(2\left(\frac{x_i + e_i}{2}\right)^2 - 1\right) - .6\left(8\left(\frac{x_i + e_i}{2}\right)^4 - 8\left(\frac{x_i + e_i}{2}\right)^2 + 1\right) \quad (26)$$

The one-dimensional graph of the function is given in Figure 25.

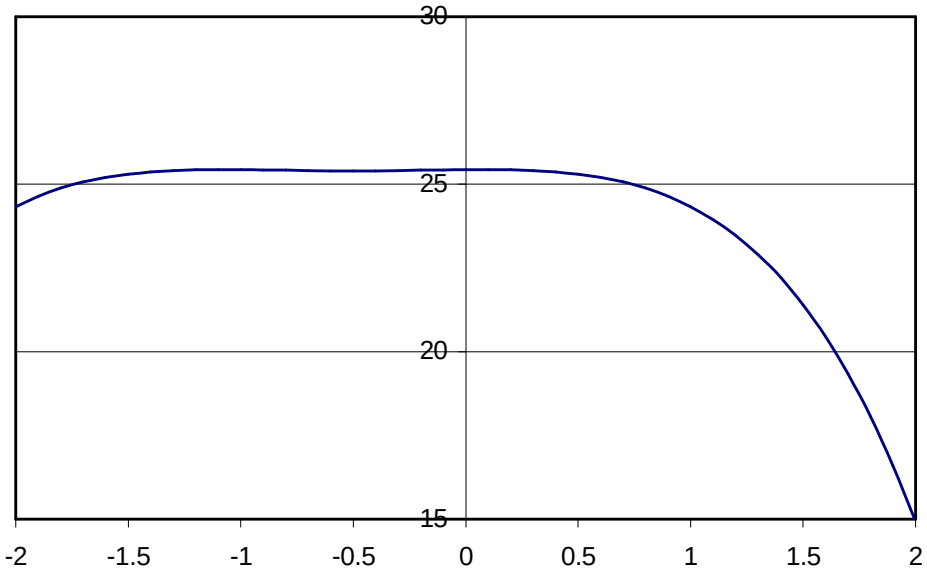


Figure 25. Low-fidelity Function in One Dimension

Both the high and low-fidelity models are polynomial functions, which means their actual execution times are trivial. In order to compare the efficiency of the multifidelity DIRECT to simply optimizing the high-fidelity problem directly, a theoretical cost ratio for the two functions can simply be designated. In this way problems with any combination of execution times can be simulated to see where the algorithm is effective.

The second test problem is the Griewank function, previously given in equation 9 and repeated here for convenience

$$F(x) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{d} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right). \tag{9}$$

A simple quadratic function centered a short distance from the optimum was used as the low fidelity analysis function.

$$F(x) = \sum_{i=1}^{n} \frac{(x_i + e_i)^2}{d}, \tag{27}$$

where e_i is a random value in the range [25,45] and d was set to 1000. The bounds for the design space were set so that the space had edges of length 1000 with the upper limit randomly set in the range [100, 900]. This was to generate different random results for each run.

Test Results

The two versions of the multifidelity DIRECT were compared against the original version of DIRECT for optimizing the high-fidelity function. For each version, the optimization was stopped once the smallest box diameter reached 0.01% of the original box side length. At that point DOT was started from up to 15 starting points separated by at least 5% of the width of the design space. Each optimizer was run 30 times for each case with random values for $\boldsymbol{e}$ and random bounds for the Griewank function and the results were averaged.

Sixth Order Test Problem

Table 10 gives the results for the constant correction factor and response surface versions of DIRECT for the first test problem.

Table 10. Multifidelity versions of DIRECT vs. single fidelity DIRECT for the sixth order test problem, Eq. 25.

	Original DIRECT	Constant Correction Factor DIRECT		Linear Response Surface DIRECT[3]	
	High-fidelity Evaluations	High-fidelity Evaluations	Low-fidelity Evaluations	High-fidelity Evaluations	Low-fidelity Evaluations
2 DV	688	1562	4170	588	439
5 DV	2202	1650	2500	2687	6710
10 DV	8345	6673	34642	10026	57555

For the CCF version of DIRECT, the 5 and 10 dimensional case resulted in a slight improvement in the number of high-fidelity analyses used by the optimization but at the cost of a large number of low-fidelity analyses. The results from the two dimensional case indicate that this correction scheme can cause problems for DIRECT. As the function values at the center of each box change from corrected values to high-fidelity values, the points that were near the convex hull can change. This can cause the optimizer to select boxes for division that would normally not be selected by the original version of DIRECT. An animation of the point selection for the two dimensional case showed the optimizer jumping to boxes where the corrected low-fidelity value is lower than the high-fidelity value. For this case the optimizer performed a wider search of the design space than the single fidelity version and required more evaluations to reach the optimum.

For the LRS version of DIRECT, the opposite trend was the case. As the number of dimensions increased, the optimizer became much less efficient at locating the global optimum. The more aggressive correction response surface tended to over correct the low-fidelity results which led to DIRECT dividing more boxes than would normally have been selected. The optimization jumped from one box to another as in the CCF version

[3] For the 10D case, 6 out of the 30 runs failed to locate the global optimum before converging normally and for 12 of the runs which did locate the global optimum, the optimizer was stopped by the upper limit on the number of points to be analyzed, which was set to 100,000 low-fidelity evaluations.

as the high-fidelity analyses were shown to be worse than the corrected value and different boxes became potentially optimal.

One difference between the two dimensional case and the other cases is that the low-fidelity function becomes negative in some areas for more than 2 dimensions. This can cause problems in areas where the objective function is changing sign. In order to test what effect this had on the optimization the high and low-fidelity functions were changed so that the low-fidelity function remained positive for the entire design space. The high-fidelity function was changed to:

$$y = \sum_{i=1}^{n} \{10 + 2(x_i + e_i)^2 - 1.5(x_i + e_i)^4 + .2(x_i + e_i)^6\} \qquad (28)$$

The low-fidelity function was set to:

$$y = \sum_{i=1}^{n} \left\{ 10 - 1 - 2\left(2\left(\frac{x_i + e_i}{2} \right)^2 - 1 \right) - .6\left(8\left(\frac{x_i + e_i}{2} \right)^4 - 8\left(\frac{x_i + e_i}{2} \right)^2 + 1 \right) \right\} \qquad (29)$$

This prevents either function from becoming negative at any point in the design space. The results for this version of the test problem is given in Table 11.

Table 11. Multifidelity DIRECT comparison for modified sixth order problem, Eq. 28, 29.

	Original DIRECT	Constant Correction Factor DIRECT[4]		Linear Response Surface DIRECT[5]	
	High-fidelity Evaluations	High-fidelity Evaluations	Low-fidelity Evaluations	High-fidelity Evaluations	Low-fidelity Evaluations
2 DV	688	1553	4069	612	541
5 DV	2181	26842	150000	2205	6288
10 DV	8333	17885	150000	8237	48884

[4] For the CCF version of DIRECT, the optimizer ran until it reached the maximum number of allowable function evaluations for the 5 and 10 DV cases which was 150,000.

[5] For the 10 DV case, the LRS version failed to locate the global optimum for 7 out of the 30 runs but none of them reached the maximum allowable number of function evaluations.

For the 2 DV problem, both versions performed approximately the same as for the original problem. This was expected since the problems were basically the same. However, for the other two cases the LRS version of DIRECT was slightly cheaper than for the original problem. This seems to indicate that it is more efficient when the high and low-fidelity functions remain the same sign during the optimization. For these problems the optimizer had the same problem that LRS version had originally. After a small number of iterations, the smallest boxes were not divided until most of the boxes that were slightly larger were divided. In order to reduce the number of function evaluations that were required the same modification to the potentially optimal box division that was used in the LRS version was used for the CCF version. Boxes on the convex hull that were smaller than the one containing the best function value but whose function value was less than 0.1% worse were selected as potentially optimal. The results from this version of the CCF DIRECT are given in Table 12.

Table 12. Multifidelity constant correction factor DIRECT with modified potentially optimal box selection

	Original DIRECT	Constant Correction Factor DIRECT	
	High-fidelity Evaluations	High-fidelity Evaluations	Low-fidelity Evaluations
2 DV	688	610	512
5 DV	2181	2350	7477
10 DV	8333	8557	49557

By letting DIRECT divide boxes that are smaller than the best function value but slightly worse, the CCF version converged in about the same number of function evaluations as the LRS version but none or the CCF runs resulted in a sub-optimal solution. With the constant correction factor, the CCF version avoids spurious corrected function values caused by large errors in the predicted slope of the correction factor.

For both versions of the Multifidelity DIRECT, the actual number of boxes was much higher than for the single fidelity version of DIRECT. Due to the errors in the corrected values, the boxes with the lowest corrected function values tended to have a larger true function value when a high-fidelity analysis was performed. This resulted in a much slower reduction to the minimum box size. However, when the optimization was stopped after a smaller number of iterations than it would require using the box size criterion, in most cases it could still locate the global optimum. This is an indication that a different stopping criterion may be useful for the Multifidelity DIRECT.

Griewank Function

The Griewank function was the second example problem tested. For this case both functions had similar global trends with the global optima separated by a small percentage of the design space. The difference in gradients near the global optimum also was not as severe as for the previous problems. Based on the results for the previous problem, the potentially optimal box calculation for the CCF DIRECT was modified to include boxes on the lower left side of the convex hull as with the LRS version. Table 13 shows the results for the ten-dimensional case of the Griewank function.

Table 13. Multifidelity DIRECT Comparison for Griewank Function

10 DV	# of optima located / # of runs	High-fidelity Evaluations / Optimum	Low-fidelity Evaluations / Optimum
Original DIRECT	56 / 100	11810	NA
CCF DIRECT	18/ 30	7754	37134
LRS DIRECT	10 / 30	12900	75800

For this problem the CCF version of DIRECT showed about a 35% improvement in the number of high-fidelity evaluations required to locate the global optimum. The low fidelity information was useful in reaching the neighborhood of the global optimum and the CCF correction was able to move the search closer to the true global optimum. About half of the high fidelity evaluations were used by the local optimizer for this problem. For this problem there are no multiple widely scattered, local optima, so the search is clustered in a small region of the design space. Thus the use of up to 15 local optimizations as with the original DIRECT algorithm is excessive and could be reduced in general when some knowledge of the design space is available before the optimization.

The LRS version of DIRECT did not perform as well in this case as the CCF version. Both the cost and reliability of the optimizer were poorer than the original version of DIRECT. This indicates that it is doing a poor job of predicting the slope of the correction ratio near the global optimum. For this low fidelity approximation, although the global optima are near to each other in the design space and the difference in the low- and high-fidelity function values near the true global optimum are not large compared to the average function value throughout the design space, the difference is large compared to the high-fidelity values near the optimum. This makes small changes in the high-fidelity value have a large affect on the correction ratio. In order to determine if this was causing the LRS version to perform poorly the high- and low-fidelity functions were modified to increase their values at the global optimum.

The Griewank function was simply increased by 100 over the entire design space

$$F(x) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{d} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 100 \quad . \tag{30}$$

The low-fidelity approximation was increased by 150 and its value for *d* was set to 2000 so the low-fidelity function would cross the high-fidelity function at some points in the design space. A one dimensional plot of both functions is given in Figure 26.

$$F(x) = \sum_{i=1}^{n} \frac{(x_i + e_i)^2}{d} \tag{31}$$

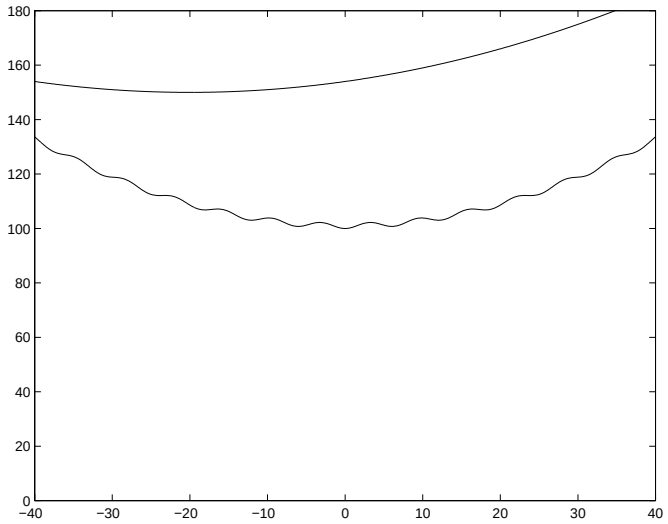


Figure 26. Modified Griewank function and low fidelity approximation

The bounds for the problem were left the same and the optimizers were run 30 times and the results were averaged. The results are given in Table 14.

Table 14. Multifidelity DIRECT comparison for modified Griewank function.

10 DV	# of optimal basins[6] / # of runs	High-fidelity Evaluations / Optimum	Low-fidelity Evaluations / Optimum
Original DIRECT	12/30	8524	NA
CCF DIRECT	15/30	4310	9090
LRS DIRECT	14/30	4840	11426

[6] Optimizers did not get as close to the actual optimum as they did for the original Griewank function.

While the performance of the DIRECT portion of the search was much better for the both multifidelity versions of DIRECT, the local search performed very poorly in refining the solution within the basin containing the global optimum. The variation in the function from the cosine portion was much smaller than the value of the function at the optimum, which caused the local optimizer to consider the problem converged before reaching the actual minimum of the global optimum basin. Therefore, the results in Table 12 consider the optimization to have reached the global optimum if it converges to any point within the basin containing the global optimum.

With this modified function all three versions of DIRECT performed better than the original Griewank function but the two multifidelity versions improved the most. The number of high-fidelity evaluations for the CCF version was slightly more than half the number of evaluations used by the original DIRECT algorithm. Of more importance however, was the fact that for this case only a quarter of the high-fidelity analyses were used by DIRECT, the rest were used by the local optimization. Thus, the reduction in cost of just the DIRECT portion was even greater.

The LRS version of DIRECT was slightly more expensive than the CCF version but it was still a large improvement over the original DIRECT algorithm. For this case also, most of the high-fidelity evaluations were used by the local optimizer after DIRECT was finished. Both versions of the multifidelity DIRECT indicate that a different method of selecting points to start a local optimization is needed for the multifidelity algorithm. Using 15 local optimizations on the original DIRECT algorithm had a small affect on the number of points analyzed when the optimization uses tens of thousands of evaluations and increases the likelihood that you will start a local optimization near the global

optimum. However, since the CCF version used on average 450 high-fidelity evaluations per optimization for this problem, the local optimizer is too expensive to use at 15 points in the design space. In order to improve the efficiency of the local portion of the search, either a multifidelity local optimizer could be employed or the number of local optimizations must be reduced.

Algorithm Status

The two versions of the multifidelity DIRECT algorithm tested here have provided mixed results on the test problems examined. On the first test problem there was only a small improvement in the number of function evaluations for the CCF version of DIRECT and the LRS version resulted in an increase in the number of function evaluations. While there was a noticeable decrease in the number of high-fidelity analyses required for the Griewank test problem, the low-fidelity function was fairly similar to the high-fidelity function. The global optima were separated by only a small percentage of the design space and the gradients of the two functions were small in the region of the global optimum. On the first test function, the gradients of both the high- and low-fidelity functions were much higher in the neighborhood of the global optimum and the optima were separated by a larger percentage of the design space. The correction scheme did not perform well for this problem, possibly due to the large difference in the gradients at the true global optimum.

These experiments have shown that it is possible to use a correction scheme with the DIRECT algorithm to select boxes to divide. Provided the high- and low-fidelity functions are not too dissimilar near the global optimum, the algorithm is able to locate the global optimum at a lower cost than the single fidelity version. The multifidelity DIRECT code performs the best when the correction ratio is not changing rapidly within

the design space, particularly in the region of the global optimum. It is errors in the prediction of this correction ratio that hurt the efficiency of this code. With additional experimentation on the correction and box selection scheme, the multifidelity DIRECT algorithm may still become an effective global optimization algorithm for a wider combination of high- and low-fidelity functions, provided better ways of dealing with these errors can be developed.

Chapter 5
PARALLEL DIRECT

The DIRECT global optimization algorithm has been found to be an efficient optimization routine for problems with as many as 20 design variables and noisy and nonconvex objective functions. It is quick to locate the region of the global optimum and when it is combined with a local optimizer for the final convergence it can locate the precise minimums with little additional work. However, this optimizer is still too expensive for use with high-fidelity evaluations and a large number of variables on a sequential computing platform. For this reason a parallel version of the DIRECT algorithm is desired.

Previous Parallel DIRECT

The DIRECT algorithm provides natural opportunities for parallelization. At each iteration, DIRECT generates a batch of points to be analyzed. These analyses can be distributed to multiple processors for evaluation and then collected back at the primary processor for sorting and generating points for the next iteration. This is the approach used by Baker et al. (2000). Provided that the analyses take longer to execute than the DIRECT overhead, which must be run serially, this method can be a very efficient way to parallelize the DIRECT code. Baker et al. found that their implementation of the DIRECT algorithm achieved up to 85% efficiency on 64 processors for a 28 design variable problem with function evaluations that required approximately 1.75 seconds.

This method does not require changing the DIRECT algorithm at all, and has the same search performance as the sequential implementation. However, the DIRECT

algorithm has some aspects which are not designed for parallel implementation. For the first few iterations, DIRECT produces a very small number of points to be evaluated. As the number of processors increases, more processors will sit idle for more iterations until the number of evaluations becomes high enough to provide work for all of them. In addition, at the end of each iteration, DIRECT must wait until all of the processors are finished evaluating the points before continuing. For iterations where there are only a few evaluations per processor, a small load imbalance can result in a large ratio of time spent sitting idle for some processors while waiting for the rest to finish.

Baker et al. (2000) have also tested an aggressive version of DIRECT in an effort to increase the number of evaluations performed at each iteration. Instead of using the DIRECT method of selecting boxes for division, the boxes are sorted according to size and the boxes with the lowest function value for each box size were selected for division. This is based on the assumption that many of these boxes would likely be divided at a later iteration as larger boxes get divided. This provides more points for evaluation at each iteration and improves the load balancing over a larger number of processors. For the 28 design variable HSCT problem, the original DIRECT used between 150 and 400 evaluations per iteration after the first few iterations while the aggressive DIRECT used over 2500 evaluations per iteration after only 4 iterations.

The cost of this method is an increase in the number of function evaluations used by the optimizer. For their experiments, the optimization of the HSCT problem was stopped after 37 iterations and 10,077 function evaluations using the original DIRECT algorithm. Using the aggressive DIRECT, the optimization was stopped at 20 iterations after using 48,577 function evaluations. The maximum efficiency of the computer code

rose to 94% for 64 processors, but at a cost of almost 5 times as many evaluations. For this reason, the aggressive DIRECT was not used in this research.

DIRECT Modifications

This research examines two modifications to increase the performance of DIRECT for single-fidelity global optimizations. At the start of the optimization, DIRECT is modified to create enough points for all of the processors before distributing them. This prevents some processors from sitting idle until there is enough work to go around. Secondly, when each processor finishes its assigned analysis it immediately is given another analysis to perform instead of waiting for all of the other processors to finish. This requires DIRECT to continuously generate points for analysis instead of using a batch method.

Accelerated Start

The first iteration of the DIRECT algorithm must be changed so that there is enough points for each of the processors to analyze. Two methods have been tried. For Method 1, the optimizer can select the largest box and divide it in the same manner as the DIRECT algorithm without analyzing any of the points. The optimizer assumes that each box has the same function value and divides the dimensions randomly (Figure 27). This continues until there are enough points for each processor to be given a task.

Method 2 simply divides the design space based on how many processors there are. Each dimension is divided into evenly sized boxes so that the total number of boxes exceeds the number of processors. In order to prevent the aspect ratio of the boxes from becoming too large, each dimension is divided into approximately the same number of partitions (Figure 28). Some of the dimensions are divided into d partitions and the rest are divided into $d+1$ partitions. If an n dimensional problem is run on P processors then

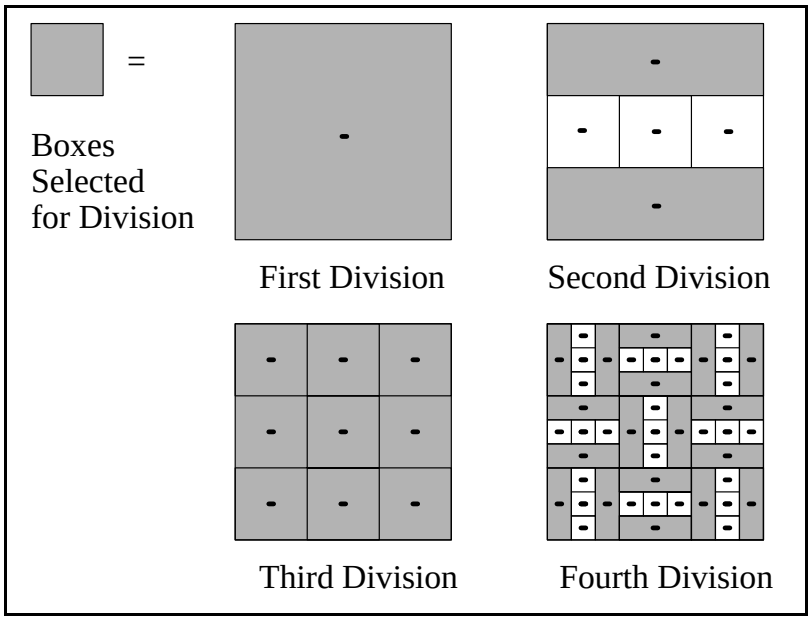


Figure 27. Initial Box Division for Parallel DIRECT

the following equation is solved for the number of partitions, *d*, and how many dimensions, *a*, are to be divided into *d* partitions. Note that *d* can be 1.

$$P \leq d^{a} + (d+1)^{(n-a)} \tag{32}$$

This method results in boxes that are not square if all of the dimensions are not divided the same number of times. This can result in additional sets of box sizes and reduce the number of dimensions that are divided by DIRECT in each potentially optimal box. DIRECT only divides the largest dimensions in a potentially optimal box. In general, the boxes will never become square after any combination of divisions so no box will be divided in every dimension by DIRECT.

For both accelerated start up methods, DIRECT is run on the entire design space at once. The design space is not divided into separate regions where independent DIRECT optimizations are run. The next modification allows DIRECT to place the extra points into a queue to be distributed as processors become available. For this reason,

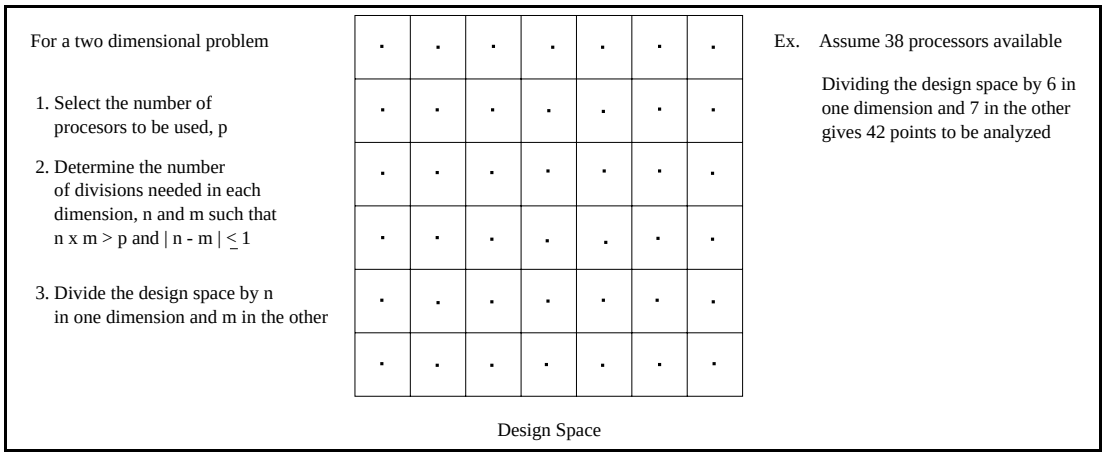


Figure 28. Alternate Initial Box Division for Parallel DIRECT

the initial set of points does not need to be matched to the number of processors and each point is simply analyzed as processors become available.

Continuous DIRECT

A second modification to the DIRECT algorithm aims to eliminate the sequential calculation of points for evaluation as well as the global synchronization it entails. The original DIRECT algorithm generates a batch of points, waits for all of the analyses to finish and then calculates the next batch based on the previous results. When one of the analysis processors finishes its tasks it must wait for all other processors to finish before it is given additional work. By modifying DIRECT to maintain a queue of tasks awaiting assignment, processors are able to receive a new task immediately after completing the previous one. For this to work, DIRECT must select boxes for division before it receives the results from all of the existing boxes.

Figure 29 illustrates the organization of the parallel DIRECT algorithm. The analysis is done on separate processors independently of the DIRECT processor. When they finish one job they send the results to the DIRECT processor and receive a new task from the queue. The parallel code is set up so that while the analysis processor is

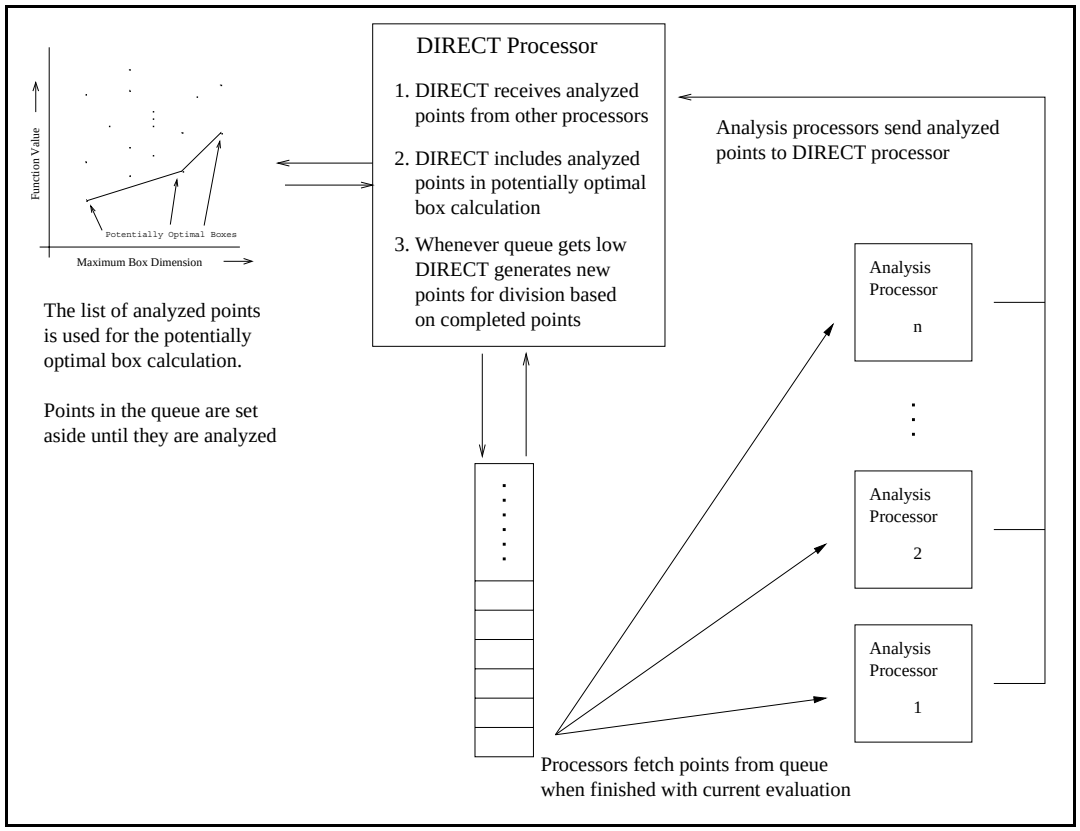


Figure 29. Continuous parallel DIRECT organization

working on one point, the DIRECT processor has already sent a second point to that processor. In this way, when the analysis processor finishes its job and sends the result, its next job is waiting for it and it does not have to wait for the DIRECT processor to handle the new data. Once the queue is empty, however, the analysis processors are not sent a waiting point to analyze until one analysis processor has finished all of its work.

The DIRECT processor is responsible for generating new points for analysis based on the results it has received. It maintains a list of the points which have been analyzed which it uses to select potentially optimal boxes. It continues to add points to the list until the queue of points to be analyzed is empty and one analysis processor has run out of points to analyze. When this happens, it uses the points that are finished to calculate a set of potentially optimal boxes. It then generates new points from the

potentially optimal boxes and sends them to the queue. In the original version of DIRECT, all of the new points are analyzed immediately and the parent box is divided based on the new function values before the next potentially optimal box is divided. Since DIRECT can not wait for all of the points to come back from the analysis processors before adding them to the potentially optimal box calculation, each box is not included until all of the child boxes of a parent box are finished. After the last new point within a parent box is finished, the parent box is divided based on the new function values within that box. These new completed boxes are then added to the potentially optimal box calculations. This means that some points which have been analyzed are not included in the next potentially optimal box selection because the other child points within the same parent box are not finished.

The parallel DIRECT algorithm is as follows. The modifications for the parallel version are in bold.

1) Normalize the search space to the unit hypercube.
    a) On the DIRECT processor divide the search space to provide two points for every analysis processor plus enough to place at least x **points into the queue.**
2) For t =1, …
    a) For the analysis processors:
        i) Read a point sent to it by the main processor.
        ii) Analyze the point and send the results to the DIRECT processor.
        iii) Terminate if directed to by DIRECT processor, else goto 3.a.
    b) For the DIRECT Processor
        (1) **Send two points to each analysis processor using non-blocking sends.**

(2) **Add the result from an analysis processor to the appropriate box.**

(3) **Send a new point to the analysis processor if the queue is not empty.**

(4) **Check list of undivided boxes to see if all of the child boxes from a potentially optimal box have been analyzed. If so:**

(a) Identify the set of dimensions *I* which were divided.

(b) While *I* is not empty:

(i) Divide $X_j(t)$ into thirds along the dimension $i \in I$ with the lowest value of $f(c_j \pm \delta e_i)$ to create two new boxes and reduce the size of box $X_j(t)$.

(ii) Remove *i* from *I* and add the new boxes to B_t.

(5) **Check queue to see if the queue is empty and one processor has run out of points to analyze, if not goto 2.b(1)**

(6) Identify the set $S_t \subseteq B_t$ of potentially optimal boxes.

(7) For each box $X_j(t) \in S_t$ where $\text{diam}(X_j(t)) > \gamma$:

(a) Identify *I,* the dimensions of $X_j(t)$ with the longest side length. Let δ equal one-third of this length.

(b) Select points $c \pm \delta e_i$ for all $i \in I$, where *c* is the center of the rectangle and e_i is the *i*th unit vector.

(c) Set the box size of each child box to the size and shape of the potentially optimal box

(d) Divide potentially optimal box size by 3 in each dimension in *I*.

(e) Send points for child boxes to queue for analysis.

(8) Terminate if an appropriate stopping rule is satisfied.

Message Traffic

Initial testing of the continuous parallel DIRECT algorithm was done on a heterogeneous network of UltraSPARC processors with speeds from 85 to 300 MHz at the High-performance Computing & Simulation Research Lab at the University of Florida. Two versions of the test problem were used which gave the same function values but allowed the algorithm to be compared with analysis functions of different run times. This was done by forcing the analysis to do 10-100 million loops of a simple floating point multiplication before returning the results of the analysis. The fast version (Prob. A) required approximately 0.07 seconds on a 300 MHz UltraSPARC while the slower analysis case (Prob. B) required about 0.8 seconds.

The test problem that was used for this initial testing is given in equation 33

$$F(x) = \sum_{i=1}^{n} \left[2.2(x_i + e_i)^2 - (x_i + e_i)^4 - 1.3 x_i \right], \qquad (33)$$

where n = 10.

The one dimensional graph of this function is given in Figure 28.

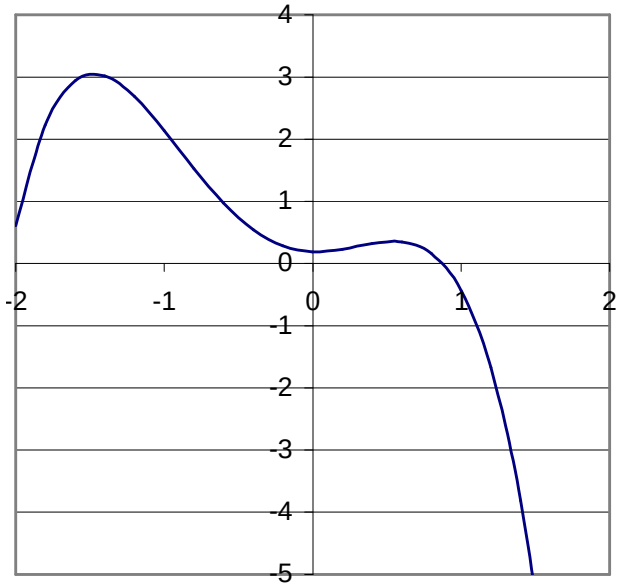


Figure 30. Test Problem in One Dimension for Parallel DIRECT (*e* = 0.2)

This is a modification of the quartic function from Equation 13 to increase the global bias of the function towards the global optimum. The quartic function was actually deceptively easy for the original DIRECT algorithm to solve. At every iteration, DIRECT divided the box which contained the global optimum due to the slight global trend towards the optimum. If the continuous DIRECT code attempts to select boxes based on incomplete information it can start to divide a box which contains some other good local optimum. Once DIRECT has divided one of these other boxes a few times, the function value it has located will be so low that it masks all but the largest remaining boxes from the potentially optimal box selection (Figure 16). Thus, DIRECT only works on this problem if it moves near the global optimum within the first few iterations. Otherwise it converges to the first deep local optimum it finds.

The log files of the message traffic were used to examine the way that the continuous algorithm was performing on this simple test problem for this network system. These tests employed the first accelerated start method which divides the design space the same way that DIRECT would. The network contains multiple clusters of processors which shared the connecting bandwidth between different clusters. Because of this, the graphs of the message traffic shows cases where messages were delayed by processes running on other parts of the network. When the communications had difficulty reaching their destination rapidly, the optimization encountered many problems. The convex hull calculations had to proceed with a smaller set of completed points which made the search less efficient and resulted in dividing more boxes. In some cases for larger numbers of processors, messages were lost completely causing DIRECT to never search the region associated with the lost result. Ultimately, I was unable to

complete runs using more than 25 processors due to the problems caused by communications and synchronization between the processors.

Figure 31 shows the behavior of the code on 4 processors at the beginning of the optimization. The communication is orderly and every message is at its destination before the processor tries to read it. The analysis processors are able to continuously work on analyzing points and the main processor takes a negligible amount of time to generate new points when the queue gets low. The main processor spends most of its time waiting for results from the analysis processor. This effectively means that there will be one idle processor most of the time. While this is poor for a four processor case, once the number of processors increases the main processor will be kept busy for a larger percentage of time.

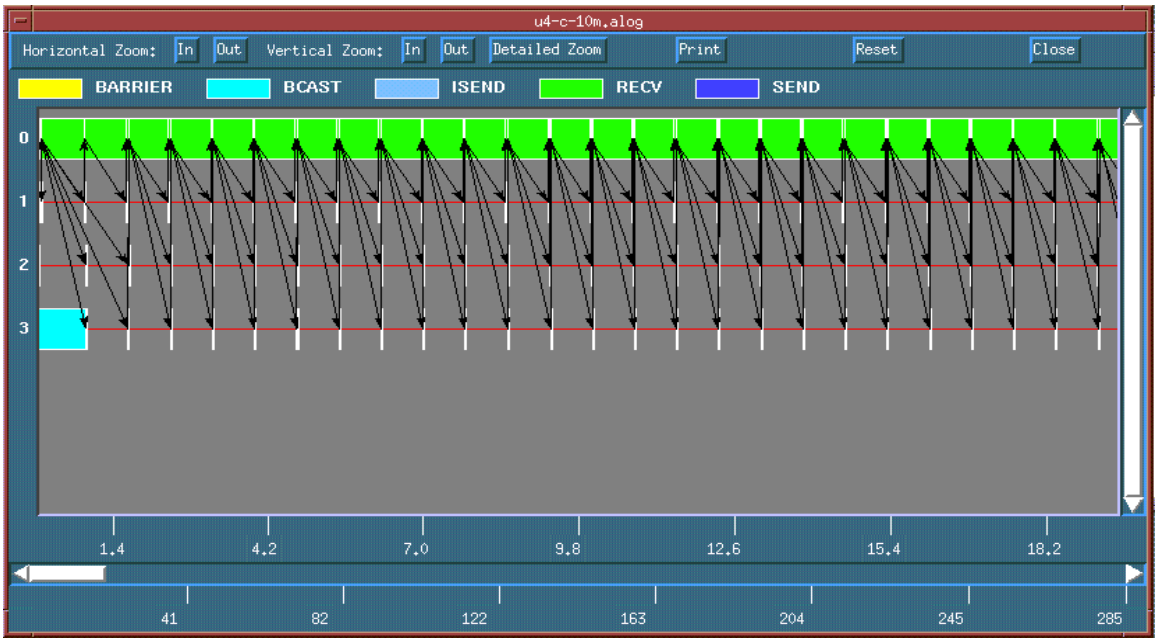


Figure 31. Initial message pattern for 4 processors using Prob. B.

Figure 32 shows the same run as Figure 31 at the end of the optimization. The main processor is still able to generate points before any analysis processor runs out of

work and the optimization is well behaved until the end. The convex hull calculation still takes a negligible amount of time compared to the analysis and the main processor has no trouble keeping up with the message traffic.

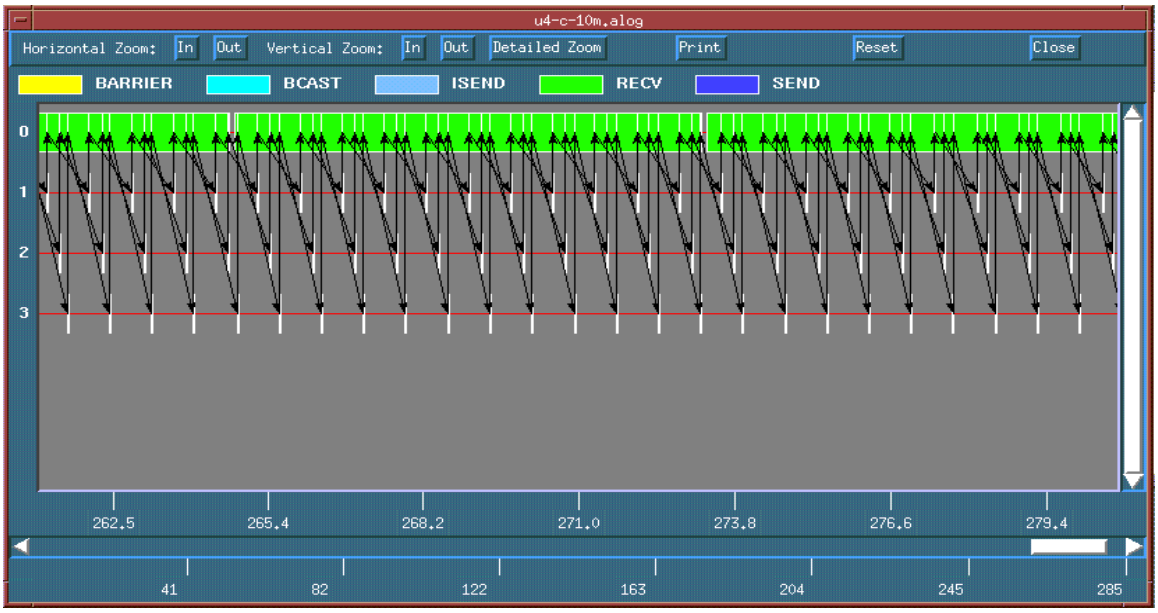


Figure 32. Final message pattern for 4 processors using Prob. B.

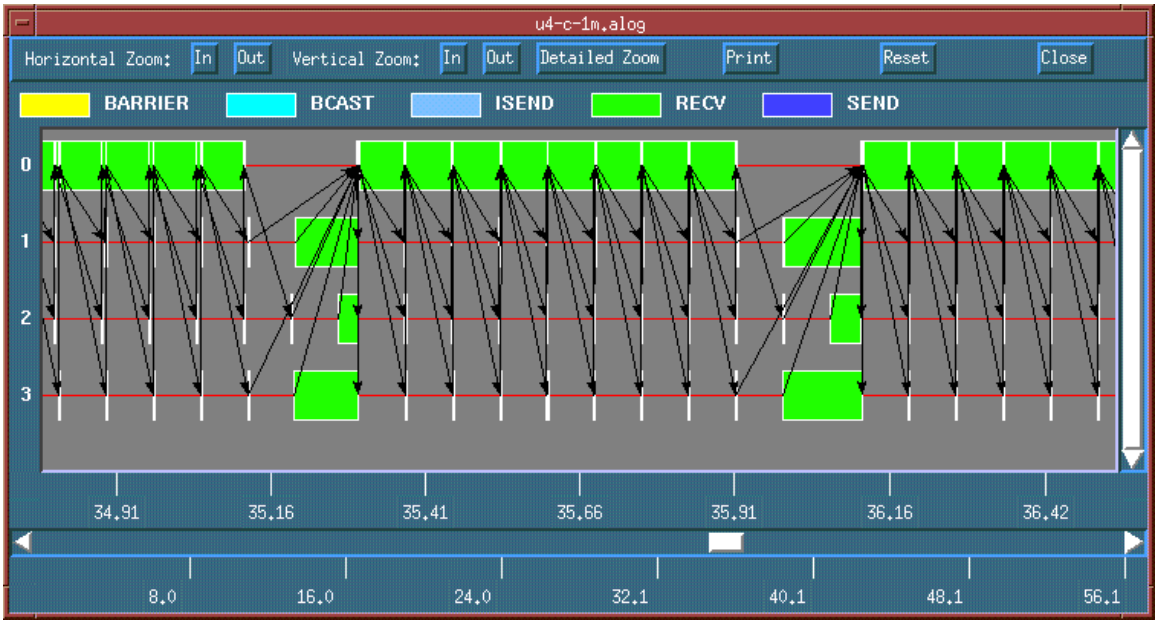


Figure 33. Final message pattern for 4 processors using Prob. A.

Figure 33 shows the effect of using faster analyses when performing the opptimization. In this case the main processor takes the same amount of time to generate new points but since the analysis processors are completing the points faster, they end up having to wait at the end of each iteration for new points to analyze. The time scale on this graph is expanded relative to the previous figures to show the activity more clearly.

Figures 34 through 36 show the effect that the lack of points for the analysis processors had on the computational efficiency of the algorithm for larger numbers of processors. Near the end of each optimization, the analysis processors spent a large amount of time waiting for points to analyze. The efficiency was further hurt by the irregularity of the message traffic between the processors that caused some processors to be idle even at the beginning of the optimization. This graphic also shows the effect of using a heterogeneous collection of processors. Due to the difference in speeds, the results for some points can take much longer to finish than the other new points from the same parent box. The size of a new box cannot be determined until all of the other child boxes from the same parent are finished. Note that the load is implicitly balanced over the analysis processors by distributing the points as they are needed instead of distributing them at the beginning without regard to relative speeds. No processor is idle for more than one analysis period longer than any other processor.

As the number of points and processors increases, the main processor takes longer to calculate the potentially optimal boxes to generate a new set of points. For this problem, as the optimization nears the end there are not many points on the convex hull. This means that there are not very many points generated at each iteration, as Figures 35 and 37 demonstrate. Thus, the analysis processors have to wait more frequently.

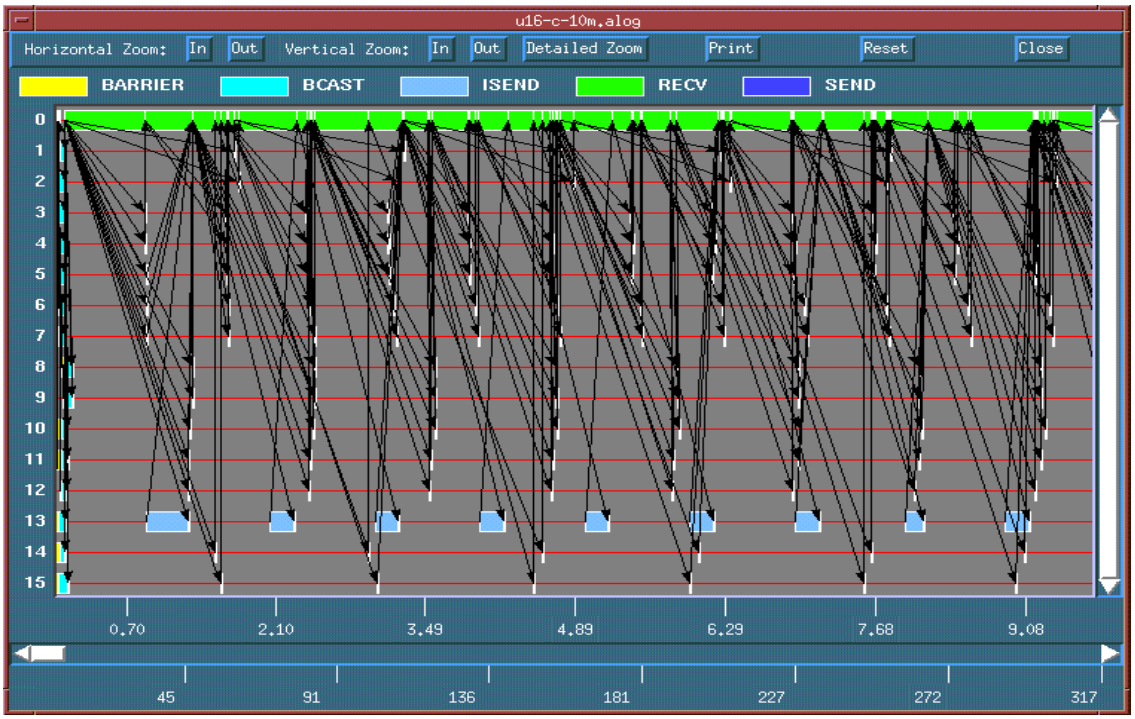


Figure 34. Initial message pattern for 16 processors using Prob. B.

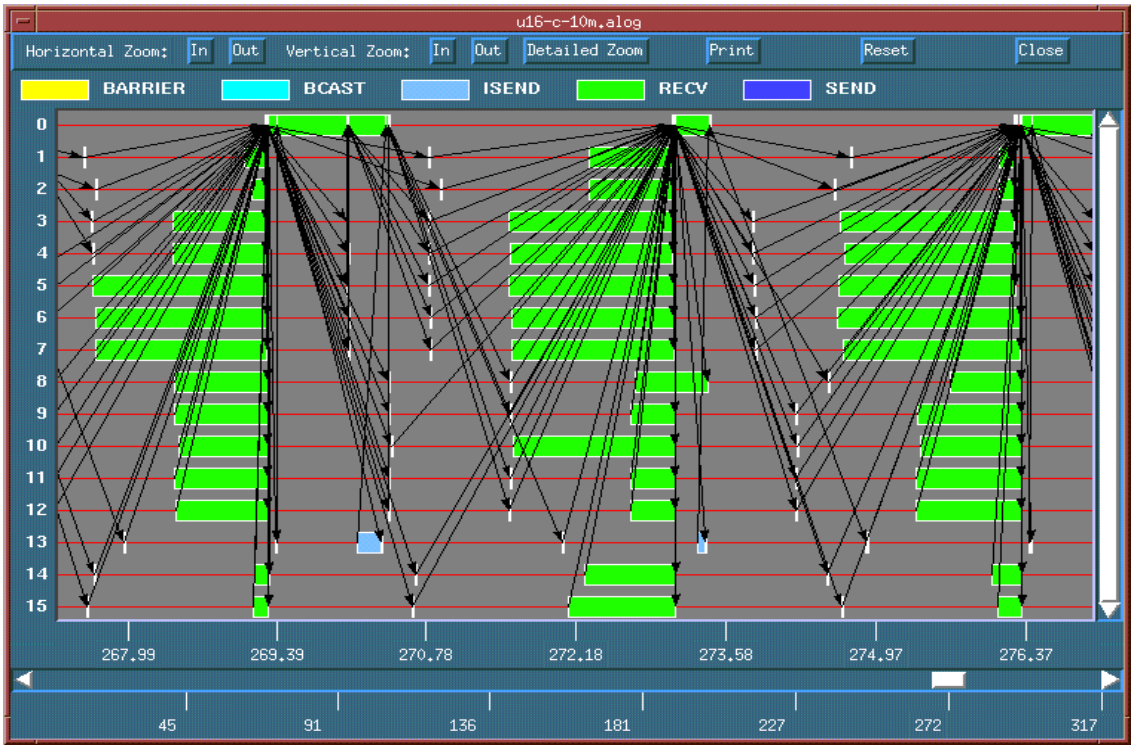


Figure 35. Final message pattern for 16 processors using Prob. B.

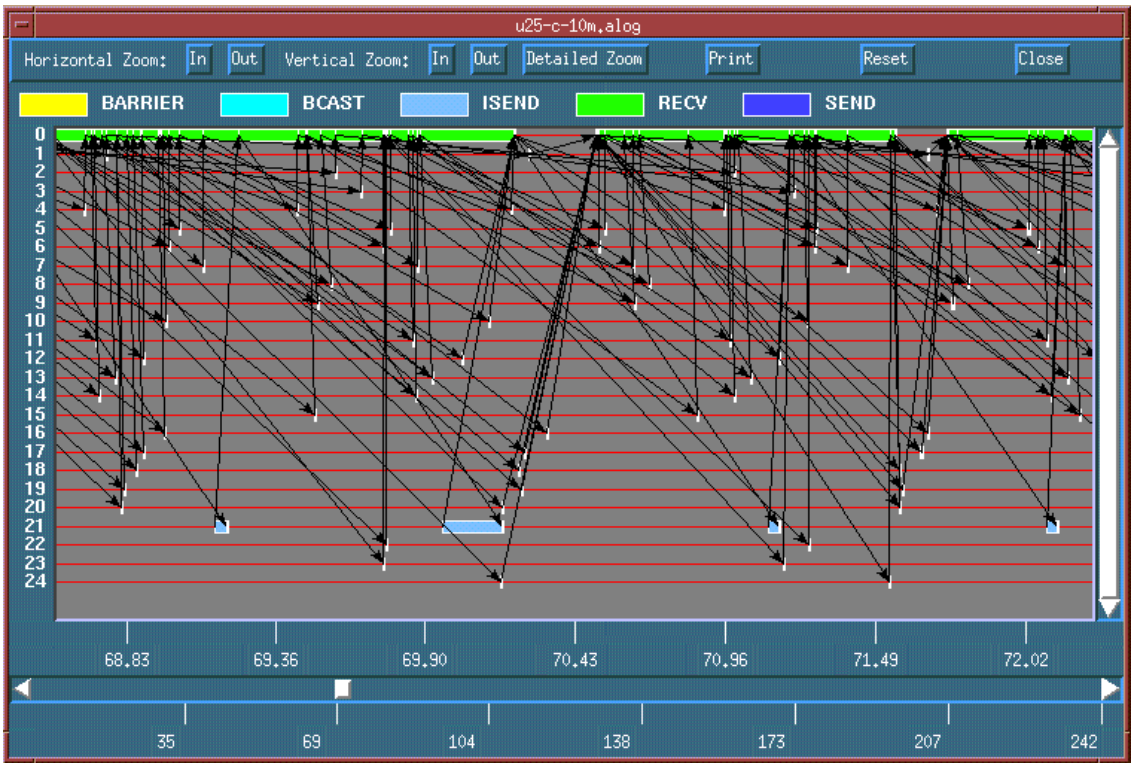


Figure 36. Initial message pattern for 25 processors using Prob. B.

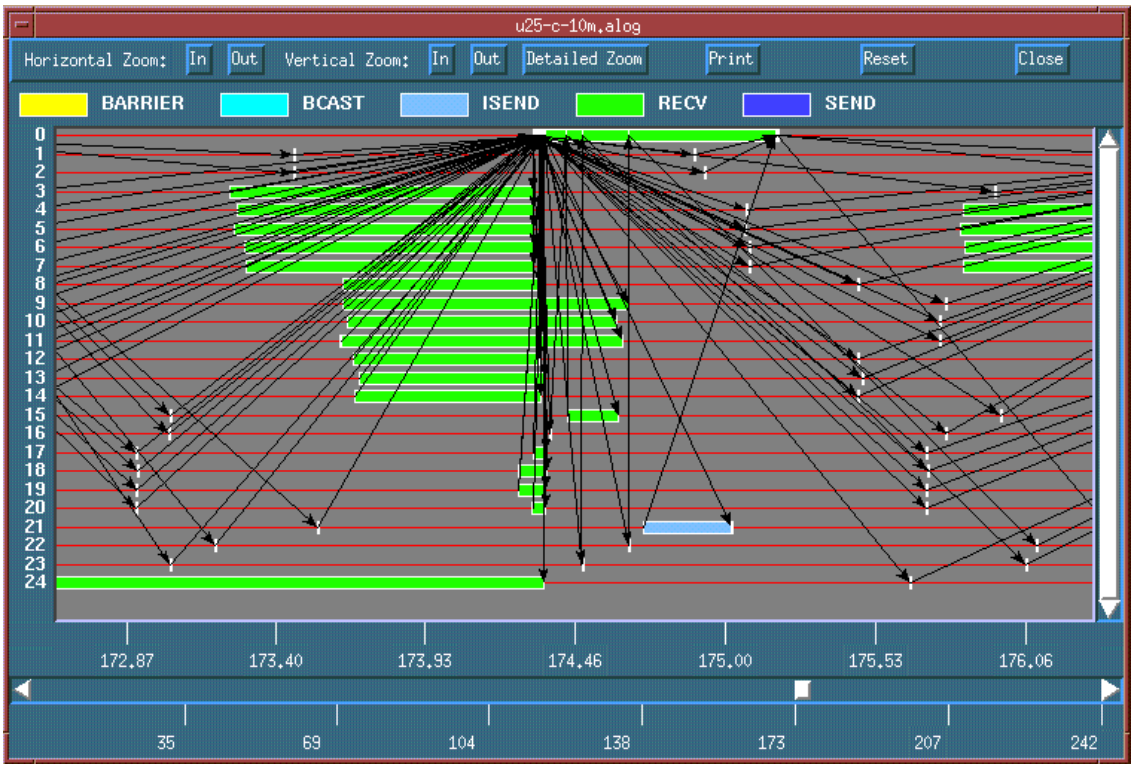


Figure 37. Final message pattern for 25 processors using Prob. B.

Performance

The parallel DIRECT code was also run on the Ross parallel cluster at Sandia National labs to examine the relative efficiency of the code as the number of processors is increased for a more complicated problem. The Ross cluster is a homogeneous collection of thousands of processors with a homogeneous network. Using this cluster, the parallel DIRECT codes were tested using 20, 40 and 80 processors each. In order to generate more points per iteration and reduce the masking effect seen on the quartic function, the twenty dimensional Griewank function was used (d = 1000, $\mathbf{x_U}$ = [100,900] with $\mathbf{x_L} = \mathbf{x_U}$-1000). The evaluation time was increased by performing 700,000,000 floating point multiplications at each evaluation. (7 times as many as with Prob. B) The optimizers stopped once the smallest box reached a minimum box diameter of 0.0001 times the original box side length. Three versions of the parallel DIRECT algorithm were used.

- Original DIRECT – Optimization waits for all points to be analyzed and boxes divided before selecting the next set of potentially optimal boxes
- Continuous DIRECT - When queue of points to analyze is empty new potentially optimal boxes are selected without waiting until all points have finished being analyzed. Two different accelerated starting methods were used.
- Hybrid DIRECT – Starts the optimization with the Original DIRECT algorithm and after 5 iterations switches to the Continuous DIRECT

Table 15 gives the performance of the four versions of DIRECT on this problem. The global optimum function value is 0.0. The continuous DIRECT version using the accelerated start method 2 reached the best solution and required the fewest function evaluations for any number of processors.

Table 15. Performance of the parallel DIRECT algorithms on a 20DV Griewank function.

	Number of Processors	Time Required (sec)	# of Iterations	Function Evaluations	Solution Found
Original DIRECT	20	6398	126	35751	.008
	40	3515	126	35751	.008
	80	1790	126	35751	.008
Continuous DIRECT-Method 1	20	7566	130	37133	.288
	40	4502	130	38199	.288
	80	2190	133	38043	.288
Continuous DIRECT-Method 2	20	6251	128	32996	.000
	40	2925	129	27824	.000
	80	1885	169	29324	.000
Hybrid DIRECT	20	6180	126	35751	.008
	40	2994	132	36861	.328
	80	1595	131	35709	.328

The optimization with a single processor takes about 117 thousand seconds. Figure 38 shows the relative analysis efficiency of the four parallel versions of DIRECT. The hybrid DIRECT analyzed the most points per second with the continuous DIRECTs analyzing the least.

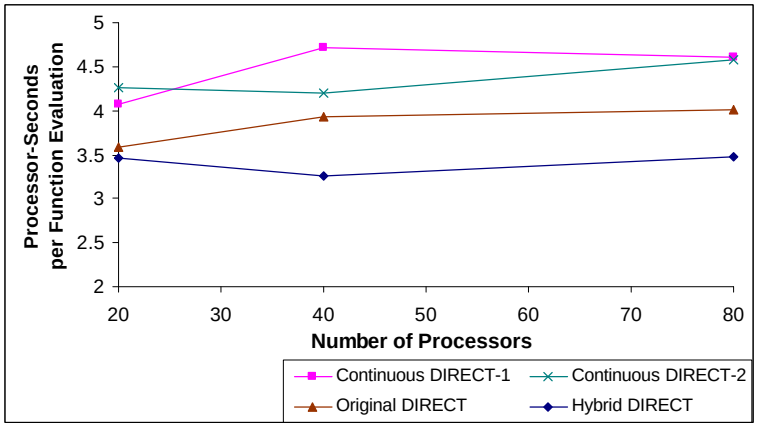


Figure 38. Comparison of computational efficiency of the three parallel DIRECT algorithms (from data from Table 16).

The continuous DIRECT algorithm using starting method 2 was the most efficient algorithm for this problem, while the continuous algorithm using starting method 1 was the worst version of DIRECT for this test problem. The continuous algorithm was

expected to be less efficient than the original algorithm since there is a chance that the boxes selected for division are not as good as they would have been if the algorithm waited until all of the points were finished. Using the second starting method changed the distribution of the initial search which allowed DIRECT to reach a better optimum with fewer function evaluations. The number of points analyzed per second per processor was lower for either continuous DIRECT algorithm than the original DIRECT.

The hybrid version of DIRECT was the most efficient at analyzing points for any number of processors. As the number of processors increased however, the algorithm did not locate as good an optimum as the continuous DIRECT but still required less time to analyze about the same number of points as the other optimizers. This is because there are more unfinished points at the end of each iteration as the number of processors increases. This causes DIRECT to divide more boxes which are not potentially optimal. The hybrid version located a better optimum than the first continuous version of DIRECT with 20 processors but was worse for more processors. This indicates that the initial search is vital, even for this problem. However, with a large number of processors, switching to the continuous DIRECT requires dividing so many extra boxes to create enough points that it eliminates the benefit of starting with the original DIRECT.

This test illustrates the importance of the initial iterations on the DIRECT algorithm. Three different starting conditions lead to three different optimums and required different amounts of time for the same number of processors. The original DIRECT algorithm places the best points in the biggest boxes and divides them preferentially from the first iteration. The first accelerated start method divides boxes into the same shapes as the original DIRECT but lacks the information on the function

values at the center of each box to put the biggest boxes in the best region of the design space. As a result, the biggest boxes do not necessarily have good function values at their centers, which places more boxes on the convex hull and increases the number of points analyzed at the beginning of the optimization. The second accelerated starting method makes all of the boxes the same size and shape. This causes DIRECT to divide only the best box finished at the end of the first iteration and does not make boxes with poor function values at the center potentially optimal at the beginning simply because they are large. This emphasizes zeroing in on the first good local optimum and results in the highest efficiency for this test problem. This method can also reduce the number of new points generated from each box division. DIRECT only divides the longest dimension of each box and the boxes may never be square after the initial division. This means that the boxes may never be divided on every dimension in one iteration. The efficiency of this starting method could also have been due to a lucky point selection at the beginning.

Alternate Convex Hull Calculation

As Figures 31 through 37 show, the DIRECT processor alternates between idle and busy periods that correspond to message handling and point generation respectively. The work could potentially be more efficiently balanced by interlacing the two functions. Many of the boxes can be eliminated from consideration as potentially optimal for the current iteration without waiting for the new points to be finished. The arrangement of the DIRECT code can be changed so that when there are no messages coming in to the main processor it will be calculating the convex hull of the existing boxes. As each new box is finished, it can be added to the boxes being considered for that iteration. Thus, much of the work for calculating the new set of potentially optimal boxes will already be finished when the queue runs out of points to analyze. This can reduce the idle time for

the analysis processors for problems where the analysis only takes a few seconds. The current arrangement of the Parallel DIRECT code is outlined in Figure 39.

A. Minimize function
    1. Set up problem, generate initial points and distribute first sets of points to the analysis processors
    2. Wait for a completed point from an analysis processor
    3. Send a new point to the analysis processor and store the finished point
    4. Divide the boxes within a parent box when all of the points from that parent box have been analyzed
    5. If the queue is not empty, go to A.2. Else:
        i. Check for convergence
            1. If converged stop
        ii. Call Sorting subroutine
        iii. Call Divider subroutine
        iv. Distribute needed points
        v. Go to A.2

B. Potentially Opotimal Box Selection Subroutine
    1. Locates potentially optimal boxes based on Graham Scan algorithm or Jones' constrained algorithm

C. Divider Subroutine
    1. Pick new points to analyze in potentially optimal boxes
    2. Place new points in queue

Figure 39. Outline of existing parallel DIRECT code arrangement.

This version waits until the queue of points to be analyzed is empty before sorting the finished points to select potentially optimal boxes. In order to better distribute the work on the DIRECT processor, a modified DIRECT code is given in Figure 40.

A. Minimize function
    1. Set up problem, generate initial points and distribute first sets of points to the analysis processors
    2. Call Sorting subroutine
    3. Call Divider subroutine
    4. Distribute needed points

B. Potential Optimal Box Selection Subroutine
    1. If results from analysis processors are waiting call Receive subroutine
    2. If the convex hull has been located and the queue is not empty go to B.1
    3. Eliminate a portion points not on the convex hull from consideration
    4. If the complete convex hull has not been located go to B.1
    5. Check for convergence and if converged stop

C. Receive Subroutine
    1. Accept result and send a new point to the analysis processor if available
    2. Divide the boxes within a parent box if all of the points from that parent box have been analyzed and add them to the list of points to consider for the convex hull
    3. If the queue is empty or there are no more messages pending, return to the Potential Optimal Box Selection subroutine, else go to C.1

Figure 40. Modification to the DIRECT code to reduce idle time.

This modification could potentially reduce the amount of time spent computing the convex hull after the queue of points waiting to be analyzed is emptied. This can reduce the amount of idle time on the analysis processors, particularly for problems where the analyses take only a few seconds.

Algorithm Status

The efficiency of the various parallel algorithms depends on the properties of the problem being optimized. The original DIRECT algorithm requires the same number of analyses as the sequential version with the differences in efficiency coming from the amount of idle time on the analysis processors. As more points are divided at each iteration, the frequency of idle processors decreases and the computational efficiency of the algorithm increases. The number of potentially optimal boxes increases when the problem does not have local optima which have much lower function values than the rest of the design space. These deep local opotima tend to mask most of the points from the convex hull and reduce the global exploration of the algorithm. Similarly, as analysis take longer, the time spent calculating the next set of potentially optimal points has a smaller relative impact on the total time. The relative effect of the overhead of the algorithm is reduced.

The efficiency of the continuous DIRECT algorithm is affected by similar parameters but in different ways. When more boxes are divided at each iteration, the selection of potentially optimal boxes at each iteration is missing a smaller percentage of the new boxes. This reduces the number of boxes which are divided when they normally would not have been. This also reduces the frequency of any idle time on the processors. Similarly, increasing the time required for each analysis gives the master processor more time to generate the next set of boxes to divide before the analysis processors run out of

work. This reduces the likelyhood of idle processors, particularly at the end of the optimization when the calculation of potentially optimal boxes takes the longest.

Based on this limited test, the continuous DIRECT algorithm still has some problems to overcome. First, the search is not as efficient when it has to calculate the potentially optimal boxes without having all of the information that the original DIRECT algorithm would have. It may be useful to recompute the potentially optimal boxes after the missing points have been completed. The queue of waiting points can then be modified to remove boxes which should not have been divided at that iteration and add points from the boxes that needed to be divided. Second, the method for generating the initial set of points to analyze needs to be examined more closely. Third, the work for the DIRECT processor is very poorly distributed. Finding a way to utilize the idle time in the DIRECT processor can increase the efficiency of the search and possibly reduce idle time on the analysis processors.

Chapter 6
CONCLUSIONS

The DIRECT algorithm has demonstrated itself to be an effective global optimization algorithm. It is capable of handling noisy functions where gradient calculations are unreliable or impossible to compute. While small fluctuations in the function values over the design space can be difficult for gradient based optimizers to pass through, they have only a minor effect on the performance of DIRECT. In addition to handling noise well in an otherwise local optimization problem, DIRECT continues to explore the entire design space up until it converges to an optimum. This broad search makes DIRECT an effective optimizer in problems where there are multiple widely separated local optima scattered throughout the design space. The efficiency of DIRECT on these types of problems compared favorably with multistart local optimizations both with approximations to the objective function and without. The advantage of the DIRECT algorithm comes in the lack of required user provided parameters. This makes the algorithm easier to use on problems where prior expert knowledge of the characteristics of the problem are unavailable. The algorithm is more robust and systematic in its search and does not rely on stochastic parameters such as random starting points.

In addition to the standard DIRECT algorithm, this dissertation examined three modifications to the DIRECT algorithm. The DIRECT-BP algorithm was proposed as a way to increase the robustness of the search in the neighborhood of the best local

optimum located as the optimization progresses in exchange for additional function evaluations. This modification forces DIRECT to analyze more points in the neighborhood of the best point found so that the region surrounding this point will be analyzed on a finer mesh than might otherwise occur. This provides a measure of confidence that DIRECT has converged to a local optimum as opposed to stopping on a boundary of the sub boxes formed by DIRECT. DIRECT-BP was found to be only moderately more expensive than DIRECT for most problems and, in addition to the increase in robustness of the algorithm, sampling the design space on a finer mesh on all sides of the best point can be useful after the optimization to determine the characteristics of the design space in that region.

A multifidelity version of DIRECT was proposed which allows the optimization to replace some of the expensive high-fidelity analyses with cheaper low-fidelity analyses while still retaining most of the accuracy of the high-fidelity analysis throughout the design space. For the test problems used here, the multifidelity DIRECT showed some improvement over the single fidelity version of DIRECT. However, the experiments also indicated that stopping when the smallest box reaches a specified size does not work well for this version of DIRECT. Due to the small difference between the corrected function values and the high fidelity function values, the smallest boxes selected for division were constantly changing which slowed the reduction in size of the smallest box. In addition, the correction ratio used here is sensitive to the relative magnitude of the high- and low-fidelity functions near the global optimum. The test problems used here had a proportionately large difference in their function values which increased the relative errors in the corrected function values. However, this research suggests that the

multifidelity version of DIRECT has promise as an effective optimization scheme for complex problems.

The parallel version of the DIRECT code was an attempt to distribute the work load over a larger number of processors to enable the optimization of problems which are too time consuming for a single processor. The algorithm was broken into a master-slave implementation with the master performing all of the work relating to the DIRECT algorithm while the slave processors performed all of the design analysis work. Three versions of the parallel code were generated which successfully distributed the work over a number of processors and contained no intrinsic restrictions on the number of processors which could be employed. One continuous version was successful in decreasing the amount of time required and finding a better solution than the parallel implementation of the original DIRECT algorithm. The parallel implementations given here shows promise as effective ways of distributing work over a large number of processors while retaining the effectiveness of the original DIRECT algorithm.

Several areas of the continuous DIRECT algorithm can potentially be modified to increase the efficiency of the algorithm. The queue of points to analyze can be examined as new points are finished to see if they are still potentially optimal. The convex hull calculation can be interlaced with the message handling on the master processor. This will take advantage of the idle time the master processor currently spends waiting for points to be analyzed and reduce the time between the queue running out of points and the new points are ready for analysis. Also, the generation of the initial set of points needs to be examined for ways to reduce the effect of not having some points when calculating the potentially optimal boxes.

APPENDIX A
Sequential DIRECT C++ Library

The DIRECT algorithm was initially coded in Fortran and later converted to C++ to take advantage of dynamic memory management and class structuring. The final version of the C++ DIRECT code is designed to interface with the COLIN optimization framework written at Sandia National Laboratories. The C++ code is designed to execute all of the different versions of DIRECT used over the course of this research but the multifidelity version. The C++ libraries can be found for release under a General Public License through Sandia National Laboratories. At the time this disseretation was written, the SGOPT software package which contains the DIRECT libraries could be located at the following link: www.cs.sandia.gov/SGOPT. The code can also be found in the DAKOTA package through Sandia. An independent implementation of DIRECT and DIRECT-BP which is not linked to COLIN is written in C++ and is given in Appendix B. The parallel DIRECT codes, also written in C++, are given in Appendix C and the multifidelity versions of DIRECT are written in Fortran and are given in Appendix D.

The C++ library is designed to use the COLIN interface to set up the optimization problem as well as perform the analysis calculations and keep track of the best point found. COLIN is a templatized version of the SGOPT toolkit distributed by Sandia National Laboratories. It is a way of interfacing between arbitrary analysis codes and optimizers. The different versions of DIRECT are selected by setting the values of different parameters when the optimization problem is set up. Additional parameters control the stopping behavior of DIRECT. These parameters are given in Table 14. All

parameters are set at run time by functions native to COLIN. Set up is performed by a simple block of code which designates the analysis function, the optimization problem, the optimizer used and any parameters needed for the optimizer or analysis function. A simple test problem code is given below.

Table 14. Parameters in Object Oriented DIRECT library code.

Parameter	Effect
Method	(1) Constrained DIRECT or (2) Unconstrained/Penalty function DIRECT
DIRECT-BP	(False) no neighborhood search, (True) neighborhood search
Aggressive	If method = 2; (True) performs aggressive search suggested by Baker et al. (**) (False) searches for the convex hull
division	(0) divides potentially optimal boxes on all long sides each iteration (1) divides a box on only 1 dimension per iteration
Ratio	Value of Γ for DIRECT-BP
Stopcond	(1) stops when smallest box is smaller than minsize (2) stops after maxit iterations (3) stops after FElimit analyses have been performed (4) (DIRECT-BP only) stops when the best box is smaller than minsize2 and the neighborhood is balanced
Minsize	Normalized diameter of smallest box before stopping for stopcond 1
Maxit	Maximum number of iterations before stopping for stopcond 2
FElimit	Maximum number of analyses before stopping for stopcond 3
Minsize2	$c\gamma$ from DIRECT-BP description
Gamma	Smallest allowed ratio of γneighbor diameter to best box diameter

Example Problem Code

```
//
// OptSolver test commandshell interface
//

#include "stl_auxillary.h"
#include <iostream.h>
#include "DIRECT.h"

const double pi = 3.141592654;

double func(vector<double>& vec)
{
double val;
for (unsigned int i=0; i<vec.size(); i++)
// Griewank Function
          double x[21];
      for (l=1;l<=dimen;l++)
        {
             x[l] = vec[l];
             val += pow(vec[l],2);
            }
      val =val/500;
      val += (1.-cos(x[1])*cos(x[2]/sqrt(2.))*cos(x[3]/sqrt(3.))*
        cos(x[4]/2.)*cos(x[5]/sqrt(5.))*cos(x[6]/sqrt(6.))*
        cos(x[7]/sqrt(7.))*cos(x[8]/sqrt(8.))*cos(x[9]/3.)*
        cos(x[10]/sqrt(10.))*cos(x[11]/sqrt(11.))*cos(x[12]/sqrt(12.))*cos(x[13]/sqrt(13.))*
        cos(x[14]/sqrt(14.))*cos(x[15]/sqrt(15.))*cos(x[16]/4.)*
        cos(x[17]/sqrt(17.))*cos(x[18]/sqrt(18.))*cos(x[19]/sqrt(19.))*
        cos(x[20]/sqrt(20.)));
return val;
}

int main()
{
OptProblem<vector<double> > prob;
OptSetup(prob,func);

vector<double> lbounds(14), ubounds(14);
lbounds << 0.0;
ubounds << 3.0;
prob.set_real_bounds(ubounds,lbounds);


DIRECT opt;
```

```
opt.set_problem(prob);
opt.set_option("dimension",14);
opt.set_option("ncon",0);
opt.set_option("DIRECT_BP",true);

opt.reset();
opt.minimize();
cout << "finished\n\n"  << "minimum = " << best_func;
return 0;
}
```

APPENDIX B
DIRECT and DIRECT-BP

This implementation of the sequential DIRECT and DIRECT-BP codes is designed as a stand alone code. The analysis functions are included at the end of the code and the code must be recompiled after changing any of the parameters. The code is an earlier version of the code included in the C++ DIRECT Library at Sandia.

Sequential DIRECT and DIRECT-BP code

```
/*******************************************************/
/*      Direct Optimizer Localized Version             */
/*      Steven Cox                                     */
/*      September 2001                                 */
/*                                                     */
/*      This is local version 2 where Basis is called  */
/*      multiple times, adding the needed V until      */
/*      a minimum positive basis is found              */
/*******************************************************/

#include <stdio.h>
#include <math.h>
#include <iostream.h>
```

```
#include <stdlib.h>
#include <cstdlib>
#include <list>
#include <time.h>

const double pi = 3.14159265358979323846;

const int nbox = 60001;                         //absolute limit on number of boxes
const int dimen = 10;
const int ni = 0;
const int d2 = dimen+2;

const int print = 1;                            //prints all points and their value
const int agressive = 0;                        //used in Graham routine to select
method
const int division = 0;                         //divide on all sides (0) or one side
(1)
const bool DIRECT_BP = true;                    //perform DIRECT-BP local search

const double minsize1 = 0.0001;                 //stopping box size
const double minsize2 = 0.001;                  //ALT min box size
const double radrem = 0.03;                     //radius of removal
const double mingain = 0.0000001;               //epsilon described in Jones' paper,
min potential improvement over fmin

const double ratio = 45. ; //3.*sqrt(dimen);
const double minsize3 = sqrt(dimen)/ratio/3;    //min box size needed to check for
neighbors

const int nopt = 800;                           //maximum number of potentially
optimal boxes
const int stopcond = 5;             //1 = stop on min box size, 2 = remove boxes less
than minsize2
                                    //3 = remove radius around minsize2 boxes. 4 =
stop on iterations
                                    //5 = local criteria of small positive basis around
local opt
const int maxit = 100;
const int loc = 15;

const int nbest = 1000;
const double mindist = 0.05;

double V[nbox][dimen+1];
```

```
struct bvector  {
  bool is_p_basis ;
  double V1[dimen+1];
  };

int numb, nold;                                  //number of boxes, max number of boxes
int final;

double xopt[dimen+1], x[dimen+1];                //point to pass to DOT
double center[nbox+1][(2*dimen+3)+1];            //value, position and dimension of each
point
int points[15+1], t[dimen+1];
int duals[nbox][2];                              // keeps track of overlapping boxes

int optimal[nopt+1], best[nbest+1], dropped;           //potentially optimal points
int totcalls, calls, count, converg, nact, itter, nsort;   //counters for FE used by DOT

double xl[dimen+1], xu[dimen+1], xbest[dimen+1], e[dimen+1], size[dimen+1];
          //current best box and function value
double fbest, dist, order2b[nbox+1], local[loc][d2], g[ni+1];

int index1[nbox+1], timeS, timeF;

std::list<int> hull;
std::list<int>::iterator hcount;

FILE *fp1, *fp2, *fp3, *fp4;

// define function prototype

void Eval();
double Func(double *y);
void Graham();
void Divider();
void con(double *y);
void Neighbor(int order[], double order2[], int initial);
bvector Basis(int nvec);
extern "C" void loccntl_(int &, int &, double &, double &, double &, double &, int &);
extern "C" double ffunc_(double *y);
extern "C" void fcon_(double *y, double g[]);

/*****************************************************************/
/* Main Program                                                  */
/*****************************************************************/
void main(void)
{
```

```
  int k, kr, i, j, l, bad, dropped;
  double func, t;


// open file "high.dat" here <-----------------------------
        fp1 = fopen("highloc.dat", "a+");
 printf("loop\n");

        srand(5);
//      xu[1] = rand()/32767.0;

// open file "points.dat" here <------------------------
        fp2 = fopen("points.dat", "w+");

// open file "points.dat" here <------------------------
        fp3 = fopen("plot.dat", "a+");

   timeS = (int)time(0);

        for (kr = 1; kr <= 1; kr++)
  {
// Setup cycles
        optimal[1] = 1;
        nold = 1;
        numb = 1;
        final = nbox-2*dimen;
        converg = 0;
        fbest = 9999090;
        totcalls = 0;
        itter = 0;
        count = 0;
        nsort = 0;
        nact = 0;

        if(division == 1)
           final = nbox -2;
// Box Function
/*      for (i = 1; i <= dimen; i++)
          {
                xu[i] = 2;
        xl[i] = -2;
        t = rand();
                e[i] = 0.2+(t/RAND_MAX)*0.2;// 0.3; //
          }
*/
printf("loop 0\n");
```

```
// Test Function
        for (i = 1; i <= dimen; i++)
          {
                xu[i] = 3;
        xl[i] = 0;
          }

// Griewank Function
/*      for (i = 1; i <= dimen; i++)
          {
        t = rand();
                xu[i] = 100.+(t/RAND_MAX)*300;
        xl[i] = xu[i]-500;
          }
*/
//
          for (i = 1; i <= nbox; i++)
           {
                for (j = 1; j <= (dimen+3); j++)
        center[i][j] = 0;
                order2b[i] = 0;
                index1[i] = 0;
                center[i][2*dimen+2] = 1;
                center[i][2*dimen+3] = 1;
           }

// find sizes and center points
          for (i = 1; i <= dimen; i++)
           {
                center[1][i+1] = (xu[i]+xl[i])/2;               //center point of box
                center[1][i+dimen+1] = 1;                       //each box is 1 unit long in
each dimension
                size[i] = (xu[i]-xl[i]);                //length of box side
           }
                Eval();                                         // analyse the function value
at the center

          do
           {
                itter++;
// locate potentially optimal points
                Graham();

// Divide Potentially optimal boxes
                Divider();
```

```
            } while ((numb < final) && (converg != 1));
          if (print == 1)
            {
           fprintf(fp2, "numb =%3d   itter =%3d    converg =%1d\n", numb, itter,
converg);
                for (i = 1; i <= numb; i++)
                  {
                    fprintf(fp2, "%5d  %18.15f",i,center[i][1]);
                    for (j = 2; j <= (2*dimen+1); j++)
                         fprintf(fp2, "  %9.4f", center[i][j]);
                fprintf(fp2, "\n");
             }
            }

// open file "best.out" here <------------------------------
            fp3 = fopen("best.out", "w+");

            for (i = 1; i <= nbest; i++)
                  best[i]=0;

/***************Replace removed center points***************
if ((stopcond == 2) || (stopcond == 3))
  {
    for (i = 1; i <= loc; i++)
      {
          for (j = 1; j <= loc; j++)
            {
              if (best[j] == 0)
                 {
                     best[j] = (int)local[i][dimen+1];
                   break;
           }
              if (center[(int) local[i][dimen+1]][1] < center[best[j]][1])
                 {
                     for (k = loc; k >= (j+1); k--)
                   best[k] = best[k-1];
                     best[j] = (int)local[i][dimen+1];
                   break;
                 }
            }
      }
  }
************************************************/

   for (i = 1; i <= numb; i++)
      {
```

```
    for (j = 1; j <= nbest; j++)
      {
       if (best[j] == 0)
          {
               best[j] = i;
            break;
     }
       if (center[i][1] < center[best[j]][1])
          {
               for (k = nbest; k >= (j+1); k--)
            best[k] = best[k-1];
               best[j] = i;
            break;
          }
      }
}

for (j = 1; j <= loc; j++)
      points[j] = 0;

points[1] = 1;
k = 0;
for (j = 1; j <= nbest; j++)
{
    if (k == loc)
    break;
  calls = 0;
  if (k != 0)
    {        bad = 0;
    for (l = 1; l <= k; l++)
         {
                dist = 0;
          for (i = 2; i <= dimen+2; i++)
                  dist += pow((center[best[points[l]]][i]-center[best[j]][i])/size[i-1],2);
          dist = sqrt(dist);
                if (dist < mindist)
                     {
                      bad = 1;
                      break;
                     }
               }
            if (bad)
               continue;

    points[k+1] = j;
  }
```

```
        k++;

    for (i = 1; i <= dimen; i++)
         xopt[i] = center[best[j]][i+1];

        int n = dimen;
        int cn = ni;
        double md = mindist;
        func = center[best[j]][1];

        printf("%4d  %7.4f :",j,func);
        for (i = 1; i <= dimen; i++)
             printf( " %6.3f",xopt[i]);
        printf("\n");

//*****************************
//       loccntl_(n,cn,xopt[1],size[1], md, func, calls);


/*      dist = 0;
    for (i = 1; i <= dimen; i++)
         dist += fabs(center[best[j]][i+1]-xopt[i]);
*/
    if (func < fbest)
      {
        fbest = func;
        for (i = 1; i <= dimen; i++)
                xbest[i] = xopt[i];
      }
      totcalls += calls;
    }

// close file "best.out" here <-------------------------------
                fclose(fp3);

        timeF = (int)time(0);

    if ((stopcond==1)||(stopcond==4)||(stopcond == 5))
        {
          fprintf(fp1,"\n %4d %6d %5d %9.4f %4d %8d",kr, numb, totcalls, fbest, itter,
(timeF-timeS));
          for (i=1; i<=dimen; i++)
             fprintf(fp1," %8.4f",xbest[i]);
          printf("\n %4d %6d %4d %9.4f",kr, numb, itter, fbest);
          for (i=1; i<=dimen; i++)
             printf(" %8.4f",xbest[i]);
```

```
            printf("\n");
          }

      if (stopcond==2)
          {
            dropped =0;
            for(i=1; i<=loc; i++)
              dropped += (int)local[i][dimen+2];
            fprintf(fp1,"\n %4d %6d %4d %9.4f %6d",kr, numb, dropped, fbest, itter);
            for (i=1; i<=dimen; i++)
               fprintf(fp1," %8.4f",xbest[i]);
          }

      if (stopcond==3)
          {
            fprintf(fp1,"\n %4d %6d %4d %9.4f %6d",kr, numb, count, fbest, itter);
            for (i=1; i<=dimen; i++)
               fprintf(fp1," %8.4f",xbest[i]);
          }

    }

          fprintf(fp3,"\n");
// close file "plot.dat" here <---------------------------
          fclose(fp3);
// close file "points.dat" here <--------------------------
          fclose(fp2);
// close file "high.dat" here <--------------------------------
          fclose(fp1);
}


/*************************************************************************
*/
/* Subroutine to evaluate the function value at a point                  */
/*************************************************************************
*/

void Eval()
{
          double y[dimen+1];//, g[ni+1];
          double pen;
          int i, j, k;

          pen = 0.51;
                                    //penalty function multiplier
```

```
  for (i = nold; i <= (numb+2*dimen); i++)
    {
     if (center[i][dimen+2] != 0)
       {
          if (fabs(center[i][1]) < 0.00001)          //check if point has been analyzed
           {
         for (j = 1; j <= dimen; j++)
                    y[j] = center[i][j+1];                                          //put
box center into y
             center[i][1] = Func(y);
                con(y);

             for(k=1;k<=ni;k++)
               {
                 if (g[k] > 0)
                    center[i][1] += pen*g[k];
               }
           }
        }
     else
           break;
    }

}

/*********************************************************************/
/* Subroutine Graham's scan                                          */
/*********************************************************************/

void Graham()
{
  double order2[dimen*nopt+1];
  double z, o, fmingain, Lmin, Ldes;
  int order[dimen*nopt+1];
  int yy, initial, m, i, j, k, l;

// sort points by distance

  printf("\n%d  %f %d\n",numb,order2b[1],itter);

  for (i = 1; i <= (dimen*nopt); i++)                    //zero the local order arrays
    {
        order[i] = 0;
      order2[i] = 0;
    }
```

```
  for (i = 1; i <= nact; i++)
    {
          yy = optimal[i];                                    //optimal contains the boxes
that were divided last round
          z = 0;
          for (k = (dimen+2); k <= (2*dimen+1); k++)
            z += pow(center[yy][k],2);                        //get the sizes of the box after
dividing
          z = sqrt(z);                                        //distance to edge for box i

        for (k = 1; k <= nsort; k++)
          {
              if (index1[k] == yy)                            //find out where the box was
in the list
                    {
            for (l = k; l <= (nsort-1); l++)                  //index1 keeps a list of the boxes in
order of size and value
                 {
                       index1[l] = index1[l+1];
                  order2b[l] = order2b[l+1];
                 }
                index1[nsort] = 0;
                      order2b[nsort] = 0;
                      break;
                     }
          }

        for (j = 1; j <= nsort; j++)                          //put the box back in where it
belongs
          {
               if (index1[j] == 0)
               {
                       index1[j] = yy;
                       order2b[j] = z;
                break;
           }
            if (z < order2b[j])
           {
                      for (k = nsort; k >= (j+1); k--)
                        {
                             index1[k] = index1[k-1];
                    order2b[k] = order2b[k-1];
                        }
                      index1[j] = yy;
                      order2b[j] = z;
```

```
                    break;
            }

                if ((z == order2b[j]) && (center[yy][1] < center[index1[j]][1]))
            {
              for (k = nsort; k >= (j+1); k--)
                        {
                            index1[k] = index1[k-1];
                            order2b[k] = order2b[k-1];
                        }
                    index1[j] = yy;
                    order2b[j] = z;
                    break;
                }
             }
        }

  for (i = (nsort+1); i <= numb; i++)                    //now do the rest of the new
points
    {
          z = 0;
       for (k = (dimen+2); k <= (2*dimen+1); k++)
            z += pow(center[i][k],2);
       z = sqrt(z);                                      //distance to edge for box i

       for (j = 1; j <= numb; j++)
         {
             if (index1[j] == 0)
           //if at end of list put box here
           {
                    index1[j] = i;
                    order2b[j] = z;
                    break;
           }
            if (z < order2b[j])
           //if smaller than this entry insert here
           {
             for (k = i; k >= (j+1); k--)
               {
                        index1[k] = index1[k-1];
                  order2b[k] = order2b[k-1];
               }
                    index1[j] = i;
                    order2b[j] = z;
                    break;
           }
```

```
          if ((z == order2b[j]) && (center[i][1] < center[index1[j]][1])) //if better and same
size insert here
          {
           for (k = i; k >= (j+1); k--)
                    {
                     index1[k] = index1[k-1];
                     order2b[k] = order2b[k-1];
                    }
              index1[j] = i;
              order2b[j] = z;
              break;
          }
         }
    }

   nsort = numb;

/**************end sorting*************/
  converg = 0;
/**************Stopcond 1***************/
  if ((stopcond == 1) && (order2b[1] < minsize1))                    //stop when
smallest box reaches certain size
      {
       converg = 1;
       return;
      }                                                                  //stop after this
loop if boxes small enough

/**************Stopcond 2***************
  m=0;
  do {
     if ((order2b[1] <= minsize2) && (stopcond == 2))                //removing boxes that
are too small
        {
          count ++;
          if (count > loc)
            {
              converg = 1;
              return;
            }

          local[count][dimen+1] = index1[1];
          for (i=1; i<=dimen; i++)
            local [count][i] = center[index1[1]][i+1];
          local[count][dimen+2] = 1;
```

```
        for (i=1; i<=nsort; i++)
             {
              order2b[i] = order2b[i+1];
              index1[i] = index1[i+1];
             }
           index1[nsort] =0;
           order2b[nsort] =0;
         }

/***************Stopcond 3****************

    else if ((order2b[1] <=minsize2) && (stopcond == 3))    //removing boxes that are too
small
         {
        count ++;
        if (count > loc)
          {
           converg = 1;
           return;
          }

        local[count][dimen+1] = index1[1];
        for (i=1; i<=dimen; i++)
          local [count][i] = center[index1[1]][i+1];
                    for (j=2; j<=nsort; j++)
             {
                  dist =0;
                  for (i=(2); i<=(dimen+1); i++)
                    dist += pow((center[j][i]-center[index1[1]][i])/size[i], 2);
                  dist = sqrt(dist);
                  if (dist < radrem)
                    {
                     for (i=j; i<=nsort; i++)
                           {
                             order2b[i] = order2b[i+1] ;
                             index1[i] = index1[i+1];
                           }
                     index1[nsort] =0;
                     order2b[nsort] =0;
                    }
             }
           for (i=1; i<=nsort; i++)
             {
               order2b[i] = order2b[i+1] ;
                  index1[i] = index1[i+1];
             }
```

```
            index1[nsort] =0;
            order2b[nsort] =0;
          }
    else
         m =1;
  } while (m == 0);

/***************Stopcond 4***************/
  if ((stopcond==4)&&(itter>(maxit-1)))                  //stop when max number of iterations
exceeded
    {
     converg = 1;                                        //stop after this loop if enough iterations
    }
/**************End Stopconds**************/

// start agressive sorting of DIRECT

     m = 0;
     j = 2;
     k = 2;
     order[1] = index1[1];
     order2[1] = order2b[1];
     while (m==0)
       {
           if ((order2b[j]-.001*minsize1)>order2b[j-1])                  //if the box size
changes then index1[j] is potentially optimal
          {
            order[k] = index1[j];
            order2[k] = order2b[j];
            k++;
          }
        if ((j>numb)||(k>(dimen*nopt)))                  //exit if you reach the end of index1
or optimal
          m = 1;
        j++;
       }

     nact = k-1;
     initial = 1;

     if (agressive == 0)
          {
// find the convex hull
           i = 1;
           j = 2;
           k = 3;
```

```
// Conversion to sample more points
            if (itter > 0)
             {
               while(k <= nact)                          //remove right turns & keep left turns
                 {
                       o = order2[i]*(center[order[j]][1]-center[order[k]][1])
            -center[order[i]][1]*(order2[j]-order2[k])
            +(order2[j]*center[order[k]][1]-order2[k]*center[order[j]][1]);
            if (o > 0.0000000000000)
              {
               i = i+1;
                  j = i+1;
                        k = j+1;
              }
            else
              {
               for (l = j; l <= nact; l++)
                              {
                               order[l] = order[l+1];
                               order2[l] = order2[l+1];
                              }
                        order[nact] = 0;
                        order2[nact] = 0;
                        nact--;
                        if (i > 1)
                              {
                               k = i+1;
                               j = i;
                               i = i-1;
                              }
                        else
                              {
                               j = i+1;
                               k = i+2;
                              }
              }
                 }

// Remove left side of convex hull
            m = 0;
            i=1;
            while(m != 1)
             {
               if ((center[order[i+1]][1] < center[order[i]][1]) && (order[i+1] != 0))
                 i++;
```

```
            else
               m = 1;
             }
          fmingain = center[order[i]][1] - mingain*fabs(center[order[i]][1]);
          initial = i;

          m=0;
             while (m != 1)
                  {
                    Lmin = (center[order[initial+1]][1]-
center[order[initial]][1])/(order2[initial+1]-order2[initial]);
                    Ldes = (center[order[initial]][1]-fmingain)/order2[initial];

                 if ((Lmin <= Ldes) && (order[initial+1] != 0))
//               if ((center[order[initial+1]][1]-center[order[initial]][1]
//                  < 0) && (center[order[initial+1]][1] != 0))
              initial++;
               else
                    m=1;
              }
          }
    }
//added 2-28-01 to see if the Graham Scan routine is correct
          fprintf(fp2,"\norder & order2 iteration # %3d     After ends are removed.%3d\n",
itter, initial);
                  for (j = initial; j <= nact; j++)
                          fprintf(fp2, "  %4d  %9.5f %12.5f \n", order[j], order2[j],
center[order[j]][1]);
            fprintf(fp2,"\n");

//end addition
   printf("  %4d  %9.5f %14.8f \n", order[initial], order2[initial],
center[order[initial]][1]);

//print progression of best point
          fprintf(fp3," %d  %14.8f %10.7f : ",itter,center[order[initial]][1], order2[initial]);
          for (i=2;i <=dimen+1;i++)
                  fprintf(fp3," %10.5f",center[order[initial]][i]);
          fprintf(fp3,"\n");


/**********Check for small boxes with large neighbors ***********/
   if (DIRECT_BP && (order2[initial] < minsize3))
    {
          printf("calling neighbor\n");
          Neighbor(order,order2,initial);
```

```
     }

// Record Potentially optimal points
    if ((nact-initial) > nopt)
            nact = nopt+initial-1;
    for (i = initial; i <= nact; i++)
              optimal[i-initial+1] = order[i];
    for (i = nact+2-initial; i <= nopt; i++)
            optimal[i] = 0;
    nact -=  (initial-1);

//added 2-28-01 to see if the Graham Scan routine is correct
           fprintf(fp2, "optimal         iteration # %3d     After ends are removed.%3d\n", itter,
initial);
                    for (j = 1; j <= nact; j++)
                              fprintf(fp2, "  %4d  \n", optimal[j]);
             fprintf(fp2, "\n");
}

/*********************************************************************/
/* Subroutine to divide the potentially optimal boxes into smaller boxes                 */
/*********************************************************************/
void Divider()
{
  double lng, z;
  int divide[2*dimen+1], split[2*dimen+1], dim[dimen+1],net;
  int i, k, j, l, m, tmin;

  for (m = 1; m <= nopt; m++)
    {
      nold = numb;
                                    //start with first optimal point in "optimal"
         if (optimal[m] < 0.5)
           break;
      k = optimal[m];

//    printf(" box  %d is being divided, numb = %d\n",k,numb);
      lng = 0.0;
                            //only divide dimensions that are longer than the rest
      z = 0.0;
      tmin = 99999;
      for (i = 1; i <= dimen; i++)
        {
            dim[i] = 0;
          z += pow(center[k][i+dimen+1],2);
          if (center[k][i+dimen+1] > lng)                          //finding longest dimension
```

```
          lng = center[k][i+dimen+1];
        }

      z = sqrt(z);
      if ((stopcond == 1)||(stopcond == 3)||(stopcond == 4)||(stopcond == 5))          //don't
divide if smaller than min box size
           if (z < minsize1)
        continue;
      else if (stopcond==2)
       if (z<minsize2)
        continue;

      if (center[k][2*dimen+2] < center[k][2*dimen+3])                 //skip if not divided
this time
           {
            center[k][2*dimen+2] = center[k][2*dimen+2]+1;
        continue;
       }
      else
           center[k][2*dimen+2] = 1;

        j=1;
      net = 0;
      for (i = 1; i <= dimen; i++)
       if (fabs(center[k][i+dimen+1]-lng) < (0.001*minsize1))
        {
         if (division == 0)
           {
            dim[j] = i;
            j++;
            net++;                                                  // # of dimensions being
divided
         //# of dimensions to divide
           }
         else
           {
            if (t[i] < tmin)                                        // divide longest side that was
divided least
              {
              fprintf(fp2," i = %d  tmin = %d, t[i] = %d \n",i,tmin,t[i]);
               dim[1] = i;
               tmin = t[i];
              }
            net =1;
           }
        }
```

```
      if (division == 1)
        t[dim[1]]++;                                    // increment number
of divisions for side being divided

      if ((numb+2*net) > nbox)
          break;

      for (i = (numb+1); i <= (numb+net*2); i++)        //create the new
points
          for (l = 2; l <= (2*dimen+1); l++)
        center[i][l] = center[k][l];

      for (i = 1; i <= net; i++)
       {
        center[numb+2*i-1][dim[i]+1] += center[numb+2*i-
1][dim[i]+dimen+1]/3*size[dim[i]];
        center[numb+2*i][dim[i]+1] -=
center[numb+2*i][dim[i]+dimen+1]/3*size[dim[i]];
        center[k][dim[i]+dimen+1] = center[k][dim[i]+dimen+1]*1/3;
       }

      Eval();                                           //evaluate new points

//divide the new boxes based on the values at the centers
//sort points
        for (i = 1; i <= (2*net); i++)
         divide[i] = 0;

        for (i = (numb+1); i <= (numb+2*net); i++)
         for (j = 1; j <= (2*net); j++)
        {
         if (divide[j] == 0)
           {
                  divide[j] = i;
           break;
           }
         if (center[i][1] < center[divide[j]][1])// < -0.00000000000001
           {
                  for (k = (2*dimen); k >= (j+1); k--)
                    divide[k] = divide[k-1];
           divide[j] = i;
           break;
           }
        }
```

```
      for (i = 1; i <= net; i++)
        //divide the boxes based on the sorted values
        {
            split[i*2] = 0;                          //has the box been divided completely yet
         split[i*2-1] = 0;
        }

      for (i = 1; i <= (2*net); i++)
        {
            k = divide[i]-numb;
         if (split[k] != 1)
           {
         for (j = (numb+1); j <= (numb+2*net); j++)
              if (split[j-numb] != 1)
                    center[j][dim[(k+1)/2]+dimen+1] = center[j][dim[(k+1)/2]+dimen+1]/3;
//integer division by 2 ??????

                  split[k] = 1;
              if ((k/2) == ((k+1)/2))
         //integer division????
                    split[k-1] = 1;
                  else
                    split[k+1] = 1;
           }
        }

      numb = numb+2*net;
    }
}

/*********************************************************************/
/* Subroutine Neighbor                                               */
/* Finds neighbors of small boxes and adds them to order list if they are too large  */
/*********************************************************************/
void Neighbor(int order[], double order2[], int initial)
{

  int i,j,k,l,ll,m,q,dim[dimen],overlap,n1,cover[2*dimen+1]={0};
  double dist,z,distance[nbox];
  bool border=0;

  bvector VM,VM2;
  double dmax,dmin,tmin,theta,mag,length;
  int nvec = 0, neighbor[4][nbox] = {0},vmax,vmin,send,vadd,s1,cc;
```

```
    if (order2[initial] < minsize3)                          // if smaller than limit check
neighbors for size ratio
       {
        n1 = nact;
        dmax = 0;
        dmin = dimen;
        for (j=1; j<= numb; j++)
          if(order[initial]!=j)
           {
            m=0;
            q=0;
            overlap = 0;

            for (k=2; k<= dimen+1; k++)
              {
                if (center[order[initial]][k] < center[j][k])
                  {
                    if (fabs(center[order[initial]][k] +
center[order[initial]][k+dimen]/2*size[k-1]+center[j][k+dimen]/2*size[k-1] -
center[j][k])< .0000001)
                     {
                      q++;
                      dim[q] = k-1;
//  printf("k = %d  j = %d  order[initial] = %d   x1 = %f, x2 = %f, d1 = %f, d2 = %f
\n",k,j,order[initial],center[order[initial]][k],center[j][k],center[order[initial]][k+dimen]/2
*size[k-1],center[j][k+dimen]/2*size[k-1]);
                     }
                    else if ((center[order[initial]][k] +
center[order[initial]][k+dimen]/2*size[k-1]+center[j][k+dimen]/2*size[k-1]) <
center[j][k])
                                {
                                 m=1;
                    //           printf("j = %4d  m = 1   due to dimen %2d nvec = %3d \n",j,k-
1,nvec);
                                }
                    else if ((center[order[initial]][k] +
center[order[initial]][k+dimen]/2*size[k-1]+center[j][k+dimen]/2*size[k-1]) >
center[j][k])
                      overlap++;               // face = k;
//   printf("   k = %d  j = %d  order[initial] = %d   x1 = %f, x2 = %f, d1 = %f/%f, d2 =
%f/%f
%f\n",k,j,order[initial],center[order[initial]][k],center[j][k],center[order[initial]][k+dimen
]/2*size[k-1],center[order[initial]][k+dimen],center[j][k+dimen]/2*size[k-
1],center[j][k+dimen],size[k-1]);
                 }
                else
```

```
                {
                  if (fabs(center[j][k] + center[order[initial]][k+dimen]/2*size[k-
1]+center[j][k+dimen]/2*size[k-1] - center[order[initial]][k]) < .0000001)
                   {
                     q++;
                     dim[q] = k-1;
//  printf(" k = %d  j = %d  order[initial] = %d   x1 = %f, x2 = %f, d1 = %f, d2 = %f
\n",k,j,order[initial],center[order[initial]][k],center[j][k],center[order[initial]][k+dimen]/2
*size[k-1],center[j][k+dimen]/2*size[k-1]);
                   }
                  else if ((center[j][k] + center[order[initial]][k+dimen]/2*size[k-
1]+center[j][k+dimen]/2*size[k-1]) < center[order[initial]][k])
                              {
                                m=1;
                             //    printf("j = %4d  m = 1   due to dimen %2d nvec = %3d \n",j,k-
1,nvec);
                              }
                  else if ((center[j][k] + center[order[initial]][k+dimen]/2*size[k-
1]+center[j][k+dimen]/2*size[k-1]) > center[order[initial]][k])
                    overlap++;              // face = k + dimen;
//  printf("   k = %d  j = %d  order[initial] = %d   x1 = %f, x2 = %f, d1 = %f/%f, d2 =
%f/%f
%f\n",k,j,order[initial],center[order[initial]][k],center[j][k],center[order[initial]][k+dimen
]/2*size[k-1],center[order[initial]][k+dimen],center[j][k+dimen]/2*size[k-
1],center[j][k+dimen],size[k-1]);
                }
              }

            if ((m == 0) && (q <= (dimen)))              // j is neighbor to i & and must be
any dimensional face
               {
                 nvec++;
//               printf("%5d is a neighbor to %5d  on dimen %2d   nvec = %2d  q = %d
\n",j,order[initial], dim[1],nvec,q);
                 dist = 0;
                 for (k=1;k<= dimen; k++)
                   {
                     V[nvec][k] = center[j][k+1] - center[order[initial]][k+1];              //add
vector to V
                     dist += pow(V[nvec][k],2);
                   }
                 dist = sqrt(dist);
          //     printf("dist = %f\n",dist);
                 for (k=1;k<= nvec; k++)
                   if ((dist < distance[k])||(distance[k] == 0))                             //sort
vectors by distance
```

```
    {
     for (l=nvec;l>=k;l--)
       {
        distance[l+1] = distance[l];
        for (ll=1;ll<=dimen;ll++)
          V[l+1][ll] = V[l][ll];
        neighbor[1][l+1] = neighbor[1][l] ;
        neighbor[2][l+1] = neighbor[2][l] ;
        neighbor[3][l+1] = neighbor[3][l] ;
       }
     distance[k] = dist;
     for (l=1;l<=dimen;l++)
       V[k][l] = V[nvec+1][l];
     neighbor[1][k] = j;
    }
if (dist > dmax)
  {
   dmax = dist;
   vmax = nvec;
  }
if (dist < dmin)
  {
   dmin = dist;
   vmin = nvec;
  }

   tmin = pi;
   for (i=1; i<= dimen;i++)              // find face V passes through
     {
      theta = acos(V[k][i]);
         if (theta < tmin )
  {
   tmin = theta;
   vmin = i;
  }
      theta = acos(-V[k][i]);
         if (theta < tmin )
  {
   tmin = theta;
   vmin = -i;
  }
     }

neighbor[2][k] = vmin;                   //closest face in neighbor[2][k]
```

```
                if (q == 1)                                     //neighbor[3][k] has
dimension touched by n-1 dimensional face
                  {
                  neighbor[3][k] = dim[1];
                  if (center[order[initial]][dim[1]+1] < center[j][dim[1]+1])
                    {
                      cover[dim[1]] = j;
                                                    }
                  else
                    {
                      cover[dim[1]+dimen] = j;          // face has a neighbor
                                                    }
                  }
                 else
                   neighbor[3][k] = 0;
                 dim[1] =0;
               }
          }

        for(i=1;i<=2*dimen;i++)
          if(cover[i] == 0)                                     //if no neighbor to face then
add dummy vector on C.D.
            {
              border = 1;                                       //best box is on a border
              for (l=nvec;l>=1;l--)
                {
                  distance[l+1] = distance[l];
                  for (ll=1;ll<=dimen;ll++)
                     V[l+1][ll] = V[l][ll];
                  neighbor[1][l+1] = neighbor[1][l] ;
                  neighbor[2][l+1] = neighbor[2][l] ;
                  neighbor[3][l+1] = neighbor[3][l] ;
                }
              distance[1] = dmin;
              for (l=1;l<=dimen;l++)
                V[1][l] = 0;
              if(i<=dimen)
                V[1][i] = 1;
              else
                V[1][i-dimen] = -1;
              neighbor[1][1] = 0;
              nvec++;
            }

/*****Find all of the short enough vectors to send for the first try*****/
        send = nvec;
```

```
            if(order2[initial] > minsize1)
              {
            for (j=2;j<=nvec;j++)
              {
                if (distance[j]/dmin > ratio)
                  {
                    send = j-1;
                    break;
                  }
              }
            printf(" There are %d neighbors, sending %3d distance[%d] = %f
\n",nvec,send,j-1,distance[j-1]);
//            printf("           distance[%d] = %f n#= %d face =
%d\n",j,distance[j],neighbor[1][j],neighbor[2][j]);
           }
            else
              {
                for (j=2;j<=nvec;j++)
                  {
                    if (distance[j] > minsize2)
                      {
                        send = j-1;
                        break;
                      }
                  }
            printf(" There are %d neighbors, sending %3d distance[%d] =
%f\n",nvec,send,j-1,distance[j-1]);
//            printf("           distance[%d] = %f n#= %d face =
%d\n",j,distance[j],neighbor[1][j],neighbor[2][j]);
              }

            if(send < dimen+1)
              send = dimen+1;

         s1 = send;

/****Send first batch****/
         VM = Basis(send);

/****Check all vectors before adding one at a time****/
         if (!VM.is_p_basis)
           {
            send = nvec;
            VM2 = Basis(send);
            if (!VM2.is_p_basis)
```

```
            goto skip_check;                    //don't bother checking subsets of vectors,
skip the next section
          else
            {
            send = s1;                          //All of them form a pb so start checking
from the s1 set

          printf("send = %d \n");
            //VM.is_p_basis = true;

//  Add all `face' neighbors to list if they are large
/**/

                          for (i=1;i<=2*dimen;i++)
                            {
                              if (cover[i] != 0)
                            {
                              for (l=1;l<=nvec;l++)
                                if (cover[i]==neighbor[1][l])
                                            {
                                  vadd =l;
                                  break;
                                    }
                                      dist = distance[vadd];
                                 if (dist/dmin > ratio)
                                {
                                        for (l=1;l<=dimen;l++)
                                  V[nvec+1][l] = V[vadd][l];
                                    neighbor[1][nvec+1] = neighbor[1][vadd] ;
                                    neighbor[2][nvec+1] = neighbor[2][vadd] ;
                                    neighbor[3][nvec+1] = neighbor[3][vadd] ;
      printf("%d adding #%d size %f  send = %d   theta =
%f\n",vadd,neighbor[1][vadd],distance[vadd],send+1,tmin);

                                   for (l=vadd-1;l>=send+1;l--)
                                {
                                  distance[l+1] = distance[l];
                                          for (ll=1;ll<=dimen;ll++)
                                    V[l+1][ll] = V[l][ll];
                                          neighbor[1][l+1] = neighbor[1][l] ;
                                          neighbor[2][l+1] = neighbor[2][l] ;
                                  neighbor[3][l+1] = neighbor[3][l] ;
                                        }
                                    distance[send+1] = dist;
                                    for (l=1;l<=dimen;l++)
                                              V[send+1][l] = V[nvec+1][l];
```

```
                                neighbor[1][send+1] = neighbor[1][nvec+1] ;
                                neighbor[2][send+1] = neighbor[2][nvec+1] ;
                                neighbor[3][send+1] = neighbor[3][nvec+1] ;

                                        send ++;
                                }
                        }
                        }
                }
        }

printf("send = %d\n\n",send);

        while (!VM.is_p_basis)
          {
            if (send == nvec)
              break;
            dist = 0;
            for (j=1; j<= dimen; j++)
              dist += pow(VM.V1[j],2);
                  dist = sqrt(dist);
                  tmin = pi;
            for (i=send+1; i<= nvec;i++)                    //calculate thetas
              {
                mag = 0;
                for (j=1; j<= dimen; j++)
                  {
                    mag += (-VM.V1[j]*V[i][j]);
                    length += pow(V[i][j],2);
                  }
                length = sqrt(length);
                theta = acos(mag/(length*dist));
                if (theta < tmin )                          //take closest vector to VM
                  {
                    tmin = theta;
                    vadd = i;
                  }
                if (theta < .35)                    //or accept shortest vector within 20 degrees
of VM (.35)
                  {
                    vadd = i;
                    break;
                  }
              }
            dist = distance[vadd];
            for (l=1;l<=dimen;l++)
```

```
              V[nvec+1][l] = V[vadd][l];
            neighbor[1][nvec+1] = neighbor[1][vadd] ;
            neighbor[2][nvec+1] = neighbor[2][vadd] ;
            neighbor[3][nvec+1] = neighbor[3][vadd] ;
          printf("%d adding #%d size %f  send = %d   theta =
%f\n",vadd,neighbor[1][vadd],distance[vadd],send+1,tmin);

            for (l=vadd-1;l>=send+1;l--)
              {
               distance[l+1] = distance[l];
               for (ll=1;ll<=dimen;ll++)
                 V[l+1][ll] = V[l][ll];
               neighbor[1][l+1] = neighbor[1][l] ;
               neighbor[2][l+1] = neighbor[2][l] ;
               neighbor[3][l+1] = neighbor[3][l] ;
              }
            distance[send+1] = dist;
            for (l=1;l<=dimen;l++)
              V[send+1][l] = V[nvec+1][l];
            neighbor[1][send+1] = neighbor[1][nvec+1] ;
            neighbor[2][send+1] = neighbor[2][nvec+1] ;
            neighbor[3][send+1] = neighbor[3][nvec+1] ;
      printf("#%d is box %d \n",send,neighbor[1][vadd]);

                 send ++;
            VM = Basis(send);
          }

skip_check :                                  // go here if all of the variables do not form a
pb

        if (VM.is_p_basis)
             {
               m=0;
               for (l=s1+1;l<=send;l++)   //nvec s1
                 {
//                 printf("checking for box %d \n",neighbor[1][l]);
                   if (((distance[l]/dmin > ratio)&&(order2[initial] > minsize1))||((distance[l]
> minsize2)&&(order2[initial] <= minsize1)))
                  for (k=initial;k<=nact+1;k++)
                   {
                    if( order[k] == 0)
                      {
                        order[k] = neighbor[1][l];
                        z=0;
                        for (j=1;j<=dimen;j++)
```

```
                    z+= pow(center[neighbor[1][l]][j+1+dimen],2);
                  z = sqrt(z);
                  order2[k] = z;
                  printf("adding box %d to optimal because it is too long of a search
direction from box %d\n",neighbor[1][l],order[initial]);
                        printf("dist = %f and order2[%d] = %f\n",dist,i,order2[i]);
                  nact++;
                  m =1;
                  break;
                 }
                else if(neighbor[1][l] == order[k])
                  break;
               }
            }

               if (border)                                          //if best box is on a
border
                {
                 cc = 0;
                 for (i=1; i<=2*dimen;i++)
                  if (cover[i] == 0)                          //if not border of this
dimension
                    cc++;
                 if(cc == 1)                                          //only at edge of one
dimension
                  for (i=1; i<=dimen;i++)
                   if ((cover[i] != 0) && (cover[i+dimen] != 0))     //if not border of this
dimension
                     {
                  z=0;
                  for (l=1;l<=dimen;l++)
                   z+= pow(center[cover[i]][l+1+dimen],2);
                  z = sqrt(z);
                  if (center[cover[i]][i+1+dimen] > center[order[initial]][i+1+dimen])
                 // add neighbor on border if it is larger
                    for (k=initial;k<=nact+1;k++)
                     {
                      if ( order[k] == 0)
                       {
                         order[k] = cover[i];
                               order2[k] = z;
                    printf("adding box %d to optimal because it is too long of a search
direction from box %d\n",cover[i],order[initial]);
//                          printf("dist = %f and order2[%d] = %f\n",dist,i,order2[i]);
                         m =1;
                         nact++;
```

```
                        break;
                      }
                    else if(cover[i] == order[k])
                      break;
                  }
                z=0;
                for (l=1;l<=dimen;l++)
                  z+= pow(center[cover[i+dimen]][l+1+dimen],2);
                z = sqrt(z);
                if (center[cover[i+dimen]][l+1+dimen] >
center[order[initial]][i+1+dimen])
                  for (k=initial;k<=nact+1;k++)
                    {
                      if ( order[k] == 0)
                        {
                          order[k] = cover[i+dimen];
                                order2[k] = z;
                        printf("adding box %d to optimal because it is too long of a search
direction from box %d\n",cover[i+dimen],order[initial]);
//                          printf("dist = %f and order2[%d] = %f\n",dist,i,order2[i]);
                          m =1;
                          nact++;
                          break;
                        }
                      else if(cover[i] == order[k])
                        break;
                    }
              }
            else                                                    //borders 2 or more edges so
all face neighbors on edge
                 for (i=1; i<=2*dimen;i++)
                  if (cover[i] != 0)                                //check all real face neighbors
                    {
                z=0;
                for (l=1;l<=dimen;l++)
                  z+= pow(center[cover[i]][l+1+dimen],2);
                z = sqrt(z);
                if(i < dimen)
                  k = i+dimen;
                else
                  k = i;

                if (center[cover[i]][k+1] > center[order[initial]][k+1])       // add
neighbor on border if it is larger
                  for (k=initial;k<=nact+1;k++)
                    {
```

```
                        if ( order[k] == 0)
                          {
                            order[k] = cover[i];
                                  order2[k] = z;
                       printf("adding box %d to optimal because it is too long of a search
direction from box %d\n",cover[i],order[initial]);
//                              printf("dist = %f and order2[%d] = %f\n",dist,i,order2[i]);
                            m =1;
                            nact++;
                            break;
                          }
                        else if(cover[i] == order[k])
                          break;
                       }
                    }
                 }
               printf("\n");
                  if(m == 1)
                  {                                                  //remove order[initial] if it is square
                    m=0;
                    for (k=1;k<=dimen-1;k++)
                      if (center[order[initial]][k+dimen+1] != center[order[initial]][k+dimen+2])
                        m=1;
                        if(m == 0)
                        {
                          for(k=initial;k<=nact;k++)
                            {
                              order[k]=order[k+1];
                              order2[k]=order2[k+1];
                            }
                          nact--;
                          printf("removed initial box from optimal.\n");
                        }
                  }


/*************************/
              if ((stopcond == 5) && (m == 0)&&(order2[initial] <= minsize1))
                converg = 1;
/*************************/
            }
          else      // NO PB COULD BE FOUND MUST CHOOSE NEIGHBORS TO
DIVIDE
            {
                                                                                    // This
version divides boxes on hull
```

```
            hcount = hull.begin();
            while (hcount !=hull.end())
              {
                for (k=initial;k<=nact+1;k++)
                  {
//                  printf("order[%d] = %d\n",k,order[k]);
                    if ( order[k] == 0)
                      {
                        nact++;
                        order[k] = neighbor[1][*hcount];
                        z=0;
                        for (l=1;l<=dimen;l++)
                          z+= pow(center[neighbor[1][*hcount]][l+1+dimen],2);
                        printf("adding box %d to optimal because it helps form a positive basis
for box %d\n",neighbor[1][*hcount],order[initial]);
                        z = sqrt(z);
                        order2[k] = z;
                        break;
                      }
                    else if(neighbor[1][*hcount] == order[k])
                      break;
                  }
                hcount++;
              }
            m=0;
            for (k=1;k<=dimen-1;k++)
              if (center[order[initial]][k+dimen+1] != center[order[initial]][k+dimen+2])
                m=1;
            if(m == 0)
              {
                for(k=initial;k<=nact;k++)
                  {
                    order[k]=order[k+1];
                    order2[k]=order2[k+1];
                  }
                nact--;
                printf("removed initial box from optimal.\n");
              }
          }
      }

    for (i=1;i<=nvec+1;i++)
      {
        for (j=1;j<=dimen;j++)
          V[i][j] = 0;
        distance[i] = 0;
```

```
      }
}

/*********************************************************************/
/* Positive Basis Calculator                                         */
/* Vectors declared globally, up to 3*dimen vectors alowed           */
/*********************************************************************/

bvector Basis(int n)
{
  const double Estart = 0.2, alphstart = 0.3,interval = 10.;
  int i,j,k,t,loop;
  double tmax,tmin,mag,alph = alphstart,E = Estart;   //theta[nbox],alpha[nbox],,S[nbox],
order[nbox]={0}
  double Vnew[dimen+1],Vold[dimen+1]={0};
  double eps = 0.01,diff, denom;
  bvector VM;

  double* theta = new double[n+3];
  double* alpha = new double[n+3];
  int* order = new int[n+3];
  int* S = new int[n+3];


  printf("n = %d \n",n);
  t = 0;
  loop = 0;
  for (i = 1; i <= n; i++)
    {
      for (k = 1; k <= dimen; k++)
        mag += V[i][k]*V[i][k];
      mag = sqrt(mag);
      for (k = 1; k <= dimen; k++)
        {
          V[i][k] = V[i][k]/mag;
          Vold[k] += V[i][k];
        }
    }

  mag = 0;
  for (k = 1; k <= dimen; k++)
     mag += (Vold[k]*Vold[k]);
  mag = sqrt(mag);

  if (mag >= 0.00001)
    {
```

```
      for (k = 1; k <= dimen; k++)
         Vnew[k] = Vold[k]/mag;
    }
  else
    {                                    //for now use ave of V[1],V[2],V[3] as Vnew if
Vave is small.
      for (k = 1; k <= dimen; k++)
         Vnew[k] = (V[1][k]+V[2][k]+V[3][k])/3.;
      for (k = 1; k <= dimen; k++)
         mag += (Vnew[k]*Vnew[k]);
      mag = sqrt(mag);
      for (k = 1; k <= dimen; k++)
         Vnew[k] = Vnew[k]/mag;
    }
//      printf(" %f  %f  %f    mag = %f\n",Vold[1],Vold[2],Vold[3],mag);
  do
    {
      t++;
      if (t > interval)
        {
          E=Estart/(t/interval);
          alph = alphstart/(t/interval);
        }

      for (k = 1; k <= dimen; k++)
         Vold[k] = Vnew[k];
      tmax = 0;
      tmin = pi;
      for (i=1; i<= n;i++)               // calculate thetas
        {
          mag = 0;
          for (j=1; j<= dimen; j++)
            mag += (Vnew[j]*V[i][j]) ;
          theta[i] = acos(mag);
//      printf(" theta = %f  \n",theta[i]);
          if (theta[i] < tmin )
            tmin = theta[i];
          if (theta[i] > tmax)
            tmax = theta[i];
        }

      for (i = 1; i <= n; i++)
           order[i]=0;

      for (i = 1; i <= n; i++)                    //Sort the thetas
        {
```

```
          for (j = 1; j <= n; j++)
            {
                if (order[j] == 0)
             //if at end of list put box here
              {
                        order[j] = i;
                        break;
              }
               if (theta[i] > theta[order[j]])
                      //if smaller than this entry insert here
              {
                for (k = i; k >= (j+1); k--)
                           order[k] = order[k-1];
                        order[j] = i;
                        break;
              }
             }
        }
//for (k=1;k<=n;k++)
//printf("order[%d] = %d\n",k,order[k]);

      j =1;                                                  //Select vectors to move towards
      for (j=1;j<= dimen;j++)
         S[j] = order[j];
      if (tmax-theta[order[j-1]] > E)
         denom = tmax-theta[order[j-1]];
      else
         denom = E;

//printf("five basis   denom = %f  alph = %f\n",denom, alph);
      for (i=j; i<= n; i++)
        if( tmax - theta[order[i]] < E)
          {
           S[j] = order[i];
           j++;
          }
        else
          break;
      j--;

      for (i=1; i<= j; i++)                                  //Calculate alphas
        {
//         printf(" %d  %d  denom = %f  %d  %f",i,j,denom, S[i], theta[S[i]]);
        alpha[i] = (1-(tmax-theta[S[i]])/(denom*.95))*alph;
//         printf(" alpha = %f \n",alpha[i]);
        }
```

```
//printf("six basis \n");
      mag = 0;
      diff = 0;
      for (i=1; i<= dimen;i++)
        {
//      printf(" %f  %f  %f  %f  %f  %f
\n",Vnew[1],Vnew[2],Vnew[3],V[S[i]][1],V[S[i]][2],V[S[i]][3]);
          for (k=1;k<=j; k++)
            Vnew[i] = Vnew[i]+alpha[k]*V[S[k]][i];
          mag += Vnew[i]*Vnew[i];
        }
      mag = sqrt(mag);
      for (i=1; i<= dimen;i++)
        {
          Vnew[i] = Vnew[i]/mag;
          diff += pow(Vnew[i]-Vold[i],2);
        }
      diff = sqrt(diff);

//      printf(" %f  %f  %f  mag = %f  t = %d  diff = %f, tmax = %f, tmin =
%f\n",Vnew[1],Vnew[2],Vnew[3],mag,t,diff,tmax,tmin);


    } while ((diff > eps));   //(tmax > pi/2.) && (tmin < pi/2) &&

 printf("tmax = %f  Tmin = %f pi/2 = %f  diff = %f  t =
%d\n",tmax,tmin,pi/2,fabs(diff),t);
/* for(i=1;i<=dimen;i++)
   printf("Vm[%d] = %7.4f ",i,Vnew[i]);
 printf("\n");
*/

 if ((tmax > (pi*19./36.)) && (tmin < pi/2.))
   VM.is_p_basis = true;
 else
   VM.is_p_basis = false;

 for (i=1;i<=dimen;i++)
   VM.V1[i]=Vnew[i];

// printf(" j= %d  E = %f\n",j,E);

 hull.clear();

 for (k=1;k<=j;k++)
```

```
  hull.push_back(S[k]);

 delete [] theta;
 delete [] alpha;
 delete [] order;
 delete [] S;

 return VM;
}
/*****************************************************************/
/* Subroutine LOW                                                */
/*****************************************************************/
double Func(double *y)
{
  double obj = 0;
  int l,m;

  m=0;                                  // Gomez Function
  if (m==1)
    obj =  ((4-2.1*y[1]*y[1]+pow(y[1],4)/3)*y[1]*y[1]+y[2]*y[1]+(-
4+4*y[2]*y[2])*y[2]*y[2]);

  m=1;                                  // Test Function
  if (m==1)
    for (l=1;l<=dimen;l++)
      obj +=  (((y[l]+1)*sin(1.5*pi*(y[l]-.1)))/1.7+pow((y[l]-.3)/1.3,2) + y[l]*l/99.0 +
.08*cos(y[l]*71));

  m=0;                                  // Quartic Function
  if (m==1)
    for (l = 1; l <= dimen; l++)
      obj += 10+2.2*pow((y[l]+e[l]),2)-1*pow((y[l]+e[l]),4);

  m=0;                                  // Upside Down Quartic Function
  if (m==1)
    for (l = 1; l <= dimen; l++)
      obj -= (1+2.2*pow((y[l]+e[l]),2)-1*pow((y[l]+e[l]),4)+ 0.5*y[l]);

  m=0;                                  // Griewank Function
  if (m==1)
    {
      for (l=1;l<=dimen;l++)
        obj += pow(y[l],2);
      obj =obj/1000;
      obj += (1.-cos(y[1])*cos(y[2]/sqrt(2.))*cos(y[3]/sqrt(3.))*
        cos(y[4]/2.)*cos(y[5]/sqrt(5.))*cos(y[6]/sqrt(6.))*
```

```
        cos(y[7]/sqrt(7.))*cos(y[8]/sqrt(8.))*cos(y[9]/3.)*
        cos(y[10]/sqrt(10.)));
    }

  m=0;                                  // Cones, min follows trend
  if (m==1)
    for (l=1;l<=dimen;l++)
      {
        obj-= .5*(y[l]+2);

            if((y[l] < 1.2) && (y[l] >0.8))
                    obj -= 4-20*fabs(y[l]-1);
            else if((y[l] < -0.2) && (y[l] > -0.6))
                    obj -= 4-20*fabs(y[l]+0.4);
            else if((y[l] < -1.4) && (y[l] > -1.8))
                    obj -= 4-20*fabs(y[l]+1.6);
      }

  m=0;                                  // Cones, min is anti-trend
  if (m==1)
    for (l=1;l<=dimen;l++)
      {
        obj-= .5*(y[l]+2);

            if((y[l] < 1.2) && (y[l] >0.8))
                    obj -= 4-20*fabs(y[l]-1);
            else if((y[l] < -0.2) && (y[l] > -0.6))
                    obj -= 6-30*fabs(y[l]+0.4);
            else if((y[l] < -1.4) && (y[l] > -1.8))
                    obj -= 8-40*fabs(y[l]+1.6);
      }
  return obj;
}
/*************************************************************************/
/* Subroutine con                                                        */
/*************************************************************************/
void con(double *y)
{
  int j;
  for (j=1; j<= ni; j++)
     g[j] = (y[j]+.01);
  return ;
}
/*************************************************************************/
/* Fortran Calls                                                         */
/*************************************************************************/
```

```
double ffunc_(double *z)
{
   double y[dimen+1];
   for (int i = 1; i <= dimen; i++)
      y[i] = z[i-1];

   return Func(y);
}
void fcon_(double *y, double g[])
{
  int j;
  for (j=1; j<= ni; j++)
     g[j] = (y[j]+.01);
  return ;
}
```

APPENDIX C
Parallel DIRECT Code

This code is similar to the sequential DIRECT code. It implements all three versions of parallel DIRECT with all of the starting and stopping choices. It must be recompiled after changing any of the parameters or the function being optimized and has hard limits on the memory allocation which is set at compile time.

Parallel DIRECT Code

```
/************************************************************/
/*      Direct Optimizer Single fidelity version-parallel   */
/*      Convert from Fortran90 to C++                       */
/*      Steven Cox and Chalermpong Dilakanont               */
/*      Febuary 2001                                        */
/*      Modified by Steven Cox                              */
/*      Summer and Fall 2001                                */
/*      Contains a continuous version with less overlap     */
/************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <mpi.h>
#include <string>
```

```
const int method = 3;                                   // 1 = original DIRECT, 2 =
        continuous DIRECT, 3 = blank entries in queue
                                                        // 4 = random divisions always,  5 =
        change from orig to cont
const int dd = 0;                                       // 1 = can't divide parent again until
        all children are finished
const int agressive = 0;                                // 0= convex hull only, 1 = add extra
        large box to list
const int switcher = 5;
const int nbox =160001;                                         //absolute limit
const int dimen = 20;
const int startcond = 1;                                // 1 = use DIRECT division, 2 = use
        even boxes
const int stopcond = 4;                                 // 1 = min box size, 2 = remove
        boxes smaller than minsize2, 3 = remove radius arround small boxes

const int print = 1;                                    //prints all points and their value

const double minsize1 = 0.0001;                                 //stopping box size
const double minsize2 = 0.005;                                  //minimum divide box size
const double radrem = 0.03;                             //radius of removal

const int nopt = 200;                                   //maximum number of potentially
        optimal boxes

const int loc = 15;
const int maxit = 100;

const int nbest = 6000;
const double mindist = 0.35;

const long evaluation = 700000000;
const int d2 = dimen+2;

int numb, ni, nold;                                                     //number of
        boxes, max number of boxes
int final;

double xopt[dimen+1], x[dimen+1];                                       //point to pass
        to DOT
double center[nbox+1][(2*dimen+3)+1];                                           //value,
        position and dimension of each point
int points[15+1];
```

```
int optimal[nopt+1], best[nbest+1], dropped, optional[nopt+1];
                //potentially optimal points
int totcalls, calls, count, converg, nact, itter;
//counters for FE used by DOT

double xl[dimen+1], xu[dimen+1], xbest[dimen+1], e[dimen+1], size[dimen+1];
                //current best box and function value
double fbest, dist, order2b[nbox+1];
int index2[nbox+1];

//parallel variables
const int master = 0;
const int qlim = 20*dimen*dimen;                                    //qlim must be larger
        than 2* number of processors
double queue[qlim+1][dimen+2];                              //index2 for the points to be
        analyzed
int nextsend, nextgen;                          //next position to get pt from queue,
        next position to add pt
int nsort, nproc;                               // number of completed boxes,
        number of processors
int children[51][2*dimen+2];                    // keeps track of boxes that are
        generated but not divided yet
int start;                                      // tells divider not to put divided
        boxes into global array
int wait2;                                      // Skips sending new points to
        analyze if waiting for all to finish

FILE *fp1;
FILE *fp2;
FILE *fp3;
FILE *fp4;
FILE *fp5;

// define function prototype

double Eval(double *y);

void insert(int notnew);
void Graham(double center[nbox+1][2*dimen+3+1], int *optimal);
void Divider(double center[nbox+1][2*dimen+3+1], int *optimal);
void initial(double center[nbox+1][2*dimen+3+1], int *optimal);
void divide2(int parent);
void initial2(int igen);

/*****************************************************************/
/* Main Program                                  */
```

```
/*****************************************************************/
int main(int argc, char** argv)
{
  int k, kr, i, m, j, n, l, complete, pid, msize, dest, temp, ierror, net, prepoints;
  double func, t, a[3], y[dimen];
  double data_in[dimen+2];   //input array for slave proc.
  double answer[3];          //result from calculation
  double starttime, endtime;
  int pfin[350]= {0},mcount=0,mcheck=0;                    //processors that have
        finished one of their tasks
  int thought_so;

  char name[50];
  char message[50];
  int namelen, igen;

  MPI_Status status,status2 ;
  MPI_Request req1 ,req[500];

  MPI_Init(&argc, &argv) ;
  MPI_Comm_size(MPI_COMM_WORLD, &nproc) ;
  MPI_Comm_rank(MPI_COMM_WORLD, &pid) ;



/* Get the processor id and name */

  fp5 = fopen("collect13.dat", "a+");

  if (pid != 0)
    {
     MPI_Get_processor_name(name, &namelen);
     sprintf(message,"pid = %d, name = %s", pid, name);
     MPI_Send(message, strlen(message)+1, MPI_CHAR, master, 1,
         MPI_COMM_WORLD);
    }
  else
    {
     fprintf(fp5,"Evaluation loops = %d\n", evaluation);
      MPI_Get_processor_name(name, &namelen);
     fprintf(fp5,"pid = %d, name = %s  ", pid, name);
     for (dest = 1; dest < nproc; dest++)
          {
           MPI_Recv(message, 100, MPI_CHAR, dest, 1, MPI_COMM_WORLD,
         &status);
```

```
            fprintf(fp5,"%s  ", message);
          }
    }

  MPI_Barrier(MPI_COMM_WORLD);

/*********************************************************/
/* Master processor part */
/*********************************************************/

  if (pid == master)
    {
     starttime = MPI_Wtime();

// open file "high.dat" here
      fp1 = fopen("high.dat", "a+");


         srand(5);
         xu[1] = rand()/RAND_MAX;

// open file "points.dat" here
         fp2 = fopen("points.dat", "w+");

// Setup cycles
                  optimal[1] = 1;
                  nold = 1;
                  numb = 1;
                  ni = 0;
                  final = nbox-2*dimen;
                  converg = 0;
                  fbest = 9999090;
                  totcalls = 0;
                  itter = 1;
                  count = 0;
                  nsort = 0;
                  nact = 0;
                  nextsend = 1;
                  nextgen = 2;
                  start = 0;
                  wait2 = 0;

                  for (i = 1; i <= nbest; i++)
                     best[i] = 0;
```

```
// Box function
                for (i = 1; i <= dimen; i++)
                {
                        t = rand();
                        xu[i] = 100 + 800.* (t/RAND_MAX);   //2;
                        xl[i] = xu[i]-1000;   //-2;
                        e[i] =  0.2+(t/RAND_MAX)*0.2;
        printf(" e%d = %f,  ",i, e[i]);
                }
        printf("\n");

                for (i = 1; i <= nbox; i++)
                {
                        for (j = 1; j <= (dimen+3); j++)
                                center[i][j] = 0;
                        order2b[i] = 0;
                        index2[i] = 0;
                        center[i][2*dimen+2] = 1;
                        center[i][2*dimen+3] = 1;
                }

// Send out values of e
                MPI_Bcast(&e[1],dimen,MPI_DOUBLE,master,MPI_COMM_WORLD);

//GENERATE INITIAL BOXES HERE AND PUT IN QUEUE


  if (agressive == 3)
     igen = 2*dimen+1+nproc*2;                                  // gives nproc*2 extra
        points in addition to boxes from 1st itteration
  else
     igen = nproc*2-1;

  printf("about to start\n");
  if (startcond == 1)
     {
// find sizes and center points
        for (i = 1; i <= dimen; i++)
                {
                        center[1][i+1] = (xu[i]+xl[i])/2;                       //center
        point of box
                        center[1][i+dimen+1] = 1;                               //each
        box is 1 unit long in each dimension
                        size[i] = (xu[i]-xl[i]);            //length of box side
                        queue[1][i+1] = center[1][i+1];
                }
```

```
                        queue[1][1] = 1;


        printf("start  igen = %d\n", igen);
        m=0;
                do
                {
                        initial(center, optimal);
                        Divider(center, optimal);

                 if ((numb >= igen) || (method == 1) || (method == 5))
                        m = 1;

        printf("loop  %d %d %d %d %d\n", numb, nproc,nextsend, nextgen, m);
                } while (m !=1);                                // loop until enough points to
        generate 2 for each with 15 extra for queue

        if ((method == 1) || (method == 3) || (method == 5))
                // boxes divided in initial box division
        prepoints = 1;
        else
        prepoints = numb;

    }
  else if (startcond == 2)
    {   printf("starting cond 2\n");
        for (i = 1; i <= dimen; i++)
          size[i] = (xu[i]-xl[i]);                              //length of box side

      initial2(igen);

        prepoints = numb;
    }
  printf("finished start\n");


        start = 1;                                              // divider should move boxes
        in global array from now on

//DISTRIBUTE INITIAL BOXES HERE

                for (j = 1; j <= 2; j++)                // loop twice to send 2 messages to
        each process
                {
                        for (i = 1; i < nproc; i++)
                        {
```

```
                                msize = dimen+1;

                                if (nextsend >= nextgen)        // Stop if you run out of
        points
                                   break;
/*
// ********* print test area below here ****************
printf("Send job no.%3d to proc no %d : ", nextsend, i);
//for (int k=1; k<=msize; k++)
        printf("%f  ",queue[nextsend][1]);
printf("\n");
// ********* print test area above here ****************
*/

                MPI_Isend(&queue[nextsend][1],msize,MPI_DOUBLE,i,10,
        MPI_COMM_WORLD,&(req[mcount]));     //Non blocked send
                                nextsend++;
                                mcount++;
                             if(mcount == 500)
                                mcount = 0;
                        }
                }
                if ((converg != 1) && ((method == 1) || (method == 5)) && (nextsend ==
        nextgen))
                  wait2 = 1;
                else
                  wait2 = 0;
//MAIN LOOP
                do {
                    m = 0;

//POLL FOR COMPLETED BOXES
                    do {
                        msize = 2;
                        MPI_Recv(&a[1],msize,MPI_DOUBLE,
        MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
                // Blocks until new message received
                        complete = (int)a[1];   //index2 of finished point       // Status
        contains the PID of the sending processor
                        center[complete][1] = a[msize];        //put function value into
        'center' array
/*
// ************ print test area below here ********************
printf("Recv job no.%3d from proc. %d ", (int)a[1], status.MPI_SOURCE);
printf(" result  %13.8f \n", a[2]);
// ************ print test area above here ********************
```

```
*/
                        MPI_Test(&(req[mcheck]),&thought_so,&status2);
                        mcheck++;
                        if(mcheck == 500)
                          mcheck = 0;

                        if (wait2 == 2)                         //Only generate more points
        if one processor is finished
                         {
                          for (j=1; j<=nproc+1; j++ )
                           {
                            if (pfin[j] == status.MPI_SOURCE)
                                  m = 1;
                            if (pfin[j] == 0)
                             {
                               pfin[j] = status.MPI_SOURCE;
                //                 printf("pfin[%d] = %d \n",j,pfin[j]);
                                  break;
                             }

                           }
                         }


                        if (wait2 == 0)                         //Only send more points if
        not waiting for all analyses to complete
                         {
                          msize = dimen+1;
                          dest = status.MPI_SOURCE;       // destination is the rank that
        just send message, in status
/*
// ********* print test area below here ****************
printf("Send job no.%3d send to proc no %d : ", nextsend, dest);
        printf("%6.0f\n",queue[nextsend][1]);
// ********* print test area above here ****************
*/
                          MPI_Isend(&queue[nextsend][1],msize,MPI_DOUBLE,dest,10,
        MPI_COMM_WORLD,&(req[mcount]));    //Non blocked send
                          mcount++;
                          if(mcount == 500)
                                mcount = 0;

                          if(nextsend == qlim)                          // if at the end of the
        queue then start over at the beginning
                                nextsend = 0;
                          nextsend++ ;
```

```
//CHECK FOR ENOUGH BOXES IN QUEUE

                        if (nextsend > nextgen)
                              temp = nextgen+qlim-nextsend;
                        else
                              temp = nextgen-nextsend;

                        if ((method == 1) || ((method == 5) && (itter < switcher)))
                          {
                              if (temp <= 0)                    //generate more points NOW
        if none left
                              {   wait2 = 1;                    // wait2=1 means wait for all
        jobs to finish
                                printf("wait2 = 1  %d  %d  %d\n", nextsend, nextgen, temp);
                              }
                          }
                        else
                          {
                              if (temp <= 0)                    // wait for 1 processor to
        finish before generating more
                                 {
                                   wait2 = 2;                   // wait2=2 means wait for one
        processor to completely finish
//                                 pfin[1] = status.MPI_SOURCE;
                printf("wait2 = 2   %d\n", status.MPI_SOURCE);
                                 }
                          }

                       }


                  if (complete != 0)
                       {
                         if ((complete <= prepoints) || (method == 4))
                              insert(complete);
                         else
                              {
//Check to see if all of the child boxes from a parent are finished if not a prepoints box

                                 for (k=1 ; k<=50; k++)
                                   {
                                     if(children[k][1] == 0)
                                          break;
```

```
                                        l=0;                                    // l= 0 means
        all boxes done
                                        for (j=2; j <= (dimen*2+1); j ++)
                                            {
                                              if (children[k] [j] != 0)
                                                if(center[children[k][j]] [1] == 0)
                                                    {l = 1;                     // l=1 means
        some children not analyzed
                                                      break;
                                                    }
                                            }
                                       if (l == 0)
                                            {   //printf("from parent # %d \n", children[k][1]);
// divide boxes
                                               divide2(k);                      // send the
        parent index to divide on
                                                 for (j=2; j <= (dimen*2+1); j ++)
                                                   if (children[k] [j] != 0)
                                                       {   complete = children[k][j];
//PUT COMPLETED BOX INTO LIST
//      printf("box added is %d  from dimension %d\n", complete, (j/2));
                                                          insert(complete);
                                                  }

//  Remove parent and children from children array since they have been added to the
        index
                                                for (l= k; l<=49; l++)
                                                  for (j=1; j <= (dimen*2+1); j++)
                                                      children[l] [j] = children[l+1] [j];       //
        move lower entries up one in the list
                                             }
                                        }
                                    }
                                }

// Check to see if you should generate new points
                        if (((method == 1) || ((method == 5) && (itter <= switcher)))&&
        (wait2 == 1))
                            {
                              j = 0;
                              for (l=1; l<=50; l++)
                                 if (children[l][1] != 0)                       // check to see that the
        children array is empty
                                       { //printf("parent =%d \n",children[l][1]);
                             j = 1;                                     // j = 1 means some child not
        finished
```

```
                        break;
                      }


                       if (j != 1)
                       {
                         m = 1;                                    //m = 1 means go to
        graham
                      wait2 = 0;
                       }
                    }

                  } while (m != 1);

        converg = 0;
        if ((stopcond == 1) && (order2b[1] < minsize1))                  //stop when
        smallest box reaches certain size
                converg = 1;                                      //stop after this loop if
        boxes small enough
        else if ((stopcond == 4)&& (itter >= maxit))              //stop after maxit
        iterations
             converg =1;

     // printf("converg = %d\n",converg);
                  if(( numb <= final) && (converg != 1))
                       {
// locate p.o. points
                        Graham(center, optimal);

// divide boxes
                        Divider(center, optimal);

                         printf(" %d  %d  \n",nextsend,nextgen);

                        msize = dimen+1;

                        k =1;
                        if((itter == switcher) && (method == 5))
                         for (j = 1; j <= 2; j++)
                              for (i = 1; i< nproc; i++)
                                {
                                      pfin[k] = i;
                                      k++;
                                }
```

```
                        if ((converg !=1) && ((method == 1) || ((method == 5) && (itter
        <= switcher))))
                          {
                              for (j = 1; j <= 2; j++)                    // loop twice to send 2
        messages to each process
                                  for (i = 1; i < nproc; i++)
                                    {
                                      if (nextsend == nextgen)      // Stop if you run out
        of points
                                         break;
/*
// ********* print test area below here ****************
printf("Send job no.%3d to proc no %d : ", nextsend, i);
//for (int k=1; k<=msize; k++)
k=1 ;
        printf("%13.8f  ",queue[nextsend][k]);
printf("\n");
// ********* print test area above here ****************
*/

        MPI_Isend(&queue[nextsend][1],msize,MPI_DOUBLE,i,10,
        MPI_COMM_WORLD,&(req[mcount]));    //Non blocked send
                                    mcount++;
                                    if(mcount == 500)
                                          mcount = 0;

                                      if(nextsend == qlim)
                                              nextsend =0;
                                      nextsend++;
                                      for (k =1; k <=(nproc*2-1); k++)
                                        pfin[k]=pfin[k+1];
                                    }
                          }

                        if ((converg !=1) && ((method == 2) || ((method == 5) && (itter
        > switcher))))
                          {
                              for (i = 1; i <= nproc*2; i++)
                                {
                                    if (nextsend == nextgen)          // Stop if you run out
        of points
                                      break;

                                    if (pfin[1] == 0)
                                      break;
/*
```

```
// ********* print test area below here ****************
printf("Send job no.%3d to proc no %d : ", nextsend, pfin[1]);
//for (int k=1; k<=msize; k++)
k=1 ;
        printf("%6.0f  ",queue[nextsend][k]);
printf("\n");
// ********* print test area above here ****************
*/

        MPI_Isend(&queue[nextsend][1],msize,MPI_DOUBLE,pfin[1],10,
        MPI_COMM_WORLD,&(req[mcount]));    //Non blocked send
                                        mcount++;
                                        if(mcount == 500)
                                              mcount = 0;

                                           if(nextsend == qlim)
                                             nextsend =0;
                                           nextsend++;

                                           for (j=1; j<=nproc+1;j++)
                                             pfin[j] = pfin[j+1];
                                      }
        //                           for (j=1;j<=nproc+1;j++)
        //                             printf("pfin[%d] = %d\n",j,pfin[j]);

                                   }
                                if ((converg !=1) && (nextsend == nextgen))
                                  {
                                    if ((method == 1) || ((method == 5) && (itter < switcher)))
                                          wait2 = 1;
                                    if ((method == 2) || ((method == 5) && (itter >= switcher)))
                                          wait2 = 2;
                                  }
                                else
                                       wait2 =0;
                              }
                        printf("nsort = %d, numb = %d, wait2 = %d, itter = %d, converg = %d
        \n",nsort,numb, wait2, itter, converg);
                itter++;
                } while (converg != 1  &&  numb <= final || wait2 == 1 );

                printf("finished\n");

// *********Finished Optimization--Present Results***********

                if (print == 1)
```

```
                {
                        fprintf(fp2, "numb =%3d   itter =%3d    converg =%1d\n", numb,
        itter, converg);
                        for (i = 1; i <= numb; i++)
                          {
                                fprintf(fp2, "%5d  %10.5f",i,center[i][1]);
                                for (j = 2; j <= (2*dimen+1); j++)
                                fprintf(fp2, " %7.4f", center[i][j]);
                                fprintf(fp2, "\n");
                          }
                }

// finish divide boxes, send all processors stop calculation

                for (i = 1; i < nproc; i++)
                 {
                  printf("stopping proc %d \n",i);

        MPI_Isend(&queue[nextsend][1],0,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&
        req1);
                 }

                fp3 = fopen("best.out", "a+");                  // open file "best.out" here

                fprintf(fp3, "\n best points\n");
                fprintf(fp2, "\n best points\n");

                for (i = 1; i <= numb; i++)
                {
                   if (center[i][1] == 0)
                     continue;

                        for (j = 1; j <= nbest; j++)
                        {
                                if (best[j] == 0)
                                {
                                        best[j] = i;
                                        break;
                                }
                                if (center[i][1] < center[best[j]][1])
                                {
                                        for (k = nbest; k >= (j+1); k--)
                                        best[k] = best[k-1];
                                        best[j] = i;
                                        break;
                                }
```

```
                }
        }

//      fp4 = fopen("results.dat", "a+");

//      fprintf(fp4," \nResults from %3d dimensions required %5d function
        evaluations\n",dimen, numb);
        fprintf(fp3," %d ",best[1]);
        for (k=1; k<=(dimen+1); k++)
                fprintf(fp3,"%6f ",center[best[1]][k]);
        fprintf(fp3,"\n");

//      fclose(fp4);

                for (j = 1; j <= loc; j++)
                        points[j] = 0;

                points[1] = 1;
                k = 0;
                for (j = 1; j <= nbest; j++)
                {
                        if (k == loc)
                                break;
                        calls = 0;
                        m=0;
                        if (k != 0)
                        {
                                for (l = 1; l <= k; l++)
                                {
                                        dist = 0;
                                        for (i = 1; i <= dimen; i++)
                                                dist += pow((center[best[points[l]]][i+1]-
        center[best[j]][i+1])
                                                /size[i],2);
                                        dist = sqrt(dist);
                                        if (dist < mindist)
                                        m=1;
                                }
                        if (m<1)
                                points[k+1] = j;
                        }
/*
                        if (m<1)
                {
                        k++;
                        for (i = 1; i <= dimen; i++)
```

```
                        xopt[i] = center[best[j]][i+1];

                func = center[best[j]][1];
// **********Call DOT here  (for now print out best points *********

                        fprintf(fp2, "%5d  %12.9f",best[j],center[best[j]]
        [1]);
                        for (i = 2; i <= (2*dimen+1); i++)
                        fprintf(fp2, "%7.4f", center[best[j]][i]);
                        fprintf(fp2, "\n");
        }

                dist = 0;
        for (i = 1; i <= dimen; i++)
                dist += fabs(center[best[j]][i+1]-xopt[i]);

        if (func < fbest)
        {
                fbest = func;
                for (i = 1; i <= dimen; i++)
                        xbest[i] = xopt[i];
        }
*/

        }
//      if ((stopcond==1)||(stopcond==4))
//              * print something out *

        fclose(fp3);                    // close file "best.out" here

    fprintf(fp1,"\n %6d %4d ", numb,  itter);
    for (k=1; k<=(dimen+1); k++)
        fprintf(fp1,"%6f ",center[best[1]][k]);

    fclose(fp2);                        // close file "points.dat" here
    fclose(fp1);
    endtime = MPI_Wtime();
//  printf("Total time = %.16f\n",endtime-starttime);                   // close file
    "high.dat" here

      fprintf(fp5,"\nMethod=%2d  agressive=%2d  stopcond=%2d  startcond=%2d
    nproc = %3d  switcher=%3d  minsize1= %.5f  minsize2= %.4f  iterations= %4d
    Total_time= %.8f  FE= %6d   \n ",method,agressive, stopcond, startcond, nproc,
    switcher, minsize1, minsize2, itter, endtime-starttime, numb);
    for (k=1; k<=(dimen+1); k++)
        fprintf(fp5,"%6f ",center[best[1]][k]);
```

```
        fprintf(fp5,"\n");

   fprintf(fp5,"---------------------------------------\n");

   fclose(fp5);


   }               //end Master processor part

/*************************************************/
/* Slave processor part */
/*************************************************/

   if (pid != master)
   {
        m=0;
        MPI_Bcast(&e[1],dimen,MPI_DOUBLE,master,MPI_COMM_WORLD);
        msize = dimen + 1;
        while (1)
        {
                MPI_Recv(&data_in[1],msize,MPI_DOUBLE,master,MPI_ANY_TAG,
        MPI_COMM_WORLD,&status);
                if (status.MPI_TAG == 0)
                        break;
                else
                {

                        for (i=1; i<= (dimen); i++)
                                y[i] = data_in[i+1];
                        answer[1] = data_in[1];
                        answer[2] = Eval(y);
                        if(m==1)
                          {
                           MPI_Test(&(req[mcheck]),&thought_so,&status);
                           mcheck++;
                          }

                MPI_Isend(&answer[1],2,MPI_DOUBLE,master,status.MPI_TAG,MPI_
        COMM_WORLD,&(req[mcount]));
                        m=1;
                mcount++;
                   if(mcount == 500)
                        mcount = 0;
                   if(mcheck == 500)
                        mcheck = 0;
```

```
                }
        }
  }
//      End Slave Processor part

  MPI_Finalize();
}


/*********************************************************************/
/* Subroutine Graham's scan                                          */
/*********************************************************************/
void Graham(double center[nbox+1][2*dimen+3+1], int *optimal)
{
  double order2[nbox+1];
  double z, o;
  int order[nbox+1], temp;
  int yy, initial, m, i, j, k, l, extra, extra2;

        for (i = 1; i <= (nsort); i++)          //zero the local order arrays
        {
                order[i] = 0;
                order2[i] = 0;
        }
        m=0;
        if (start != 0)
   {
        do{

           if ((order2b[1] <=minsize1) && ((stopcond == 1)||(stopcond == 4)))
                //removing boxes that are too small
           {    for (i=1; i<=nsort; i++)
                 { order2b[i] = order2b[i+1] ;
                   index2[i] = index2[i+1];
                 }
                  index2[nsort] =0;
                  order2b[nsort] =0;
                  nsort-- ;
           }
            else if ((order2b[1] <= minsize2) && (stopcond == 2))          //removing
        boxes that are too small
           {    for (i=1; i<=nsort; i++)
                 { order2b[i] = order2b[i+1] ;
                   index2[i] = index2[i+1];
                 }
                  index2[nsort] =0;
```

```
            order2b[nsort] =0;
            nsort-- ;
   }
    else if ((order2b[1] <=minsize2) && (stopcond == 3))                //removing
boxes that are too small
   {  printf("removing box #1 \n");
          for (j=2; j<=nsort; j++)
     {
            dist =0;
            for (i=(2); i<=(dimen+1); i++)
                    dist += pow((center[j][i]-center[index2[1]][i])/size[i], 2);
            dist = sqrt(dist);
            if (dist < radrem)
       {  printf("removing box #%d due to radius",j);
             for (i=j; i<=nsort; i++)
              { order2b[i] = order2b[i+1] ;
                index2[i] = index2[i+1];
              }
             index2[nsort] =0;
             order2b[nsort] =0;
             nsort-- ;
           }
     }
          for (i=1; i<=nsort; i++)
           { order2b[i] = order2b[i+1] ;
             index2[i] = index2[i+1];
           }
            index2[nsort] =0;
            order2b[nsort] =0;
            nsort-- ;
   }
    else
          m =1;
  }while (m == 0);
}                                                                       //end box removal

j=2;
k=2;
m=0;
order[1] = index2[1];
order2[1] = order2b[1];

do
{
          if (order2b[j] > order2b[j-1])
```

```
                { temp =j;                                  //temp is the start of the
        largest boxes
                  order[k] = index2[j];
                  order2[k] = order2b[j];
                 if (j <= (nact-1))
                  extra = index2[j+1];
                 if (j <= (nact-2))
                  extra2 = index2[j+2];
                  k ++;
                }

        if (j >= numb)
                  m=1;

                j ++;
        } while (m == 0);

        nact = k - 1;

//Start Convex Hull version of DIRECT
        initial = 1;

        i = 1;
        j = 2;
        k = 3;

//Conversion to sample more points at the beginning if > 0
        if (itter > 0)
        {
                while (k <= nact)                   //remove right turns & keep left
        turns
                {
                        o = order2[i]*(center[order[j]][1]-center[order[k]][1])
                                -center[order[i]][1]*(order2[j]-order2[k])
                                +(order2[j]*center[order[k]][1]-
        order2[k]*center[order[j]][1]);
                        if (o > 0.000000000000)
                        {
                                i = i+1;
                                j = i+1;
                                k = j+1;
                        } else
                        {
                                for (l = j; l <= nact; l++)
                                {
                                        order[l] = order[l+1];
```

```
                                order2[l] = order2[l+1];
                        }
                        order[nact] = 0;
                        order2[nact] = 0;
                        nact--;
                        if (i > 1)
                        {
                                k = i+1;
                                j = i;
                                i = i-1;
                        } else
                        {
                                j = i+1;
                                k = i+2;
                        }
                }
        }
}
m = 0;
while (m != 1)                                          //remove left side of convex
hull
{
        if ((center[order[initial+1]][1]-center[order[initial]][1] < 0.000000000000)
                && (center[order[initial+1]][1] != 0))
                initial++;
        else
                m = 1;
}


if ((nact-initial) > nopt)
        nact = nopt+initial-1;
for (i = 1; i <= (nact-initial+1); i++)
        optimal[i] = order[i+initial-1];
for (i = nact+2-initial; i <= nopt; i++)
        optimal[i] = 0;

nact = nact-initial+1;
if (((agressive == 1) || (agressive == 2)) && (extra != 0))
{
nact ++;
  optimal [nact] =  extra;
}

if ((agressive == 2) && (extra2 != 0))
{
```

```
          nact ++;
            optimal [nact] =  extra2;
          }

          if(agressive == 3)
          {  for (i=1; i<=nopt; i++)
             optimal[i] = index2[temp+i-1];
             printf("%d   %d\n",optional[1],optional[2]);
             if (nact > 1)
              { for (i=1; i<=(nact-1); i++)
                optimal[i] = optimal [i+1];
                optimal[nact] =0;
                nact --;
              }
             else
              { for (i=1; i<=(nopt-1); i++)
                optional[i] = optional [i+1];
                optional[nopt] =0;
              }
          }

printf("optimal     optional    iteration # %3d   nact = %d \n", itter, nact);
for (j = 1; j <= nact; j++)
          printf("  %4d   %4d \n", optimal[j],optional[j]);
printf("\n");
}

/*********************************************************************/
/* Subroutine to divide the potentially optimal boxes into smaller boxes                */
/*********************************************************************/

void Divider(double center[nbox+1][2*dimen+3+1], int *optimal)
{
  double lng, z, t;
  int divide[2*dimen+1], split[2*dimen+1], dim[dimen+1],net,agg;
  int i, k, j, l, m, ll, temp, yy, gstart, gfin, count;

  nold = numb;                                          //start with first optimal point in
          "optimal"
  agg = 0;
  m = 0;
  while (true)
    {
      m ++;
      if ((numb+2*dimen) > nbox)
          {
```

```
            printf("skipping box %d and after\n\n ",optimal[m]);
            break;
           }

      if (((m > nopt) || (optimal[m] < 0.5)) && (agg == 0))
           {  agg = 1;  m =1;  count = numb; }

      if (agg == 0)
           k = optimal[m];
      else
           {
            if ((m > nopt) || (optional[m] < .5) || ((numb - count) >=(2*nproc)))
              break;
            else
              k = optional[m];
           }

      if (dd ==1)
           {
            j=0;
            for (i=1; i<=50;i++)
              {
                if (children[i][1] == k)
                  {
 //printf(" %d   %d ",children[i][1],k);
                    j=1; break;
                      }
              }
            if(j == 1)
              continue;
           }

//         printf("agg = %d\n",agg);
//         printf("  %d  is an optimal box and will be divided\n", k);
//         printf(" m = %d  order2b = %f\n ",m, order2b[optimal[m]]);

           lng = 0.0;                                                  //only divide
           dimensions that are longer than the rest
      z = 0;
      for (i = 1; i <= dimen; i++)
           {
            dim[i] = 0;
            z += pow(center[k][i+dimen+1],2);
            if (center[k][i+dimen+1] > lng)                        //longest dimension
              lng = center[k][i+dimen+1];
           }
```

```
z = sqrt(z);
if (((stopcond == 1)||(stopcond == 3)||(stopcond == 4))&& (z < minsize1))       //don't
     divide if smaller than min box size
     continue;

if (center[k][2*dimen+2] < center[k][2*dimen+3])                    //skip if not divided
     this time
     {
      center[k][2*dimen+2] = center[k][2*dimen+2]+1;
      continue;
     }
else
     center[k][2*dimen+2] = 1;

j = 1;
net = 0;
for (i = 1; i <= dimen; i++)
     {
      if (fabs(center[k][i+dimen+1]-lng) < 0.0001)
        {
         dim[j] = i;
         j++;
         net++;                                                //# of dimensions to divide
        }
     }

for (i = (numb+1); i <= (numb+net*2); i++)                                        //create
     the new points
     for (l = 2; l <= (2*dimen+1); l++)
      center[i][l] = center[k][l];

for (i = 1; i <= net; i++)
     {
       center[numb+2*i-1][dim[i]+1] += center[numb+2*i-
     1][dim[i]+dimen+1]/3*size[dim[i]];
       center[numb+2*i][dim[i]+1] -=
     center[numb+2*i][dim[i]+dimen+1]/3*size[dim[i]];
       center[k][dim[i]+dimen+1] = center[k][dim[i]+dimen+1]*1/3;
     }

for (i = 1; i <= (net*2); i++)
     {
      queue[nextgen][1] = numb+i;
      for(j = 2; j <= (dimen+1); j++)                                  // <------- edit
     j<=(dimen+1) instead of (dimen+2) ..3/28/01
      queue[nextgen][j] = center[numb+i][j];              // stores x position in queue
```

```
          if (nextgen == qlim)                    // if at the end of the queue then start over at
        the beginning
            nextgen = 0;
          nextgen++;                              //increment the counter for generating the
        points
        }

// move center[k] in the global index

    z=0;
    for (i=(dimen+2); i<=(2*dimen+1); i++)
        z += pow(center[k] [i], 2);
    z=sqrt(z);

    for(j=1; j<=nsort; j++)
        {
          if (( z < order2b[j]) || ((z == order2b[j]) && (center[k][1] <
        center[index2[j]][1])))
            {
              gstart = j;                         // gstart is the point where the box should be
        now
              break ;
            }
        }

    for (j = gstart; j <=nsort; j++)
        {
          if (index2[j] == k)
            {
              gfin = j;                    // gfin is the current location of the box
              break ;
            }
        }

    for ( j= gfin ; j >= (gstart+1); j--)
      {
          index2[j]  = index2[j-1] ;
          order2b[j] = order2b[j-1];
        }

    index2 [gstart] = k;
    order2b[gstart] = z;
//end of move
```

```
    if(((start == 0) && ((method != 1) && ((method != 5) || (itter >= switcher)))) ||
        (method == 4))
        {
//randomly sort points
          for (i = 1; i <= (2*net); i++) //start with the points in the order that they were
        generated
            divide[i] = i+numb;

          for (i = 1; i <= (2*net); i++) //randomly switch each point with some other
            {
              t = rand();
              j = (int)((t*2*net)/RAND_MAX + 1);
              temp = divide[i];
              divide[i] = divide[j];
              divide[j] = temp;
            }

//divide the new boxes based on random order
          for (i = 1; i <= net; i++)                    //divide the boxes based on
        the sorted values
            {
              split[i*2] = 0;                    //has the box been divided completely yet
              split[i*2-1] = 0;
            }
          for (i = 1; i <= (2*net); i++)
            {
              k = divide[i]-numb;
              if (split[k] != 1)
                  {
                    for (j = (numb+1); j <= (numb+2*net); j++)
                      {
                        if (split[j-numb] != 1)
                            center[j][dim[(k+1)/2]+dimen+1] =
        center[j][dim[(k+1)/2]+dimen+1]/3;
//                  printf(" box %d was just divided in dimension %d \n",j, dim[(k+1)/2]);

                  }
                    split[k] = 1;
                    if ((k/2) == ((k+1)/2))
                      split[k-1] = 1;
                    else
                      split[k+1] = 1;
                  }
            }
        }
```

```
    if (((start >= 1) && (method != 4)) || ((method == 1) || ((method == 5) && (itter <
        switcher))))
        {
// put divided boxes into children array
         for (j=1; j<=50; j++)
           {
            if (children[j][1] ==0)
              {
                  children[j][1] = k;
               ll = numb +1;              // the number of the next child to put in array
               for (i=1; i<=net; i++)
                 {
                     children[j][dim[i]*2]= ll;
                  ll ++;
                  children[j][dim[i]*2+1] = ll;
                  ll ++;
//        printf(" boxes %d and +1 were just put in children position %d and +1 \n",(ll-2),
        (dim[i]*2) );
                 }
               break;
              }
           }
        }
    numb = numb+2*net;

    if(nextsend <= nextgen)
        {
         temp = nextsend+qlim-nextgen;
         if (temp <= (2*dimen))                  //don't generate more points if the queue
        might overlap itself
           {
            printf("skipping box %d and after,queue too small\n\n ",optimal[m+1]);
            break;
           }
        }
    else
        {
         if ((nextsend-nextgen) <= (2*dimen))
           {
            printf("skipping box %d and after,queue too small\n\n ",optimal[m+1]);
            break;
           }
        }
    if ((method == 5) && (itter <= switcher) && (agg == 1))
        break;
  }
```

```
// add blank lines to the queue
  if((method == 3) && (start > 0))
    {
     for (i = 1; i <= ((nproc-1)*2); i++)
          {
           queue[nextgen][1] = 0;
           for(j = 2; j <= (dimen+1); j++)
             queue[nextgen][j] = 0;                    //stores x position in queue

           if(nextgen == qlim)                         //if at the end of the queue then start
          over at the beginning
             nextgen = 0;
             nextgen++;                                //increment the counter for
          generating the points
          }
    }
}
/*************************************************************************/
/* Subroutine Insert   --  adds new points to the list                   */
/*************************************************************************/
void insert(int notnew)
{
   double z;
   int m, i, j, k, l;

// sort points by distance
          z = 0;
          for (k = (dimen+2); k <= (2*dimen+1); k++)
                    z += pow(center[notnew][k],2);
          z = sqrt(z);                                                    //distance to
          edge for box i
          for (j = 1; j <= nsort+1; j++)
          {
                    if (index2[j] == 0)
                              //if at end of list put box here
                    {
                              index2[j] = notnew;
                              order2b[j] = z;
                              break;
                    }
                    if (z < order2b[j])
                              //if smaller than this entry insert here
                    {
                              for (k = nsort+1; k >= (j+1); k--)
                              {
```

```
                        index2[k] = index2[k-1];
                        order2b[k] = order2b[k-1];
                }
                index2[j] = notnew;
                order2b[j] = z;
                break;
        }
//              if ((z == order2b[j]) && (center[i][1] < center[index2[j]][1])) //if better
        and same size insert here
        if ((z == order2b[j]) && (center[notnew][1] < center[index2[j]][1]))
        {
//              for (k = i; k >= (j+1); k--)
                for (k = nsort+1; k >= (j+1); k--)
                {
                        index2[k] = index2[k-1];
                        order2b[k] = order2b[k-1];
                }
//              index2[j] = i;
                index2[j] = notnew;
                order2b[j] = z;
                break;
        }
    }
    nsort++ ;
}

/*****************************************************************************/
/* Subroutine for initial selection for divisions                            */
/*****************************************************************************/

void initial(double center[nbox+1][2*dimen+3+1], int *optimal)
{
  double order2[qlim+1];
  double z, o;
  int order[qlim+1];
  int yy, initial, m, i, j, k, l;

// sort points by distance

    for (i = 1; i <= qlim; i++)             //zero the local order arrays
    {
        order[i] = 0;
        order2[i] = 0;
    }

    for (i=1; i<=(nopt+1); i++)
```

```
            optimal[i] = 0;

        for (i = 1; i <= numb; i++)                    //sort all of the points largest to
        smallest
        {
            z = 0;
            for (k = (dimen+2); k <= (2*dimen+1); k++)
                z += pow(center[i][k],2);
            z = sqrt(z);                                //distance to edge for box i
            for (j = 1; j <= numb; j++)
            {
                if (order[j] == 0)                          //if at end of list put
        box here
                {
                    order[j] = i;
                    order2[j] = z;
                    break;
                }
                if (z >= order2[j])                         //if larger than this
        entry insert here
                {
                    for (k = i; k >= (j+1); k--)
                    {
                        order[k] = order[k-1];
                        order2[k] = order2[k-1];
                    }
                    order[j] = i;
                    order2[j] = z;
                    break;
                }
            }
        }
// end sorting
        m = 0;
        j = 2;
        k = 2;
        optimal[1] =order[1] ;
        while (m == 0)
        {
            if (order2[j]==order2[1])                   //if the box stays the same
        size
            {
                optimal[k] = order[j];
                k++;
            }
```

```
            if ( ( j>numb )||( k>nopt )||( (k*dimen+numb) >= qlim ) )          //exit
        if you reach the end of index2 or optimal
                    m = 1;
            j++;
        }

        nact = k-1;

}

/*********************************************************************/
/* Subroutine Divide2                                                */
/*********************************************************************/

void divide2(int parent)
{
  double lng, z;
  int divide[2*dimen+1], split[2*dimen+1], dim[dimen+1],net;
  int i, k, j, l, m, start;

        for (i=2; i<=(2*dimen+1); i+= 2)
        { if (children[parent][i] >=1 )
          { start = children[parent][i] ;
//          printf("start is %d, dimen is %d,  Parent is %d \n", start, (i/2), parent);
            break;
          }
        }

//divide the new boxes based on the values at the centers

    for (i = 1; i <= dimen; i++)
        dim[i] = 0;                         // the dimensions to divide on

// find number of new boxes
        net = 0;                            // the number of new boxes
        j=1;
        for (i=2; i<=(2*dimen+1); i+= 2)
          {  if (children[parent] [i] >= 1)
            { dim[j] = i/2;
              net++;
              j++;
            }
          }

//sort points
                for (i = 1; i <= (2*net); i++)
```

```
        divide[i] = 0;

                for (i = start; i <= (start+2*net-1); i++)
    {
        for (j = 1; j <= (2*net); j++)
      {
        if (divide[j] == 0)
        {
                divide[j] = i;
          break;
        }
        if (center[i][1] - center[divide[j]][1] < -0.00000000000001)
        {
                for (k = (2*dimen); k >= (j+1); k--)
                divide[k] = divide[k-1];
          divide[j] = i;
          break;
        }
      }
    }
    for (i = 1; i <= net; i++)
                //divide the boxes based on the sorted values
    {
        split[i*2] = 0;                         //has the box been divided completely yet
     split[i*2-1] = 0;
    }
    for (i = 1; i <= (2*net); i++)
    {
        k = divide[i]-start+1;
      if (split[k] != 1)
      {
        for (j = start; j <= (start-1+2*net); j++)
        {       if (split[j-start+1] != 1)
                center[j][dim[(k+1)/2]+dimen+1] = center[j][dim[(k+1)/2]+dimen+1]/3;
        }
                split[k] = 1;
        if ((k/2) == ((k+1)/2))
                //integer division????
                split[k-1] = 1;
        else
                split[k+1] = 1;
      }
    }
}

/*****************************************************************************/
```

```
/* Subroutine Initial2                                                    */
/************************************************************************/

void initial2(int igen)
{
  int a[dimen], b[dimen];                    // indexes for division
  int i,j,k,l,m, prod;

  for (i=1; i<=dimen; i++)
    {
      a[i] = 1;
      b[i] = 1;
    }

// Find number of divisions to make
  m = 0;
  do {
    for (i = 1; i<=dimen; i++)
     {
      a[i] ++;
      prod = 1;
      for (j = 1; j<= dimen; j++)
        prod = prod *a[j];
      if (prod >= igen)
        {
          m = 1;
            break;
         }
    }

  }while (m == 0);

// Divide Boxes
  j = 1;
  m = 0;
  do {
     for (i = 1; i <= dimen; i++)
       {
         center[j][i+1] = xl[i] + size[i]/a[i]*(b[i]-0.5);
         center[j][i+1+dimen] = 1./a[i];
       }
     j ++;

     for (k =1; k<=dimen; k++)
       if (b[k] < a[k])
         {
```

```
            b[k] ++;
            for (l = 1; l <=(k-1); l++)
              b[l] =1;
            break;
          }
        else if (k == dimen)
          m = 1;
      } while (m == 0);

  numb = j-1;
  nextgen --;

          for (i = 1; i <= numb; i++)
          {
                  queue[nextgen][1] = i;
                  for(j = 2; j <= (dimen+1); j++)
                          queue[nextgen][j] = center[i][j];       // stores x position in queue

                  if(nextgen == qlim)                    // if at the end of the queue then start
          over at the beginning
                          nextgen = 0;
                  nextgen++;                             //increment the counter for
          generating the points
          }
}


/*************************************************************************/
/* Subroutine Eval                                                       */
/*************************************************************************/

double Eval(double *y)
{
   double obj = 0, a;
   int l,m;

   m=0;                                  // Quartic Function
   if (m==1)
          for (l = 1; l <= dimen; l++)
            obj += 10 + 2.2*pow((y[l]+e[l]),2)-1*pow((y[l]+e[l]),4);//-1.3*y[l];

   m=1;
   if (m==1)                             //Griewank Function
     {
          double x[21]={0};
```

```
        for(l=1;l<=dimen; l++)
          {
            x[l] = y[l];
            obj += pow(y[l],2);
          }
        obj =obj/20000.;
        obj += (1.-cos(x[1])*cos(x[2]/sqrt(2.))*cos(x[3]/sqrt(3.))*
            cos(x[4]/2.)*cos(x[5]/sqrt(5.))*cos(x[6]/sqrt(6.))*
            cos(x[7]/sqrt(7.))*cos(x[8]/sqrt(8.))*cos(x[9]/3.)*
            cos(x[10]/sqrt(10.))*cos(x[11]/sqrt(11.))*cos(x[12]/sqrt(12.))*
            cos(x[13]/sqrt(13.))*cos(x[14]/sqrt(14.))*cos(x[15]/sqrt(15.))*
            cos(x[16]/4.)*cos(x[17]/sqrt(17.))*cos(x[18]/sqrt(18.))*
            cos(x[19]/sqrt(19.))*cos(x[20]/sqrt(20.)));
 }

m=0;                                    // Upside Down Quartic Function
if (m==1)
  for (l = 1; l <= dimen; l++)
    obj -= 1+2.2*pow((y[l]+e[l]),2)-1*pow((y[l]+e[l]),4)+ 2*y[l];

m=0;                                    // Cones, min follows trend
if (m==1)
  for (l=1;l<=dimen;l++)
    {
      obj-= .5*(y[l]+2);

        if((y[l] < 1.2) && (y[l] >0.8))
                obj -= 4-20*fabs(y[l]-1);
        else if((y[l] < -0.2) && (y[l] > -0.6))
                obj -= 4-20*fabs(y[l]+0.4);
        else if((y[l] < -1.4) && (y[l] > -1.8))
                obj -= 4-20*fabs(y[l]+1.6);
    }

m=0;                                    // Cones, min is anti-trend
if (m==1)
  for (l=1;l<=dimen;l++)
    {
      obj-= .5*(y[l]+2);

        if((y[l] < 1.2) && (y[l] >0.8))
                obj -= 4-20*fabs(y[l]-1);
        else if((y[l] < -0.2) && (y[l] > -0.6))
                obj -= 6-30*fabs(y[l]+0.4);
        else if((y[l] < -1.4) && (y[l] > -1.8))
                obj -= 8-40*fabs(y[l]+1.6);
```

```
    }

        for (l = 1; l <= evaluation; l++)
                a=1.1*1.34;

        return obj;
}
```

APPENDIX D
Multifidelity DIRECT Code

The multifidelity DIRECT code was written in Fortran 77 with some portions taking advantage of Fortran 90 constructs. The parameters are all set at compile time and the code is currently set up to have the analysis coded as a subroutine within the code which makes the code completely self contained, but forces a new compilation after any change. There are two versions of the multifidelity code, the *C*onstant *C*orrection *F*actor (CCF) and the *L*inear Correction *R*esponse *S*urface (LRS) versions.

CCF Multifidelity DIRECT code

```
c       Direct optimizer, Multifidelity CCF version-sequential setup
c       Steven Cox
c       October '00
c*************************************************************************
        module param
        integer, parameter :: nbox = 100001, dimen = 10, d2 = dimen+2      !used in every
module
        integer, parameter :: print = 0
        end module param

        module dividing
        parameter (agressive = 0)                                  !used in Graham
routime to select method
        real(8),parameter::minsize1=0.0001,minsize2=0.001,radius=0.05    !stopping box
sizes and radius of removal
        parameter (nopt = 800, stopcond = 4)                       !maximum number of
potentially optimal boxes
        integer, parameter :: loc = 15 ,maxit = 100
        end module dividing

        use param
```

```
      use dividing

      parameter nbest = 8000                              !parameters for the main
routine only

c     declare variables
      implicit double precision (a-g,o-z) , integer (i-n)
      real(8), parameter :: mindist = 0.05

      double precision xopt(dimen),x(dimen),points(15)    !point to pass to DOT
      double precision center(nbox,(2*dimen+5)),func,dist       !value, position and
dimentions of each point
      integer optimal(nopt),best(nbest),dropped, totmult              !potentialy
optimal points
      integer totcalls,calls,count,converg,nact                 !counters for FE used
by DOT
      double precision xl(dimen),xu(dimen),xbest(dimen),fbest         !current best
box and function value
      double precision local
      common /design/ numb,ni,nold                              !number of boxes,
max number of boxes
      common /sizes/ size(dimen)
      common /bound/ xl,xu,e(dimen)                             !box limits
      common /cross/ calls
      common /itteration/ itter
      common /convergence/  local(loc,d2),converg,count
      common /sorter/ order2b(nbox), index(nbox),nsort,nact

c*************************************************************************
c     Main Code
c*************************************************************************
c*************************************************************************
      open (4,file='grie_CRS.dat',access='append')
      open(7,file = "CRS_steps_grie.dat")   !,access='append')
      xu(1)=rand(267)*10

      do kr = 1,30
c     setup cycles
      optimal(1)=1
      nold=1
      numb = 1
      ni=0
      final = nbox-2*dimen                !buffer size so you can't overflow center
      converg = 0
      fbest = 9999090
      totmult = 0
```

```
        totcalls = 0
        itter =1
        count =0
        nsort = 0
        nact = 0
        calls = 1

c*****************  define box size  *****************

c************************* Griewank Function ***************
        do i=1,dimen
        xu(i) =  100. + rand(0)*800.
        xl(i) =  xu(i)-1000.
        e(i) =  45+rand(0)*40!
        enddo
c************************* Box Function ********************
c       do i=1,dimen
c       xu(i)=2
c       xl(i)=-2
c       e(i) = .2+rand(0)*.2! .3!
c       enddo
c*************************

        do i=1,nbox
         do j=1,dimen+3
          center(i,j)= 0
         enddo
        order2b(i)=0
        index(i) =0
        center(i,2*dimen+2)=1
        center(i,2*dimen+3)=1
        center(i,2*dimen+4)=0
        center(i,2*dimen+5)=0
        enddo

c       find dimentions

        do i=1,dimen
                center(1,i+1)=(xu(i)+xl(i))/2           ! center point of box
                center(1,i+dimen+1)=1                   ! each box is 1 unit long in
each dim
                size(i)=(xu(i)-xl(i))                   ! length of box side
        enddo

c       analyze the function value at the center, high and low fidelity
```

```
        call ehigh(center)
        correct =0
        call eval(center,correct)

        center(1,1) =  center(1,2*dimen+5)          !corrected value is the high fid value

c       locate potentially optimal points
25      continue

        call graham(center,optimal)

c       divide optimal boxes

        call divider(center,optimal)

        if (print .eq. 1) then
        write(7,*)numb
        do i=1,numb
        write(7,32)(center(i,j),j=1,2*dimen+5)
        enddo
        write(7,*)
        endif

32      format(2x,20(f9.4,2x))

c      if (kr .lt. 28) converg = 1

c       loop untill convergence criteria is met
        write(6,*)"         ",calls, kr
        if (((numb .lt. final).and.(converg.ne.1)))then
        go to 25
        endif

        totmult = calls

c************
        if (print .eq. 1) then                  !gives output file of all points
        open (5,file = 'points.dat')

        do i= 1,numb
        write(5,3)i,(center(i,j),j=1,dimen+1),(center(i,2*dimen+j),j=4,5)
        enddo
3       format(i6,x,21f10.4)
        endif
c************
        open (3,file = 'best.out',access = 'append')
```

```
c       sort points
        do i=1,nbest
        best(i)=0
        enddo

c*********************                    !add removed points if stopcond = 2 or 3
        if (stopcond .eq. (2.or.3)) then
        do i=1,loc
          do j=1,loc
           if (best(j) .eq. 0) then
                best(j) = local(i,dimen+1)
                exit
           endif
           if (center(local(i,dimen+1),1) .lt. center(best(j),1)) then
                do k=loc,j+1,-1
                        best(k) = best(k-1)
                end do
                best(j)=i
                exit
           end if
          enddo
        enddo
        endif
c*********************                    !finish adding removed points

        do i=1,numb
          do j=1,nbest
                if (best(j) .eq. 0) then
                        best(j) = i
                        exit
                endif
                if (center(i,1) .lt. center(best(j),1)) then
                        do k=nbest,j+1,-1
                                best(k) = best(k-1)
                        end do
                        best(j)=i
                        exit
                end if
          enddo
        enddo


        do k=1,nbest
        write(3,23) k,best(k), (center(best(k),j),j=1,dimen+1)
        enddo
23      format(3x,i4,x,i6,f9.3,20f6.2)
```

```
      write(3,*)

      do j=1,loc
      points(j)=0
      enddo

       points(1)=1
      k=0
      do j=1,nbest                                  !loop to find best points separated by
%
      if (k.eq.loc) exit
      calls = 0
      if (k.ne.0) then
      do l=1,k
       dist = 0.
       do i=1,dimen
        dist = dist+((center(best(points(l)),i+1)-center(best(j),i+1)
     &  )/size(i))**2
       enddo
       dist = sqrt(dist)
       if (dist.lt.mindist) goto 12
      enddo
       points(k+1)=j
      endif
       k=k+1

c     write(3,*)best(j)
c     write(3,*)(center(best(j),i),i=1,dimen+1)

      do i=1,dimen
      xopt(i)=center(best(j),i+1)
      enddo
      func = center(best(j),1)

c     write(4,300)k,func,(xopt(i),i=1,dimen)
300   format(20f9.4)
c*******
      call dotcntl (xopt,func)              !Here DOT is called to do local opt
c*******

       dist = 0.
       do i=1,dimen
        dist = dist+abs(center(best(j),i+1)-xopt(i))
       enddo

c     write(3,348),'calls =',calls,'Distance moved =',dist
```

```
348     format(x,a7,x,i4,8x,a16,f10.6)

        totcalls = totcalls+calls

c       write(4,*)func,fbest

c       update best optimum location
        if (func .lt. fbest) then
         fbest = func
         do i=1,dimen
          xbest(i) = xopt(i)
         enddo
        endif

12      continue
        enddo

        write(3,213),"high fidelity calls =",totcalls,
    & " ,  Low fidelity calls =",numb ,"Iterations =",itter
213     format(2x,a21,x,i5,a24,x,i6,2x,a12,x,i4)

c****************************          !First and Fourth Stopping
Method
        if ((stopcond .eq. 1).or.(stopcond .eq.4)) then

        write(4,187)kr,fbest,totmult,totcalls,numb,(xbest(i),i=1,dimen)
c       write(4,16)(e(i),i=1,dimen)
187     format(x,i5,x,f11.5,x,3i9,2x,20f11.6)
16      format (11x,20f11.6)
        endif

c****************************          !Second Stopping Method

        if(stopcond .eq. 2) then
        dropped = 0

        do i=1,loc
        dropped = dropped+ local(i,dimen+2)
        enddo

        write(4,186)kr,fbest,totcalls,numb,dropped,(xbest(i),i=1,dimen)
186     format(x,i5,x,f11.5,x,3i7,2x,20f11.6)

        endif

c****************************          !Third Stopping Method
```

```
        if(stopcond .eq. 3) then

        write(4,187)kr,fbest,totcalls,numb+loc,(xbest(i),i=1,dimen)
c       write(4,16)(e(i),i=1,dimen)
        endif

c*************************************************     !End Stopping conds
        close (3)

        enddo
        write(7,*) 0
        write(7,*) 0
        write(4,*)

        close(7)
        close(4)
        end

c*******************************************************************************
c*******************************************************************************
c       subroutine to evaluate the function value at a point
        subroutine eval(center,correct)

        use param

        implicit double precision (a-g,o-z)
        implicit integer (i-n)
        double precision y(dimen),g(ni), obj            ! current design
        dimension center(nbox,(2*dimen+5))
        common /design/ numb,ni,nold
        common /bound/ xl(dimen),xu(dimen),e(dimen)
c*****************************************
        pen = 0.01              !pen = penalty function multiplier

        do i = nold,numb+2*dimen
        if (center(i,dimen+2) .ne. 0.) then
                if (abs(center(i,2*dimen+4)) .lt. 0.00001) then         !check if point has
been analyzed
                do j=1,dimen
                        y(j) = center(i,j+1)                    !put box center into y
                enddo

        call low(y,obj)

         center(i,2*dimen+4)= obj
```

```
c       compute penalty function

c       call con(y,g,ni)

c               do k=1,ni
c                       if (g(k) .gt. 0) then
c                       center(i,4) = center(i,1) +pen*g(k)
c                       end if
c               end do

         if (correct .ne. 0) then
         center(i,1) = obj * correct
         end if

          end if
         else
         exit
         end if
        end do

        return
        end
c*******************************************************************************
c       subroutine to evaluate the High Fidelity function value at a point
        subroutine ehigh(y,func)

        use param

        implicit double precision (a-g,o-z)
        implicit integer (i-n)
        double precision y(dimen),g(ni)                         ! current design
        common /design/ numb,ni,nold
        common /bound/ xl(dimen),xu(dimen),e(dimen)
c*****************************************
        call High(y,func)

c       compute penalty function

c       call con(y,g,ni)

c               do k=1,ni
c                       if (g(k) .gt. 0) then
c                       center(i,4) = center(i,1) +pen*g(k)
c                       end if
c               end do
```

```
      return
      end


C**********************************************************************
C**********************************************************************
c     Graham's scan routine
      subroutine graham(center,optimal)

      use param
      use dividing

      implicit double precision (a-g,o-z), integer (i-n)
      double precision order2(nbox), y(dimen)     !sizes of boxes in order, center of
box to be removed
      double precision local
      integer order(nbox),converg                 !order of points for scan,
      dimension center(nbox,(2*dimen+5))
      integer optimal(nopt), count, nact          !optimal boxes,# of remaining pts
      common /design/ numb,ni,nold
      common /convergence/ local(loc,d2),converg, count !local is the removed points
from stopcond = 2
      common /itteration/ itter
      common /sorter/ order2b(nbox), index(nbox),nsort,nact
c*********************************************

c     sort points by distance
      write(6,*) numb

      do i=1,nact+30                              !zero the local order arrays
      order(i)=0
      order2(i)=0
      enddo


      do i=1,nact
      yy=optimal(i)                       !optimal containes the boxes that were
divided last round
            z=0
            do k=dimen+2,2*dimen+1
            z = z+center(yy,k)**2         !get the sizes of the box after dividing
            end do
            z=sqrt(z)                     !distance to edge for box i

       do k=1,nsort
```

```
                if(index(k) .eq. yy)then        !find out where the box was in the list
                 do l=k,nsort-1                 !index keeps a list of the boxes in order of
size and value
                  index(l)=index(l+1)
                  order2b(l)=order2b(l+1)
                 enddo                          ! THIS SECTION REMOVES THE BOX
FROM THE LIST AND IT IS LATER PLACED IN THE NEW POSITION
                 index(nsort)=0
                 order2b(nsort)=0.
                 exit
                 endif
          enddo

          do j=1,nsort                          !Put the box back in where it belongs
                 if (index(j) .eq. 0.) then
                 index(j) = yy
                 order2b(j) = z
                 exit
                 endif
                          if (z .lt. order2b(j)) then
                          do k=nsort,j+1,-1
                                   index(k) = index(k-1)
                                   order2b(k) = order2b(k-1)
                          end do
                          index(j) = yy
                          order2b(j) = z
                          exit
                          end if

                          if (z .eq. order2b(j)) then
                          if (center(yy,1).lt.center(index(j),1)) then
                           do k=nsort,j+1,-1
                                   index(k) = index(k-1)
                                   order2b(k) = order2b(k-1)
                           end do
                           index(j) = yy
                           order2b(j) = z
                           exit
                          endif
                          end if

            enddo
          enddo

          do i=nsort+1,numb                                 !now do the rest of the new
points
```

```
                z=0
                do k=dimen+2,2*dimen+1
                z = z+center(i,k)**2
                end do
                z=sqrt(z)                                   !distance to edge for box i


          do j=1,numb
                if (index(j) .eq. 0.) then                  !if at end of list put box here
                index(j) = i
                order2b(j) = z
                exit
                endif
                        if (z .lt. order2b(j)) then         !if smaller than this entry insert here
                        do k=i,j+1,-1
                                index(k) = index(k-1)
                                order2b(k) = order2b(k-1)
                        end do
                        index(j) = i
                        order2b(j) = z
                        exit
                        end if

                        if (z .eq. order2b(j)) then         !if better and same size insert here
                        if (center(i,1).lt.center(index(j),1)) then
                         do k=i,j+1,-1
                                index(k) = index(k-1)
                                order2b(k) = order2b(k-1)
                         end do
                         index(j) = i
                         order2b(j) = z
                         exit
                        endif
                        end if
            enddo
          enddo
          nsort = numb

c         end sorting

          converg = 0

c*********************************                    !stop when smallest box
reaches certain size
          if (stopcond .eq. 1)then
          if (order2b(1).lt.minsize1) then
```

```
        converg         =1                                  !stop after this loop if boxes
small enough
        endif

        endif
c*********************************                          !remove boxes if they are in
neighborhood of small box
        if (stopcond .eq. 2) then
        m=0

        do while (m .eq. 0)
        if (order2b(1) .lt. minsize2) then
        count = count+1
         if (count .gt.(loc-1)) then                        !remove small boxes 15 times
then stop
          m=1
          converg = 1
         endif

                do k=1,dimen
                y(k)= center(index(1),1+k)                  !y is center of radius for the
removal of boxes
                local(count,k) = y(k)                       !save the good points
                enddo

          do k=numb,1,-1                                    !start at end so fewer shifts
are required
                dist = 0
                do i=1,dimen
                 dist = dist + (y(i)-center(index(k),1+i))**2
                enddo
                dist = sqrt(dist)

                if (dist .lt.radius) then           !if boxes are within 'radius' of the smallest
box then remove
                  do l=k,numb-1
                   index(l)=index(l+1)
                   order2b(l)=order2b(l+1)
                  enddo
                  index(numb)=0
                  order2b(numb)=0.
                  numb = numb -1                            !update numb to indicate 1
box removed
c       write(6,*)dist,index(k),k
                endif
          enddo
```

```
          local(count,dimen+2) = nsort - numb                    !how many points
were removed this round
          nsort = numb
          else
          m=1
          endif
          enddo
          endif
c*********************************                    !remove boxes if they are in
neighborhood of small box
          if (stopcond .eq. 3) then
          m=0

          do while (m .eq. 0)
          if (order2b(1) .lt. minsize2) then
          count = count+1
           if (count .gt.(loc-1)) then                    !remove small boxes 15 times
then stop
           m=1
           converg = 1
          endif

                  do k=1,dimen
                  y(k)= center(index(1),1+k)              !y is center of radius for the
removal of boxes
                  local(count,k) = y(k)                   !save the good points
                  enddo
                  local(count,dimen+1) = index(1)
          do k=numb,2,-1                                  !start at end so fewer shifts
are required
                  dist = 0
                  do i=1,dimen
                   dist = dist + (y(i)-center(index(k),1+i))**2
                  enddo
                  dist = sqrt(dist)

                  if (dist .lt.radius) then          !if boxes are within 'radius' of the
smallest box then remove
                  center(index(k),2*dimen+3) =center(index(k),2*dimen+3) +3
                  endif
          enddo
                   do l=1,numb-1
                    index(l)=index(l+1)
                    order2b(l)=order2b(l+1)
                   enddo
                   index(numb)=0
```

```
                    order2b(numb)=0.
                    numb = numb -1                          !update numb to indicate 1
box removed

           nsort = numb
          else
          m=1
          endif
          enddo
          endif
c*********************************                          !stop when max number of
iterations exceeded
          if (stopcond .eq. 4)then
          if (itter.gt.maxit-2) then
          converg          =1                                !stop after this loop if enough
iter.
          endif
          endif
c***********************************************
          m=0
          j=2
          k=2
          order(1) = index(1)
          order2(1) = order2b(1)
c           write(4,*)1,index(1),order2b(1),k-1

          do while (m .eq. 0)

          if (order2b(j) .gt. order2b(j-1)) then   !if the box size changes then index(j)is
potentially optimal
            order(k) = index(j)
            order2(k) = order2b(j)
c           write(4,*)j,index(j),order2b(j),k
            k=k+1
          endif

          if (j.ge.numb) then      !exit if you reach the end of index or optimal
            m=1
          endif
          j=j+1
          enddo
          nact = k-1
c***********************************************
          initial = 1                                       !start with the first entry when
dropping left side
```

```
      if (agressive .eq. 0) then                    !START ORIGINAL VERSION OF
DIRECT

c     find the convex hull

      i = 1
      j = 2
      k = 3

c**********************************
c     Conversion to sample more points in the beginning
c***************
      if (itter .gt. 0) then
c**********************************
c     remove right turns & keep left turns
      do while(k .le. nact)

      o = order2(i)*(center(order(j),1)-center(order(k),1))
     &  -center(order(i),1)*(order2(j)-order2(k))
     &  +(order2(j)*center(order(k),1)-order2(k)*center(order(j),1))

      if (o.gt.0.)then
      i=i+1
      j=i+1
      k=j+1
      else
       do l=j,nact
            order(l)=order(l+1)
            order2(l)=order2(l+1)
       enddo
      order(nact)=0
      order2(nact)=0
      nact =nact-1
       if (i.gt.1) then
            k=i+1
            j=i-0
            i=i-1
       else
       j=i+1
       k=i+2
       endif
      endif
      enddo
c**********************************
      end if
      itter =itter +1
```

```
c*********************************
      endif                          !END CONVEX HULL VERSION OF DIRECT

c     remove left side of convex hull
c     write(6,976)nact,((order(i),order2(i)),i=1,nact)

      m=0
      do while (m .ne. 1)

      if (((center(order(initial+1),1)-center(order(initial),1)).lt.
     &  -(0.001*center(order(initial),1))).and.(center(order(initial+1
     &  ),1).ne.0.)) then
      initial = initial+1                !if worse then skip to next entry
      else
      m=1
      endif
      enddo

      if (nact-initial .gt. nopt) nact = nopt+initial-1

c     record the potentially optimal points
      do i=initial,nact
            optimal(i-initial+1) = order(i)
      enddo
      do i=nact+2-initial,nopt
            optimal(i) = 0
      enddo
      nact = nact-initial +1
      return
      end

c*******************************************************************************
c*******************************************************************************
c     subroutine to divide the potentially optimal boxes into smaller boxes
      subroutine divider(center,optimal)

      use param
      use dividing

      implicit double precision (a-g,o-z)
      implicit integer (i-n)
      double precision lng , y(dimen)                !lng = longest length of a
side, y = position for high fidelity analysis
      dimension center(nbox,2*dimen+5)
      integer divide (2*dimen),split(2*dimen),dim(dimen),net
      integer optimal(400),calls                     !potentially optimal boxes
```

```
      common /design/ numb,ni,nold
      common /sizes/ size(dimen)
      common /cross/ calls
c*************************************************
c     start with first optimal point in "optimal"
      nold =numb
      do m=1,nopt
      if (optimal(m) .lt. .5) exit
      k= optimal(m)

c************ Only divide dimensions that are longer than the rest **********
      lng = 0.0
      z=0
      do i=1,dimen
      dim(i) =0
      z=z+ center(k,i+dimen+1)**2
      if (center(k,i+dimen+1).gt.lng) lng=center(k,i+dimen+1)     !longest dimention
      enddo
      z=sqrt(z)
      if (stopcond .eq. (1 .or. 3 .or. 4)) then          !Dont divide if smaller than min box
size
      if (z .lt. minsize1) goto 14
      elseif (stopcond .eq. 2) then
      if (z .lt. minsize2) goto 14
      endif

      if (center(k,2*dimen+2) .lt. center(k,2*dimen+3)) then                !skip if not
divided this time
      center(k,2*dimen+2) = center(k,2*dimen+2) +1
      goto 14
      else
      center(k,2*dimen+2) = 1
      endif

      j=1
      net =0
      do i=1,dimen
      if(abs(center(k,i+dimen+1)-lng).lt. minsize1)then
      dim(j)=i
      j=j+1
      net = net+1                              !# of dimensions to divide
      endif
      enddo
c*************************************************
c     Calculate correction factor
```

```
          if (center(k,2*dimen+5) .eq. 0) then
            do i=1,dimen
            y(i) = center(k,i+1)
            enddo

            call ehigh(y,value)
            calls = calls + 1
c           write(6,*)value
            center(k,2*dimen+5) = value
            correct = value / center(k,2*dimen+4)
            center(k,1) = center(k,2*dimen+4) * correct
          else
            correct =  center(k,2*dimen+5) / center(k,2*dimen+4)
          end if

c*************************************************
          if (numb+2*net .gt. nbox)exit
          do i=numb+1,numb+net*2                    !create the new points
                  do l=2,2*dimen+1
                  center(i,l)=center(k,l)
                  enddo
          enddo
          do i=1,net
           center(numb+2*i-1,dim(i)+1)=center(numb+2*i-1,dim(i)+1)
     & +center(numb+2*i-1,dim(i)+dimen+1)/3.*size(dim(i))

           center(numb+2*i,dim(i)+1)=center(numb+2*i,dim(i)+1)
     & -center(numb+2*i,dim(i)+dimen+1)/3.*size(dim(i))

           center(k,dim(i)+dimen+1) = center(k,dim(i)+dimen+1)*1./3.
          enddo

c         evaluate the new points
          call eval(center,correct)

c         divide the new boxes based on the values at the centers

c         sort points

          do i=1,2*net
          divide(i)=0
          enddo

          do i=numb+1,numb+2*net

            do j=1,2*net
```

```
            if (divide(j) .eq. 0) then
                  divide(j) = i
                  exit
            endif
            if (center(i,1) .lt. center(divide(j),1)) then
                  do k=2*dimen,j+1,-1
                        divide(k) = divide(k-1)
                  end do
                  divide(j)=i
                  exit
            end if
      enddo

      enddo

c     divide the boxes based on the sorted values
      do i=1,net
      split(i*2)=0     ! has the box been divided completely yet
      split(i*2-1)=0
      enddo

      do i=1,2*net
            k=(divide(i)-numb)

      if (split(k) .ne. 1) then
       do j=numb+1,numb+2*net
            if ((split(j-numb).ne.1))then
            center(j,dim((k+1)/2)+dimen+1) = center(j,dim((k+1)/2)+dimen+1)/3.
         endif
       enddo
       split(k)=1
       if (k/2 .eq.(k+1)/2) then
        split(k-1)=1
        else
        split(k+1)=1
       endif
      endif

      enddo
      numb = numb+2*net
14    continue
      enddo

      return
      end
```

```
c**********************************************************************
c**********************************************************************
          subroutine dotcntl(x,obj)

          use param

      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      implicit integer (i-n)
      parameter( ncon = 2 )
      parameter( NRWK = 6200, NRIWK = 300 )
      integer iseed, m,calls,IPRM(20)
      double precision xl1(dimen),xu1(dimen)
      double precision G(40),WK(6200),RPRM(20),IWK(300),X(dimen)
      common /design/ numb,ni,nold                    !number of boxes, max number of
boxes
          common /cross/ calls
      common /bound/ xl(dimen),xu(dimen),e(dimen)
c**********************************************************************
c  Check to make sure input number of design variables and constraints
c  matches NDV and NCON.
c
          n=dimen

c zero WK and IWK
      do 200 i = 1,NRWK
        WK(I) = 0.0
  200 continue
      do 210 i = 1,NRIWK
        IWK(I) = 0
  210 continue

C ZERO RPRM AND IPRM
      DO 10 I=1,20
       RPRM(I)=0.0
10      IPRM(I)=0

c set convergence tolerance criteria to other than default
      IPRM(4) = 4
      IPRM(8) = 50
      IPRM(9) =  5
      RPRM(1) = -1.0d-1
      RPRM(2) =  1.0d-3
      RPRM(3) =  1.0d-3
      RPRM(4) =  1.0d-3
      RPRM(5) =  3.0
      RPRM(6) =  3.0
```

```
      RPRM(7) = 3.
      RPRM(8) = 4.0
      RPRM(9) = 0.02

C DEFINE IPRINT, MINMAX, INFO
C PRINT CONTROL
      IPRINT=1

C MINIMIZE
      MINMAX=-1

C DEFINE METHOD,NDV,NCON
C 0,1 = MODIFIED METHOD OF FEASIBLE DIRCTIONS
C 2   = sequential linear program
C 3   = sequential quadratic program
      METHOD=3


c INITIALIZE INFO TO ZERO
      INFO=0

C OPTIMIZE


100    CALL DOT (INFO,METHOD,IPRINT,n,NCON,X,XL,XU,
     * OBJ,MINMAX,G,RPRM,IPRM,WK,NRWK,IWK,NRIWK)


C FINISHED?
      IF(INFO.EQ.0) then
        call High(X,OBJ)

        call con( X, G,ncon)
        calls = calls+1
186     format(xf9.5,2x,20f15.6)

      endif

C EVALUATE OBJECTIVE AND CONSTRAINTS.
          if (info.ne.0) then

        call High(X,OBJ)
        call con( X, G,ncon)
        calls = calls+1
c         write(4,300)obj,(x(i),i=1,dimen)
300     format(20f9.4)
```

```
C GO CONTINUE WITH OPTIMIZATION
     GO TO 100
          endif


     end
c****************************************
          subroutine High(x,obj)

          use param

       IMPLICIT double precision (A-H,O-Z),INTEGER(I-N)
       DIMENSION x(dimen), y(10)
          common /design/ numb,ni,nold
          common /bound/ xl(dimen),xu(dimen),e(dimen)


          do i= 1,dimen
            y(i)= x(i)
          enddo
          do i= dimen+1,10
            y(i) = 0
          enddo

          m=1
          if (m .eq. 1) then
            obj =(y(1)**2+y(2)**2+y(3)**2+y(4)**2+y(5)**2+y(6)**2+y(7)**
     & 2+y(8)**2+y(9)**2+y(10)**2)/1000.
     & +1.-cos(y(1))*cos(y(2)/sqrt(2.))*cos(y(3)/sqrt(3.))*
     & cos(y(4)/2.)*cos(y(5)/sqrt(5.))*cos(y(6)/sqrt(6.))*
     & cos(y(7)/sqrt(7.))*cos(y(8)/sqrt(8.))*cos(y(9)/3.)*
     & cos(y(10)/sqrt(10.))
          endif

          m=0
          if (m .eq. 1) then
        obj=0
        do l=1,dimen
          obj= obj+10+2.0*(x(l)+e(l))**2-1.5*(x(l)+e(l))**4
     &     +.2*(x(l)+e(l))**6
        enddo

          endif

          return
          end
```

```
c*****************************************
      subroutine Low(y,obj)

      use param
     IMPLICIT double precision (A-H,O-Z),INTEGER(I-N)
     DIMENSION y(dimen)
      common /design/ numb,ni,nold
      common /bound/ xl(dimen),xu(dimen),e(dimen)

      m=1
      if (m .eq. 1) then                !griewank
      obj = 5
      do i=1,dimen
      obj = obj + (y(i)+e(i))**2/5000.
      enddo
      endif

      m=0
      if (m .eq. 1) then
    obj=0
    do l=1,dimen
      obj= obj+10+2.5*(y(l)+e(l))**2-1.2*(y(l)+e(l))**4
    enddo
      endif

      m=0
      if (m .eq. 1) then
    obj=0
    do l=1,dimen
      obj= obj +10-1 -2*(2*((y(l)+e(l))/2.)**2-1)
&   -.6*(8*((y(l)+e(l))/2.)**4-8*((y(l)+e(l))/2.)**2+1)
    enddo
      endif

      m=0
      if (m .eq. 1) then                !chebychev low + part of high
    obj=0
    do l=1,dimen
      obj= obj + 10 - 1 -2*(2*((y(l)+e(l))/2.)**2-1)
&   -.6*(8*((y(l)+e(l))/2.)**4-8*((y(l)+e(l))/2.)**2+1)
&  + .5*(.4*(32*((y(l)+e(l))/2.)**6-48*((y(l)+e(l))/2.)**4
&  +18*((y(l)+e(l))/2.)**2-1))

    enddo
      endif
```

```
          m=0
          if (m .eq. 1) then                !linear addition to high fidelity function
        obj=0
        do l=1,dimen
          obj= obj+10+2.0*(y(l)+e(l))**2-1.5*(y(l)+e(l))**4
     &     +.2*(y(l)+e(l))**6  ! +    .6* y(l)+.02*y(l)**2
        enddo

          do l=1,dimen                      !piecewise linear
          if (y(l) .lt. -.8) then
           obj = obj+ y(l)+.8
          else if (y(l) .gt. .4) then
           obj = obj - 1.2 * y(l)+1.08
          else
           obj = obj +.5* y(l)+.4
          endif
          enddo
          endif

          return
          end
c*****************************************
          subroutine con(x,g,ncon)

          use param

          IMPLICIT REAL*8 (A-H,O-Z),INTEGER(I-N)
          common /design/ numb,ni,nold
          common /bound/ xl(dimen),xu(dimen),e(dimen)
        DIMENSION X(dimen),G(ncon)

          do j=1,ncon
          g(j)=(-2000.+x(j))/100.
          enddo

           RETURN
        END
```

LRS Multifidelity DIRECT Code

```
c       Direct optimizer Multifidelity LRS version-sequential setup
c       Steven Cox
c       Feb '00
c**********************************************************************
        module param
        integer, parameter :: nbox = 100001, dimen = 10, d2 = dimen+2      !used in every
module
        integer, parameter :: print = 0
        end module param

        module dividing
        parameter (agressive = 0)                                          !used in Graham
routime to select method
        real(8),parameter::minsize1=0.0001,minsize2=0.001,radius=0.05    !stopping box
sizes and radius of removal
        parameter (nopt = 800, stopcond = 1)                               !maximum number of
potentially optimal boxes
        integer, parameter :: loc = 5 ,maxit = 60, nrs = 80
        end module dividing

        use param
        use dividing
        parameter nbest = 8000                                  !parameters for the main
routine only
c       declare variables
        implicit double precision (a-g,o-z) , integer (i-n)
        real(8), parameter :: mindist = 0.05

        double precision xopt(dimen),x(dimen),points(15)    !point to pass to DOT
        double precision center(nbox,(2*dimen+6)),func,dist          !value, position and
dimentions of each point
        integer optimal(nopt),best(nbest),dropped, totmult                 !potentialy
optimal points
        integer totcalls,calls,count,converg,nact, RScount                 !counters for
FE used by DOT
        double precision xl(dimen),xu(dimen),xbest(dimen),fbest            !current best
box and function value
        double precision local
        common /design/ numb,ni,nold                                       !number of boxes,
max number of boxes
        common /sizes/ size(dimen)
        common /bound/ xl,xu,e(dimen)                                      !box limits
        common /cross/ calls
        common /itteration/ itter
        common /convergence/  local(loc,d2),converg,count
```

```
      common /sorter/ order2b(nbox), index(nbox),nsort,nact
      common /RS/ RS(nrs,dimen) , RScount                    !Array of RS slopes
and current RS to generate

c*********************************************************************
c     Main Code
c*********************************************************************
c*********************************************************************
      open (4,file='grie_LRS.dat',access='append')
      open(7,file = "lrs_steps_grie.dat")   !,access='append')
      xu(1)=rand(267)*10
      do kr = 1,30

c     setup cycles
      optimal(1)=1
      nold=1
      numb = 1
      ni=0
      final = nbox-2*dimen                  !buffer size so you can't overflow center
      converg = 0
      fbest = 9999090
      totmult = 0
      totcalls = 0
      itter =1
      count =0
      nsort = 0
      nact = 0
      calls = 1
      RScount = 1

c*****************  define box size  *****************
c************************* Griewank Function ***************
      do i=1,dimen
      xu(i) =  100. + rand(0)*800.
      xl(i) =  xu(i)-1000.
      e(i) =  45+rand(0)*40!
      enddo
c************************* Box Function ********************
c     do i=1,dimen
c     xu(i)=2
c     xl(i)=-2
c     e(i) =  .2+rand(0)*.2!.1!
c     enddo
c*************************
      do i=1,nbox
       do j=1,dimen+3
```

```
        center(i,j)= 0
       enddo
      order2b(i)=0
      index(i) =0
      center(i,2*dimen+2)=1
      center(i,2*dimen+3)=1
      center(i,2*dimen+4)=0
      center(i,2*dimen+5)=0
      center(i,2*dimen+6)=0
      enddo

      do j=1,nrs
      do i=1,dimen
      RS(j,i) =0
      enddo
      enddo

c     find dimentions

      do i=1,dimen
            center(1,i+1)=(xu(i)+xl(i))/2          ! center point of box
            center(1,i+dimen+1)=1                  ! each box is 1 unit long in each dim
            size(i)=(xu(i)-xl(i))                  ! length of box side
      enddo

c     analyze the function value at the center, high and low fidelity

      call ehigh(center)
      call eval(center)

      center(1,1) =  center(1,2*dimen+5)           !corrected value is the high fid value

c     locate potentially optimal points
25    continue

      call graham(center,optimal)

c     divide optimal boxes

      call divider(center,optimal)

      if (print .eq. 1) then
      write(7,*)numb
      do i=1,numb
      write(7,32)(center(i,j),j=1,2*dimen+6)
      enddo
```

```
        write(7,*)
        endif

32      format(2x,9(f9.4,x),2(f4.0,x),3(f9.4,x))

c       loop untill convergence criteria is met
c       write(6,*)"         ",calls, converg
        if (((numb .lt. final).and.(converg.ne.1))) then
        go to 25
        endif

        totmult = calls

c************
        if (print .eq. 1) then                  !gives output file of all points
        open (5,file = 'points.dat')
c       do j=1,dimen
c         write(5,3)j,xu(j),xl(j),size(j)
c       enddo

        do i= 1,numb
        write(5,3)i,(center(i,j),j=1,dimen+1),(center(i,2*dimen+j),j=4,5)
        enddo
3       format(i6,x,21f10.4)
        endif
c************
        open (3,file = 'best.out',access = 'append')
c       sort points
        do i=1,nbest
        best(i)=0
        enddo
c*********************                   !add removed points if stopcond = 2 or 3
        if (stopcond .eq. (2.or.3)) then
        do i=1,loc
          do j=1,loc
            if (best(j) .eq. 0) then
                best(j) = local(i,dimen+1)
                exit
            endif
           if (center(local(i,dimen+1),1) .lt. center(best(j),1)) then
                do k=loc,j+1,-1
                        best(k) = best(k-1)
                end do
                best(j)=i
                exit
            end if
```

```
          enddo
        enddo
        endif
c*********************                    !finish adding removed points

        do i=1,numb
          do j=1,nbest
                if (best(j) .eq. 0) then
                        best(j) = i
                        exit
                endif
                if (center(i,1) .lt. center(best(j),1)) then
                        do k=nbest,j+1,-1
                                best(k) = best(k-1)
                        end do
                        best(j)=i
                        exit
                end if
          enddo
        enddo

        do k=1,nbest
        write(3,23) k,best(k), (center(best(k),j),j=1,dimen+1)
        enddo
23      format(3x,i4,x,i6,f9.3,20f6.2)
        write(3,*)

        do j=1,loc
        points(j)=0
        enddo

         points(1)=1
        k=0
        do j=1,nbest                            !loop to find best points separated by
%
        if (k.eq.loc) exit
        calls = 0
        if (k.ne.0) then
        do l=1,k
         dist = 0.
         do i=1,dimen
          dist = dist+((center(best(points(l)),i+1)-center(best(j),i+1)
    &    )/size(i))**2
         enddo
         dist = sqrt(dist)
         if (dist.lt.mindist) goto 12
```

```
          enddo
           points(k+1)=j
          endif
           k=k+1

c         write(3,*)best(j)
c         write(3,*)(center(best(j),i),i=1,dimen+1)

          do i=1,dimen
          xopt(i)=center(best(j),i+1)
          enddo
          func = center(best(j),1)

c         write(4,300)k,func,(xopt(i),i=1,dimen)
300       format(20f9.4)
c*******
          call dotcntl (xopt,func)                !Here DOT is called to do local opt
c*******

           dist = 0.
           do i=1,dimen
            dist = dist+abs(center(best(j),i+1)-xopt(i))
           enddo

c         write(3,348),'calls =',calls,'Distance moved =',dist
348       format(x,a7,x,i4,8x,a16,f10.6)

          totcalls = totcalls+calls

c         write(4,*)func,fbest

c         update best optimum location
          if (func .lt. fbest) then
           fbest = func
           do i=1,dimen
            xbest(i) = xopt(i)
           enddo
          endif

12        continue
          enddo

          write(3,213),"high fidelity calls =",totcalls,
     & " ,  Low fidelity calls =",numb ,"Iterations =",itter
213       format(2x,a21,x,i5,a24,x,i6,2x,a12,x,i4)
```

```
c*****************************          !First and Fourth Stopping
Method
      if ((stopcond .eq. 1).or.(stopcond .eq.4)) then

      write(4,187)kr,fbest,totmult,totcalls,numb,(xbest(i),i=1,dimen)
c     write(4,16)(e(i),i=1,dimen)
187   format(x,i5,x,f12.5,x,3i9,2x,20f11.6)
16    format (11x,20f11.6)
      endif

c*****************************          !Second Stopping Method

      if(stopcond .eq. 2) then
      dropped = 0

      do i=1,loc
      dropped = dropped+ local(i,dimen+2)
      enddo

      write(4,186)kr,fbest,totcalls,numb,dropped,(xbest(i),i=1,dimen)
186   format(x,i5,x,f11.5,x,3i7,2x,20f11.6)

      endif

c*****************************          !Third Stopping Method
      if(stopcond .eq. 3) then

      write(4,187)kr,fbest,totcalls,numb+loc,(xbest(i),i=1,dimen)
c     write(4,16)(e(i),i=1,dimen)
      endif

c*************************************************    !End Stopping conds
      close (3)

      enddo
      write(7,*) 0
      write(7,*) 0
      write(4,*)

      close(7)
      close(4)
      end

c*******************************************************************************
c*******************************************************************************
c     subroutine to evaluate the function value at a point
```

```
      subroutine eval(center)

      use param
      implicit double precision (a-g,o-z)
      implicit integer (i-n)
      double precision y(dimen),g(ni), obj            ! current design
      dimension center(nbox,(2*dimen+6))
      common /design/ numb,ni,nold
      common /bound/ xl(dimen),xu(dimen),e(dimen)
c*****************************************

      pen = 0.1               !pen = penalty function multiplier

      do i = nold,numb+2*dimen
      if (center(i,dimen+2) .ne. 0.) then
            if (abs(center(i,2*dimen+4)) .lt. 0.00001) then        !check if point has
been analyzed
            do j=1,dimen
                  y(j) = center(i,j+1)                    !put box center into y
            enddo

      call low(y,obj)

       center(i,2*dimen+4)= obj

c     compute penalty function

c     call con(y,g,ni)

c           do k=1,ni
c                 if (g(k) .gt. 0) then
c                 center(i,4) = center(i,1) +pen*g(k)
c                 end if
c           end do

        end if
       else
       exit
       end if
      end do

      return
      end

c*********************************************************************
c     subroutine to evaluate the High Fidelity function value at a point
```

```
      subroutine ehigh(y,func)

      use param
      implicit double precision (a-g,o-z)
      implicit integer (i-n)
      double precision y(dimen),g(ni)                    ! current design
      common /design/ numb,ni,nold
      common /bound/ xl(dimen),xu(dimen),e(dimen)
c*****************************************

      call High(y,func)

c     compute penalty function

c     call con(y,g,ni)

c           do k=1,ni
c                 if (g(k) .gt. 0) then
c                 center(i,4) = center(i,1) +pen*g(k)
c                 end if
c           end do

      return
      end

C*********************************************************************
C*********************************************************************
c     Graham's scan routine
      subroutine graham(center,optimal)

      use param
      use dividing

      implicit double precision (a-g,o-z), integer (i-n)
      double precision order2(nbox), y(dimen)      !sizes of boxes in order, center of
box to be removed
      double precision local
      integer order(nbox),converg                  !order of points for scan,
      dimension center(nbox,(2*dimen+6))
      integer optimal(nopt), count, nact           !optimal boxes,# of remaining pts
      common /design/ numb,ni,nold
      common /convergence/ local(loc,d2),converg, count !local is the removed points
from stopcond = 2
      common /itteration/ itter
      common /sorter/ order2b(nbox), index(nbox),nsort,nact
c*********************************************
```

```
c       sort points by distance
        write(6,*) numb

        do i=1,numb                     !zero the local order arrays
        order(i)=0
        order2(i)=0
        enddo


        do i=1,nact
        yy=optimal(i)                   !optimal containes the boxes that were
divided last round
                z=0
                do k=dimen+2,2*dimen+1
                z = z+center(yy,k)**2   !get the sizes of the box after dividing
                end do
                z=sqrt(z)               !distance to edge for box i

          do k=1,nsort
                if(index(k) .eq. yy)then        !find out where the box was in the list
                 do l=k,nsort-1                 !index keeps a list of the boxes in order of
size and value
                  index(l)=index(l+1)
                  order2b(l)=order2b(l+1)
                 enddo                          ! THIS SECTION REMOVES THE BOX
FROM THE LIST AND IT IS LATER PLACED IN THE NEW POSITION
                 index(nsort)=0
                 order2b(nsort)=0.
                 exit
                 endif
           enddo

           do j=1,nsort                 !Put the box back in where it belongs
                if (index(j) .eq. 0.) then
                index(j) = yy
                order2b(j) = z
                exit
                endif
                        if (z .lt. order2b(j)) then
                        do k=nsort,j+1,-1
                                index(k) = index(k-1)
                                order2b(k) = order2b(k-1)
                        end do
                        index(j) = yy
                        order2b(j) = z
```

```
                    exit
                    end if

                    if (z .eq. order2b(j)) then
                    if (center(yy,1).lt.center(index(j),1)) then
                     do k=nsort,j+1,-1
                            index(k) = index(k-1)
                            order2b(k) = order2b(k-1)
                     end do
                     index(j) = yy
                     order2b(j) = z
                     exit
                    endif
                    end if

          enddo
        enddo


        do i=nsort+1,numb                                   !now do the rest of the new
points
                z=0
                do k=dimen+2,2*dimen+1
                z = z+center(i,k)**2
                end do
                z=sqrt(z)                                   !distance to edge for box i


          do j=1,numb
                if (index(j) .eq. 0.) then                  !if at end of list put box here
                index(j) = i
                order2b(j) = z
                exit
                endif
                        if (z .lt. order2b(j)) then         !if smaller than this entry insert here
                        do k=i,j+1,-1
                                index(k) = index(k-1)
                                order2b(k) = order2b(k-1)
                        end do
                        index(j) = i
                        order2b(j) = z
                        exit
                        end if

                        if (z .eq. order2b(j)) then         !if better and same size insert here
                        if (center(i,1).lt.center(index(j),1)) then
```

```
                        do k=i,j+1,-1
                                index(k) = index(k-1)
                                order2b(k) = order2b(k-1)
                        end do
                        index(j) = i
                        order2b(j) = z
                        exit
                       endif
                       end if
          enddo
        enddo
        nsort = numb

c       end sorting

        converg = 0
        write(6,35)index(1),order2b(1),center(index(1),1),
    &   center(index(1),2*dimen+4),center(index(1),2*dimen+5)
35      format(2x,i8,4f10.4)

c*********************************           !stop when smallest box
reaches certain size
        if (stopcond .eq. 1)then
        if (order2b(1).lt.minsize1) then
        converg         =1                      !stop after this loop if boxes
small enough
        endif

        endif

c*********************************           !remove boxes if they are in
neighborhood of small box
        if (stopcond .eq. 2) then
        m=0

        do while (m .eq. 0)
        if (order2b(1) .lt. minsize2) then
        count = count+1
          if (count .gt.(loc-1)) then           !remove small boxes 15 times
then stop
            m=1
            converg = 1
           endif

                do k=1,dimen
```

```
                y(k)= center(index(1),1+k)                    !y is center of radius for the
removal of boxes
                local(count,k) = y(k)                         !save the good points
                enddo

          do k=numb,1,-1                                      !start at end so fewer shifts
are required
                dist = 0
                do i=1,dimen
                 dist = dist + (y(i)-center(index(k),1+i))**2
                enddo
                dist = sqrt(dist)

                if (dist .lt.radius) then           !if boxes are within 'radius' of the smallest
box then remove
                  do l=k,numb-1
                   index(l)=index(l+1)
                   order2b(l)=order2b(l+1)
                  enddo
                  index(numb)=0
                  order2b(numb)=0.
                  numb = numb -1                              !update numb to indicate 1
box removed
c         write(6,*)dist,index(k),k
                endif
           enddo

           local(count,dimen+2) = nsort - numb                      !how many points
were removed this round

          nsort = numb

         else
         m=1

         endif
         enddo

         endif

c********************************                              !remove boxes if they are in
neighborhood of small box
         if (stopcond .eq. 3) then
         m=0

         do while (m .eq. 0)
```

```
          if (order2b(1) .lt. minsize2) then
          count = count+1
            if (count .gt.(loc-1)) then                    !remove small boxes 15 times
then stop
             m=1
             converg = 1
            endif

                  do k=1,dimen
                  y(k)= center(index(1),1+k)               !y is center of radius for the
removal of boxes
                  local(count,k) = y(k)                    !save the good points
                  enddo

                  local(count,dimen+1) = index(1)
            do k=numb,2,-1                                 !start at end so fewer shifts
are required
                  dist = 0
                  do i=1,dimen
                   dist = dist + (y(i)-center(index(k),1+i))**2
                  enddo
                  dist = sqrt(dist)

                  if (dist .lt.radius) then                !if boxes are within 'radius' of the
smallest box then remove

                  center(index(k),2*dimen+3) =center(index(k),2*dimen+3) +3

                  endif
            enddo

                    do l=1,numb-1
                     index(l)=index(l+1)
                     order2b(l)=order2b(l+1)
                    enddo
                    index(numb)=0
                    order2b(numb)=0.
                    numb = numb -1              !update numb to indicate 1 box removed

           nsort = numb

          else
          m=1

          endif
          enddo
```

```
        endif
c********************************                    !stop when max number of
iterations exceeded
        if (stopcond .eq. 4)then
        if (itter.gt.maxit-2) then
        converg         =1                          !stop after this loop if enough
iter.
        endif

        endif

c*********************************************
        m=0
        j=2
        k=2
        order(1) = index(1)
        order2(1) = order2b(1)
c         write(4,*)1,index(1),order2b(1),k-1

        do while (m .eq. 0)

        if (order2b(j) .gt. order2b(j-1)) then   !if the box size changes then index(j)is
potentially optimal
          order(k) = index(j)
          order2(k) = order2b(j)
c         write(4,*)j,index(j),order2b(j),k
          k=k+1
        endif

        if (j.ge.numb) then      !exit if you reach the end of index or optimal
          m=1
        endif

        j=j+1

        enddo

        nact = k-1

c*********************************************

        initial = 1                                  !start with the first entry when
dropping left side
        if (agressive .eq. 0) then        !START ORIGINAL VERSION OF DIRECT
```

```
c       find the convex hull

        i = 1
        j = 2
        k = 3

c**********************************
c       Conversion to sample more points in the beginning
c***************
        if (itter .gt. 0) then
c**********************************

c       remove right turns & keep left turns

        do while(k .le. nact)

        o = order2(i)*(center(order(j),1)-center(order(k),1))
     &  -center(order(i),1)*(order2(j)-order2(k))
     &  +(order2(j)*center(order(k),1)-order2(k)*center(order(j),1))

        if (o.gt.0.)then
        i=i+1
        j=i+1
        k=j+1
        else
         do l=j,nact
                order(l)=order(l+1)
                order2(l)=order2(l+1)
         enddo
        order(nact)=0
        order2(nact)=0
        nact =nact-1
         if (i.gt.1) then
                k=i+1
                j=i-0
                i=i-1
         else
         j=i+1
         k=i+2
         endif
        endif

        enddo
c**********************************
        end if
        itter =itter +1
```

```
c*********************************

        endif                           !END CONVEX HULL VERSION OF DIRECT

c       remove left side of convex hull
c       write(6,976)nact,((order(i),order2(i)),i=1,nact)

        m=0
        do while (m .ne. 1)

        if (((center(order(initial+1),1)-center(order(initial),1)).lt.
     & -(0.001*center(order(initial),1))).and.(center(order(initial+1)
     & ,1).ne.0.)) then
        initial = initial+1                     !if worse then skip to next entry
        else
        m=1
        endif
        enddo

        write(6,35)order(initial),order2(initial),center(order(initial),1),
     & center(order(initial),2*dimen+4),center(order(initial),2*dimen+5)

        if (nact-initial .gt. nopt) nact = nopt+initial-1

c       record the potentially optimal points
        do i=initial,nact
                optimal(i-initial+1) = order(i)
        enddo
        do i=nact+2-initial,nopt
                optimal(i) = 0
        enddo

        nact = nact-initial +1

        return
        end

c*******************************************************************************
c*******************************************************************************
c       subroutine to divide the potentially optimal boxes into smaller boxes
        subroutine divider(center,optimal)

        use param
        use dividing

        implicit double precision (a-g,o-z)
```

```
      implicit integer (i-n)
      double precision lng , y(dimen)                          !lng = longest length
of a side, y = position for high fidelity analysis
      dimension center(nbox,2*dimen+6) , slope(dimen)   !slope = slopes of the RS in
use
      integer divide (2*dimen),split(2*dimen),dim(dimen),net
      integer optimal(400),calls, RSuse, RSok,RScount          !potentially optimal
boxes , RSuse = RS to use this time
      common /design/ numb,ni,nold
      common /sizes/ size(dimen)
      common /cross/ calls
      common /RS/ RS(nrs,dimen), RScount

c*************************************************
c     start with first optimal point in "optimal"

      nold =numb
      do m=1,nopt
      if (optimal(m) .lt. .5) exit
      k= optimal(m)

c************ Only divide dimensions that are longer than the rest **********

      lng = 0.0
      z=0
      do i=1,dimen
      dim(i) =0
      z=z+ center(k,i+dimen+1)**2
      if (center(k,i+dimen+1).gt.lng) lng=center(k,i+dimen+1)     !longest dimention
      enddo
      z=sqrt(z)

      if (stopcond .eq. (1 .or. 3 .or. 4)) then          !Dont divide if smaller than min box
size
      if (z .lt. minsize1) goto 14
      elseif (stopcond .eq. 2) then
      if (z .lt. minsize2) goto 14
      endif

      if (center(k,2*dimen+2) .lt. center(k,2*dimen+3)) then                !skip if not
divided this time
      center(k,2*dimen+2) = center(k,2*dimen+2) +1
      goto 14
      else
      center(k,2*dimen+2) = 1
      endif
```

```
          j=1
          net =0
          do i=1,dimen
          if(abs(center(k,i+dimen+1)-lng) .lt. .0000000001)then
          dim(j)=i
          j=j+1
          net = net+1                              !# of dimensions to divide
          endif
          enddo
c*************************************************
c         Calculate correction factor @ center of box

          if (center(k,2*dimen+5) .eq. 0) then
            do i=1,dimen
            y(i) = center(k,i+1)
            enddo

            call ehigh(y,value)
            calls = calls + 1

            center(k,2*dimen+5) = value

            correct = value / center(k,2*dimen+4)

            old = center(k,1)/center(k,2*dimen+4)              !previously assumed CR

            center(k,1) = center(k,2*dimen+4) * correct

            error = abs(correct - old) / (abs(correct + old)/2.)    !%error in the predicted CR
c         write(6,*)error
c************** CF error limit is set here ********************************
            if (((error .lt. .2).and.(center(k,2*dimen+6) .ge. 1))
     &    .or. (RScount .gt. nrs)) then
c*****************************************************************************

             RSok = 1                                    ! Use old RS
            else
             RSok = 0                                    ! Generate new RS
            endif

          else
            correct =  center(k,2*dimen+5) / center(k,2*dimen+4)
            RSok = 1
          end if
c*************************************************
```

```
      if (numb+2*net .gt. nbox)exit
      do i=numb+1,numb+net*2                    !create the new points
            do l=2,2*dimen+1
            center(i,l)=center(k,l)
            enddo
            center(i,2*dimen+6) = center(k,2*dimen+6)
      enddo

      do i=1,net
       center(numb+2*i-1,dim(i)+1)=center(numb+2*i-1,dim(i)+1)
     & +center(numb+2*i-1,dim(i)+dimen+1)/3.*size(dim(i))

       center(numb+2*i,dim(i)+1)=center(numb+2*i,dim(i)+1)
     & -center(numb+2*i,dim(i)+dimen+1)/3.*size(dim(i))

       center(k,dim(i)+dimen+1) = center(k,dim(i)+dimen+1)*1./3.
      enddo

c     evaluate the new points ****************************************
      call eval(center)

c*********************************************************************
c              Apply linear CF
c*********************************************************************

      RSuse = center(k,2*dimen+6)                   !get RS of parent box
      if (RSok) then

        do i=1,dimen
          slope(i) = RS(RSuse,i)
        enddo

c     apply correction
      do i=1,2*net          !loop through the new points
        CF = correct        !calculate CF
        do j=1,dimen
          CF = CF + slope(j)*(center(numb+i,j+1)-center(k,j+1))
        enddo

        center(numb+i,1) = center(numb+i,2*dimen+4) * CF
c       write(6,*)correct,CF, numb+i, k
      enddo

      else                          ! generate new RS  ***********
```

```
          do j=1,2*net                    !loop through the new points
           do i=1,dimen
           y(i) = center(numb+j,i+1)
           enddo

           call ehigh(y,value)            !calculate high fidelity values
           calls = calls+1
           center(numb+j,1) = value
           center(numb+j,2*dimen+5) = value
          enddo
c          write(6,*)numb, "RScount =",RScount

c         Calculate slopes of RS
          if (RScount .gt. 1) then
          do i=1,dimen
           RS(RScount,i)= RS(RSuse,i)          !put slopes of RS from parent into new RS
for dimensions that weren't divided
          enddo
          endif

          do i=1,net
          RS(RScount,dim(i)) = (center(numb+2*i-1,1)/center(numb+2*i-1,
     &  2*dimen+4)-center(numb+2*i,1)/center(numb+2*i,2*dimen+4))/
     &  (center(numb+2*i-1,dim(i)+1)-center(numb+2*i,dim(i)+1))            !calculate
slopes in dim dimensions
           center(numb+2*i-1,2*dimen+6) = RScount                        !update RS to
use
           center(numb+2*i,2*dimen+6) = RScount
          enddo
          center(k,2*dimen+6) = RScount                                  !change RS to
use of parent box

c         write(6,*)(RS(RScount,l),l=1,dimen)

          RScount = RScount+1

          endif

c*************************************************************************
c         divide the new boxes based on the values at the centers
c         sort points
          do i=1,2*net
          divide(i)=0
          enddo

          do i=numb+1,numb+2*net
```

```
      do j=1,2*net
            if (divide(j) .eq. 0) then
                  divide(j) = i
                  exit
            endif
            if (center(i,1) .lt. center(divide(j),1)) then
                  do k=2*dimen,j+1,-1
                        divide(k) = divide(k-1)
                  end do
                  divide(j)=i
                  exit
            end if
      enddo

      enddo

c     divide the boxes based on the sorted values
      do i=1,net
      split(i*2)=0     ! has the box been divided completely yet
      split(i*2-1)=0
      enddo

      do i=1,2*net
            k=(divide(i)-numb)

      if (split(k) .ne. 1) then
       do j=numb+1,numb+2*net
            if ((split(j-numb).ne.1))then
            center(j,dim((k+1)/2)+dimen+1) = center(j,dim((k+1)/2)+dimen+1)/3.
         endif
       enddo


       split(k)=1
       if (k/2 .eq.(k+1)/2) then
        split(k-1)=1
        else
        split(k+1)=1
       endif
      endif
      enddo

      numb = numb+2*net
c     write(4,*)optimal(m),net,numb
```

```
14      continue
        enddo

        return
        end

c**********************************************************************
c**********************************************************************

        subroutine dotcntl(x,obj)

        use param

    IMPLICIT DOUBLE PRECISION (A-H,O-Z)
    implicit integer (i-n)
    parameter( ncon = 2 )
    parameter( NRWK = 6200, NRIWK = 300 )
    integer iseed, m,calls,IPRM(20)
    double precision xl1(dimen),xu1(dimen)
    double precision G(40),WK(6200),RPRM(20),IWK(300),X(dimen)
    common /design/ numb,ni,nold                    !number of boxes, max number of
boxes
        common /cross/ calls
    common /bound/ xl(dimen),xu(dimen),e(dimen)
c**********************************************************************

c  Check to make sure input number of design variables and constraints
c  matches NDV and NCON.
c
        n=dimen

c zero WK and IWK
    do 200 i = 1,NRWK
      WK(I) = 0.0
 200 continue
    do 210 i = 1,NRIWK
      IWK(I) = 0
 210 continue

C ZERO RPRM AND IPRM
    DO 10 I=1,20
     RPRM(I)=0.0
10     IPRM(I)=0

c set convergence tolerance criteria to other than default
    IPRM(4) = 4
```

```
      IPRM(8) = 50
      IPRM(9) =  5
      RPRM(1) = -1.0d-1
      RPRM(2) =  1.0d-3
      RPRM(3) =  1.0d-3
      RPRM(4) =  1.0d-3
      RPRM(5) =  3.0
      RPRM(6) =  3.0
      RPRM(7) = 3.
      RPRM(8) = 4.0
      RPRM(9) = 0.02

C DEFINE IPRINT, MINMAX, INFO
C PRINT CONTROL
      IPRINT=1

C MINIMIZE
      MINMAX=-1

C DEFINE METHOD,NDV,NCON
C 0,1 = MODIFIED METHOD OF FEASIBLE DIRCTIONS
C 2   = sequential linear program
C 3   = sequential quadratic program
      METHOD=3


c INITIALIZE INFO TO ZERO
      INFO=0

C OPTIMIZE


100    CALL DOT (INFO,METHOD,IPRINT,n,NCON,X,XL,XU,
     * OBJ,MINMAX,G,RPRM,IPRM,WK,NRWK,IWK,NRIWK)


C FINISHED?
      IF(INFO.EQ.0) then
        call High(X,OBJ)

        call con( X, G,ncon)
        calls = calls+1
186      format(xf9.5,2x,20f15.6)

      endif
```

```
C EVALUATE OBJECTIVE AND CONSTRAINTS.
          if (info.ne.0) then

         call High(X,OBJ)
         call con( X, G,ncon)
         calls = calls+1
c          write(4,300)obj,(x(i),i=1,dimen)
300      format(20f9.4)
C GO CONTINUE WITH OPTIMIZATION
      GO TO 100
          endif


      end
c**************************************
          subroutine High(x,obj)

          use param

       IMPLICIT double precision (A-H,O-Z),INTEGER(I-N)
       DIMENSION x(dimen), y(10)
          common /design/ numb,ni,nold
          common /bound/ xl(dimen),xu(dimen),e(dimen)

          do i= 1,dimen
            y(i)= x(i)
          enddo
          do i= dimen+1,10
            y(i) = 0
          enddo

          m=1
          if (m .eq. 1) then
            obj =(y(1)**2+y(2)**2+y(3)**2+y(4)**2+y(5)**2+y(6)**2+y(7)**
     & 2+y(8)**2+y(9)**2+y(10)**2)/1000.
     & +1.-cos(y(1))*cos(y(2)/sqrt(2.))*cos(y(3)/sqrt(3.))*
     & cos(y(4)/2.)*cos(y(5)/sqrt(5.))*cos(y(6)/sqrt(6.))*
     & cos(y(7)/sqrt(7.))*cos(y(8)/sqrt(8.))*cos(y(9)/3.)*
     & cos(y(10)/sqrt(10.))
          endif

          m=0
          if (m .eq. 1) then
        obj=0
        do l=1,dimen
          obj= obj+10+2.0*(x(l)+e(l))**2-1.5*(x(l)+e(l))**4
```

```
     &     +.2*(x(l)+e(l))**6
           enddo

             endif

             return
             end
c****************************************
             subroutine Low(y,obj)

             use param

            IMPLICIT double precision (A-H,O-Z),INTEGER(I-N)
            DIMENSION y(dimen)
             common /design/ numb,ni,nold
             common /bound/ xl(dimen),xu(dimen),e(dimen)

             m=1
             if (m .eq. 1) then                !griewank
             obj = 0
             do i=1,dimen
             obj = obj + (y(i)+e(i))**2/5000.
             enddo
             endif

             m=0
             if (m .eq. 1) then                !original low
           obj=0
           do l=1,dimen
             obj= obj+10+2.5*(y(l)+e(l))**2-1.2*(y(l)+e(l))**4
           enddo
             endif

             m=0
             if (m .eq. 1) then                !chebychev low
           obj=0
           do l=1,dimen
             obj= obj +10-1 -2*(2*((y(l)+e(l))/2.)**2-1)
     &   -.6*(8*((y(l)+e(l))/2.)**4-8*((y(l)+e(l))/2.)**2+1)
           enddo
             endif

             m=0
             if (m .eq. 1) then                !chebychev low + part of high
           obj=0
           do l=1,dimen
```

```
          obj= obj + 10 - 1 -2*(2*((y(l)+e(l))/2.)**2-1)
     &   -.6*(8*((y(l)+e(l))/2.)**4-8*((y(l)+e(l))/2.)**2+1)
     &  + .5*(.4*(32*((y(l)+e(l))/2.)**6-48*((y(l)+e(l))/2.)**4
     &  +18*((y(l)+e(l))/2.)**2-1))

        enddo
          endif

           m=0
          if (m .eq. 1) then                !linear addition to high fidelity function
        obj=0
        do l=1,dimen
          obj= obj+10+2.0*(y(l)+e(l))**2-1.5*(y(l)+e(l))**4
     &     +.2*(y(l)+e(l))**6  ! +    .6* y(l)+.02*y(l)**2
        enddo

          do l=1,dimen                      !piecewise linear
          if (y(l) .lt. -.8) then
           obj = obj+ y(l)+.8
          else if (y(l) .gt. .4) then
           obj = obj - 1.2 * y(l)+1.08
          else
           obj = obj +.5* y(l)+.4
          endif
          enddo

          endif
          return
          end
c*****************************************
          subroutine con(x,g,ncon)

          use param

          IMPLICIT REAL*8 (A-H,O-Z),INTEGER(I-N)
          common /design/ numb,ni,nold
          common /bound/ xl(dimen),xu(dimen),e(dimen)
        DIMENSION X(dimen),G(ncon)

          do j=1,ncon
          g(j)=(-2000.+x(j))/100.
          enddo

           RETURN
        END
```

LIST OF REFERENCES

Alexandrov, N. M., Dennis, J. E., Lewis, R. M., and Torczon, V., "A Trust-region Framework for Managing the use of Approximation Models in Optimization," *Structural Optimization*, Vol. 15, 1998, pp. 16-23

Alexandrov, N. M., Lewis, R. M., Gumbert, C.R., Green, L. L. and Newman, P.A., "Optimization with Variable-fidelity Models Applied to Wing Design," *38th AIAA, Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, Jan 2000, AIAA Paper-2000-0841

Audet, C. and Dennis JR., J. E., "Analysis of Generalized Pattern Search," *Technical Report, Department of Computational and Applied Mathematics*, Rice Univ., February 2000, TR00-07

Audet, C. and Dennis JR., J. E., "A Pattern Search Filter Method for Nonlinear Programming without Derivatives," *Technical Report, Department of Computational and Applied Mathematics*, Rice Univ., March 2000, TR00-09

Baker, C. A., Grossman, B., Haftka, R. T., Mason, W. H. and Watson, L. T., "HSCT Configuration Design Space Exploration using Aerodynamic Response Surface Approximations," *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, St Louis, Missouri, September 2-4, 1998, pp. 769-777, AIAA Paper-98-4803

Baker, C. A., Watson, L. T., Grossman, B., Haftka, R. T. and Mason, W. H., "Study of a Global Design Space Exploration Method for Aerospace Vehicles," *8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Long Beach, California, September 6-8, 2000, AIAA Paper 2000-4763

Baker, C. A., Watson, L. T., Grossman, B., Mason, W. H. and Haftka, R. T., " Parallel Global Aircraft Configuration Design Space Exploration," *International Journal of Computer Research*, Vol. 10, No. 4, 2001, pp. 501-515

Balabanov, V., Haftka, R. T., Grossman, B., Mason, W. H. and Watson, L. T. "Multifidelity response surface model for HSCT wing bending material weight," *7th AIAA/USAF/NASA/ISSMO Symposium on Multi Disciplinary Analysis and Optimization*, St. Louis, Missouri, 1998, pp. 778-788.

Bartholomew-Biggs, M. C., Parkhurst, S. C. and Wilson, S. P., "Global Optimization Approaches to an Aircraft Routing Problem," *Computational Optimization and Applications*, Vol. 21, No. 3, 2003, pp.311-323

Bhardwaj, M., Reese, G., Driessen, B., Alvin, K. and Day, D., "Salinas – an Implicit Finite Element Structural Dynamics Code Developed for Massively Parallel Platforms," *41st AIAA/ASME/ASCE/AHS/ASC SDM*, Atlanta, Georgia, 2000, AIAA Paper-2000-1651

Boender, C. G. E. and Kan, A. H. G. R., "A Bayesian Analysis of the Number of Cells of a Multinomial Distribution," *The Statistician*, Vol. 32, 1983, pp. 240-248

Burgee, S., Giunta, A. A., Balabanov, V., Grossman, B., Mason, W. H., Narducci, R., Haftka, R. T. and Watson, L. T. "A coarse-grained parallel variable-complexity multidisciplinary optimization paradigm," *The International Journal of supercomputing applications and high performance computing*, Vol. 10, No. 4, 1996, pp. 269-299

Carlisle, A. and Dozier, G. "An off the Shelf PSO," *Proc. Workshop on Particle Swarm Optimization*, Purdue School of Engineering and Technology, Indianapolis, Indiana, 2001

Carlson, H. W. and Miller, D. S., "Numerical Methods for the Design and Analysis of Wings at Supersonic Speeds," NASA TN D-7713, December 1974

Carlson, H.W. and Walkley, K.B., "Numerical Methods and a Computer Program for Subsonic and Supersonic Aerodynamic Design and Analysis of Wings with Attainable Thrust Corrections," NASA CR-3808, 1984

Chan, K. L., Kennedy, D. and Williams, F. W., "Hybrid Optimization Strategy Beyond Local Optima in Aerospace Panel Designs," *AIAA Journal*, Vol. 37, No. 5, May 1999, pp. 588-593

Chang, K. J., Haftka, R. T., Giles, G. L. and Kao, P. J. "Sensitivity-based scaling for approximating structural response," *Journal of Aircraft*, Vol. 30, No. 2, 1993, pp. 283-287.

Clerc, M., "The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization," *Proc. Congress of Evolutionary Computation*, Vol. 3, IEEE Press, Mayflower Hotel, Washington D.C., July 1999, pp.1951-1957

Cox, S., Haftka, R.T., Baker, C., Grossman, B., Mason, W. and Watson, L. T., "Global Optimization of a High Speed Civil Transport," *Journal of Global Optimization*, Vol. 21, No. 4, December 2001, pp.415-433

Culler, D. E., Singh, J. P. and Gupta, A., *Parallel Computer Architecture A Hardware / Software Approach*, Morgan Kaufmann Publishers, Inc., 1999

Dennis, J. E. and Torczon, V., "Direct search methods on parallel machines," *SIAM Journal of Optimization*, Vol. 1, 1991, pp.448-474

Floudas, C.A. and Pardalos, P.M., *State of the Art in Global Optimization*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996

Floudas, C.A., Pardalos, P.M., Adjiman, C. S., Esposito, W. R., Gumus, Z. H., Harding, S. T., Klepeis, J. L., Meyer, C. A. and Schweiger, C. A., *Handbook of Test Problems in Local and Global optimization*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999

Fourie, P. and Groenwold, A., "Particle Swarms in size and Shape Optimization," *Proc. Workshop on Multidisciplinary Design Optimization*, Pretoria, South Africa, August 2000, pp. 97-106

Giunta, A.A., Narducci, R., Burgee, S., Grossman, B., Haftka, R.T., Mason, W.H., and Watson, L.T., "Variable-Complexity Response Surface Aerodynamic Design of an HSCT Wing," *Proceedings of the 13th AIAA Applied Aerodynamics Conference*, San Diego, California, June 19-22, 1995, pp. 994-1002, AIAA Paper 95-1886

Giunta, A. and Watson, L. T., "A Comparison of Approximation Modeling Techniques: Polynomial Versus Interpolating Models," *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, St. Louis, Missouri, September 2-4, 1998, pp. 392-404, AIAA Paper-98-4758

Groenwold, A., Schutte, J., and Bolton, P. "Competing Parallel Algorithms and Multiple Local Searches in Global Optimization," *International Workshop on Multidisciplinary Design Optimization*, Pretoria, South Africa, August 7-10, 2000, pp. 134-143

Haftka, R. T. "Combining global and local approximations," *AIAA Journal*, Vol. 29, No. 9, 1991, pp. 1523-1525.

Haftka, R. T. and Gürdal, *Elements of Structural Optimization*, Third Revised and Expanded Edition, Kluwar Academic Publishers, The Netherlands, 1992

Haim, D., Giunta, A.A., Holzwarth, M.M., Mason, W.H., Watson, L.T. and Haftka R.T., "Comparison of Optimization Software Packages for an Aircraft Multidisciplinary Design Optimization Problem," *Design Optimization*, Vol. 1, 1999, pp. 9-23.

Hart, W., "A Convergence Analysis of Unconstrained and Bound Constrained Evolutionary Pattern Search," *Evolutionary Computation*, Vol. 9, No. 1, 2001, pp. 1-23

Hopkins, E. J., "Charts for Predicting Turbulent Skin Friction from the van Driest Method," NASA TN D-6945, October 1972

He, J., Watson, L. T., Ramakrishnan, N., Shaffer, C. A., Verstak, A., Jiang, J., Bae, K. and Tranter, W. H., "Dynamic Data Structures for a DIRECT Search Algorithm," Computational Optimization and Applications, Vol. 23, Issue 1, October 2002, pp. 5-25

Ishikawa, T., Tsukui, Y. and Matsunami, M., "A Combined Method for the Global Optimization Using Radial Basis Function and Deterministic Approach," *IEEE Transactions on Magnetics*, Vol. 35, No. 3, May 1999, pp. 1730-1733

Jones, D.R., Perttunen, C.D. and Stuckman, B.E., "Lipschitzian Optimization Without the Lipschitz Constant," *Journal of Optimization Theory and Application*, Vol. 79, 1993, pp. 157-181.

Jones, D. R., Schonlau, M. and Welch, W. J., "Efficient Global Optimization of Expensive Black-Box Functions," *Journal of Global Optimization*, Vol. 13, 1998, pp. 455-492

Kam, T. Y., Lai, F. M. and Liao, S. C., "Minimum Weight Design of Laminated Composite Plates Subject to Strength Constraint," *AIAA Journal*, Vol. 34, No. 8, August 1996, pp. 1699-1708

Kaufman, M., Balabanov, V., Burgee, S. L., Giunta, A. A., Grossman, B., Mason, W. H. and Watson, L. T. "Variable complexity response surface approximation for wing structural weight in HSCT design," *AIAA 34th Aerospace Sciences Meeting & Exhibit*, Reno, Nevada, January 1996, pp. 112-127, AIAA Paper 96-0089

Kaufman, M., Balabanov, V., Grossman, B., Mason, W. H., Watson, L. T. and Haftka, R. T., "Multidisciplinary Optimization Via Response Surface Techniques," *Proceedings of the 36th Israel Conference on Aerospace Sciences*, Israel, February 21-22, 1996, pp. A-57 to A-67

Kennedy, J. and Eberhart, R., "Particle Swarm Optimization," *Proc. 1995 IEEE International Conference on Neural Networks*, Vol. 4, Perth, Australia, IEEE Service Center, Piscataway, New Jersey, 1995, pp. 1942-1948

Knill, D. L., .Giunta, A. A., Baker, C. A., Grossman, B., Mason, W. H., Haftka, R. T. and Watson, L. T. "Response surface model combining linear and euler aerodynamics for super sonic transport design," *Journal of Aircraft*, Vol. 36, No. 1, 1999, pp. 75-86.

Korngold, J. C. and Gabriele, G. A., "Multidisciplinary Analysis and Optimization of Discrete Problems using Response Surface Methods," *Journal of Mechanical Design*, Vol. 119, December 1997, pp. 427-433

Krasteva, D. T., Watson, L. T., Baker, C., Grossman, B., Mason, W. H. and Haftka, R. T., "Distributed control parallelism in multidisciplinary aircraft design," *Concurrency: Practice and Experience*, Vol. 11, No. 8, 1999, pp. 435-459

Kroo, I., Altus, S., Braun, R., Gage, P. and Sobieski, I. "Multidisciplinary Optimization Methods for Aircraft Preliminary Design," *5th AIAA/USAF/NASA/ISSMO Symposium on multidisciplinary analysis and optimization*, Panama City, Florida, 1994, pp. 697-707.

Lamberti, L., Venkataraman, S., Haftka, R. T. and Johnson, T. F., "Preliminary Design Optimization of Stiffened Panels Using Approximate Analysis Models," *Accepted for publishing in the International Journal for Numerical Methods in Engineering*, 2002

Liu, B., Haftka, R. T. and Akgün, "Two-Level Composite Wing Structural Optimization Using Response Surfaces," *Structural Multidisciplinary Optimization*, Vol. 20, 2000, pp. 87-96

Lombardi, M., Haftka, R.T. and Cinquini, C., "Optimization of Composite Plates for Buckling by Simulated Annealing," *33rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, Dallas, Texas, April 1992, pp. 2552-2563

McQuade, P.D., Eberhardt, S. and Livne, E., "CFD-Based Aerodynamic Approximation Concepts Optimization of a Two-Dimensional Scramjet Vehicle," *Journal of Aircraft*, Vol. 32, No. 2, March-April 1995, pp. 262-269

MacMillin, P.E., Golovidov, O.B., Mason, W.H., Grossman, B. and Haftka, R.T., "An MDO Investigation of the Impact of Practical Constraints on an HSCT Configuration," *AIAA 35th Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, January 1997, AIAA Paper 97-0098

McGrory, W.D., Slack, D.C., Applebaum, M.P. and Walters, R.W. *GASP Version 2.2 Users Manual*, Aerosoft, Inc., Blacksburg, Virginia, 1993

Nelson II, S.A. and Papalambros, P.Y., "A Modification to Jones' Global Optimization Algorithm for Fast Local Convergence," *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, St Louis, Missouri, September 1998, pp. 341-348, AIAA Paper-98-4751

Ramírez-Rosado, I. J. and Bernal-Augustín, J.L., "Genetic Algorithms Applied to the Design of Large Power Distribution Systems," *IEEE Transactions on Power Systems*, Vol. 13, No. 2, May 1998, pp. 696-703

Raymer, D. P., *Aircraft Design: A Conceptual Approach*, AIAA Education Series, Washington, DC, 1992

Savic, D. A. and Walters, G. A., "Genetic Algorithms for Least-Cost Design of Water Distribution Networks," *Journal of Water Resources Planning and Management*, Vol. 123, No. 2, March/April 1997, pp. 67-77

Schutte, J. F. and Groenwold, A. A., "Optimal Sizing Design of Truss Structures Using the Particle Swarm Optimization Algorithm," *Proc. 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization Conference,* Atlanta, Georgia, September 4-6, 2002, AIAA Paper-02-5639

Shi, Y. and Eberhart, R., "A Modified Particle Swarm Optimizer," *Proc. IEEE International Conference on Evolutionary Computation*, IEEE Press, Piscataway, New Jersey, 1998, pp. 67-73

Simpson, T. W., Mauery, T. M., Korte, J. J. and Mistree, F., "Comparison of Response Surface and Kriging Models for Multidisciplinary Design Optimization," *7th*

AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St Louis, Missouri, September 2-4, 1998, pp. 381-391, AIAA Paper-98-4755

Snyman, J.A., "An Improved version of the original Leap-Frog dynamic method for unconstrained minimization: LFOP1(b) ," *Appl. Math. Modelling*, Vol.7, 1983, pp. 216-218.

Snyman, J. A. and Fatti, L. P., "A Multi-Start Global Minimization Algorithm with Dynamic Search Trajectories," *Journal of Optimization Theory and Applications*, Vol 54, No. 1, July 1987, pp. 121-141

Soh, C. K. and Yang, J., "Fuzzy Controlled Genetic Algorithm Search for Shape Optimization," *Journal of Computing in Civil Engineering*, April 1996, pp. 143-150

Swift, R. A. and Batill, S. M., "Simulated Annealing Utilizing Neural Networks for Discrete Variable Optimization Problems in Structural Design," *33rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, Dallas, Texas, April 1992, pp. 2536-2544

Torczon, V., "On the Convergence of Pattern Search Algorithms," *Siam Journal of Optimization*, Vol 7, No. 1, February 1997, pp. 1-25

Tzan, S. and Pantelides, C. P., "Annealing Strategy for Optimal Structural Design," *Journal of Structural Engineering*, Vol. 122, No. 7, July 1996, pp. 815-826

Vanderplaats Research & Development, Inc., *DOT: Design Optimization Tools*, Version 4.20, 1995

Vasconcelos, J. A., Saldanha, R.R., Krahenbühl, L. and Nicolas, A., "Genetic Algorithm Coupled with a Deterministic Method for Optimization in Electromagnetics," *IEEE Transactions on Magnetics*, Vol. 33, No. 2, March 1997, pp. 1860-1863

Vitali, R., Haftka, R. T. and Sankar, B. V., "Correction Response Surface Approximations for Stress Intensity Factors for Composite Stiffened Plates," *AIAA/ASME/ASCE/ AHS/ASC Structures, Structural Dynamics and Material Conference*, Long Beach, California, 1998, Vol. 4, pp. 2917- 2922, AIAA Paper-98-2047

Vitali, R. and Sankar, B.V. "Correction Response Surface Design of Stiffened Composite Panel with a Crack," *AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Material Conference*, St. Louis, Missouri, 1999, AIAA Paper-99-1313

Weston, R. P., Townsend, J. C., Eidson, T. M. and Gates, R. L., "A Distributed Computing Environment for Multidisciplinary Design," *5th AIAA/USAF/NASA/ISSMO Symposium*, Panama City Beach, Florida, September 1994, pp. 1091-1097, AIAA Paper-94-4372-CP

BIOGRAPHICAL SKETCH

Steven Cox was born in Jacksonville, Florida where he lived until starting college at the University of Florida. He received a Bachelor of Science in Aerospace Engineering in December of 1997 with a minor in Materials Engineering. He then started graduate school under the supervision of Dr Raphael Haftka where he earned a Master of Science in Aerospace Engineering in December 2000. He has worked on internships with Natalia Alexandrov at NASA Langley Research center and William Hart at Sandia National Labs.