RICE UNIVERSITY

A Framework for Managing Models in Nonlinear Optimization of Computationally Expensive Functions

by

David B. Serafini

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

John E. Dennis, Jr., Chairman
Noah Harding Professor of Computational and Applied Mathematics

Danny C. Sorensen
Professor of Computational and Applied Mathematics

Andrew J. Meade, Jr.
Associate Professor of Mechanical Engineering and Materials Science

Houston, Texas

November, 1998

Abstract

A Framework for Managing Models in Nonlinear Optimization of Computationally Expensive Functions

by

David B. Serafini

One of the most significant problems in the application of standard optimization methods to real-world engineering design problems is that the computation of the objective function often takes so much computer time (sometimes hours) that traditional optimization techniques are not practical. A solution that has long been used in this situation has been to approximate the objective function with something much cheaper to compute, called a "model" (or surrogate), and optimize the model instead of the actual objective function. This simple approach succeeds some of the time, but sometimes it fails because there is not sufficient *a priori* knowledge to build an adequate model. One way to address this problem is to build the model with whatever a priori knowledge is available, and *during the optimization process* sample the true objective at selected points and use the results to monitor the progress of the optimization and to *adapt* the model in the region of interest. We call this approach "model management". This thesis will build on the fundamental ideas and theory of pattern search optimization methods to develop a rigorous methodology for model management. A general framework for model management algorithms will be presented along with a convergence analysis. A software implementation of the framework, which allows for the reuse of existing modeling and optimization software, has been developed and results for several test problems will be presented. The model management methodology and potential applications in aerospace engineering are the subject of an ongoing collaboration between researchers at Boeing, IBM, Rice and College of William & Mary.

Acknowledgments

This dissertation and the research it documents would not have been possible without the guidance and support of my advisor, John Dennis. His concern and patience are greatly appreciated. I'd like to acknowledge the other members of my thesis committee, Virginia Torczon, Dan Sorensen and Andrew Meade, for their efforts in maximizing the quality of this thesis. Prof. Torczon, in particular, has made many valuable contributions to my work. I appreciate her efforts more than she knows.

This work was supported by the NSF, through the Center for Research on Parallel Computation at Rice Univesity, the Air Force Office of Sponsored Research, The Boeing Company, IBM and NASA. My thanks to everyone involved.

In particular, many thanks must go to Greg Shubin of the Research and Technology division of the Boeing Company. His support of the collaboration between Rice and Boeing was invaluable to my work. Boeing provided the main test problem I used in this thesis and some significant pieces of software that would have been very time-consuming to write from scratch. Greg also made it possible for me to visit his group for a year, which was a great help to my work. My collaborators at Boeing – Andrew Booker, Evin Cramer and Paul Frank – were extremely generous with their time and expertise. Other Boeing people I'd like to thank are Tom Grandine, Rich Mastro and Michael Carter, who all helped with acquiring and using various pieces of Boeing software. I'm especially indebted to Andrew Booker for assistance with his DACEPAC software.

Paul Frank of Boeing and Katya Mints of IBM provided computational results that are reported herein; I thank them for their assistance.

My personal thanks go to Ellen, Jo and Drew, Bonnie, Rocky, Dave, Ingrid, Maeve and Karin for their friendship and support. I'm very grateful to Mike Fagan and Keith Berrier at Rice and Mike Epton at Boeing for the many happy hours I spent in random conversations with them. These diversions helped keep me sane. Special thanks go to Michael Lewis for the week at ICASE and an apology for turning his office into a war zone.

Before graduate school, Bill Rose was my collaborator and mentor while I worked on several NASA-sponsored projects in computational aerodynamics. Working with him taught me a great many things about doing research (and making a living at it). Everything I have learned about research since then has increased my appreciation of those lessons. I hope this thesis lives up to the standards of our joint work.

My parents, Robert and Marie Serafini, always supported me without reservation and with infinite patience and love, through all the twists and turns my life has taken. Ever since my mother died, my father has born the brunt of this alone and I've relied on his constancy more than I can say. This thesis is dedicated to the memory of my mother, whose encouragement invariably gave me the confidence to continue to pursue my goals whenever I doubted I could achieve them. I wish she was here to share in the achievement of this particular goal.

Contents

Illustrations

Chapter 1

Introduction

There are many applications of optimization in engineering and science where the cost of computing the objective function values (or derivatives) is so large that standard optimization methods cannot produce results in reasonable time or at reasonable cost. Engineering design is probably the primary example of this kind of application but certainly not the only one. Optimization traditionally has important applications in process control and has seen increasing use recently in biology and chemistry. Many of these applications have extremely large computational requirements.

The research described in this thesis was done as part of an ongoing collaboration between Rice University, Boeing, IBM and the College of William & Mary [6, 7]. As a result, the emphasis of this thesis will be on engineering design applications, although the methods are applicable to non-design problems as well.

1.1 Justification of current work

There is growing interest in the use of computational optimization in many disciplines because, compared to existing practices, computational optimization can offer better results, shorter design cycle times, and reduced human effort. Many examples in engineering design applications have been identified during the course of our collaboration with Boeing.

The primary motivation for this work is that the rewards of finding better solutions to these optimization problems are very large and justify the efforts required to develop methods that can produce these solutions. In engineering, better designs lead to better performance of the product or system being designed. In the highly competitive environment that many engineering disciplines face, better performance (or a reduction in cost) are a major advantage. In other areas, where current designs are often already at the limits of technology, even small improvements in performance can be the difference between success and failure. In scientific applications, better optimization results can lead to better simulations of natural phenomena and a better understanding of the universe. It should be noted that in many of these applications the ultimate goal is not necessarily to find the exact optimizer of the objective function; just finding a better solution can be a satisfactory result.

Shorter design cycle times can improve time-to-market, another major competitive advantage. Faster turn-around can improve the quality of the design early in the design process, when time is the limiting factor and many of the features of a design are "locked in", meaning they cannot be changed later in the design process without significant (and often unacceptable) cost and delay. Thus small improvements early in the design process are magnified later in the process.

Clearly, extremely long solution times negate many of the benefits just described and thus are a serious impediment to the continued growth in the use of computational optimization methods in engineering and science applications. In engineering design, the two main reasons the objective function evaluations are so costly are that the systems being engineered are very complex (leading to complex simulations) and the accuracy requirements for the simulations are high. After years of research and development effort, existing designs already take maximum advantage of the simple, low-fidelity simulations. Higher fidelity (hence higher cost) simulations and more sophisticated solution methods are required to make the major improvements in system quality and design cycle time that are the goals in current design environments.

The work discussed in this thesis is an attempt to address the inadequacies of existing optimization methods for this class of applications. The essential feature of our approach is to replace the expensive functions with ones that are cheaper to evaluate and use the cheaper functions in the optimization procedure. The contributions of this work are:

1. a general purpose, abstract framework for methods of this type
2. a convergence analysis for the framework
3. a general purpose software implementation of the framework that supports a wide variety of methods.

1.2 Background

The basic idea of replacing the objective (or constraint) function in an optimization problem in order to make the problem easier to solve is an old one. The use of Taylor series approximations goes back to Raphson, if not all the way to Newton himself. The use of modeling has a long history in engineering applications of optimization and a great deal of work has been done in that area. For example, see [2] and [65] for reviews of the structural and aerospace design fields and see [11], [30], [31], [60], [63], [67] for details of recent work in this area.

The methods developed in engineering have tended to be very application-specific, depending on specific characteristics of either the problem or the modeling strategy or both. The more general work that has been the subject of research in the field of nonlinear programming has tended to focus strongly on Taylor series models and gradient-based methods. Much less work on general purpose methods for derivative-free models has been done (see [32],[50],[53] [59] and [14, 15]).

In engineering applications, the models used can be differentiated into two basic types: numerical solutions of governing equations of physical systems and functional approximations of the solutions of the equations constructed without resort to knowledge of the physical system, that is, by using the values of the functions only. For convenience, we will denote these types as "physical" and "functional" models, respectively. The key distinguishing feature is that the former embodies knowledge of the physical system, while the latter is purely a mathematical construct. A functional model only embodies knowledge of the behavior of the function it is approximating at the points for which function values are given, while the physical model embodies knowledge of the behavior of the system at all points.

The distinction between these two types of model is not absolute. There are some models that can be defined arguably as either type. For example, a Taylor series model is arguably a functional model since differentiation of the governing equations is a purely mathematical process that can be done without any knowledge of the system. It is also arguably a physical model since the derivatives of the governing equations describe the behavior of the same system that the governing equations describe.

Another difference is that typically some number of evaluations of the actual function are required to construct a model of the functional type, whereas a physical model can be used without such a startup cost. Since we are assuming the evaluations of the actual function are very costly, the cost of constructing a functional model can be a significant factor in the choice of a modeling strategy for a particular application.

Examples of the governing equation type of modeling are the use of the potential flow or Euler equations as models of the Navier-Stokes equations. The fidelity of a physical model can vary depending on something as simple as a grid spacing parameter, or as complex as a turbulence model formulation in a Navier-Stokes equation solver.

Examples of the functional type of model are polynomial interpolation [15, 20, 59], splines[13], neural nets and other type of curve fits (particularly response surface models [9][43]), and least-squares models (linear or nonlinear) [45].

Hybrids that have characteristics of both types can be used. For example, one could take a simplified version of the equations describing a physical system, evaluate both the simplified and complete equations at a set of points in the design space and fit a linear least-squares model to the differences. To evaluate the model at some other point, the values of the simplified equations and the least-squares fit of the difference are computed and summed. The hybrid model captures some of the behavior of the full equations at essentially the cost of the simplified equations.

Throughout the rest of this thesis, the term "model" will be used to refer to any function that is used as an approximation of another function for the purpose of reducing the cost of evaluation or improving the behavior of the function. Although the discussion will concentrate on modeling the objective function, it applies to modeling constraint functions as well. The term "model management" will be used to describe optimization methods that use models as surrogates of the problem functions and modify the models during the optimization process.

It is assumed that for the applications of interest in this work the derivative information necessary to build Taylor series models for use in gradient-based optimization methods is not available or cannot be computed except at great cost. A frequently-used approach to overcoming this cost involves replacing the objective function in the optimization method with a model which is cheaper to evaluate. In addition to the obvious cost advantage, this approach may have an additional benefit if the surrogate function is smoother than the actual function, thus making the optimization problem easier. Another approach is to use an optimization method that does not require derivative information.

The major disadvantage of simply replacing the objective with a model is that the solution of the optimization problem using the model is not necessarily the solution to the original problem. Furthermore, the difference in the two solutions usually is not known *a priori* and cannot be determined without significant effort, negating the cost advantage. The major disadvantage of most derivative-free optimization methods is that they require many more function evaluations than derivative-based methods. This motivates our efforts to develop derivative-free optimization methods that use models to ameliorate the cost of the extra function evaluations.

A wide variety of approaches is used in practice to build models of the functional type that do not require gradient information. One characteristic shared by many approaches is that the model is built before doing the optimization and typically it is not modified until after the optimum of the model has been found, if it is modified at all. A basic premise of this work is that the model should be modified during the optimization process and that the strategy for modifying (or replacing) the model should be driven by values of the actual objective function that are computed during the process. When and how these modifications are made, especially when and where the actual objective is evaluated, and how these choices are integrated into an optimization method, define the "management" part of a model management method.

The motivation for the present work stems from what we consider to be deficiencies in current practice in methods for using models in optimization. One deficiency is the lack of generality in some methods. This arises in methods where some special characteristics of the application are critical to the success of the method. In the application domain for which the method is intended, this is usually an advantage because it can lead to better performance. But it can restrict the usability of the method in other application domains. One of our goals is to develop a general-purpose method that facilitates, but does not require, the incorporation of application-specific knowledge.

Some existing methods exhibit a reliance on the special behavior of derivative-based models, especially Taylor series models. This leads to general-purpose methods that work well for problems in many application domains, but which are inappropriate for the applications we are interested in because the cost of computing derivative information is too high. A secondary issue that arises in some applications is that even though the derivative information is computable, it is unreliable enough (noisy, inaccurate, non-smooth) to hamper the effectiveness of the optimization method. Thus, another of our goals is to develop a derivative-free method.

Another deficiency from which some methods suffer is a lack of theoretical support. This occurs more often among methods that use models other than Taylor series models. We are interested in these models because they are more practical for the applications we have targeted. We consider it particularly important that the methodology we propose has a general convergence analysis that can be applied to realistic algorithms using any type of model.

1.3 An algorithmic framework for model management

For the purposes of this work, how the model is used in the optimization procedure and how it is determined when the model needs to be modified are more important than how the model is constructed and evaluated. We want our methodology to be independent of the type of model used. This gives the methodology greater generality. For a particular application, the choice of modeling strategy will be extremely important, but that is a decision that should be based on issues such as performance, available expertise, and the nature of the application, *not* on the ability of the model to work within the framework.

We seek to generalize current practice in using models in optimization, address the deficiencies noted, and provide a capability for designing and expressing model management algorithms. To do this we develop a framework, called the Model Management Framework, that expresses an abstract model management algorithm. To make a practical algorithm within the Framework it is necessary to specify implementations for certain specific operations that are required in the abstract algorithm but are not defined *a priori*. The abstraction of a model that is used in the Framework requires only evaluation and modification operations. Any modeling technique that can provide these capabilities can be used with the Framework.

The abstract optimization algorithm used in the Framework is based on a general form for pattern search methods. This generalized method and the convergence theory for it was originally developed by Torczon in [72] and refined by Lewis and Torczon in [47], [48] and [49].

The Model Management Framework also uses ideas (but not theory) from the trust-region class of quasi-Newton methods [22]. The idea of combining these methods was introduced by Dennis and Torczon in [21] in a somewhat different form than is presented here. The pattern search convergence theory is used as the theoretical basis in both cases.

Pattern search methods are a subclass of direct search methods. Direct search methods, as their name implies, operate by evaluating the objective function at individual points in the problem space. Direct search methods do not explicitly use any derivative information. There are many different versions of direct search methods dating back to the early days of numerical nonlinear optimization [10, 37, 55, 66, 69]. The characteristic that distinguishes pattern search methods from other direct search methods is the use of a pattern of points to define where to evaluate the objective function. The pattern need not be static; it may vary during the course of the search.

A pattern search method [72] has the basic form:

1. Construct or update a pattern of points around the current iterate.

2. Search the pattern to find a point in the pattern that reduces the objective function subject to a requirement that certain points must be evaluated under certain conditions.
3. Depending on whether the search found a point with decrease, decide if and how to change the pattern.
4. Iterate until convergence

The key features of pattern search methods that are important to the current work are the independence from explicit derivative information, the discreteness of the pattern, and the existence of a convergence theory. This will be discussed in detail in Chapter 2.

The algorithm used in the Framework is developed and analyzed as a kind of pattern search method, but it is also motivated by trust-region methods. A typical trust region algorithm [22] has the form:

1. Build (update) a quadratic (Taylor series) model of the objective function.
2. Optimize the model subject to a bound on the step size (the trust region radius).
3. Evaluate the objective function at the step-bounded approximate minimizer of the model.
4. Compute the ratio of actual reduction in the objective to the predicted model reduction.
5. Based on this ratio, decide whether to accept the step and how to change the trust region radius.
6. Iterate until convergence.

The Model Management Framework we propose modifies this basic methodology in two ways, based on one key observation. The quadratic model is generalized to any kind of model of the objective. This model can be of the physical or functional types described above or a hybrid of the two. The model can even be different in different parts of the domain. The consequence of this relaxation of the trust region methodology is that the convergence theory for trust region methods does not apply.

The second modification of the trust region approach follows from the first: since the model does not have a predetermined form, the standard trust region quadratic subproblem is replaced by a more general subproblem. In our approach, the optimization subproblem is reduced to the subproblem of the pattern search method: find a step in the pattern that improves the objective function (or some other merit function). As in the trust region quadratic subproblem, the pattern search subproblem in the Framework is treated as a "black box". This means that the algorithm used to solve the subproblem is not specified by the Framework and does not affect the Framework in any way except through the results it produces. This generalization of the subproblem allows a wide variety of methods to be used, and specifically allows, but does not require, the method to be tailored to the application domain. The key differences between the pattern search subproblem and the trust region subproblem are that the former requires the solution to be taken from the discrete points in the pattern and has weaker conditions on acceptability of trial iterates.

The insight that motivates these generalizations is this: trust-region methods work well because there is a direct relationship between the accuracy of the Taylor series model and the trust region radius. Thus the control algorithm in the trust-region method fulfills two purposes with one variable. The trust region radius controls the accuracy of the model and limits the amount of decrease required in the optimization subproblem. The smaller the region, the higher the accuracy of the model and the less decrease required from the solution to the model subproblem.

The Model Management Framework has to sacrifice this nice feature because it cannot rely on the model to be more accurate locally. Perforce we must replace it with some other mechanism to guarantee that the method converges to a solution. The

Framework inherits the convergence properties from the pattern search methods. The sufficient decrease condition that governs the trust region subproblem is replaced with a simple decrease condition on the pattern search subproblem. The discrete nature of the pattern ultimately provides the guarantee of convergence.

The control algorithm in a model management method is more complex that that of a trust-region method or a model management method that uses derivative information (e.g. [1]), which can rely on the local behavior of the Taylor series model. This is the price we must pay for a derivative-free method.

Using the ideas from trust region methods discussed above and merging them with the pattern search method we can express the basic form of the Model Management Framework:

1. Build (update) a model of the objective function.
2. Generate a trial step that decreases the model of the objective and satisfies the requirements of the pattern search convergence theory, if possible.
3. Evaluate the actual objective function at the trial step.
4. Based on the comparison of the model and actual values, and possibly other criteria, decide whether to accept or reject the trial step.
5. Iterate until convergence.

In the next chapter we will develop these concepts into a precise, rigorous, general purpose method for nonlinear optimization for computationally expensive functions.

1.4 A simple example

To demonstrate what we mean by model management we now present a very simple example. This is not precisely the form that the Framework we propose will take, but it is sufficient for illustrative purposes.

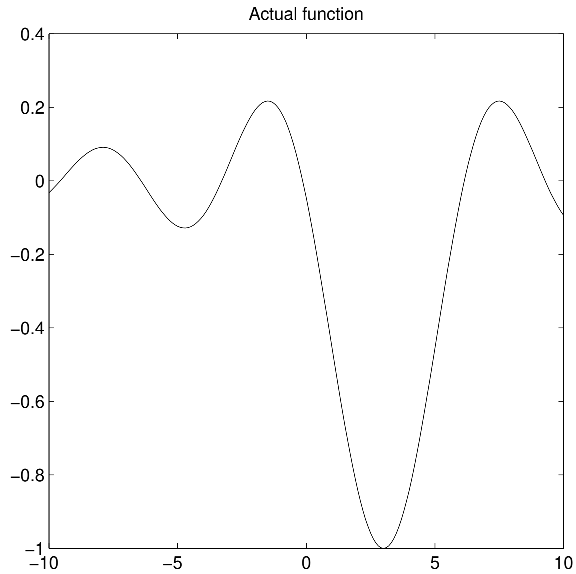


Figure 1.1 Plot of example objective function.

Take the scalar function $F(x)$ of one variable on the bounded interval [-10:10] defined by:

$$F(x) = -\frac{\sin(x - 3 + \epsilon)}{x - 3 + \epsilon},$$

where ϵ is a very small number. Figure 1.1 is a plot of this function. As our class of models we will use a simple cubic spline (as implemented by the MATLAB* "spline" function).

The procedure we will follow is:

1. construct the model (spline) using the function values we have computed so far
2. compute the minimizer of the model (spline)
3. evaluate the function at the minimizer
4. accept the minimizer as the new iterate if the function decreased, else keep the current iterate

*MATLAB Version 4.2c was used for this example. MATLAB is a registered trademark of The MathWorks, Inc.

5. repeat 1–4.

We start with two points equally spaced in the bounded interval, namely -15/3 and +15/3. We take the one with the smaller function value as our initial iterate. Figure 1.2 shows the models that result from following this procedure four times. The dashed lines in each plot are the model at that iteration and the "*" symbol indicates the minimizer of each model. The "o" symbols indicate the points at which the actual objective function has been evaluated.

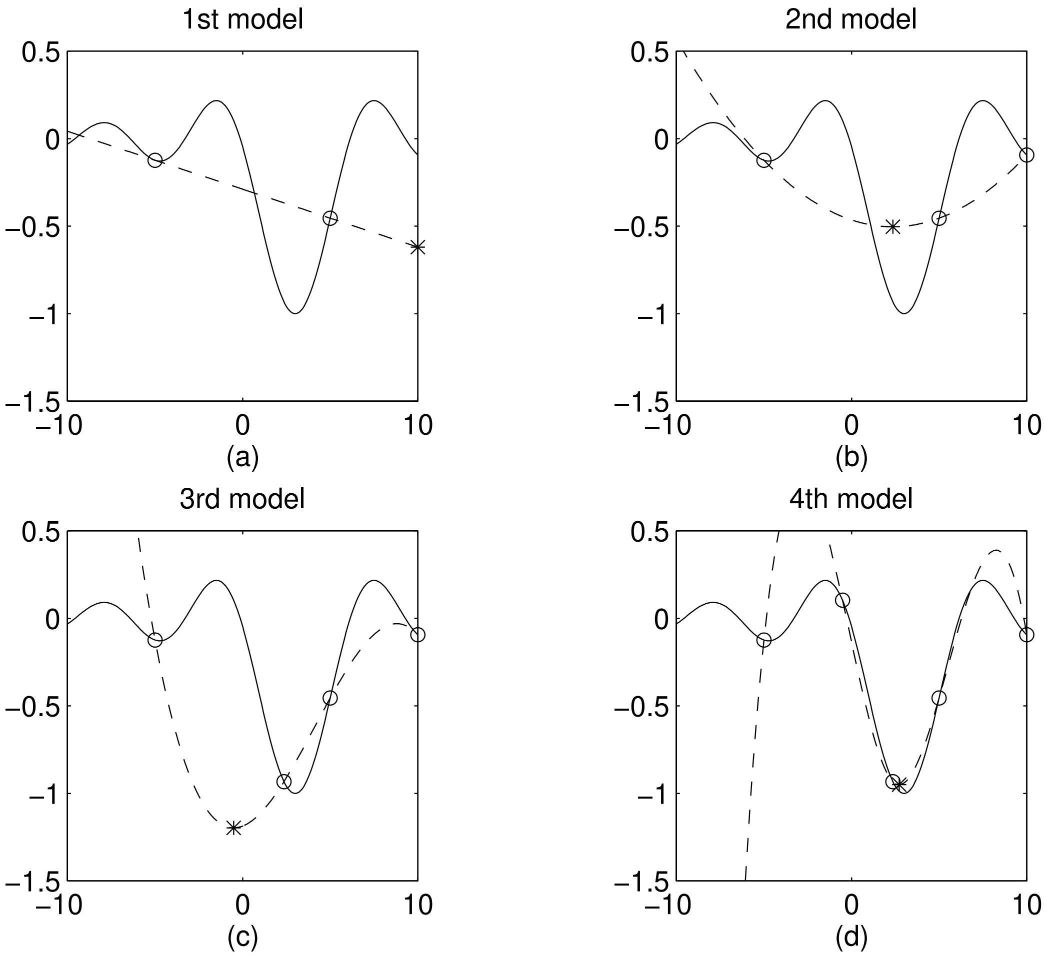


Figure 1.2 Plots of models of $F(x)$ at each model management iteration.

In the first iteration (Figure 1.2a), the model is linear and the minimizer is at the upper bound. The function value at $x = 10$ is not decreased, so the trial iterate would be rejected. The model in the second iteration (Figure 1.2b) is quadratic and its minimizer does produce decrease so it would be accepted. In fact, we get much more decrease than the model predicted (sometimes you get lucky). The next two iterations (Figures 1.2c,d) show the essence of the model management approach. The third model predicts decrease (the model minimum is less than the value at the current iterate) but is incorrect; the function does not decrease. However, the new model, the fourth, is much better than the third in the neighborhood of the current iterate. Note that it is much worse far away, in the region $x < 0$, but that part of the space is no longer of interest. The minimizer of the fourth model is very close to the minimizer of the actual function (2.717 and 3.0, respectively), and the function values are even closer (-0.987 and -1.0, respectively).

The total work included building the spline model four times, performing four separate nonlinear minimization operations on the models, and computing the actual function value six times. Half of the iterations found decrease in the actual objective function, and the model improved (for our purposes) at every iteration. Though it is likely we would not do this well on a real problem, we would hope to achieve similar ratios. The key is that the model improves our chances of finding a trial iterate that decreases the actual objective each time we evaluate it.

Also note that we found the global minimizer of the function even though we only used a local minimization method on the model. There is no guarantee this will always happen on real problems, although the model can increase the likelihood of this happening if it smooths out some of the local minima.

A two-dimensional problem using a more realistic algorithm is presented in §4.2.1. Figures 4.1–4.3 present 2D plots similar to Figures 1.1 and 1.2.

1.5 Zen and the art of model management

In the philosophy espoused by the Model Management Framework there are two levels of success that an algorithm should aspire to achieve. First, it must *work*. That is, it must compute the correct answer to the optimization problem (or at least as good an approximation to the solution as the user desires). Second, it must work *efficiently*. A correct answer is not worth anything if it comes too late to be used. The problems we are interested in are extremely expensive. Efficiency in practice will be the yardstick by which we measure the usefulness of algorithms. It is likely that in practice the accuracy of the solution may be traded off against performance, but there is a qualitative difference between a method that can compute a less accurate answer at lower cost but can always produce a more accurate answer at higher cost and a method that cannot be relied upon to produce a more accurate answer regardless of the cost.

Another aspect of the Zen of model management is that it is not required for the model to be an accurate approximation of the objective function. The management algorithm can be used to make up for inaccuracy in the model, albeit usually at the cost of additional computation. In fact, good model management algorithms will be designed with just that requirement in mind. Ideally, the accuracy of the model should increase as the solution is approached. The mantra of the model management algorithm designer should be: "do less work when the solution is less accurate." The converse has important implications as well. The more accuracy in the solution you want to get, the more work you have to do to get it.

Another aspect of the accuracy issue relates to local versus global behavior. If the model captures the global (general) behavior of the objective reasonably well but not the local (detailed) behavior, it may still do well in finding the basin of the global optimum of the actual objective. By removing small scale variations in the function, the model may allow the search (optimization) method to avoid local minima that it would find if acting directly on the actual objective, as in the example in §1.4. The recent work of C. T. Kelley on "implicit filtering" [41] successfully applies this same idea to the optimization of noisy objective functions using a gradient-based method.

In the common case of non-convex problems, the use of a model frees the algorithm designer to address the issue of global optimization to an extent not possible with an expensive objective function. This is not to say that a model management method will be able to find a global minimum just because the objective is cheap. The exponential growth in the size of the problem space with dimension precludes truly global methods in more than a few dimensions for most non-convex objective functions, regardless of cost. Despite this, it may be possible to design a model management algorithm to be less sensitive to local minimizers. We cite Kelley [41] and Trosset and Torczon [73] as examples.

The next chapter addresses the theoretical issues of the definition and convergence analysis of the model management framework. Subsequent chapters deal with the practical issues of algorithm design and software implementation and present numerical results of an example implementation on some test problems.

Chapter 2

Model Management Convergence Theory

2.1 Introduction

The convergence theory for the Model Management Framework depends on the convergence theory for pattern search optimization methods developed by Torczon in [72] and Lewis and Torczon in [48] for unconstrained problems and in [47] for bound constrained problems. Section 2.2 reviews and summarizes the results from these papers that will be used in this thesis. Section 2.3 shows how these results can be applied specifically to model management. Section 2.4 presents an abstract framework for model management algorithms that fits the pattern search theory and defines conditions under which algorithms expressed using this abstraction are provably convergent. Issues of algorithm design and implementation using the Model Management Framework will be addressed in Chapter 3.

The pattern search theory is appropriate for use with the model management approach proposed here because both are concerned only with objective function values, not derivatives. It is possible to construct a model management approach that uses derivative information. In [1], Alexandrov et al. use the trust region methodology to derive and prove convergence for such a method. Other derivative-based approaches to using models are discussed in [30] and [40], although no convergence theory is given.

2.2 Review of pattern search theory

This section summarizes material from §1–2 of [48] for unconstrained problems and §2–3 of [47] for bound constrained problems. The convergence theorems for unconstrained and bound constrained problems are similar, except that the bound constrained case imposes some additional conditions.

We consider here the unconstrained optimization problem:

$$\begin{array}{c} \text{minimize} \quad F(x) \\ x \in \mathbf{R}^n \end{array} \tag{2.1}$$

and the bound constrained optimization problem:

$$\begin{array}{c} \text{minimize} \quad F(x) \\ x \in \Omega \end{array} \tag{2.2}$$

where $\mathbf{R}$ is the set of real numbers, $F : \mathbf{R}^n \mapsto \mathbf{R}$ is the objective function, $x, l, u \in \mathbf{R}^n$, and the feasible region is $\Omega = \{x \in \mathbf{R}^n \mid l \leq x \leq u\}$. In most of the discussion we will treat both problems at the same time, noting explicitly when the additional conditions for bound constraints are relevant.

Pattern search methods are characterized by three specific features that distinguish them from the majority of optimization methods. Firstly, at each iteration the next iterate must be selected from a discrete set of points which is determined by means of a *pattern.* No other explicit restriction is placed on the magnitude of $\|x_{k+1} - x_k\|$. Secondly, the derivatives of the objective function F are not used explicitly in the method. Only function values are needed. Thirdly, each iterate must satisfy a *simple decrease* condition: if $x_{k+1} \neq x_k$ then $F(x_{k+1}) < F(x_k)$. No restriction is placed on the magnitude of $F(x_{k+1}) - F(x_k)$. That is, there is no explicit sufficient decrease condition.

In general terms, a pattern is a collection of steps in the problem space. Each step can be added to the current iterate to produce a different trial iterate. The orientation and scaling of the pattern can be adapted as the algorithm proceeds. An alternate view is to think of the pattern as containing vectors, each of which defines a direction and length from the current iterate to a possible trial iterate. The vectors may be scaled and rotated across iterations, but they are limited to a discrete set at any given iteration. The important concept is that all possible next iterates can be computed by combining the current iterate with a step from the pattern.

There are multiple ways to represent a pattern. In [47] and [72] a matrix with the steps as columns is used. In Section 2.3 we will represent the pattern as a set.

For convenience in defining and discussing pattern search algorithms we introduce an additional object, the pattern *scale factor*† that can vary at each iteration. The scale factor is a real, scalar multiplier applied to every step in the pattern before it is used to compute a trial iterate. This provides a convenient mechanism for uniformly controlling the length of all steps in the pattern. In any discussion where the scale factor is not explicitly mentioned it should be assumed that a step is taken from the scaled pattern.

The pattern matrix is denoted in [48] by $P_k \in \mathbf{R}^{n \times p}, p > n+1$, where k is the iteration counter in the pattern search algorithm. The scale factor is denoted by $\Delta_k \in \mathbf{R}$, so the scaled pattern at iteration k is $\Delta_k P_k$. Thus the set of possible next iterates is defined by $\{x_k + \Delta_k P_k e_i \mid i \in \{1, \cdots, p\}\}$, where e_i is the unit coordinate vector of the appropriate dimension. In the bound constrained case the next iterate is further restricted to the feasible points in this set.

It is intended that Δ_k approach zero as x_k converges. Separating this scaling from the pattern itself allows the algorithm to converge without requiring modifications to the pattern. This is convenient for both the theoretical discussion and the software implementation.

†This differs from the terminology in [72] and [47], where the equivalent scalar is referred to as the *step length control parameter*.

The theory for pattern search methods for unconstrained problems in [23, 48] is very general and applies to a variety of existing methods, some very old. Coordinate search with fixed step lengths [58] and the Hooke and Jeeves method [37] are examples of pattern searches. Other examples are described in [72]. The popular Nelder-Mead simplex method [55] is similar to a pattern search, but does not meet all the theoretical requirements and is not provably convergent in dimensions higher than one [44][51]. The extension for bound constrained problems in [47] adds relatively few additional restrictions.

The Generalized Pattern Search method (GPS) has the form outlined in Figure 2.1.

Generalized Pattern Search Method (GPS):
0) Given $x_0 \in \mathbf{R}^n$ or $x_0 \in \Omega$ (as appropriate), $P_0 \in \mathbf{R}^{n\times p}$ and $\Delta_0 \in \mathbf{R} > 0$.
1) for $k = 0,1,\ldots$
2) compute a step s_k using an *exploratory moves* algorithm
3) if $F(x_k + s_k) < F(x_k)$
4) $x_{k+1} \Leftarrow x_k + s_k$
5) else
6) $x_{k+1} \Leftarrow x_k$
7) *update* P_k and Δ_k (to produce P_{k+1} and Δ_{k+1})

Figure 2.1 Algorithm for the Generalized Pattern Search method (GPS)

The convergence theory specifies rules governing the contents of the pattern, the outcome of the exploratory moves, and how to modify P_k and Δ_k. Note that for the bound constrained problem, the pattern search is a *feasible point* method. The initial point x_0 and all subsequent iterates x_k are required to be feasible, as will be seen later. The bound constrained case has an additional requirement on the contents of the pattern that is not needed for unconstrained problems (see Hypothesis 1).

The *exploratory moves* algorithm referenced in Step 2 of Figure 2.1 can be any method for selecting one of the steps from the scaled pattern matrix $\Delta_k P_k$. The step is used to define the next trial iterate x_{k+1}. The pattern search convergence theory defines hypotheses on the result of the exploratory moves algorithm but does not restrict the implementation of the algorithm. In Section 2.3 we will decompose the exploratory moves into two separate sub-algorithms that we call *oracle* and *poll*, and redefine the hypotheses in terms of them. Any implementation of oracle and poll that can satisfy these hypotheses may be used as the exploratory moves algorithm in the GPS method. It is this flexibility that will enable us to build the model management convergence theory using the pattern search theory.

The Generalized Pattern Search method defined in Figure 2.1 is provably convergent, subject to conditions on P_k, Δ_k, and the result of the exploratory moves algorithm. These conditions are presented in [47] and [48] and are restated here in a slightly different form.

First we repeat some definitions from [48] that we need for the hypotheses. These definitions are taken from the theory of positive linear dependence [17]. The *positive span* of a set of vectors $\{a_1, \cdots, a_r\}$ is the cone

$$\{a \in \mathbf{R}^n \mid a = c_1 a_1 + \cdots + c_r a_r, c_i \geq 0 \ \ \forall i\}.$$

The set $\{a_1, \cdots, a_r\}$ is called *positively dependent* if one of the a_i's is a nonnegative combination of the others; otherwise the set is *positively independent.* A *positive basis* is a positively independent set whose positive span is $\mathbf{R}^n$. A positive basis must contain at least $n+1$ vectors and may contain no more than $2n$ vectors. Figure 2.2 shows two examples of sets of positively independent vectors in $\mathbf{R}^2$. The set in Figure 2.2(a) is not a positive basis, whereas the set in Figure 2.2(b) is. The difference is that no direction in the negative x_2 half-plane can be formed by a positive combination of the vectors in Figure 2.2(a).

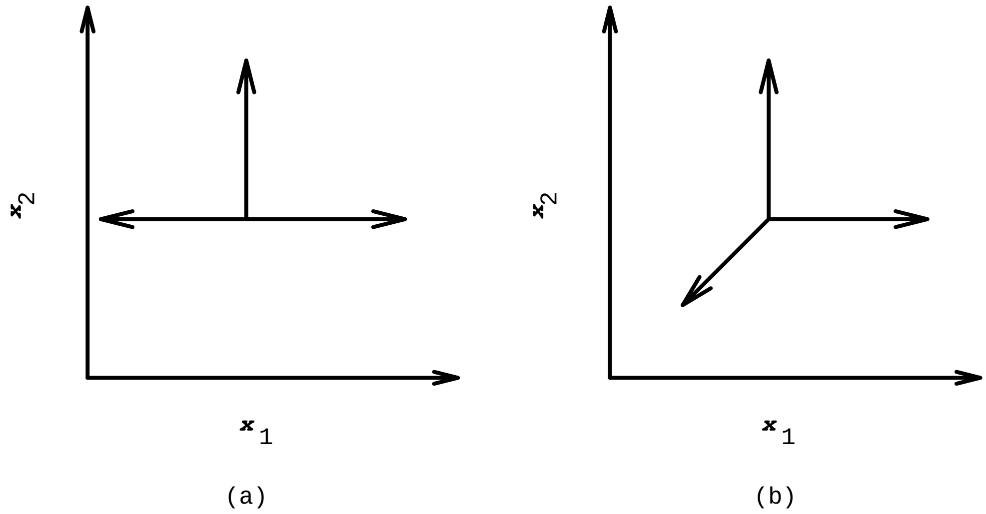


Figure 2.2 Examples of (a) positively independent set and (b) positive basis in $\mathbf{R}^2$.

Hypothesis 1 The pattern matrix

$$P_k \in \mathbf{R}^{n \times (g_k + l_k + 1)}$$

and the core matrix

$$C_k \in \mathbf{R}^{n \times (g_k + 1)},$$

with $g_k > n, l_k \geq 0$, must satisfy:

$$\begin{aligned} P_k &= [B\Gamma_k \vdots BL_k \vdots 0] \\ C_k &= [B\Gamma_k \vdots 0] \end{aligned}$$

where $B \in \mathbf{R}^{n \times n}$ is nonsingular. The matrix $\Gamma_k \in \mathbf{G} \subset \mathbf{Z}^{n \times g_k}$ and Γ_k is a positive basis of $\mathbf{R}^n$. The cardinality of the set $\mathbf{G}$ is finite.‡ The number of columns in the matrix $L_k \in \mathbf{Z}^{n \times l_k}$ need not be finite. We use 0 to denote a single column of zeros.

‡This use of C_k and $\mathbf{G}$ differs from that in [47] and [72].

Additionally, in the bound constrained case (2.2), $B\Gamma_k$ must be a maximal positive basis, that is, it must contain $2n$ orthogonal vectors.

Note that the restriction that $g_k > n$ comes from the definition of a positive basis.

Hypothesis 2 The result of the exploratory moves s_k must satisfy:

$$s_k = \Delta_k P_k e_i, \quad i \in \{1, \ldots, (g_k + l_k + 1)\}$$

where e_i is the unit coordinate vector.

In the unconstrained case s_k must also satisfy:

$$\text{if} \quad \min_{i \in \{1,\ldots,g_k\}} \{F(x_k + \Delta_k C_k e_i)\} < F(x_k), \text{ then } F(x_k + s_k) < F(x_k).$$

In the bound constrained case x_k will be feasible so s_k must satisfy:

$$x_k + s_k \in \Omega$$

to ensure x_{k+1} is also feasible and s_k must also satisfy:

$$\text{if} \quad \min_{\substack{i \in \{1,\ldots,g_k+1\} \\ x_k + \Delta_k C_k e_i \in \Omega}} \{F(x_k + \Delta_k C_k e_i)\} < F(x_k), \text{ then } F(x_k + s_k) < F(x_k).$$

Hypothesis 3 Let $\mathbf{Q}$ be the set of rational numbers and $\mathbf{Z}$ the set of integers, $\tau \in \mathbf{Q}$, $\tau > 1$, $\{w_0, w_1, \ldots, w_L\} \subset \mathbf{Z}$, $w_0 < 0$, $w_i \geq 0, i = 1, \ldots, L$, and $\Lambda = \{\tau^{w_1}, \ldots, \tau^{w_L}\}$. Given $\Delta_0 > 0$, $\theta = \tau^{w_0}$ and $\lambda_k \in \Lambda$, at each iteration k the scale factor Δ_k must satisfy:

$$\begin{aligned} \text{if} \quad F(x_k + s_k) \geq F(x_k) \quad \text{then} \quad & \Delta_{k+1} = \theta \Delta_k \\ \text{else} \quad & \Delta_{k+1} = \lambda_k \Delta_k \end{aligned}$$

In essence, what the hypotheses say is this: the pattern and the scale factor define a set of points around and including the current iterate. A subset of the steps in the pattern, when interpreted as vectors based at the current iterate, must contain a positive basis as defined above. These steps and the zero step form the *core* pattern $\mathcal{C}_k$. The positive basis provides the coverage of the space that guarantees that in the limit, as Δ_k converges to zero, the core pattern will capture descent in the function if the current iterate x_k is not a stationary point.

The exploratory moves algorithm must choose a step from the pattern which decreases the objective or is zero, but it must not be zero if **any** non-zero step in the core pattern gives simple decrease. The step need not be in the core pattern, and it need not have as much decrease as the steps in the core pattern, but it must have **some** decrease. The consequence of this is important in practice: an algorithm is allowed to stop searching as soon as a step with simple decrease is found. If no step with decrease is found and there is no step in the core pattern with decrease, the scale factor Δ_k *must* be reduced by a constant factor θ before the next iteration. The finiteness on the number of unique core patterns guarantees that the maximum size (in some appropriate norm) of the core patterns is bounded. Without this limitation, the core pattern could grow as rapidly as the pattern scale factor contracts, leaving the lengths of the core steps unchanged, which could lead to a failure to converge.

The Model Management Framework discussed later will be defined as an exploratory moves algorithm where the trial step and perhaps the pattern are determined using a model of the objective function rather than by necessarily evaluating the objective at the steps in some fixed pattern.

Notice that the hypotheses fall into two categories: requirements on the parameters (P_k, Δ_k, θ and λ_k) and requirements on the behavior of the algorithm (i.e. on the intermediate results, s_k and x_k). The only part of the method that is specified completely is the test to determine if a step is acceptable (GPS steps 3–6); the other parts of the method are specified only in terms of the results they produce. This approach of specifying the output without specifying the implementation will be used later in developing the convergence theory for the Model Management Framework.

Using the new hypotheses we can merge and restate Theorem 2.1 from [48] for the unconstrained case with Theorem 3.2 from [47] for the bound constrained case.

To state the theorem for the bound constrained case we introduce the following notation:

$$p_j(t) = \begin{cases} l_j & \text{if } t < l_j \\ t & \text{if } l_j \le t \le u_j \\ u_j & \text{if } t > u_j \end{cases},$$

$$P(x) = \sum_{j=1}^{n} p_j(x_j) e_j,$$

and

$$q(x) = P(x - \nabla F(x)) - x, \tag{2.3}$$

where $p_j(t)$ is the projection of x onto the feasible region Ω and $q(x)$ is a measure of the closeness of x to a constrained stationary point. Thus x is a stationary point for problem (2.2) if and only if $q(x) = 0$.

The proof of the convergence theorem is presented in [48] for the unconstrained case and in [47] for the bound constrained case. The proof is based on technical lemmas whose proofs are too long to reproduce here. Rather, we state the necessary supporting results without proof and use them to prove the convergence theorem.

The first lemma guarantees that some portion of the descent direction in the objective function at any non-stationary point x_k can be captured in the core pattern by reducing the scale factor Δ_k.

Lemma 2.1 (Proposition 2.3 from [48]) There exists a constant $c > 0$ and an iteration k of a generalized pattern search (GPS) method such that for any $\eta > 0$ there is a nonzero core step $s \in \{\Delta_k \mathcal{C}_k e_i, i \in \{1, \ldots, (g_k + 1)\}\}$ that satisfies

$$-\nabla F(x_k)^T s \geq c \|\nabla F(x_k)\| \|s\|$$

The second lemma guarantees that if there is a descent direction of the objective function at the current iterate, then the GPS method will eventually find a step with decrease.

Lemma 2.2 (Proposition 2.15 from [48]) Assume $L(x_0)$ is compact and that F is continuously differentiable on an open neighborhood of $L(x_0)$. Then for any $\eta > 0$ there exists $\delta > 0$, independent of k, such that if $\Delta_k < \delta$ and $\|\nabla F(x_k)\| > \eta$, then the GPS method computes a nonzero step s_k that satisfies $F(x_k + s_k) < F(x_k)$.

The previous two lemmas are used to prove the next two lemmas, which are used to prove the convergence theorem.

Lemma 2.2 says that if the gradient at x_k is nonzero then the scale factor Δ_k will eventually be small enough that the GPS method will find a step that decreases the objective. The next lemma says that if this happens the scale factor will not converge to zero.

Lemma 2.3 (Theorem 2.16 from [48]) Suppose that $L(x_0)$ is compact, that f is continuously differentiable on a neighborhood of $L(x_0)$, and that $\liminf_{k \to \infty} \|\nabla F(x_k)\| \neq 0$. Then there exists a constant $\Delta_* > 0$ such that for all k, $\Delta_k > \Delta_*$.

The next lemma guarantees that the scale factor in the GPS method is not bounded away from zero. Recall that we defined the GPS method (Hypothesis 3) so that Δ_k must be positive.

Lemma 2.4 (Theorem 2.18 from [48]) Suppose that $L(x_0)$ is compact. Then

$$\liminf_{k\to\infty} \Delta_k = 0.$$

We now use these results to prove convergence of the generalized pattern search method.

Theorem 2.1 Consider a generalized pattern search method of the form in Figure 2.1 that satisfies Hypotheses 1, 2 and 3. Let $q(x)$ be defined as in Equation 2.3. Assume that the level set $L(x_0)$ is compact and that F is continuously differentiable on an open neighborhood of $L(x_0)$. Let $\{x_k\}$ be the sequence of iterates produced by a pattern search method. Then, in the unconstrained case,

$$\liminf_{k\to\infty} \|\nabla F(x_k)\| = 0$$

and in the bound constrained case,

$$\liminf_{k\to\infty} \|q(x_k)\| = 0.$$

Proof (unconstrained case) The proof is by contradiction. Suppose that $\liminf_{k\to\infty} \|\nabla F(x_k)\| \neq 0$. Then Lemma 2.3 says there exists $\Delta_* > 0$ such that for all k, $\Delta_k \geq \Delta_*$. But this contradicts Lemma 2.4, and so

$$\liminf_{k\to\infty} \|\nabla F(x_k)\| = 0.$$

□

The proof for the bound constrained case follows from similar arguments, with the additional restriction that the core pattern must be a maximal positive basis to guarantee that a solution on the boundary will be found. The proof is presented in [47].

2.3 Pattern search theory applied to model management

The convergence theory for pattern search methods reviewed in the previous section relies on two fundamental concepts. The first is that the pattern must contain a set of steps (the *core*) which forms a positive basis for $\mathbf{R}^n$ (a maximal positive basis in the bound constrained case) and the steps in the core must be exhaustively searched before the trial step can be set to zero, thus forcing the pattern scale factor Δ_k to be reduced. If the current iterate is not a stationary point of the objective, then the positive basis is guaranteed to define at least one direction of descent from the current iterate. Asymptotically, as the scale factor shrinks, some step in the scaled core pattern will capture a nonzero fraction of the steepest descent direction, thus ensuring the search will find a step with decrease in the objective function (Proposition 2.3 in [48]). This prevents the method from converging to a point that is not a critical point by satisfying an analog of a *fraction of Cauchy decrease* condition without explicitly computing the gradient. The second concept is that the scale factor may be reduced only when the trial step is zero, and it must be reduced by a constant rational factor (see Hypothesis 3). This guarantees that the scale factor does not converge to zero prematurely. This provides an analog of the Armijo-Goldstein-Wolfe condition. The proof of Theorem 2.1 relies heavily on these two concepts.

In this section we specialize the pattern search theory in two distinct ways. First we choose a particular class of patterns with an infinite number of steps. After making some changes in notation, we restate the hypotheses and the convergence theorem. Second, we define an approach to designing exploratory moves algorithms tailored specifically to model management strategies. In the next section we define conditions on the model management algorithms and prove convergence.

2.3.1 Patterns for model management

In using the pattern search theory to develop model management algorithms it will be convenient to make some straightforward changes in notation. These changes are motivated by the desire to exploit two particular aspects of the theory. First, the exploratory moves algorithm is not required to evaluate the objective function at all the steps in the pattern in order to find a step that produces decrease. Hypotheses 1 and 2 define conditions on the result of the exploratory moves, not on the implementation. Second, the pattern is not restricted to a finite number of steps. The notational changes will not require any substantive changes to the proof of Theorem 2.1.

We will restate Hypotheses 1 and 2 using the new notation and show that the convergence theorem (Theorem 2.1) still holds. There is no need to restate Hypothesis 3, which places conditions on the update of the scale factor Δ_k, as it is unaffected by these changes.

We define a pattern for use in the Generalized Pattern Search method (Figure 2.1) by taking a real nonsingular basis matrix $B \in \mathbf{R}^{n \times n}$ multiplied by all possible integer vectors in $\mathbf{Z}^n$. We represent this pattern by the set $\mathcal{S} = \{Bz \mid z \in \mathbf{Z}^n\}$. This pattern defines a regular lattice. Note that Hypothesis 1 requires that the size of the pattern be at least $n + 1$ ($2n$ in the bound constrained case). We want to make explicit the fact that the pattern may contain an infinite number of steps, hence our definition of the set $\mathcal{S}$. The lattice points, when scaled by Δ_k, define the set of potential trial steps which, when added to the current iterate x_k, generate the potential trial iterates.

In a similar fashion let us define a set $\mathcal{C}_k$ which contains the steps in the core pattern matrix C_k defined in Hypothesis 1. By definition, the steps in the core matrix C_k are contained in the pattern matrix P_k and by construction the steps in the pattern matrix are contained in the set $\mathcal{S}$, so the set $\mathcal{C}_k$ is a strict subset of $\mathcal{S}$. The definition of positive basis implies that the core pattern must contain between

$n+1$ and $2n$ nonzero steps. Hypothesis 1 requires that is also contain the zero step. Note that the steps in the core pattern may change across iterations (subject to the requirements of Hypothesis 1) but every possible core pattern is contained in $\mathcal{S}$.

We use script letters $\mathcal{S}, \mathcal{C}, \mathcal{G}$ to denote sets that correspond to the matrices in the pattern search theory denoted by capital Roman and Greek letters. Using this notation, Hypothesis 1 can be restated as follows:

Hypothesis 4 The set of possible trial steps $\mathcal{S}$ and the set of core steps $\mathcal{C}_k$ must satisfy:

$$\begin{aligned} \mathcal{S} &= \{Bz \mid z \in \mathbf{Z}^n\} \\ \mathcal{C}_k &= \{Bg \mid g \in \{\mathcal{G}_k \cup \{0\}\}\} \subset \mathcal{S} \end{aligned}$$

where $B \in \mathbf{R}^{n \times n}$ is nonsingular, $\mathcal{G}_k \in \widehat{\mathcal{G}}$, $\mathcal{G}_k$ is a positive basis of $\mathbf{R}^n$, $\widehat{\mathcal{G}}$ is a finite set of sets such that $\widehat{\mathcal{G}} = \{\mathcal{G}_i \mid \mathcal{G}_i \text{ is a positive basis of } \mathbf{Z}^n, i = 1, \ldots, q, q < \infty\}$, and $\{0\}$ is the set containing the zero vector in $\mathbf{R}^n$.

It should be reiterated that the set $\mathcal{S}$ contains all steps that may be contained in any possible pattern matrix P_k because each column of P_k is composed of the product of B and an integer vector (see Hypothesis 1). So, clearly, once a basis B is chosen, all columns of P_k are elements of $\mathcal{S}$. By similar arguments, at iteration k of GPS the set of core steps $\mathcal{C}_k$ can contain the same steps contained in the core matrix C_k.

Using set notation to represent a pattern of infinite size, the hypothesis on the result of the exploratory moves (Hypothesis 2) can be restated as follows:

Hypothesis 5 An exploratory moves algorithm must satisfy:

$$s_k \in \Delta_k \mathcal{S}.$$

In the unconstrained case (2.1) it must satisfy:

$$\text{if} \min_{y \in \Delta_k \mathcal{C}_k} \{F(x_k + y)\} < F(x_k) \text{ then } F(x_k + s_k) < F(x_k).$$

In the bound constrained case (2.2) it also must satisfy feasibility:

$$x_k + s_k \in \Omega$$

$$\text{if} \min_{\substack{y \in \Delta_k \mathcal{C}_k \\ (x_k + y) \in \Omega}} \{F(x_k + y)\} < F(x_k) \text{ then } F(x_k + s_k) < F(x_k).$$

Hypothesis 5 differs from Hypothesis 2 only in notation and in the specification of the fixed infinite set $\mathcal{S}$ of potential trial steps rather than the possibly varying pattern P_k. Only one trial step s_k is ultimately selected in each iteration of the pattern search method, and a pattern matrix P_k can always be defined that contains s_k, so the behavior of an exploratory moves algorithm under Hypothesis 5 can be achieved by an exploratory moves algorithm under Hypothesis 2. Similarly, since any possible P_k is contained in $\mathcal{S}$ (because $\mathcal{S}$ contains all possible trial steps), the behavior of an exploratory moves algorithm under Hypothesis 2 can be achieved by an exploratory moves algorithm under Hypothesis 5. Thus, the restated hypotheses are functionally equivalent to the original pattern search hypotheses and so Theorem 2.1 still applies. We restate it now using the new Hypotheses 4 and 5:

Theorem 2.2 Given a pattern search method whose steps satisfy Hypotheses 3, 4 and 5, assume that the level set $L(x_0)$ is compact and that F is continuously differentiable on an open neighborhood of $L(x_0)$. Let $q(x)$ be defined as in Equation 2.3. Then, in the unconstrained case, for the sequence of iterates $\{x_k\}$ produced by such a pattern search method,

$$\liminf_{k\to\infty} \|\nabla F(x_k)\| = 0,$$

and in the bound constrained case,

$$\liminf_{k\to\infty} \|q(x_k)\| = 0.$$

Proof: Hypothesis 4 states that the sets $\mathcal{S}$, $\mathcal{C}_k$ and $\mathcal{G}_k$ are equivalent to the matrices P_k, $[B\Gamma_k \vdots 0]$ and Γ_k, respectively. With these substitutions, Hypothesis 5 is equivalent to Hypothesis 2. Hypothesis 3 is unchanged. Thus the hypotheses of Theorem 2.2 are equivalent to those of Theorem 2.1 and the proofs of Theorem 2.1 and the supporting lemmas hold without change. □

In [48] a stronger result is derived that achieves $\lim_{k\to\infty} \|\nabla F(x_k)\| = 0$ rather than the lim inf. One of the additional hypotheses for the stronger result is that the columns of the pattern matrix P_k are bounded in norm. Unfortunately, this condition can not be met under Hypothesis 4 because the set $\mathcal{S}$ is unbounded by definition.

2.3.2 Decomposition of the exploratory moves for model management

Given the definition of the behavior of an exploratory moves algorithm in Hypothesis 5, the algorithm can be decomposed easily into two separate phases. Figure 2.3 presents an outline of an exploratory moves algorithm showing this decomposition. The first phase searches the set of possible trial steps (not necessarily exhaustively) and either finds one that produces decrease in the objective function or returns the zero step. This corresponds to line 1 of Figure 2.3. The second phase is only used when the first returns the zero step. It corresponds to lines 2–3 of Figure 2.3. It searches the steps in the core pattern (exhaustively in the worst case) looking for one that decreases the objective. It may return the zero step only if *none* of the steps in the core produce decrease. This guarantees that the second part of Hypothesis 5 is satisfied. In fact, the second phase alone is sufficient to satisfy all the requirements of Hypothesis 5; just evaluating the objective at the core steps is a complete exploratory moves algorithm. This decomposition will be used later in the development of the Model Management Framework.

---

0) Given a current iterate x_k, an objective function F, and $\Delta_k and \mathcal{S}$ as defined in Hypotheses 3 and 4
1) select a step s_k from $\Delta_k \mathcal{S}$ that produces decrease in F or is zero
2) if ($s_k = 0$)
3) reset s_k to a step from the core pattern $\Delta_k \mathcal{C}_k$ that produces decrease in F, if there is one
4) return s_k

Figure 2.3 Outline of an exploratory moves algorithm

In the rest of this thesis the first phase will be referred to as the "oracle", and the second as the "poll", or (in the context of model management) as "polling truth".

The term oracle is used in some parts of the statistics and operations research communities to mean a function that produces observations or responses. This harkens back to the classic Greek mythological meaning of oracle. In our case, the oracle is asked which pattern point to look at next and responds with a step from the pattern. The analogy to the mythic oracle is especially appropriate to the model management context because of the implication that the oracle has knowledge that the questioner does not have, which in this context will be the knowledge of the model. The oracle may use any strategy for finding the next step that fits the outline in Figure 2.3. In fact, the convergence theory requires no knowledge of the model, nor even that a model exists. The restriction of the trial steps to the discrete grid $\Delta_k \mathcal{S}$ and the condition enforced in Steps 2–3 of Figure 2.3 are sufficient to guarantee convergence.

The term poll is meant to emphasize the idea that the evaluation of the actual objective function at the core steps is a sampling operation, like taking a poll. The goal of the poll is to determine if there is any decrease to be found in the pattern at the current scale factor. We call the actual function "truth" even though it is often an approximation of the "real" (physical) system being optimized because it provides the most accurate representation of the real system to which we will have access.

There are three important characteristics of this particular decomposition of exploratory moves algorithms: the oracle is free to use any method to choose a trial step from anywhere in the feasible region, as long as it is in the pattern; the poll cannot search outside the core; if the oracle succeeds in finding a step that decreases the objective there is no need to poll.

The decomposition of exploratory moves into oracle and poll is formalized in the following hypotheses and lemma.

Hypothesis 6 The result s_+ of an oracle algorithm must satisfy:

$$s_+ \in \Delta_k \mathcal{S}.$$

$$\text{if } s_+ \neq 0 \text{ then } F(x_k + s_+) < F(x_k).$$

In the bound constrained case it must also satisfy feasibility:

$$x_k + s_+ \in \Omega.$$

Hypothesis 7 Given s_+ satisfying Hypothesis 6, the result s_k of a poll algorithm must satisfy:

$$\begin{array}{lll} \text{if } s_+ \neq 0 & \text{then} & s_k = s_+, \\ \text{if } s_+ = 0 & \text{then} & s_k \in \Delta_k \mathcal{C}_k \\ & & \quad \textbf{and} \\ & & s_k = 0 \ \textbf{ or } \ F(x_k + s_k) < F(x_k), \end{array}$$

subject to, in the unconstrained case,

$$\text{if} \min_{y \in \Delta_k \mathcal{C}_k} \{F(x_k + y)\} < F(x_k) \text{ then } s_k \neq 0,$$

or, in the bound constrained case,

$$x_k + s_k \in \Omega$$

and

$$\text{if} \min_{\substack{y \in \Delta_k \mathcal{C}_k \\ (x_k + y) \in \Omega}} \{F(x_k + y)\} < F(x_k) \text{ then } s_k \neq 0.$$

Lemma 2.5 Any combination of an oracle algorithm satisfying Hypothesis 6 followed by a poll algorithm satisfying Hypothesis 7 also satisfies Hypothesis 5.

Proof: In the unconstrained case, Hypothesis 6 guarantees that $s_+ \in \Delta_k \mathcal{S}$ and Hypothesis 7 guarantees that $s_k = s_+$ or $s_k \in \Delta_k \mathcal{C}_k$ or $s_k = 0$. Hypothesis 4 guarantees

$$\Delta_k \mathcal{C}_k \subset \Delta_k \mathcal{S}$$

and

$$0 \in \Delta_k \mathcal{C}_k$$

so it follows that

$$s_k \in \Delta_k \mathcal{S}$$

and the first condition of Hypothesis 5 is satisfied.

In the case where $s_+ \neq 0$, we have from Hypothesis 6 that

$$F(x_k + s_+) < F(x_k)$$

and from Hypothesis 7 that

$$s_k = s_+$$

so

$$F(x_k + s_k) < F(x_k)$$

and the second condition of Hypothesis 5 is satisfied.

In the case where $s_+ = 0$, the simple decrease requirement in Hypothesis 7 is identical to Hypothesis 5, so again the latter is satisfied.

The bound constrained case follows from similar arguments. □

As stated previously, a poll algorithm by itself satisfies the requirements for an exploratory moves algorithm, but in combination with an oracle it gains tremendous flexibility (and, one hopes, efficiency). This flexibility stems from the use of the infinite set of potential trial steps and the way the hypotheses are constructed. This allows any method to be used for the oracle, not just one based on sampling the function at some predetermined steps in a finite pattern, as is the usual implementation strategy for pattern search methods. This is discussed in [21]. This is critical in the development of the Model Management Framework, which will be presented in the next section.

In practice, the efficiency of the oracle method will likely depend on a variety of application-specific factors. This is totally independent of the convergence theory and its requirements. The theory provides guidelines for the development of methods, but cannot guarantee efficiency.

In contrast to the oracle, the cost of the poll method can be defined precisely. It has to be at least the cost of one objective function evaluation (in the best case of finding decrease on the first try) and not more than $2n$ function evaluations (in the worst case of not finding decrease in any step in the core pattern). The worst case is determined by the number of steps in the core, which is limited to $2n$ by the size of the maximal positive basis.[§] By using a minimal positive basis in the core pattern, the worst case cost is reduced to $n+1$ function evaluations. However a fixed minimal positive basis may be used only for unconstrained problems. Constrained problems require at least some use of a larger basis. In practice, the key efficiency issue will be how often the poll algorithm must be executed. As in the case of the oracle, the theory provides no assistance in predicting this behavior in practice.

[§]The pattern search theory does not prohibit core patterns larger than $2n$, but there is no theoretical advantage to using one.

2.4 The Model Management Framework as an exploratory moves algorithm

In this section we propose a general-purpose framework for model management algorithms, called the Model Management Framework, and analyze its convergence properties. The Framework provides an abstraction of an exploratory moves algorithms that is tailored to the use of models and model management concepts. The analysis follows that of the Generalized Pattern Search method given in Section 2.3. The Framework is shown to satisfy the requirements of an exploratory moves algorithm in GPS.

The Framework defines an exploratory moves algorithm in terms of several simpler sub-algorithms, much as the Generalized Pattern Search method (Figure 2.1) is defined in terms of the exploratory moves and update algorithms. For convenience, these sub-algorithms are called "component algorithms." By specifying conditions on the behavior of the component algorithms, the behavior of the whole Framework can be analyzed and convergence proven. It is not necessary to specify the precise implementations of the components in the convergence theory.

First we discuss the use of models and model management strategies in pattern search methods. Next we extend the decomposition of the exploratory moves algorithm developed in Section 2.3.2 to support the development of algorithms that use models and model management. We use this decomposition to define the Model Management Framework and specify rigorously the behavior of the component algorithms and prove convergence of a form of the GPS method that uses the Model Management Framework.

2.4.1 Use of models in pattern search methods

The generalized pattern search method GPS in Figure 2.1 can be divided into three independent parts: the exploratory moves; the decision whether to accept the trial step s_k; and the update of P_k and Δ_k. As discussed earlier, the convergence theory defines the computed result s_k of the exploratory moves but intentionally leaves undefined how that computation is carried out. On the other hand, the acceptance criteria in the second part is defined very precisely. There is no room for interpretation in steps 3–6 of GPS. The third part allows for some flexibility, but less so in our specialization of the original hypotheses since extending the pattern to a countably infinite number of steps eliminates the usefulness of modifying the pattern across iterations. However, the possibility of modifying either the scale factor or the core pattern remains.

The theoretical conditions on the result of the exploratory moves do not require that the objective function be evaluated at any of the steps in the pattern except for the result step. A straightforward approach then is to use a model of the objective function as a surrogate objective in searching the pattern steps. Most of the high cost of the evaluations of the expensive objective function at the steps in the pattern can be avoided. As in any method that uses models, the difficulty arises in handling the cases where the model result does not satisfy the theoretical requirements (Hypotheses 4 and 5): either the actual objective is not reduced or the step is not in the pattern. We will assume for the present that the second case can be rectified easily. Our major concern is the first case. The simplicity of the conditions allow for a wide variety of strategies for handling this case.

An advantage of the pattern search approach for problems with expensive functions is that it guarantees that the intermediate solutions behave monotonically: the method never accepts an intermediate solution that is worse than any previous solution. This behavior is extremely desirable when each iteration is very expensive and any improvement in the solution, however small, is worth keeping.

The decomposition of the exploratory moves into oracle and poll components was motivated by the cost of evaluating the objective function. This decomposition separates the exploratory moves into two distinct phases: a potentially cheap phase (oracle) that may use whatever means are appropriate to try to find a trial step which produces decrease; and a potentially expensive phase (poll) that is constrained by the theory to search the core steps (possibly exhaustively).

Recall that Hypothesis 6 requires that if the step computed by the oracle is nonzero it must reduce the (actual) objective function. If the step is zero the poll must be executed. So clearly, whatever model management strategy is used in the oracle, if it fails then the poll will ensure that the exploratory moves conditions are met. This approach satisfies the hypotheses on the pattern search theory and therefore is convergent.

To be worthwhile, a model management strategy must also reduce the overall cost of getting a solution (compared to a more traditional pattern search method that does not use an explicit model or any additional knowledge of the function). A perfect model management algorithm would require only one evaluation of the actual objective function per iteration compared to a typical pattern search method that evaluates the objective at some or all of the points in the pattern.

The general approach that the Framework is meant to describe is an oracle algorithm that uses a model to search the pattern and has some strategy for dealing with the case when the actual objective does not decrease. If the strategy fails the oracle can just return a zero step. Clearly almost any strategy can be made to fit this paradigm. The efficiency of this approach will depend on how well the model predicts the decrease in the actual objective function, the effectiveness of the method used in searching the pattern, and the effectiveness of the failure-recovery strategy (how well the model is managed).

An obvious approach for searching the pattern is to apply a nonlinear optimization method to the model of the objective function. This approach reduces the original expensive optimization problem to a cheaper optimization problem. The reduced problem may be simpler as well as cheaper if the optimization method applied to the model can take advantage of some characteristics of the particular model used. For example, if the derivatives of the model are easily computed, a gradient-based optimization method may be used. This approach of reducing a difficult problem to a simpler one is at the heart of many successful optimization methods. Trust region, sequential linear programming (SLP) and sequential quadratic programming (SQP) methods are good examples [22],[29].

A key feature of any practical use of models in pattern search methods will be how the modifications to the model are handled. We assume that a static model is not adequate for our purposes. This is based on the argument that it is not cost-effective to build a model that is of high accuracy everywhere in the problem space. So unless there is *a priori* knowledge of where a solution lies, the model would have to be constructed to be uniformly accurate everywhere. Since it must be fairly accurate near a solution to get an accurate answer, the *a priori* accuracy would have to be high. Inevitably much of the work in building such a model would go to waste because the accuracy ultimately would be needed only in the part of the space containing a solution. This is addressed in more detail in [73].

2.4.2 Extending the decomposition of the exploratory moves

The previous section presented some general ideas about the use of models and model management for exploratory moves algorithms. Section 2.3.2 presented a simple decomposition of the exploratory moves algorithm. In this section we use the ideas of the previous section and extend the decomposition of Section 2.3.2 to explicitly include the model management concept.

The theoretical conditions on the result of the exploratory moves led to the initial decomposition of the exploratory moves algorithm into oracle and poll components. Hypotheses 6 and 7 define this decomposition. The pattern scale factor may be reduced (an operation required for convergence) only if there is no step in the core pattern that decreases the objective function. The simplest way to satisfy this requirement is to evaluate the objective function at every step in the core and compare the function values. The definition of the poll algorithm follows directly. The definition of the oracle algorithm follows from the observation that Hypothesis 5 allows any step from the pattern to be chosen as the next iterate as long as it decreases the objective.

The conditions on the result of poll are precise enough that there is no benefit to be gained by decomposing the poll algorithm any further. No practical method for enforcing the conditions of Hypothesis 7 except evaluating the function at all the steps in the core pattern has been encountered. There are variations on this implementation, but they are relatively minor.

Most of the algorithmic flexibility in the exploratory moves (in fact, most of the algorithmic flexibility in the Generalized Pattern Search method) is in the oracle algorithm. The goal of the oracle is to find a (feasible) step in the pattern that reduces the objective function. As discussed in the previous section, it is in this search process that a model of the objective function can be used as a surrogate for the objective.

This is also where the model management strategy first comes into play. Simply substituting the model for the actual objective in the search process is not sufficient to guarantee that the result will satisfy the hypotheses. It is necessary to check the value of the actual objective function at the resulting point to make sure it decreases. If it does not, there are two options: give up and return the zero step or try again. In the latter case, the search can be repeated with a different algorithm, or the model can be changed and the search repeated with the same algorithm, or both the algorithm and the model can be changed.¶ A "different" search algorithm would be one that eliminates the previously computed step and returns the next best step in the pattern. It does not have to be a completely different method.

The oracle algorithm is decomposed into four component algorithms. The first is the search using the model. To increase flexibility we allow the search to produce several trial steps rather than just one. The next component evaluates the trial steps to determine if one produces decrease in the actual objective function. The third component modifies the model. This definition is necessarily vague because there is little commonality among the many different types of models that might be used. Iterating on these three components generates the need for the fourth: a component to decide when to stop the iteration. Any fixed method for terminating the loop would not be flexible enough to allow for all possible algorithm choices for the other components.

It would be possible to eliminate the fourth component by not iterating over the other three. In this approach, if none of the results computed by the search algorithm produce decrease in the actual objective the oracle is forced to return the zero step and the poll algorithm is invoked to satisfy the convergence theory. This is a workable approach but not ideal with regard to efficiency. The cost of the poll is known to be large in the worst case. It is desirable to balance the cost of the oracle against

¶This assumes the search algorithm and the model are deterministic: if not, it may be possible to run the same algorithm with the same model and get a different result. Nothing in the theory prevents this.

the cost of the poll. The oracle may be very inexpensive to execute, and it would be inefficient to have to invoke an expensive poll because an inexpensive search failed when it might be possible to make a change to the model or search strategy that would succeed without executing the poll.

It should be noted that the decomposition described here is not the only one that is possible. However it is believed to be flexible enough to express a wide variety of search, modeling, and management approaches.

2.4.3 The Model Management Framework definition

The Model Management Framework implements an exploratory moves algorithm as defined by the pattern search theory. In this section we give a precise definition of the Framework as a generic algorithm that uses the decomposition of the exploratory moves given in the previous section. This generic algorithm uses several component algorithms. Conditions on the results of the components are presented. This approach to algorithm definition mimics that used for the Generalized Pattern Search method. In the next section the Framework is shown to satisfy the requirements of the pattern search convergence theory.

The Model Management Framework Exploratory Moves algorithm (MMFEM) is presented in Figure 2.4. The component algorithms appear in the special `typewriter` font. There are four component algorithms defined in the Framework: `SEARCH`, `MANAGEMODEL`, `TERMINATE` and `POLL`. The notation "..." when used with the arguments to a component algorithm indicates that additional arguments to the algorithm are allowed by the definition. This allows for data that is not necessary to the convergence analysis but is needed in a particular implementation.

The inputs to the Framework algorithm are the current solution (x_k and f_k), the objective function (F), the current model (M_k) and the current pattern data (the

0) **Given** $x_k, f_k = F(x_k), F, M_k, \Delta_k, \mathcal{S}, \mathcal{C}_k$
1) $f_+ \Leftarrow f_k$; $s_+ \Leftarrow 0$; $j \Leftarrow 1$; $M_{k_j} \Leftarrow M_k$
2) **iterate**
3) $\quad \{s_t\} \Leftarrow \texttt{SEARCH}(x_k + s_+, \Delta_k \mathcal{S}, \ldots)$
4) $\quad$ choose $s_j \in \{s_t\}$ such that $F(x_k + s_+ + s_j) < f_+$ **or** $s_j = 0$
5) $\quad s_+ \Leftarrow s_+ + s_j$; $f_+ \Leftarrow F(x_k + s_+)$
6) $\quad M_{k_{j+1}} \Leftarrow \texttt{MANAGEMODEL}(M_{k_j}, \ldots)$
7) $\quad j \Leftarrow j + 1$
8) **until** $\texttt{TERMINATE}(j, s_+, f_+, f_k, \ldots)$
9) $M_{k+1} \Leftarrow M_{k_j}$
10) **if** ($s_+ = 0$)
11) $\quad s_+ \Leftarrow \texttt{POLL}(x_k, \Delta_k \mathcal{C}_k, F, \ldots)$
12) return s_+

Note: the notation "…" signifies that additional arguments to the component may be added.

Figure 2.4 Model Management Framework Exploratory Moves algorithm (MMFEM)

pattern $\mathcal{S}$, the core pattern $\mathcal{C}_k$ and the scale factor Δ_k). The algorithm returns the step s_+ and the model M_{k+1}. Line 1 initializes the internal state of the algorithm. Line 2 starts the loop that ends on line 8. Line 3 executes the search, returning a set of trial steps $\{s_t\}$ from the pattern. Line 4 selects one of the trial steps that satisfies a simple decrease condition on the actual objective. If none of the trial steps are satisfactory, the zero step is selected. Line 5 sums the step from this iteration with the steps from previous iterations and saves the new objective value as the target in the simple decrease condition for the next iteration. Line 6 updates the model. Line 7 increments the iteration counter. Line 8 executes the component algorithm that determines whether to terminate the loop or execute another iteration. The iteration continues until the `TERMINATE` algorithm returns the value $True$. Lines 10 and 11 execute the poll algorithm when the search loop fails to find a step that decreases the objective.

The definitions of the component algorithms are presented below. The next section extends the convergence analysis for the general pattern search method to include the Model Management Framework version of the exploratory moves algorithm.

We begin by defining precisely what we mean by *model*, then by defining the behavior of the components in the Framework.

Definition 1 Given a function $F : \mathcal{X} \mapsto \mathcal{Y}$, for arbitrary spaces $\mathcal{X}$ and $\mathcal{Y}$, a function M is a *model* of F on $\mathcal{X}$ if $M : \mathcal{X} \mapsto \mathcal{Y}$.

This says that a model M is defined on the same domain and maps into the same range space as the function it models. Note that this definition says nothing about the values of the actual and model functions. Model accuracy will have a major impact on the computational efficiency of the algorithm, but it has no theoretical effect. Also note that although the definition mentions a single model, it can be applied to the case of multiple models of the function F trivially by combining the results of the multiple models into a single value in $\mathcal{Y}$.

Definition 2 Given an iterate x_k and the set of possible trial steps $\Delta_k\mathcal{S}$, then

$$\{s_t\} = \mathtt{SEARCH}(x_k + s_+, \Delta_k\mathcal{S}, \ldots)$$

must satisfy:

$$s_{t_i} \in \{s_t\} \subset \Delta_k\mathcal{S} \quad \forall i \in \{1, \ldots, |s_t|\}$$

and for bound constrained problems it must also satisfy:

$$x + s_{t_i} \in \Omega \quad \forall i \in \{1, \ldots, |s_t|\}.$$

It is assumed that a practical `SEARCH` algorithm will make use of the model, even though the definition specifies no requirement to do so. `SEARCH` is allowed to return multiple candidate steps from the scaled pattern $\Delta_k \mathcal{S}$. It is *not* required that the candidates produce decrease in the the actual objective F or even the model M_{k_j}. Steps 4 and 5 of MMFEM guarantee that the step s_j is zero unless decrease in the actual objective is found.

Definition 3 Given a model M of the objective function F, then

$$M_+ = \texttt{MANAGEMODEL}(M, \ldots)$$

is also a model of F.

Note that M_+ may be the same as M and that no conditions on the behavior of the model are required by the convergence analysis. The model may be modified regardless of whether the trial step produces decrease in the actual objective function.

Definition 4 Given an iteration counter j for the loop in Steps 2–8 of Figure 2.4, a trial step s_+ and current and trial values f_k, f_+ of the objective function, then

$$b = \texttt{TERMINATE}(j, s_+, f_+, f_k, \ldots)$$

must satisfy:

$$b \in \{True, False\}$$

and

$$b = True \text{ iff } (\ f_+ < f_k \quad \text{OR} \quad s_+ = 0\ )$$

and

$$\exists j < \infty \text{ s.t. } b = True.$$

Definition 5 Given an iterate x_k and a core pattern $\Delta_k \mathcal{C}_k$, then

$$s_+ = \texttt{POLL}(x_k, \Delta_k \mathcal{C}_k, F, \ldots)$$

must satisfy

$$s_+ \in \Delta_k \mathcal{C}_k$$

and

$$\begin{array}{lll} \text{if } \min\limits_{s \in \Delta_k \mathcal{C}_k} \{F(x_k + s)\} < F(x_k) & \text{then} & F(x_k + s_+) < F(x_k) \\ & \text{else} & s_+ = 0. \end{array}$$

2.5 Convergence Analysis

The requirement that `TERMINATE` evaluate to $True$ for $j < \infty$ guarantees the finite termination of the loop in Steps 2–8. The component algorithms are implicitly assumed to terminate in finite time. It is not necessarily difficult to make `TERMINATE` behave as required, although in practice it will likely require some coordination with the algorithms used for `SEARCH` and `MANAGEMODEL`.

By construction, all the non-zero candidate steps ($s_j > 0$) satisfy the simple decrease condition on the actual objective. Thus their sum s_+ also satisfies this condition. This satisfies Hypothesis 6 on the oracle.

The definition of `POLL` is precisely the same as Hypothesis 7, so Steps 11 and 12 must satisfy that hypothesis. This leads to the main convergence theorem for the Model Management Framework.

> **Theorem 2.3** Let `SEARCH`, `MANAGEMODEL`, `TERMINATE` and `POLL` satisfy Definitions 2–5 and assume Hypotheses 3 and 4 are satisfied, that the level set $L(x_0)$ is compact and that F is continuously differentiable on a neighborhood of $L(x_0)$. Let $q(x)$ be defined as in Eqn. 2.3. Then for the sequence of iterates $\{x_k\}$ produced by a pattern search method that uses as its exploratory moves any algorithm that fits the Model Management

Framework,

$$\liminf_{k\to\infty} \|\Phi(x)\| = 0$$

where $\Phi(x) \equiv \nabla F(x)$ in the unconstrained case and $\Phi(x) \equiv q(x)$ in the bound constrained case.

Proof: For convenience, let $\Omega \equiv \mathbf{R}^n$ in the unconstrained case. By the definition of `SEARCH`, the candidate steps $\{s_t\}$ produced in Step 3 of MMFEM all satisfy

$$s_{t_i} \in \{s_t\} \subset \Delta_k \mathcal{S} \text{ and } (x_k + s_+ + s_{t_i}) \in \Omega, \quad \forall i \in \{1, \ldots, |s_t|\}.$$

By construction, at each iteration j the candidate step s_j selected in Step 4 satisfies

$$\text{if } s_j \neq 0 \text{ then } F(x_k + s_+ + s_j) < F(x_k).$$

It follows that the accumulation of the candidate steps s_+ computed in Step 5 satisfies

$$\text{if } s_+ \neq 0 \text{ then } F(x_k + s_+) < F(x_k).$$

Given $s_1, s_2 \in \Delta_k \mathcal{S}$, Hypothesis 4 states

$$s_1 = \Delta_k B z_1, \quad s_2 = \Delta_k B z_2, \quad z_1, z_2 \in \mathbf{Z}^n$$

so

$$s_1 + s_2 = \Delta_k B z_1 + \Delta_k B z_2 = \Delta_k B(z_1 + z_2) = \Delta_k B z_3, \quad z_3 \in \mathbf{Z}^n$$

and it follows from Hypothesis 4 that

$$s_3 \equiv \Delta_k B z_3 \in \Delta_k \mathcal{S}.$$

Thus the sum of two steps in $\Delta_k \mathcal{S}$ is also in $\Delta_k \mathcal{S}$ and so it follows that the accumulation of the candidate steps s_+ computed in Step 5 satisfies

$$s_+ \in \Delta_k \mathcal{S}.$$

Therefore s_+ satisfies Hypothesis 6. By construction, the result of the function `POLL` satisfies Hypothesis 7. By Lemma 2.5, Hypotheses 6 and 7 are equivalent to Hypothesis 5, so MMFEM satisfies the requirements of an exploratory moves algorithm, so Theorem 2.2 applies.$\square$

Chapter 3

Model Management Algorithm Design

In this chapter we discuss issues in model management algorithm design within the context of the Model Management Framework. We address each of the algorithmic components of the Framework individually, and then we define and analyze complete algorithms. The main concerns in this discussion will be fulfilling the requirements of the convergence theory and developing efficient implementations for our target application class. Some of these issues are specific to the Framework; others apply generally to all types of model management algorithms.

The algorithm design issues are divided into two groups. Since the Model Management Framework is defined as the exploratory moves for a pattern search method, issues relating to the design of the pattern search method are addressed first. Design issues involving the component algorithms for the Model Management Framework itself are addressed next.

We take a data-oriented view of the issues in pattern search method design. The discussion will address each of the major data structures present in a pattern search method: what operations are performed on them; what approaches can be used to perform these operations; and what special considerations apply to model management methods. The pattern search convergence theory described in Chapter 2 determines many of the operations.

We take a function-oriented view of the design of Framework component algorithms. Algorithm design for each of the `SEARCH`, `MANAGEMODEL`, `TERMINATE` and `POLL` component algorithms used in the Framework will be addressed. Most of the algorithmic flexibility in the Framework resides in `SEARCH` and `MANAGEMODEL`.

3.1 Pattern search methods for the Model Management Framework

Since the Model Management Framework is constructed as an instantiation of the general pattern search method GPS (Figure 2.1), there are algorithm design issues related to pattern search methods that arise. These issues are not strongly dependent on the design of the Framework algorithm components. We address these issues in terms of the major data structures used in the GPS method.

The GPS method uses only a few important pieces of data. Among these are the basis B and scale factor Δ_k used to generate the pattern, the steps used to generate the core pattern $\mathcal{C}_k$, and the contraction ratio θ used to reduce the pattern scale factor.

3.1.1 Choice of basis

The sole purpose of the basis in the GPS method is to determine the grid of points from which the trial steps are taken. The basis specifies the relationship between adjacent points in the grid, i.e. the relative spacing in each dimension and the angles between the lines connecting the grid points. The theory requires that the basis be constant, and that it must span the space (Hypothesis 1).

The issues that arise in choosing a basis can be reduced to: (1) whether the grid is to be orthogonal or not; (2) whether the spacing in the grid is to be uniform in all dimensions or allowed to vary across dimensions.

There seems to be no general reason to prefer a skewed (non-orthogonal) grid over an orthogonal one. The latter is simpler, easier to manipulate and visualize, and corresponds to the usual method for approximating derivatives by finite differences. Also, it can be represented using a diagonal matrix. A non-orthogonal grid may be preferable if some *a priori* application-specific knowledge of the function or constraints is available.

Scaling in the basis can be used to capture problem-dependent scaling by "stretching" the grid so that the relative grid spacing matches the relative magnitudes of the problem variables.

3.1.2 Choice of core patterns

The pattern search convergence theory places two requirements on the core patterns $\mathcal{C}_k$: 1) every core pattern, when multiplied by the basis, must contain a positive basis for $\mathbf{R}^n$; 2) the number of core patterns must be finite (Hypothesis 1). The latter requirement is necessary because, unlike the basis, the core pattern can be changed from one iteration of the pattern search method to the next (line 7 of Figure 2.1). If the set of different core patterns is not finite, an infinite sequence of core patterns could be constructed that would cancel the convergence provided by shrinking the scale factor, leading to a non-convergent algorithm. This is not a severe restriction on algorithm development.

For our purposes, the most important issue in the choice of core pattern (from a performance standpoint) is the number of steps in the core. Since the pattern search theory requires that the actual objective function must be evaluated at every step in the core pattern before the pattern scale factor can be reduced, it is usually desirable for the core pattern to contain as few steps as possible in order to reduce the computational cost. Recall that a positive basis for an n-dimensional space can contain from $n+1$ to $2n$ vectors, so the cost difference between minimal and maximal core patterns is almost 50% (assuming constant function evaluation cost). One of the fundamental premises of the model management approach is that the cost of function evaluations is very high and dominates the total cost of solving the optimization problem. Thus the difference between minimal and maximal patterns can amount to a significant fraction of the cost of a model management solution.

Another issue in the choice of number of steps in the core pattern is the matter of sampling effectiveness. A core pattern with more steps provides a better sampling of the local neighborhood of the current iterate than a smaller one, increasing the likelihood of finding a step which decreases the objective function. Thus an algorithm using a larger core pattern may take fewer iterations to achieve a given amount of decrease in the objective function than an algorithm using a smaller core pattern, although each iteration may take longer. The size of the core pattern may also affect the quality of the model if the values of the actual function that are computed at the core steps are used to update the model in some fashion. The better sampling provided by a larger core pattern may produce a better model of the local neighborhood, which may make the algorithm converge in fewer iterations.

In bound constrained problems the pattern search theory requires the use of the maximal core pattern to ensure a sufficiently rich set of search directions to guarantee that a solution that lies on a boundary can be found. This extra cost may be avoided by using the maximal core only when one or more of the core steps would touch or cross a boundary. When all the core steps are in the interior of the bounded region the problem can be treated as unconstrained and a minimal core pattern used. The pattern search convergence theory for linear inequality constrained problems formalizes this approach [49]. This is also akin to the "active set" strategy used in many gradient-based optimization algorithms for constrained problems [29] in that constraints that are not active are ignored until they become active.

The design of the `POLL` algorithm can have a significant impact on the choice of the size of the core pattern. In particular, if the function values at the core steps are computed in parallel, the elapsed time to evaluate all the core steps will be equal to the elapsed time of one function evaluation (assuming each evaluation does the same amount of computation). This reduces the effective cost of using the maximal core pattern to the same as the cost of the minimal core, and raises to primary importance issues other than the computational cost of the function evaluations. The design of `POLL` algorithms is discussed further in §3.2.4.

Another important issue relating to the core pattern is how to change the steps in the core from one pattern search iteration to the next. The pattern search theory does not require that the core pattern be constant. It does require that the set of core patterns be finite, which translates into a requirement that the norms of the vectors in the core patterns be bounded across all iterations.

The simplest choice is to select a constant core pattern. The $2n$ coordinate vectors (scaled or not) are an obvious choice. This would form a maximal positive basis, as discussed above.

A more complex approach is to change the core pattern at some or all of the iterations of the pattern search. One motivation for this approach is to adapt the core according to the available knowledge of the objective function. The maximal positive basis is not unique, and different bases sample different parts of the space. This can be used to advantage by looking at the points where the objective function value is already known (from previous evaluations) and biasing the core pattern toward the points with lower objective values, increasing the probability that one of the core steps will produce decrease. This idea is used in the Multi-Directional Search (MDS) pattern search method to choose some of the core steps as well as the additional steps in the pattern outside the core [69],[70].

3.1.3 Updating the pattern scale factor

The pattern search theory almost completely specifies the behavior of the scale factor (Hypothesis 3). The model management algorithm designer is left with only a few decisions. One is the choice of the initial value of the scale factor. The other is the choice of contraction and expansion factors.

The contraction factor must be a constant rational number[‖], and the theory specifies precisely when it must be applied. The traditional value is 1/2. The smaller the value, the fewer executions of the `POLL` algorithm must be executed to achieve a given convergence level. The disadvantage of a smaller contraction factor is the danger of premature convergence. This can occur when the descent direction is "between" the directions present in the pattern. The theory guarantees that a descent direction will be found, if one exists, in the limit as the scale factor goes to zero, but it says nothing about what will happen along the way. If the pattern does not capture the descent direction and the current iterate is far from a local minimizer it may require many more steps to get close to that minimizer because the smaller scale factor will make the steps shorter. More steps implies more iterations, more function evaluations and higher computational cost.

The choice of method for expanding the pattern scale, as well as the choice of expansion factor, sometimes can counteract this undesirable behavior. In general it can accelerate convergence by allowing long steps to be taken. The convergence theory does not address scale expansion except indirectly in as much as it defines when the scale must be contracted (since both cannot be done at the same time).

[‖] It can be proved that the contraction factor may be allowed to vary across iterations as long as it is bounded above and below and all the factors used are integer powers of a constant rational number. The same holds for the expansion factor.

The danger of premature expansion exists, but is somewhat less than that of premature contraction since its effect cannot last long. When the pattern is expanded and no point with decrease is found, the pattern must be contracted. Thus the negative effect of expansion is self-limiting. At worst it wastes an iteration (assuming the contraction and expansion factors are equal). The positive effect of expansion is that longer steps can continue to be taken as long as the pattern finds decrease. From this one can argue that perhaps the expansion factor should be a multiple of the contraction factors, since an occasional few wasted iterations may be worth the iterations saved by taking longer steps. The truth of this argument certainly will depend on the problem.

In practical problems, the model management algorithm is often run for relatively few iterations because of the large computational cost, especially of the `POLL` algorithm. Thus the choice of contraction and expansion factors may have a large effect because even a few wasted iterations may be a significant portion of the total solution cost. Unfortunately, the "right" choices are very problem dependent. It seems that only experience can lead us to good ones.

3.1.4 Termination criteria

Specifying termination conditions for a pattern search method is somewhat more complicated than for a gradient-based optimization method because of the lack of explicit gradient information. In the model management context, good termination criteria are important because of the high cost of each iteration.

In general, termination conditions for pattern search methods are fairly limited because there is not much data used in the method.

From the viewpoint of the convergence theory, the obvious termination condition is a lower limit on the value of the pattern scale factor, since the theory guarantees that this value goes to zero in the limit. The length of the step taken at each iteration ($||X_{k+1} - X_k||$) could also be used in a termination condition with similar results, since it depends on the scale factor.

In practice, other factors can also be considered. The most practical is an upper bound on the execution time allowed. A limit on the number of objective or constraint function evaluations gives a similar behavior, since in most pattern search methods the time spent outside the function evaluations is relatively small. In parallel implementations, the number of iterations may be a preferable measure as it is less dependent on the load balance.

Between these theoretical and practical extremes are termination conditions based on other variables. Termination can be based on a target objective function value, or on some measure of the reduction in objective function value per iteration. For example, the reduction in objective divided by the step length can be view as an approximation of a directional derivative and should tend toward zero as the method converges (called a "simplex gradient" in [8, 42]).

In the model management context, the high cost of the function evaluations biases the choice of termination conditions. We tend to be more concerned with stopping when the process is not making much progress. This may occur because the current iterate is close to an optimizer or because the method is not doing well, probably because the model is not a sufficiently good predictor.

The quality of the model and the success of the search strategy are the two most basic measures of the performance of a model management algorithm. Implementation of conditions based on these measures can be difficult. In trust region methods the model quality is measured by the ratio of the decrease in the objective predicted by the model to the decrease actually found. In model management methods, the success of the search strategy can be measured by how often the `POLL` algorithm (Definition 5) is invoked.

3.2 Component algorithm design

A model management method expressed using the Model Management Framework is defined by the algorithms used to implement `SEARCH`, `MANAGEMODEL`, `TERMINATE` and `POLL`. In practice the implementations of the first three components almost surely will be interdependent to some degree, though not all of the components will necessarily depend on all others. In particular, since the actual changes made to the model do not play an active role in the convergence analysis, `MANAGEMODEL` may well be independent. Thus, it is possible for the changes in the model to be transparent to the `SEARCH` component algorithm. On the other hand, if the quality of the model is involved in achieving finite termination of the loop at Steps 2–8 in MMFEM, a strong interdependence between `MANAGEMODEL` and `TERMINATE` can result, as will be seen in the examples to be discussed. The `POLL` component is conceptually independent because only the actual function F is required, not the model M, but in practice its behavior also is likely to be influenced by the behavior of the other component algorithms.

The next four sections (§3.2.1–3.2.4) present some general ideas about methods for `SEARCH`, `MANAGEMODEL`, `TERMINATE` and `POLL`, and example implementations that demonstrate the issues involved in algorithm design for these methods. Section 3.3 will present specific implementations that are more likely to be useful in practice. Issues of performance and reuse of existing software will drive the choice of implementation for real applications.

3.2.1 SEARCH methods

The convergence analysis does not specify anything about the method used to implement SEARCH. The only requirement is that the steps computed by SEARCH $\{s_t\}$ must be taken from $\Delta_k \mathcal{S}$. Additional requirements may be added in order to insure finite termination of the loop in Steps 2–8 of MMFEM as long as they do not interfere with this requirement.

Possibly the simplest method to implement SEARCH would be the following:

SEARCH$(x, \Delta_k \mathcal{S})$

1) randomly choose a step $s \in \Delta_k \mathcal{S}$

2) return $\{s\}$

Although fast, simple and provably correct, this method is not likely to succeed very often. This is an extremely undesirable property, to say the least. If none of the steps computed by SEARCH produce decrease in F, then eventually POLL must be invoked.

This trivial implementation does not use the model and does not measure the quality of the selected step in any way. An obvious extension of this implementation would be to select several steps randomly, rather than one, and only return those steps that have a lower model value than the current iterate.

SEARCH$(x_k, \Delta_k \mathcal{S}, M)$

1) choose q random steps $\{s_j\}$ from $\Delta_k \mathcal{S}$, $\quad (0 < q < \infty)$

2) select $\{s_t\} \subseteq \{s_j\}$, such that $M(x_k + s_{t_i}) < M(x_k)\ \forall i$, or set $\{s_t\} = \emptyset$

3) return $\{s_t\}$

Another implementation approach that uses randomness is to use a genetic or evolutionary algorithm [52] to select steps. The crossover and mutation operators in a genetic algorithm can be manipulated to guarantee that the results are taken from the grid defined by $\Delta_k\mathcal{S}$. The combination of genetic algorithms with pattern search methods has been investigated previously and discussed in [35] and [36].

Returning to deterministic approaches, a simple implementation of `SEARCH` that uses the model to assist in the search for decrease in the actual function is:

`SEARCH`$(x_k, \Delta_k\mathcal{S}, M, F)$

1) $\{s\} \Leftarrow \emptyset$; $m \Leftarrow F(x_k)$

2) for $i \in \mathcal{I}$

3) if ($M(x_k + \Delta_k\mathcal{S}_i) < m$)

4) $\{s\} \Leftarrow \Delta_k\mathcal{S}_i$; $m \Leftarrow M(x_k + \Delta_k\mathcal{S}_i)$

5) return $\{s\}$

where $\mathcal{I}$ is an index set of finite size. This evaluates the model at a fixed number of steps in $\mathcal{S}$ and returns the minimizer. The choice of which steps to evaluate can be made *a priori* or can be determined adaptively each time `SEARCH` is executed. This implementation of `SEARCH` is effectively an oracle algorithm applied to the model instead of the actual objective function. It can be extended by selecting a finite number of the best steps and returning them.

Another type of approach involves using an optimization method to find an approximate minimizer of the model M.** This is the intuitively obvious approach to finding decrease in the model function. Since existing implementations of optimization methods are abundant, this approach should be relatively easy to implement. Because the model is relatively cheap to evaluate (compared to the actual objective), the computation should be fast. The only complication is the requirement that the steps computed by `SEARCH` must be taken from $\Delta_k\mathcal{S}$. This can be satisfied by either enforcing it directly in the optimization method or taking the approximate minimizer

**See [73] for an example.

and "moving" it to a nearby step in $\Delta_k \mathcal{S}$ (subject to the constraints, if any). A different approach to enforcing the requirement directly would be to use a pattern search method such as the multi-directional search (MDS) algorithm described in [69] and [70] to optimize the model function, using a pattern consistent with the one defined by $\Delta_k \mathcal{S}$.

Assume that we are given an algorithm `OPT` that computes an approximation to a minimizer of the model objective M. Two generic algorithms for a `SEARCH` method of this type can be specified, depending on whether or not `OPT` insures the result is on the grid.

If `OPT` always computes steps on the grid the requirements on the result of `SEARCH` are satisfied and no further work is needed. An implementation using this `OPT` is:

`SEARCH`$(x_k, \Delta_k \mathcal{S}, M)$
1) $\{s\} \Leftarrow$ `OPT`$(x_k, M, \Delta_k \mathcal{S})$
2) return $\{s\}$.

If `OPT` does not always compute steps on the grid, the `SEARCH` implementation must choose a (feasible) step in $\Delta_k \mathcal{S}$ that is near the result of the optimization procedure. Obviously it is desirable that the step selected produce decrease in the model. If we are given a function `NEARBY` that computes a subset of steps from $\Delta_k \mathcal{S}$ that are "close" to s (and feasible, if the problem is constrained), then we can implement `SEARCH` as follows (f_k denotes the actual function value at the current iterate x_k).

`SEARCH`$(x_k, \Delta_k \mathcal{S}, M, f_k)$

1) $t \Leftarrow$ `OPT`(x_k, M)

2) if ($t \in \Delta_k \mathcal{S}$)

3) $\quad \{s\} \Leftarrow t$

4) else

5) $\quad$ choose $\{s\}$ in `NEARBY`$(x_k, t, \Delta_k \mathcal{S})$ that minimizes $M(x_k + s)$ or $\{s\} = \emptyset$

6) return $\{s\}$

There are various ways to define the set of nearby steps, but they have no effect on the convergence or termination analysis as long as the set computed by `NEARBY` is finite. A general approach is to solve $By = x_k + s$ and round each of the components of y to an integer value. Each of the n components of y may be rounded up or down, giving the 2^n vertices of the hypercube of grid points surrounding y. The problem of choosing which of the vertices to use is conceptually straightforward, but in practice is complicated by the large number of vertices. If the gradient of the model is available, it can be used to select a subset of the vertices that may be worth looking at. A simpler approach is to round y to the nearest grid point and look at only the $2n$ grid points that are nearest in each of the coordinate directions. A backup strategy may be necessary in case none of these are acceptable.

Each of the optimization-based approaches can be generalized to return several possible steps. One way to accomplish this is to perform several optimizations on the model starting from different initial points. This technique is used in some global optimization methods. The literature on global optimization [26] [38] [57] is a useful source of strategies for `SEARCH` algorithms.

There are two conflicting goals in the design of the `SEARCH` algorithm. On the one hand it is desirable to search the local neighborhood of the current iterate as well as possible, to get fast local convergence when the current iterate is near the solution. On the other hand, it is desirable to search outside the local region in order to identify other basins that might have better solutions. This is especially important in the model management context because the target applications are assumed to have multiple local minimizers. The global search problem is equivalent to the global optimization problem and as such, cannot be solved in general with reasonable efficiency. However, it is possible to develop heuristics that produce better solutions in many cases than methods that only search locally. The advantage of the model management approach is that the small evaluation cost of the model allows considerable flexibility in the implementation of these heuristics.

The two goals are in conflict in the model management context because information about the actual function is a limited resource, and the method ultimately requires evaluations of the actual function to check the search results. Thus, generating more local trial steps in `SEARCH` reduces the number of global trial steps that can be chosen because the number of actual function evaluations that can be performed is limited and the actual function must be evaluated at a trial step before it can be accepted as the next iterate.

A reasonable compromise is to combine local and global search methods in a `SEARCH` algorithm and use some kind of merit function that considers both objective value and distance from the current iterate (and possibly distance from the other trial steps) to choose which steps to return to the Framework. Choosing a merit function is not a simple task because the amount of decrease in the objective function is not likely to have the same scale as the measure of distance. There has been some work in statistics in response surface modeling and global optimization that has addressed this issue [39] [54] [61] [73].

The issue also arises in the modeling strategy itself as a tradeoff between local and global accuracy. In addition to the constraint of limited information, there is the difficulty of generating a model that has the appropriate accuracy both locally and globally. The local accuracy needs to be high, since small changes in the function are likely to be involved, whereas the global accuracy can be quite low since only the general trend in the function needs to be captured in order to find descent when the current iterate is far from a solution.

3.2.2 `MANAGEMODEL` methods

An important component of any practical model management algorithm is the method used to manipulate the model as the algorithm progresses. One of the virtues of the framework described here is that theoretical considerations place no restrictions on the implementation of the `MANAGEMODEL` component algorithm.

A note should be made here about model accuracy. In the model management framework, there are two separate considerations in which the accuracy of the models plays an important part. In each individual model management iteration, it is desirable that the model accurately predict the location of the minimizer of the actual function (or *a* minimizer, if there is more than one). For a model management method to be successful, the model's prediction of the location of the actual minimizer must improve as more data is added to the model.

The lack of theoretical requirements on the implementation of `MANAGEMODEL` means that no assumptions need be made about the accuracy of the model. In the worst case of an extremely inaccurate model, the Model Management Framework effectively reduces to a pattern search method applied directly to the actual objective function. Thus convergence is guaranteed regardless of the accuracy of the model. The model is used purely as a device designed to accelerate convergence of the pattern search by reducing the number of function evaluations needed to find decrease. This does not mean than all modeling strategies are equivalent. The efficiency of a particular algorithm will usually depend on how well the model predicts descent for the actual function at each iteration and how well the model update strategy maintains the accuracy of the model during the optimization process.

Because `MANAGEMODEL` is relatively isolated from the rest of the components of the framework (in the theoretical sense), there is little that can be said about its behavior in general. The intent of including `MANAGEMODEL` in the Framework is to allow the model to be changed as execution progresses. Note that each time `SEARCH` is executed, the actual objective function is evaluated at least once. This function value represents a piece of information that can be used to modify the model. One of the fundamental motivations of the model management concept is that the potential exists to improve the model incrementally by reusing the values of the actual objective that must be computed anyway to satisfy the convergence theory.

One particular approach to `MANAGEMODEL` is expressed in a method developed at Boeing called "Balanced Local-Global Search" (BLGS)[6]. The basic idea of BLGS is to modify the model by incorporating multiple function values into the model all at once, essentially rebuilding the model with a larger amount of data. This approach incrementally improves the model, but does it with a fairly large granularity. The data points to be added to the model are chosen to fulfill one of two different goals: to improve the accuracy of the model in areas of the space that have been identified as being of interest (local); to add data to the model in areas that have not been sampled (global). As the name implies, the point is to balance these two goals given a limited number of function evaluations. The cost of function evaluations is too high to use this approach in every iteration of the Model Management Framework so some mechanism is needed to determine when it is appropriate to rebuild the model. This mechanism will depend on the type of model used.

An alternative approach to balancing the local and global accuracy of a single model is to maintain different models for different parts of the solution space or for different length scales. For example, one model could be constructed over the entire space and a second model could be constructed on a small region around the current iterate. If a low-fidelity physical model is available, it could be used as the global model and an interpolatory model could be used for the local model. This approach allows the modeling strategy to be tailored for specific applications and available modeling capabilities.

A different situation arises if the model is (or depends on) a variable-fidelity approximation of a governing equation. In this case, the management strategy may simply reduce to increasing the accuracy of the approximation (and presumably increasing its cost) as the optimization process progresses and the optimal solution is approached. A common example of this kind of model is a numerical solution of a differential equation that relies on a discrete grid, with the grid spacing as the means of controlling the fidelity. The `SEARCH` strategy should adjust to this variation so that the cost of executing `SEARCH` does not grow to be unreasonably large.

3.2.3 `TERMINATE` methods

In general, the implementation of the `TERMINATE` component algorithm tends to be the most dependent on the implementations of the other components. This is because there is little data in the algorithm to use in the termination decision that is not affected by the implementations of the other components. Only very simple termination strategies can hope to be completely general. A couple are presented here.

`TERMINATE`(j)

0) Given $j_{max} < \infty$

1) if $j \geq j_{max}$

2) return $True$

3) else

4) return $False$

`TERMINATE`(j, s_j)

1) if $s_j = 0$

2) return $True$

3) else

4) return $False$

The first simply checks the iteration count and terminates the loop when a particular number of iterations has been completed. The second terminates whenever the `SEARCH` component has failed to find decrease in the objective. Note that the second version does not use the iteration counter j but it is left in the function interface to be consistent with Definition 4.

It can be shown that both of these implementations satisfy the theoretical requirements of Definition 4. Step 4 of the MMFEM algorithm guarantees that any non-zero step must give decrease in the actual objective function, so the first part of the definition is always satisfied. Thus the necessary theoretical behavior effectively reduces to the finiteness requirement (`TERMINATE` evaluates to $True$ for some $j < \infty$). This requirement is obviously satisfied in the first implementation above.

The second implementation also has the finite termination property, but it comes about for a more subtle reason. By construction, `TERMINATE` returns $True$ when the step s_j is zero, so if `TERMINATE` is executed with a non-zero step only a finite number of times, the resulting Framework algorithm must have the desired termination property. The following lemma explains why the number of nonzero steps is always finite.

Lemma 3.1 Given a current iterate x_k that satisfies $F(x_k) \leq F(x_0)$, let $\{s_i\} \equiv \{s_i | s_i \in \Delta_k \mathcal{S}, F(x_k + s) < F(x_k)\}$. Then $|\{s_i\}| < \infty$.

Proof: By assumption the level set of $F(x_0)$ is bounded. Hypothesis 2 guarantees that the sequence of iterates x_k produces a monotonically non-increasing sequence of objective function values. Thus $F(x_k) \leq F(x_0)$, $\forall k$, so the level set of $F(x_k) \leq F(x_0)$ must be bounded. Hypothesis 3 guarantees $\Delta_k > 0$ so the number of elements of $\{x_k + \Delta_k \mathcal{S}\}$ inside the level set must be finite, so the number of elements $s \in \Delta_k \mathcal{S}$ that also satisfy $F(x_k + s) < F(x_k)$ must be finite.□

Lemma 3.2 The number of iterations of Steps 2–8 of MMFEM in which the trial step s_j is nonzero is finite.

Proof: Steps 4 and 5 of MMFEM guarantee that f_+ at iteration j must be less than at the previous iteration if $s_j \neq 0$. Therefore f_+ is monotonically decreasing in iterations when $s_j \neq 0$. By Lemma 3.1, the number of unique steps in the pattern that decrease the actual objective function is finite so the number of unique function values is finite. It follows from the monotonicity that f_+ can take on each value in only one iteration in which $s_j \neq 0$ so the number of iterations in which $s_j \neq 0$ must be finite.□

Lemma 3.2 guarantees that $s_j = 0$ for some $j < \infty$, and so this instantiation of `TERMINATE` satisfies Definition 4.

In general, the guarantee of finite termination required by Definition 4 will be an important consideration in designing model management algorithms using the Framework. Metaphorically speaking, it is the glue that can be used to bind together strategies for `SEARCH` with update strategies for `MANAGEMODEL`. The issue of how to design model management algorithms to guarantee finite termination is important also because it impacts when and how often the `POLL` algorithm is executed.

3.2.4 `POLL` methods

In the Model Management Framework, the `POLL` function is the tactic of last resort to satisfy the hypotheses on the results of an exploratory moves algorithm. It does not rely on the results of the other component algorithms because it necessarily satisfies the hypotheses. The essential idea of the Model Management Framework approach is that the other components are included to achieve satisfactory efficiency in practice, not to achieve necessary behavior in theory. As happens repeatedly in the model management paradigm, the theoretically necessary functionality can be achieved quite simply, but efficiency usually requires adding complexity.

A straightforward implementation of `POLL` is

`POLL`$(x_k, \Delta_k \mathcal{C}_k, F)$

1) for $s \in \Delta_k \mathcal{C}_k$

2) if ($F(x_k + s) < F(x_k)$) then return s

3) return 0

This implementation evaluates the objective at the steps in the core one at a time and terminates at the first step that produces decrease in the objective. A more complicated approach uses the model to determine the order in which the steps in the core pattern should be evaluated. This is intended to reduce the number of function evaluations by evaluating the steps that are most likely to produce decrease first. An implementation using this idea is:

POLL$(x_k, \Delta_k \mathcal{C}_k, F, M)$

1) $\widehat{S} \Leftarrow$ steps in $\Delta_k \mathcal{C}_k$ sorted by ascending value of M

2) for $s \in \widehat{S}$

3) if ($F(x_k + s) < F(x_k)$) then return s

4) return 0

Like the simpler version, this version terminates at the first step in the core that produces decrease. It is independent of the other component algorithms in the framework. It uses the model M but does not modify it in any way that could affect the implementation of any of the other components. The behavior of the model (in particular, how well it captures descent away from x_k) will determine the efficiency with which this version of POLL executes, but not its correctness.

Since POLL must evaluate the actual objective function, it is possible to implement it to modify the model to incorporate the new objective values into the model as they are computed. Such an implementation would depend on the modeling method used, and would thus be interdependent with the other functions that use or modify the model, particularly MANAGEMODEL. An example will be presented in a later section.

In the worst case, any poll algorithm will be required to evaluate the actual objective function at all steps in the core, an operation we assume is expensive. It is possible to compute all the function values independently, since the order of evaluation is irrelevant. So by using a parallel computer with a sufficient number of processors the worst case cost can be effectively reduced to one function evaluation in elapsed (wall clock) time by evaluating the objective at all steps in the core simultaneously and selecting one that produces decrease. Obviously, such a difference in performance can have a profound impact on the design of algorithms for this and the other component algorithms.

3.3 Two complete model management algorithms

The next sections describe two complete model management algorithm implementations using the Framework. We analyze the behavior of each method to show it conforms to the convergence theory given in Chapter 2. These algorithms are intended to serve as examples of what can be done within the Model Management Framework and as foundations for the development of algorithms better suited to practical applications.

3.3.1 Pattern search-based algorithm

This algorithm is a straightforward application of the general pattern search idea to the `SEARCH` algorithm. The advantages are that it is simple to analyze and implement, and does not require the model to interpolate exactly. It assumes the model is cheap to evaluate and takes advantage of this assumption by performing many evaluations of the model. The other framework component algorithms are kept simple.

The method used for `SEARCH` samples the model objective at the steps in a pattern centered on the current iterate. The steps s_i in this pattern are generated using the same formula as in the GPS method:

$$s_i = \Delta_k B \mathcal{L}_i, \quad i \in \{1, \cdots, |\mathcal{L}|\}.$$

where Δ_k and B are the scale factor and basis matrix at iteration k of the Model Management Framework, $\mathcal{L}$ is a finite set of vectors in $\mathbf{Z}^n$. For simplicity we make the set $\mathcal{L}$ constant and specify it as a parameter to the `SEARCH` algorithm. A more complex approach would be to adapt $\mathcal{L}$ during the optimization process.

Each step in the pattern that produces decrease in the model is added to a list. After all the steps are evaluated, the list is sorted by model value and the n_s steps with the lowest values are returned as the result of `SEARCH`. The value of n_s is specified as a parameter to `SEARCH`. It determines the maximum number of trial steps that `SEARCH` may return, and thus determines the maximum number of actual function evaluations that will be executed by the Framework on the trial steps. If the model accurately captures the trend information in the actual function, the trial steps with the lowest model values are the most likely to produce decrease in the actual objective function. It should be obvious that the choice of $\mathcal{L}$ and n_s will have a significant impact on the performance of the algorithm in practice. The pattern $\mathcal{L}$ determines how well the space is sampled and n_s determines the worst case cost of evaluating the trial steps.

The `SEARCH` algorithm can be summarized as follows:

`SEARCH`$(x_k, f_k, M_k, \Delta_k, B; \mathcal{L}, n_s)$

0) $\widehat{\mathcal{S}} = \emptyset$

1) for i in $1, \cdots, |\mathcal{L}|$

2) $\quad m \leftarrow M_k(x_k + \Delta_k B \mathcal{L}_i)$

3) $\quad$ if $(m < f_k)$ add $\Delta_k B \mathcal{L}_i$ to $\widehat{\mathcal{S}}$

4) end for

5) sort $\widehat{\mathcal{S}}$ by increasing model value

6) return $\{\widehat{\mathcal{S}}_i | i = 1, \cdots, \min(n_s, |\widehat{\mathcal{S}}|)\}$

The intent of this algorithm design is to have $\mathcal{L}$ be very large. This is to allow the pattern to sample a large part of the design space in the hope of finding improvement in the model outside the local region of the current iterate. Ideally the steps in $\mathcal{L}$ should sample the local region more densely than the non-local region, since there is a greater likelihood that decrease will be found near the current iterate. We can accept the large number of model evaluations because we assume the model is very cheap to evaluate. Also, we hope that the additional cost is more than compensated for by the increased chances of finding a better solution far away from the current iterate, or finding the best local solution in one `SEARCH` execution rather than several, thus reducing the number of actual evaluations that must be performed.

This approach is motivated by the observation that many problems to which the model management approach is targeted are complex pieces of software and tend to be noisy, non-smooth, sometimes discontinuous, and generally lacking in the properties associated with well-behaved optimization problems. This poor behavior is often a property of the software implementation of the function, not of its mathematical formulation, so it only rarely is addressed in theoretical nonlinear programming discussions (see [12] for one such treatment). The fact that all computer calculations are, to varying degree, discrete, rears its ugly head often in these applications.

The algorithm for `MANAGEMODEL` requires the capability for the new information (points and function values) to be integrated into the model at a reasonable cost. It is not required that the new model interpolate the new values, but it is necessary for the efficient operation of this algorithm that the model be relatively close to the actual function in the neighborhood of the new points. Typically the model would be modified in such a way as to improve the accuracy of the model at (and probably in some neighborhood of) the point where the actual function was evaluated. There is no theoretical requirement that this goal be accomplished, nor is it the only goal one might have in modifying a model. We call this procedure `CALIBRATE` because the typical use is to adjust the model to make it agree with the new information obtained about the actual objective.

Using this function, the algorithm for `MANAGEMODEL` can be expressed as follows:

`MANAGEMODEL`$(x_k, M, \widehat{\mathcal{S}}, \widehat{f})$

1) for i = $1, \cdots, |\widehat{\mathcal{S}}|$

2) `CALIBRATE`$(M, x_k + \widehat{\mathcal{S}}_i, \widehat{f}_i)$

3) end for

4) return M

where $\widehat{\mathcal{S}}$ denotes the set of trial steps at which the actual function has been evaluated and $\widehat{f}$ denotes the corresponding set of function values.

The `TERMINATE` algorithm is simple. It returns $True$ if none of the current trial steps produced decrease in the actual objective.

`TERMINATE`(x_k, M, F, s_+)

1) if $(s_+ = 0)$

2) return $True$

3) else

4) return $False$

If the trial step s_+ computed by the `SEARCH/MANAGEMODEL/TERMINATE` loop in MMFEM is zero, the `POLL` algorithm will be invoked. Steps 4 and 5 of MMFEM and the definition of `TERMINATE` guarantee that `POLL` cannot be invoked if the loop terminates on any iteration except the first, since the step s_+ will have to be nonzero in any iteration after the first. It follows that `POLL` will only be invoked when `SEARCH` has failed twice in a row, once in the last iteration of the previous execution of MMFEM and again in the first iteration of the current execution. This is a reasonable compromise between the need to invoke `POLL` to ensure convergence and the desire to avoid invoking `POLL` because it is potentially so expensive. If two successive executions of `SEARCH` fail to find a point with decrease then either the current iteration is close to the solution and reducing the pattern scaling is the appropriate thing to do, or the solution is still not close but the model's predictions are so poor that sampling the actual function directly is probably the only way to make progress. This exemplifies one of the key features of the model management convergence theory: the `POLL` algorithm alone guarantees that the hypotheses of the pattern search theory are satisfied.

The algorithm used to implement `POLL` takes advantage of the model to try to reduce the number of actual function evaluations that must be performed. The steps in the core pattern are ranked in order according to their model value and the steps are evaluated in this order. The new values are added to the model using the `CALIBRATE` function defined for the `MANAGEMODEL` algorithm.

`POLL`$(x_k, \Delta_k \mathcal{C}_k, F, M)$

1) $\hat{\mathcal{S}} \Leftarrow$ elements of $\Delta_k \mathcal{C}_k$ sorted by value of $M(x_k + c), \quad c \in \Delta_k \mathcal{C}_k$

2) for $s \in \hat{\mathcal{S}}$

3) $\quad f \Leftarrow F(x_k + s)$

4) $\quad$ `CALIBRATE`$(M, x_k + s, f)$

5) $\quad$ if $(f < F(x_k))$ return s, M

6) end for

7) return $0, M$

In the case where at least one of the steps in the core produces decrease, it is reasonable to believe that this step is more likely to have a lower model value, and will be found sooner by considering the model values than it would be by choosing a fixed or random order to evaluate the steps. In the case where none of the steps in the core produce decrease, the order of evaluation is irrelevant and the work done to order the steps is wasted. But since it is assumed the model is cheap to evaluate and n is such that the cost of the sort is small, it follows that the cost of this work is not significant.

An obvious variant of this `POLL` algorithm removes unsuccessful steps from $\hat{\mathcal{S}}$ as they are evaluated and sorts the remaining steps with the new model. This increases the cost of the algorithm in the worst case, but if the model evaluation cost is small compared to that of function evaluation the increase may not be significant. The benefit is it may require fewer actual function evaluations in the average case, which should result in a significant reduction in the total cost (assuming actual function evaluations are expensive).

To prove the method is convergent it is necessary to show that each component behaves according to the definitions in §2.4.3. The following corollary to Theorem 2.3 serves this purpose.

Corollary 3.1 The algorithms for `SEARCH`, `MANAGEMODEL`, `TERMINATE` and `POLL` defined above satisfy Hypotheses 3–5.

Proof The definition of `SEARCH` (Definition 2) requires each step that is returned must be from the current scaled pattern. Each step s_i used in the `SEARCH` algorithm given above satisfies this by construction:

$$s_i \in \Delta_k B\mathcal{L} \Rightarrow s_i = \Delta_k Bl \text{ for some } l \in \mathcal{L},$$

$$l \in \mathcal{L} \subset \mathbf{Z}^n \Rightarrow Bl \in \mathcal{S} \Rightarrow s_i \in \Delta_k \mathcal{S}.$$

The `MANAGEMODEL` algorithm returns a model, so its behavior satisfies Definition 3 trivially.

Definition 4 requires the `TERMINATE` algorithm to return either $True$ or $False$ and to return $True$ for some finite value of the iteration counter j. The algorithm satisfies the first by construction. To satisfy the second it suffices to show that the trial step is nonzero for only finitely many values of j. Let the trial step at iteration j be denoted by s_j. By construction we have that

$$s_j = 0 \Rightarrow \texttt{TERMINATE} = True.$$

It follows from the definition of the Framework (line 4 of Figure 2.4) that

$$s_j \neq 0 \Rightarrow f_j < f_{j-1} \Rightarrow f_j < f_0,$$

and it follows that the objective value is unique in each iteration in which $s_j \neq 0$. Hypotheses 3 and 4 guarantee that all the steps in the pattern are distinct, so there must be a finite number of pattern steps within the level set of the initial objective value $L(x_0)$. Thus there can be only a finite number of steps that decrease the objective and so there can be only a finite number of iterations in which the step is nonzero. Thus `TERMINATE` must return $True$ in a finite number of iterations, satisfying Definition 4.

Definition 5 requires the `POLL` algorithm to return a step from the core pattern $\Delta_k \mathcal{C}_k$ that either decreases the objective function or is zero. Line 1 of `POLL` defines the set $\widehat{\mathcal{S}}$ as containing only steps from $\Delta_k \mathcal{C}_k$, and the rest of the algorithm only considers steps from $\widehat{\mathcal{S}}$ so the first part of the definition is satisfied by construction. Line 5 allows the step s to be returned as the result only if the objective function decreases. If the condition in line 5 is not satisfied by any step in $\widehat{\mathcal{S}}$, line 7 returns a result of zero. Thus the second part of Definition 5 is also satisfied.

Thus all the component algorithms satisfy the definitions from the Model Management Framework, so this model management algorithm satisfies Hypothesis 5 on the exploratory moves, and the resulting pattern search algorithm satisfies Theorem 2.3. □

3.3.2 Trust region-based algorithm

Section 3.2.1 discussed the use of optimization methods to implement the `SEARCH` algorithm. In this section the concepts of a class of optimization methods – trust region methods – are used to design a complete model management algorithm. In particular, two of the principal ideas of trust regions methods are used. These ideas were discussed in a slightly different form in [21]. Note that the model management algorithm, being gradient-free, does not retain the strong theoretical properties of quasi-Newton trust region methods.

The first idea from trust region methods that we use is to reduce the original optimization problem to a simpler optimization subproblem on a model of the original objective function. The second idea is to solve the optimization subproblem on a bounded region (called the trust region) and reduce the size of this region when the accuracy of the model is seen to be poor, as determined by the amount of decrease predicted by the model compared to the amount of decrease found in the actual function.

In trust region methods the model is usually a quadratic Taylor series expansion of the objective function around the current iterate. The property of a Taylor series model that is most important in the trust region context is that the errors in the value and gradient of the model decrease closer to the current iterate. Thus the model can be relied upon eventually to produce a good approximation of a direction of descent of the actual function as the trust region radius is decreased.

Many types of models that would be considered for use in a model management method do not have this behavior. Thus the trust region approach must be relaxed for use in the model management context. In this section we present a model management algorithm in which the model is required only to be continuous and to interpolate the actual function at the current iterate. Because of continuity, this guarantees that for some value of the trust region radius the integrated error in the value of the model over the region will decrease if the radius is decreased. This requirement, combined with the method for managing the trust region radius, is used to guarantee finite termination of the subproblem algorithm. The rest of the model management algorithm guarantees that if the subproblem is solved in finite time then the whole method will converge. Note that this does not guarantee that the gradient of the model will approach the gradient of the actual function. Thus it cannot be assumed that a descent direction for the model will be a descent direction for the actual function. The algorithm accounts for this possibility by executing `POLL` when the trial steps computed using the model all fail to produce decrease in the actual function.

Although interpolation at the current iterate does not guarantee that the model will predict decrease successfully in the actual function, as a heuristic it will succeed some of the time if the model is reasonably accurate. We hope that the model will capture at least *some* of the descent in the actual function much of the time.

It is possible to construct an interpolatory model from a non-interpolatory one by fitting the error between the model and the actual function with an interpolatory approximation and using the non-interpolatory model of the objective plus the interpolatory model of the error. This is a useful technique when the non-interpolatory model is a reduced-fidelity solution of a set of governing equations or some other approximation that cannot be made to fit a specific set of data. The error model has specific behavior requirements that are easier to satisfy than the requirements for a general model. The accuracy needs to be highest near the current point. Further away, the accuracy need not be as high.

Using only the assumption that the model interpolates the actual function value at the current iterate, it is possible to use the trust region idea to define a subclass of model management algorithms that satisfy Theorem 2.3. To achieve this, the definitions of the `SEARCH` and `MANAGEMODEL` algorithms are modified to include additional conditions, and a particular implementation of `TERMINATE` is specified. The strategy for invoking `POLL` handles the case where the model consistently fails to produce a good solution to the optimization subproblem.

A precise specification of the algorithm in terms of the Model Management Framework components follows.

Let ρ represent the trust region radius. Assume that ρ is initialized at Step 1 of the Model Management Framework (Figure 2.4) so that it satisfies $\Delta_k < \rho < \infty$. This initial condition guarantees that there will be at least one step from $\Delta_k\mathcal{S}$ that is inside the trust region. We want the behavior of the algorithm to satisfy Definitions 2–3.

Hypothesis 8 Given an iterate x_k, the set of possible trial steps $\Delta_k\mathcal{S}$, and $\Delta_k < \rho < \infty$. Let

$$\delta = \min_{s \in \mathcal{S}} \|s\|.$$

Then

$$\{s_{+_i}\} = \texttt{SEARCH}(x, \Delta_k\mathcal{S}, \rho, M, \ldots)$$

must satisfy:

$$s_{+_i} \in \Delta_k\mathcal{S} \ \textbf{ and } \ \|s_{+_i}\| \leq \rho\delta, \quad \forall i,$$

and for bound constrained problems must also satisfy:

$$x + s_{+_i} \in \Omega, \quad \forall i.$$

Hypothesis 9 Given a model M of the objective function F,

$$M_+ = \texttt{MANAGEMODEL}(M, F, x_k, s_+, \ldots)$$

must be a model of F and must satisfy

$$M_+(x_k + s_+) = F(x_k + s_+).$$

Given component algorithms with this behavior, `TERMINATE` is implemented as follows:

TERMINATE(j, s_+, ρ, Δ_k)

1) if ($s_+ = 0$)

2) $\quad \rho \Leftarrow \rho/2$

3) if ($\rho < \Delta_k$)

4) $\quad$ return $True$

5) else

6) $\quad$ return $False$.

In this context, the SEARCH function serves one of the purposes that solving the optimization subproblem serves in the trust region context: to produce a trial step in the trust region at which the actual function is likely to decrease. Note that Hypothesis 8 does not require finding a minimizer of the model or even producing decrease in the model. All it does is restrict the steps from which SEARCH may choose to those within the trust region. No restrictions are placed on the method used to choose these steps.

The trust region constraint can be applied to any of the approaches to SEARCH described in §3.2.1. Since the trust region defines a bounded domain, it also makes it possible to implement SEARCH using some of the wide variety of sampling methods that have been developed in the statistics community that require a bounded region [56] [74]. Once a representative sampling of points from the trust region has been defined, the model values at the points can be used to select trial steps. These alternatives to optimizing the model may be preferable if the model is highly nonlinear (many local minima) or has other characteristics that make for difficult optimization problems.

The simplest approach to using sampling in SEARCH is to evaluate the model at all points in the sample and select the point(s) with the lowest value(s). Depending on the choice of sampling method it might be necessary to modify the point(s) to satisfy the requirement that the results from `SEARCH` be from the set $\Delta_k \mathcal{S}$ (Definition 2). Other approaches might use different selection criteria besides lowest model value. For example, if the values of the actual function will be used to modify the model, it may be preferable to select points where the reliability of the model is most in doubt [75]. The two criteria are combined in an approach known in statistics as "expected improvement" [54, 61].

The `MANAGEMODEL` definition for this algorithm fulfills the interpolation requirement. The model must interpolate the actual function only at the latest trial iterate $x_k + s_+$. Note that this is not necessary in a rigorous sense. The only behavior that is rigorously necessary is the finite termination of the loop in Steps 2–8 of MMFEM, which follows from Definition 4 of `TERMINATE`. However, without the interpolation property the analogy with the trust region method breaks down, and it is unlikely that the algorithm would be efficient in practice. It is the guarantee of interpolation and the local accuracy that it implies that supports the argument that reducing the trust region radius increases the likelihood of finding decrease in the actual function. Without additional knowledge of the model we cannot guarantee the model solution will improve. However, we still can guarantee the method will converge because the `POLL` algorithm is executed if no decrease in the actual objective is found.

The interpolation property and the use of the trust region radius allow us to satisfy the finite termination condition from Definition 4. Given this we can prove convergence by appealing to Theorem 2.3.

Theorem 3.1 Assume Hypotheses 3, 4, 8 and 9, Definition 5, and the conditions on F from Theorem 2.3 are all satisfied. Let $q(x)$ be defined as in Equation 2.3. Then any Model Management Framework algorithm using the implementation of `TERMINATE` given above produces a sequence of iterates $\{x_k\}$ that satisfies, in the unconstrained case,

$$\liminf_{k\to\infty} \|\nabla F(x_k)\| = 0$$

and in the bound constrained case,

$$\liminf_{k\to\infty} \|q(x_k)\| = 0.$$

Proof:

- Hypothesis 8 satisfies Definition 2
  - The only difference between Hypothesis 8 and Definition 2 is the additional restriction based on the trust region radius, so any step that satisfies Hypothesis 8 satisfies Definition 2.
- Hypothesis 9 satisfies Definition 3
  - The conditions in Hypothesis 9 are identical to Definition 3.
- The implementation of `TERMINATE` satisfies Definition 4
  - It is assumed that the level set of $F(x_0)$ is compact and F is continuous. By construction $F(x_k) < F(x_0)$ so the level set of $F(x_k)$ must be bounded.
  - By Hypothesis 4, the set $\mathcal{S}$ forms a regular grid of points with spacing $\Delta_k > 0$, so the number of points $x_k + s$ for $s \in \Delta_k\mathcal{S}$ that lie within the bounded level set $F(x_k)$ is finite.
  - Thus the number of points $x_k + s$ for $s \in \mathcal{S}$ that satisfy $F(x_k + s) < F(x_k)$ must be finite
  - At Step 4 in MMFEM, s_j is nonzero only if it produces decrease in F. Since there are only a finite number of points which satisfy this requirement,

there must be an iteration $j = n$ for which s_j does not produce decrease and thus is zero.

- Therefore, the number of times `TERMINATE` is evaluated with $s_j \neq 0$ is finite.
- By definition $\Delta_k > 0$ and $\rho < \infty$ so the number of times `TERMINATE` is evaluated with $s_j = 0$ before it returns $True$ is finite.

Therefore, Definitions 2–5 are satisfied and Theorem 2.3 holds.□

Both of these algorithms are presented as examples of what can be done with the Model Management Framework. The key point is the straightforward way the theoretical requirements impact the algorithm. This greatly simplifies the task of designing extensions to the algorithms that improve their behavior while maintaining the convergence property.

Chapter 4

Numerical Results

In this chapter, results are presented for three test problems. The Model Management Framework is used with the component algorithms described in §3.3.1. These tests are intended primarily to demonstrate the effectiveness of the Model Management Framework and the various issues that arise in using the Framework in practice. The results are not intended to demonstrate the best possible performance on these particular test problems. The component algorithms used, particularly the algorithm for `MANAGEMODEL`, are simpler than would be used for a "real" problem and probably do not perform as well as is possible. The simple algorithm design was chosen because it is easier to implement and understand. Despite this caveat, the performance on these test problems is quite respectable.

The first test problem is a simple two-dimensional problem drawn from the statistics literature on design of computer experiments. The objective function is trivial to compute. Results are compared for two types of interpolatory models using different values for some of the parameters present in the model management algorithm. This test was chosen because of its simplicity and the ability to visualize the results.

The second test problem is another simple objective function, but in six dimensions. This problem is drawn from the global optimization literature. The objective function is more complex than in the first test case, but still costs almost nothing to compute. This test is simple enough that a large number of test runs could be performed, but complex enough to capture some of the characteristics of the much more difficult optimization applications that are the real targets of this methodology. As in the first test, the different test runs correspond to different settings of the algorithm control parameters with the objective function and initial point held constant.

The third test problem is drawn from a practical application in engineering design provided by our collaborators at Boeing. This test is presented because it is indicative of the type of application the Model Management Framework is intended to target. Because of the much higher cost of solving this problem, only a few runs were performed.

It must be noted that very little detailed information about the objective function or the models used is presented; certainly not enough to replicate the results. Some of the data and software used in this test was developed at The Boeing Company and is of some commercial importance. We respect Boeing's proprietary interests in this software and data and have chosen not to reveal details that might lessen any competitive advantage related thereto. We think this lack of detail does not detract from the value of this problem as an example of the general issues involved in the design and use of model management methods.

4.1 MMF algorithm implementation details

The MMF component algorithms described in §3.3.1 have several control parameters that must be chosen in order to execute the algorithm. These parameters specify details of initialization, execution and termination of various parts of the model management algorithm and the pattern search method that encompasses it.

The basis matrix used for all three problems is the identity matrix. The corresponding set of basis vectors contains the positive unit coordinate vectors. The core pattern used corresponds to the maximal positive basis. The corresponding set of core steps contains the positive and negative unit coordinate vectors. This core pattern was chosen for two reasons: all three problems include bounds on the variables, so the use of the maximal positive basis is a simple means of satisfying the theoretical requirement for bound constrained problems. In real applications more consideration likely will be given to the choice of core patterns, as the choice can have a significant impact on the efficiency of the method.

The approach implemented for setting and adjusting the pattern scale factor Δ_k uses three control parameters to specify the initial value of Δ_k, the ratio by which Δ_k is contracted whenever the `POLL` operation fails, and the value of Δ_k at which the Framework terminates. (Other termination criteria will be discussed later.) Additionally, the Model Management Framework implements several methods for expanding the scale factor after a successful step, and the choice of method is specified via another control parameter. All four of these control parameters are varied in some or all of the test problems.

Other aspects of the algorithm that can be varied are the type of model and the initial model. Two methods were used to generate the models. One method involves fitting a variable-order multivariate polynomial to the given data [19] as implemented by Grandine [34]. The number of terms in the polynomial and its order increases with the number of data points to fit. The more data given, the more terms in the polynomial and the higher the order of the polynomial. The other method uses a best linear unbiased predictor (BLUP) model as discussed in [74] and implemented in the DACEPAC software by Booker [4]. We will henceforth refer to these models as polynomial and DACE models, respectively. The software for both models is implemented to the same interface specification, so the models can be substituted for each other without changes to the model management software. It is likely that existing implementations of other models could be substituted without significant coding effort.

A DACE model M of a function F has the general form:

$$M(x) = \widehat{\beta} + r(x)^T R_d^{-1}(F(x_d) - \widehat{\beta}\mathbf{1}),$$

where $r(x)$ is the vector of values of the correlation function of x with all the data points used to build the model, R_d is the matrix of values of the correlation function of all the data points with each other, x_d is the set of data points at which the model has been fit, $F(x_d)$ is the vector of objective function values at x_d, $\widehat{\beta}$ is a scalar constant related to the mean value of $F(x_d)$ and $\mathbf{1}$ is a vector composed of all 1s..

In the DACE models, the correlation function is defined as

$$\rho(x, w) = \prod_{j=1}^{n} e^{-\theta_j |x_j - w_j|^2}.$$

Thus the entries of the correlation vector $r(x)$ are defined by

$$r_i(x) = \rho(x, x_{d_i}),$$

and the elements of the matrix R_d is defined by

$$R_d(i, j) = \rho(x_{d_i}, x_{d_j}).$$

This matrix is symmetric positive definite with unit diagonals. The scalar $\widehat{\beta}$ is defined by

$$\widehat{\beta} = (\mathbf{1}^T R_d^{-1} \mathbf{1})^{-1}\ \mathbf{1}^T R_d^{-1} F(x_d).$$

To construct an initial DACE or polynomial model, the objective function must be evaluated at a set of points before starting the model management process. The cost of these evaluations and the cost of constructing the models is an overhead cost. For both types of models the major cost of construction (aside from the function evaluations) is that of computing the coefficients in the model. In the polynomial model this requires a singular value decomposition of a matrix of order the number of terms in the polynomial. In the DACE model the coefficients θ are computed by solving an optimization problem with the coefficients as the unknowns. Thus, the DACE models are generally more expensive to construct than the polynomial models. In practice the DACE models tend to be less expensive to evaluate than the polynomial models. The computational cost was not an issue in the choice of models to use in the test problems.

There are other uses to which the model(s) can be put, so it may be possible to reuse the model and/or the function data outside of the Model Management Framework. Some of these uses in an engineering design context are discussed in [6, 7].

It should be reemphasized that the Model Management Framework is not limited to the two types of models used in the test problems, nor is it limited to interpolating models. See [2, 65] for other choices.

4.2 Test function #1

This function was used by Welch, et al. in [74]. It is defined on $\mathbf{R}^2$, with two minimizers in the interval $[0:5]^2$. The function is defined by:

$$F(x) = (30 + x_1 \sin(x_1))(4 + e^{-x_2}).$$

The primary purpose of this test was algorithm and code validation. It is not representative of the behavior of the models and algorithms on realistic problems. Figure 4.1 is a contour plot of the actual objective function showing the global minimizer (*) and the points used to construct the initial models ($\oplus$). The contours are labeled with the associated function values.

Six data points were used to construct the initial DACE and polynomial models. An orthogonal array-based Latin hypercube sample [68] computed using Owen's `oa`

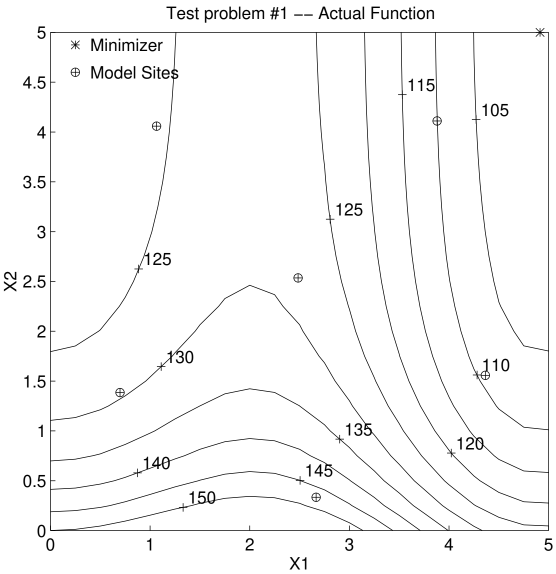


Figure 4.1 Contours of actual objective function for test problem #1.

software (available from STATLIB (`http://lib/stat/cmu.edu/designs`)) was used to select these points. The same points were used to construct both initial models. This may not seem like many points but it is indicative of the density of sampling that one can afford to do in higher dimensions on the expensive problems in which we are interested.

Figure 4.2 contains plots similar to Figure 4.1 for the initial DACE and polynomial models. The plots show clearly that both models capture the qualitative behavior of the actual function fairly well, although the polynomial model captures the location of the basin more accurately.

The actual function is non-convex, with local minimizers near the upper left and upper right corners of the bounded domain. Considering the small number of points used to construct the models, both models effectively serve the purpose of identifying a direction of descent in the actual function from most points in the domain. The major exception is that the DACE model mis-predicts the shape of the basin of the global minimizer in the upper right corner. This isn't necessarily indicative of the general behavior of polynomial and DACE models on other problems.

The starting point used for the optimization was $X = (2\frac{2}{3}, \frac{1}{3})$. This point was also used as one of the points in the initial models. The other five initial points are: $(3.88, 4.11), (4.364, 1.558), (2.485, 2.536), (0.699, 1.385) and (1.066, 4.059)$. Different initial points would produce different models.

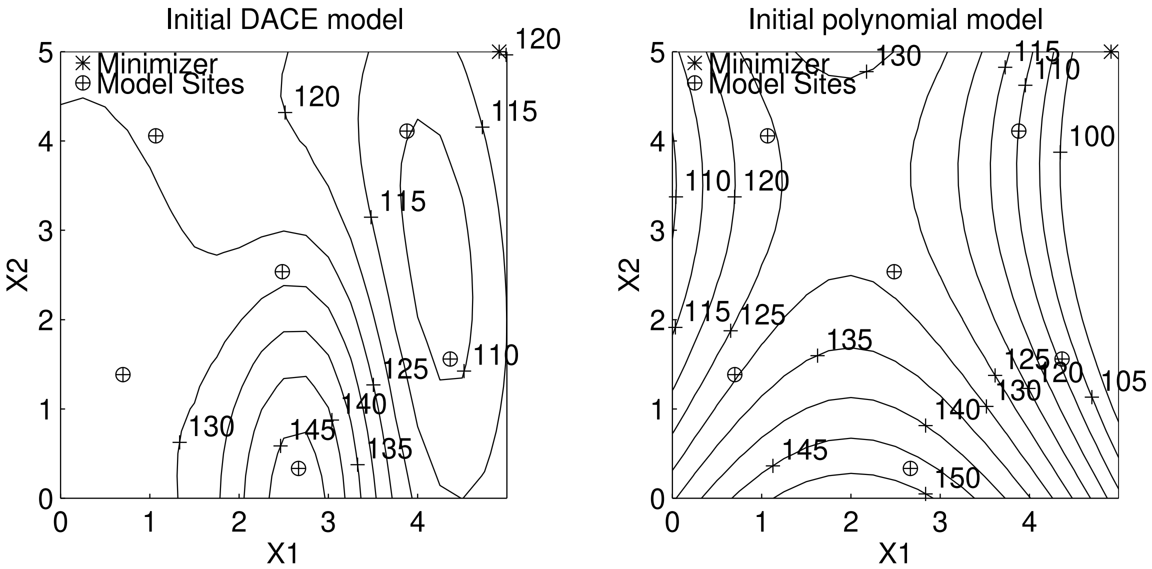


Figure 4.2 Contours of initial DACE and polynomial models for test problem #1.

Results are presented from two sets of test runs. In the first set, the model management algorithm is run once with each type of model and with the algorithm parameters set to default values. These two runs will be used for visualization purposes. In the second set of runs the algorithm parameters are varied across runs. These runs will be used to correlate the behavior of the Framework with the variations in algorithm parameters.

4.2.1 Visualization test

Figure 4.3 presents contour plots of the two sets of models at intermediate steps in the optimization. Each row of plots shows how the models change *after* the model is adjusted to fit the values computed during the iteration noted. The path taken by the optimization is also shown. These plots show how the models adapt to the actual function as they are updated with function values computed during the optimization process. Three plots for each type of model are presented, corresponding to the state of the computation at the completion of the 1st, 3rd and 5th iterations of the Model Management Framework. The path taken by the solution is shown by the lines marked with "$\times$" symbols. Each "$\times$" indicates an accepted intermediate step. Recall that the method only accepts steps when the value of the actual objective function decreases. The number of steps taken and the number of actual function evaluations performed in each iteration are not the same for the two runs. Also, some iterations did not produce nonzero steps. Thus the number of steps marked on the plots are different for the two runs. The global minimizer is marked by the "$\star$" symbol near the upper right corner. As before, the "$\oplus$" symbols mark the points used to construct the initial models.

The two types of models react differently even though both start from the same initial point and take the same first step. The solution using the polynomial model takes two steps in the first iteration compared to just one for the DACE model (top row of Figure 4.3), and stays one step ahead for the rest of the iterations.

Here we see the effect the quality of the model can have on the performance of the method. Both models take the same first step (to $x = (4\frac{2}{3}, 2\frac{1}{3})$). The initial pattern scale factor was 1, so this step corresponds to moving two pattern points in each direction. The run with the DACE model stops here because the initial model predicts the minimizer is close by (see Figure 4.2) and the method cannot find another

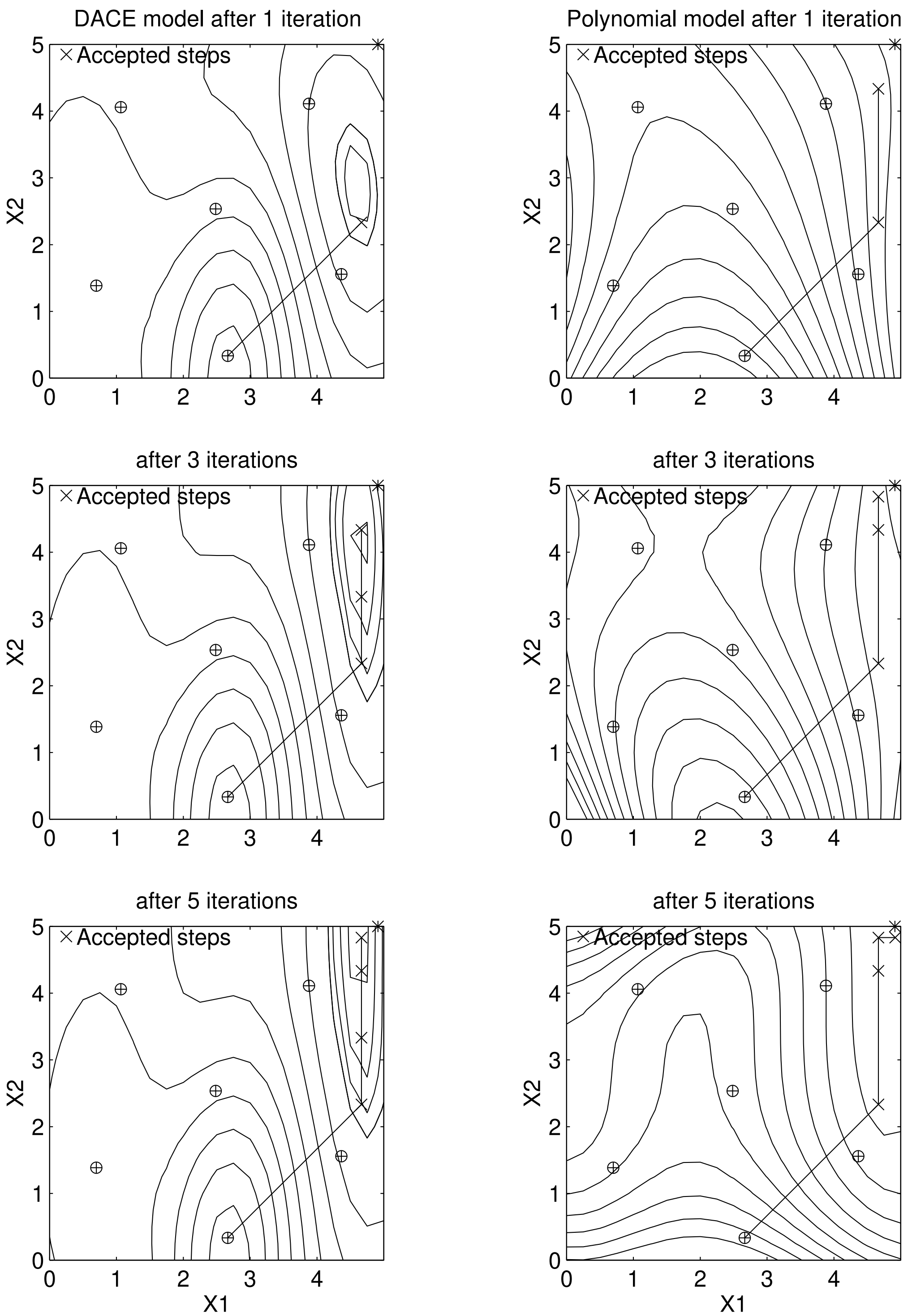


Figure 4.3 Contours of DACE and polynomial models at intermediate stages of MMF execution on Test Problem #1. (top) 1 iteration (middle) 3 iterations (bottom) 5 iterations.

step (in the pattern) that decreases the model value. The run with the polynomial model takes another step (to $x = (4\frac{2}{3}, 4\frac{1}{3})$) because the initial model captures the shape of the actual function better *in that particular part of the domain* (compare Figure 4.2 with Figure 4.1).

After the first iteration, the DACE model (on the left) still mis-predicts the location of the minimizer and the general shape of the basin of the minimizer. The polynomial model captures the shape of the basin much better.

In this iteration and the iterations that follow, the polynomial models change more than the DACE models. The only significant change in the DACE models is the basin of the global minimizer moves closer to the shape of the basin of the minimizer of the actual function. The polynomial models change significantly in the part of the domain away from the global minimizer. This exemplifies one of the fundamental differences between the DACE and polynomial models: the DACE models change locally in response to new data whereas the polynomial models change globally. One can easily imagine problems that reward either property.

The next two rows of Figure 4.3 show the steps accepted in the solutions and the models that result after 3 and 5 MMF iterations. In both cases the solution with the polynomial models is closer to the global minimizer of the actual function.

Although the solution using the polynomial models has a better solution after the 5th iteration, it took more function evaluations than the solution with the DACE models (15 vs. 9) because it selected trial steps that did not decrease the actual function more often than the solution with the DACE models. Running the Framework with the DACE models to the same number of function evaluations (15) took 8 iterations (instead of 5) and produced the same solution as the polynomial model. This shows that the polynomial model gets close to the solution in fewer iterations than the DACE model, most likely because initially it has a better approximation of the location of the basin of the solution. However, this more rapid approach to the solution leaves the polynomial model with less accuracy and thus it uses more function evaluations than the DACE model because it errs in predicting decrease.

Although this is a simple example, it demonstrates that success on a model management problem does not depend only on the initial accuracy of the model. It should be stressed that no general conclusions can be drawn about the behavior of these types of models from this particular problem.

4.2.2 Parameter variation test

The second set of tests investigates the effect the various parameters in the model management algorithm have on performance. Six algorithmic parameters were varied during these tests. These parameters are described below.

Model Type The two different model types (DACE, polynomial) used in the previous test were also used for this test.

Initial Pattern Scale Factor The pattern scale factor Δ_k determines the spacing between the steps in the pattern. Trial iterates are chosen from the pattern. A larger scale factor allows longer steps to be taken. A smaller scale factor implies a denser sampling of the problem space. Since the algorithm converges by reducing the scale factor, a smaller initial scale factor risks premature convergence and a larger initial scale factor takes longer to converge. Four initial scale factor values were used: 1, 1/2, 1/4 and 1/8.

Final Pattern Scale Factor The program terminates when the pattern scale factor Δ_k reaches this value. Three values were tested: 1/16, 1/64 and 1/256. The Framework algorithm reduces the scale factor when a `POLL` fails.

Pattern Scale Contraction Ratio The pattern search convergence theory described in Chapter 2 requires that the pattern scale factor be reduced by a constant whenever it is reduced. The contraction ratio defines this constant. A smaller contraction ratio reduces the pattern scale factor more quickly, and generally leads to smaller steps being taken. Ideally, a smaller ratio would cause the method to converge more quickly, but it may also cause premature convergence, requiring more iterations. Three contraction ratio values were used: 1/2, 1/4 and 1/8.

Pattern Update Method The Model Management Framework supports several methods for updating the pattern scale factor in iterations of the pattern search that are successful in finding decrease in the objective function (line 7 of method GPS in Figure 2.1). Five different methods were tested:

1. the pattern is never expanded
2. the pattern is expanded by 1/pattern_scale_contraction_ratio
3. same as 2 except a contraction that follows an expansion is taken without invoking `POLL`. This reduces the cost without affecting the theoretical behavior.[††]
4. same as 2 except the pattern is not expanded when the step with decrease was found by `POLL`. This prevents unwarranted expansion of the pattern.
5. same as 3 except the pattern is not expanded when the step with decrease was found by `POLL`.

[††]An expansion/contraction pair can only occur a finite number of times before a solo contraction or expansion happens, so as far as the convergence theory is concerned the paired expansion and contractions cancel out and are equivalent to a successful step taken at a constant scale factor.

Function Evaluation Limit This limits the number of times the actual objective function can be evaluated on the steps computed by `SEARCH`. In other words, it limits the size of the set of trial steps returned by `SEARCH`. This limits the cost of evaluating the steps to determine if one is acceptable. In the model management algorithm used in the tests, this value also affects when `POLL` will be invoked. Three limit values were tested: 1, 2, and 4 (i.e. $1, n$ and $2n$).

Model Interpolation Error In the DACE model only, a small amount of error in the interpolation of the model to the actual function values can be introduced to improve the condition number of the correlation matrix used to compute the model values. Increasing this error tolerance tends to lower the condition number of the correlation matrix associated with a given set of sample points, reducing the likelihood that the matrix will become singular.

The `SEARCH` algorithm uses the interpolation property to guarantee that a step that is rejected once is never selected again. Without it, the model may continue to predict decrease at a point even after the actual function value has been computed. If this happens and the point is selected again as a trial step by `SEARCH`, the Framework will reject the step again. Either another step will be accepted or `POLL` will be invoked. Thus, relaxing the interpolation requirement improves the conditioning of the DACE model at the risk of an increase in computational cost.

The Model Interpolation Error is also known as "measurement error" in the literature on response surface models [25, 54]. Three error values were tested: 0, 10^{-6} and 10^{-3}.

A total of 2160 runs were performed for this problem. Over all the runs, only 6 different results were computed. This is not surprising: the basin of the global minimizer is narrow and the number of grid points "near" the global minimizer is small (near in pattern spacing), both because the problem is only 2D (at most 9 grid points within one "unit" of pattern distance) and because the solution is on the boundary (eliminating 3 or 4 grid points, depending on pattern spacing). The small number of unique results is a positive outcome in that it shows the algorithm always reaches nearly the same answer regardless of the algorithm parameter settings.

In the discussion that follows, three measures of performance will be considered: the accuracy of the final objective function value; the number of function evaluations; and the number of executions of the `POLL` routine. The first two are the primary measures. The third indirectly measures how effectively the model is being used. If `POLL` is invoked frequently, the model is not helping very much. Also, in higher dimensional problems, `POLL` is likely to be the most expensive operation in the algorithm.

Table 4.1 summarizes the results of the test runs. The column labeled "Occurrences of Sol'n" contains the number of runs that produced each solution.

Sol'n	$\|\|X - X_*\|\|$	$f - f_*$	Occurrences of Sol'n	X
(1)	4.352×10^{-4}	5.695×10^{-4}	418	(4.916667, 4.997396)
(2)	1.098×10^{-3}	1.903×10^{-3}	721	(4.916667, 4.989583)
(3)	2.627×10^{-3}	4.604×10^{-3}	1	(4.916667, 4.973958)
(4)	4.181×10^{-3}	7.347×10^{-3}	780	(4.916667, 4.958333)
(5)	1.667×10^{-2}	3.090×10^{-2}	180	(4.916667, 4.833333)
(6)	2.976×10^{-2}	6.427×10^{-1}	60	(4.666667, 4.833333)
Total			2160	
Global solution: $f_* = 100.01182; X_* = (4.91318, 5.0)$				

Table 4.1 Results of 2D visualization test runs.

As would be expected, the accuracy of the final objective values correlates well with the choice of the final pattern scale factor parameter. Table 4.2 shows the number of occurrences of each of the 6 different solutions for each of the 3 values of the final pattern scale factor parameter. Reducing this value generally leads to better solutions.

	Sol'n $f - f_*$	Occurrences of Sol'n			
		Final Scale Factor			Total
		1/16	1/64	1/256	
(1)	5.695×10^{-4}	0	0	418	418
(2)	1.903×10^{-3}	0	420	301	721
(3)	4.604×10^{-3}	0	0	1	1
(4)	7.347×10^{-3}	480	300	0	780
(5)	3.090×10^{-2}	180	0	0	180
(6)	6.427×10^{-1}	60	0	0	60
Total		720	720	720	2160

Table 4.2 Results of 2D visualization test for different termination criteria.

Variations in the model have no effect on the results. The polynomial model and the three DACE models that were tested have essentially identical occurrences of each of the five solutions that occur more than once. (The only variation is the single occurrence of solution (3).) The performance varies across models (in terms of number of function evaluations), but the same solutions are produced. This strengthens the observation that the algorithm responds very predictably to the final pattern size termination condition.

We look next at performance in terms of number of actual function evaluations. This measure is used because it is assumed that the cost of actual function evaluations dominates the cost of the other parts of the algorithm, including searching for trial steps using the model. Though this is not the case for this test problem because the actual function is trivial, it is assumed to be true in the problems for which this method was designed.

Although the many runs produced only 6 different solutions, they took many different paths, and this is reflected in the number of function evaluations used by each run. Table 4.3 summarizes this data. For each solution, the minimum, maximum and median number of function evaluations used in the runs that produced that solution are given. The results are separated into the type of model used. Recall that solution (3) occurred only once. The solution numbers are the same as in Table 4.2.

Sol'n	Number of Function Evaluations											
	Polynomial			DACE error 0			DACE error 10^{-6}			DACE error 10^{-3}		
	Min	Max	Med	Min	Max	Med	Min	Max	Med	Min	Max	Med
(1)	15	55	32	21	50	32	23	50	32	33	100	46
(2)	11	45	25	18	44	24	19	44	25	26	100	39
(3)	0	0	0	0	0	0	0	0	0	100	100	100
(4)	7	35	18	14	37	18	15	37	18	19	78	27
(5)	7	22	16	14	31	15	14	31	15	16	42	22
(6)	8	16	12	11	20	11	11	20	11	13	28	13

Table 4.3 Results of 2D visualization test for different models.

There are several trends that are clearly represented in this data. The less accurate solutions took fewer function evaluations. The interpolating polynomial and the interpolating DACE model (error 0) have very similar behavior. The polynomial model performs slightly better for low accuracy solutions and sometimes performs better at high accuracy (as exhibited by the lower minimum number of evaluations for solutions (1) and (2) and nearly equal median and maximum). The contour plots of the actual function and the initial models (Figures 4.1 and 4.2) show that the polynomial model captures the behavior of the basin of the global minimizer (X=4.91,5.00) more accurately than the DACE models (the effect of the interpolation error is too small to be seen so the same plot applies to all three DACE models). This advantage will be particularly significant for the less accurate solutions, since the algorithm will be able to get into the neighborhood of the minimizer with less work using the polynomial model. In the more accurate solutions the model management process has sufficient information (i.e. function values) to adapt both models to the shape of the basin of the minimizer, thus explaining the similarity in performance in those cases.

Table 4.4 shows the results of varying the initial pattern scale factor parameter. The table contains the number of occurrences of each unique solution for the four values of the parameter that were tested.

Sol'n	$f - f_*$	Occurrences of Sol'n				
		Initial Scale Factor				Total
		1	1/2	1/4	1/8	
(1)	5.695×10^{-4}	60	178	60	120	418
(2)	1.903×10^{-3}	180	121	240	180	721
(3)	4.604×10^{-3}	0	1	0	0	1
(4)	7.347×10^{-3}	240	180	120	240	780
(5)	3.090×10^{-2}	60	0	120	0	180
(6)	6.427×10^{-1}	0	60	0	0	60

Table 4.4 Results of 2D visualization test for different initial conditions.

The results for solution (6) are the most easily explained. All 60 runs correspond to the same combination of parameters: initial scale factor = 1/2, final scale factor = 1/16 and contraction ratio = 1/8. This combination of settings guarantees the program will terminate after the 2nd contraction occurs because the first contraction reduces the scale factor from the initial value to the final value. This pattern repeats for all the other cases. There are 36 different combinations of (initial scale factor, final scale factor, contraction ratio) giving rise to 2—9 contractions. There are 60 test runs for each combination. Variations in model type, initial model size and pattern update method account for the 60 runs. All 60 runs for a given combination produce the same solution.

Figure 4.4 shows the different solutions plotted against the number of function evaluations for all the test runs. Each model is plotted separately for each solution. The plot shows that the polynomial models ($\circ$ symbols) tend to perform better than any of the DACE models. Among the DACE models, the performance of the interpolating models ($+$ symbols) and the non-interpolating models with the smaller error ($*$ symbols) are nearly identical, while the models with the larger error ($\times$ symbols) perform worse in many cases.

Figure 4.5 shows the same plot as Figure 4.4 except the results are separated by the pattern update method used in the run. The performance of methods 1, 3 and 5 is very similar and is somewhat better than the performance of methods 2 and 4. Update methods 2 and 4 share the common characteristic that once the pattern scale factor is increased, `POLL` must be executed in order to reduce it. The other update methods do not have this characteristic. Method 1 never increases the scale factor, and methods 3 and 5 allow the scale factor to be reduced without executing `POLL` only if the reduction cancels a previous increase.

If the pattern is expanded too much, `SEARCH` may fail to find acceptable trial steps, which causes `POLL` to be invoked. This is one potential source of the extra function evaluations. Another possibility is that the larger pattern scale factors encourage `SEARCH` to select trial steps that are further from the current iterate, where presumably the model is more likely to predict decrease incorrectly. In either case, the source of the poor performance is the pattern scale factor being increased too much.

The motivation for update method 3 is to avoid the `POLL` cost associated with decreasing the pattern scale factor after an increase due to a successful trial step.

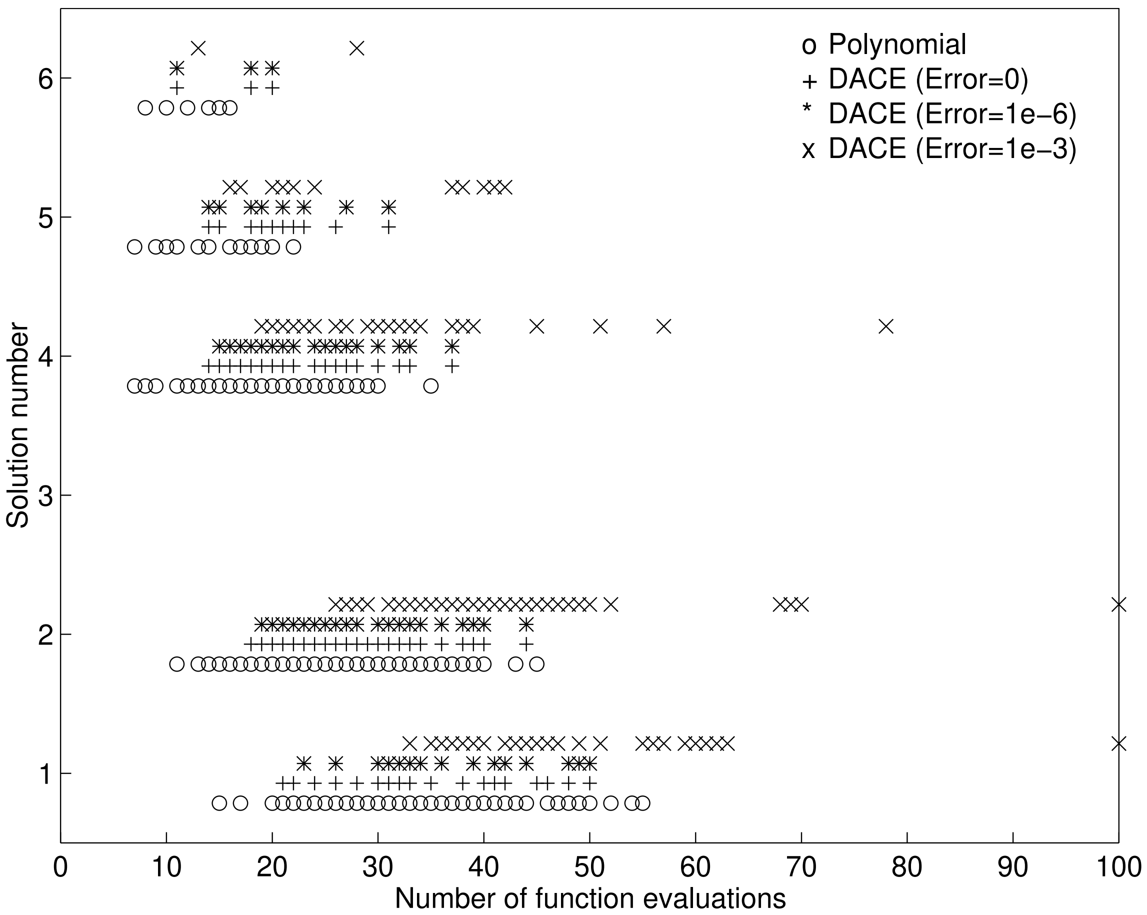


Figure 4.4 Function evaluation counts for all models and solutions for test problem #1. Solution accuracy decreases with increasing solution number.

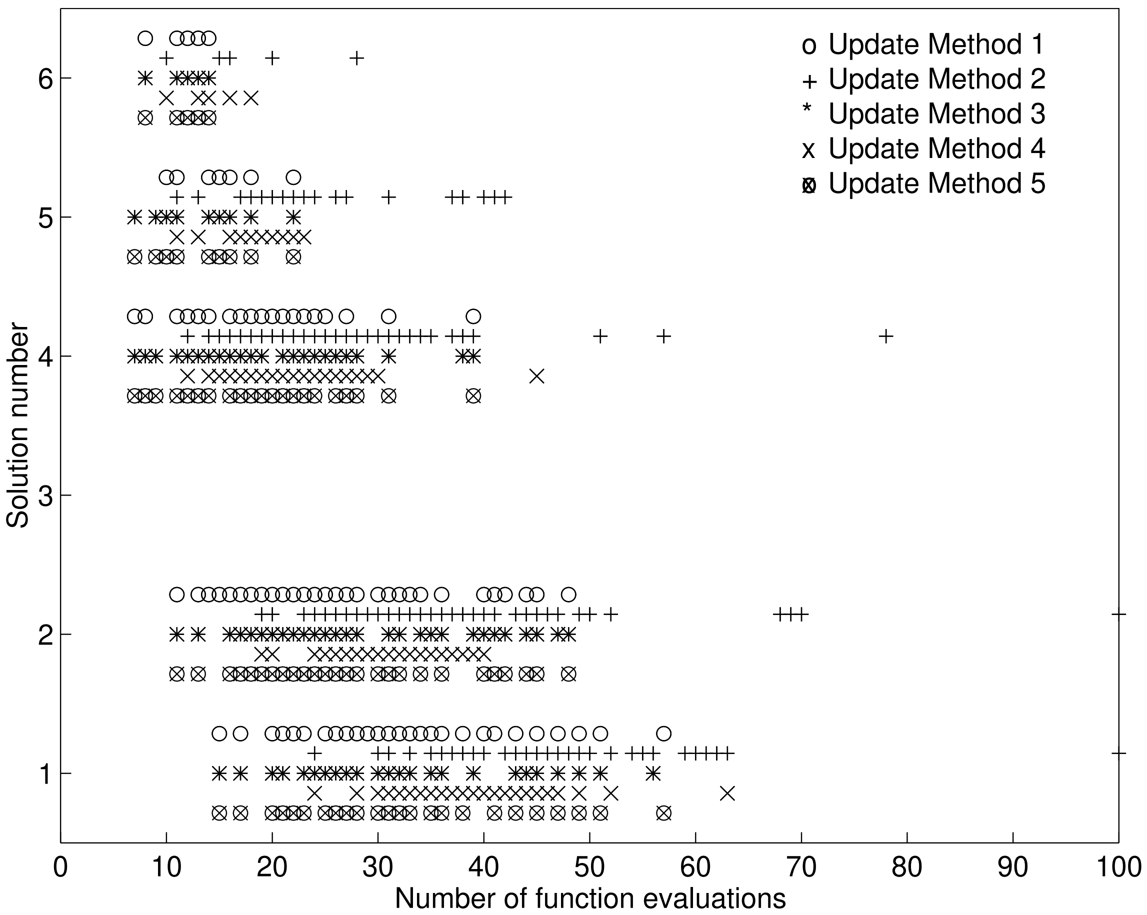


Figure 4.5 Function evaluation counts for all pattern update methods and solutions for test problem #1.

Update method 5 tries to avoid some of the `SEARCH` failures associated with larger pattern scale factors by not expanding the pattern scale factor after a successful `POLL`.

A further modification to the pattern update method would be to use the trust region-inspired approach and only expand the pattern scale factor when the ratio of the decrease in the objective function predicted by the model and the actual decrease is high. However, since this does not guarantee the accuracy of the model, this heuristic is not guaranteed to produce better results.

4.3 Test function # 2

The second test problem is taken from the global optimization literature [24], where it is known as the Hartman function. It is a six-dimensional problem with four local minima. The global minimizer is unique.

The algorithm parameters that were varied for this test include all those used in Test #1 except the final pattern scale factor which was kept constant. For these test runs termination also depends on a limit on the number of actual objective function evaluations. In addition to the algorithm parameters from Test #1, the number of data points used to construct the initial models were also varied.

Three initial models of each type were constructed, using 16, 32 and 64 data points. As in Test #1, orthogonal array-based Latin hypercube samples were used to select the data points. In each case, the DACE and polynomial models were constructed from the same set of data points.

Several values for each algorithm parameter were selected and the Framework was run for all possible combinations of these values. A limit of 100 function evaluations was set for each run. This was deemed high enough to separate successful parameter settings from unsuccessful ones.

The purpose of running tests for all the parameter values was to see which ones would perform well and to understand the reasons for the performance, both good and bad. Real applications of model management will be too expensive to run several solutions just to find a good set of algorithm parameters. Ideally, we would like to be able to justify a particular set *a priori*. More realistically, we would like to be able to make informed choices for the parameter values.

The parameter values used in the tests are given below. The meanings of the parameters are given in §4.2.2.

- Model Types: Interpolating polynomial; DACE with InterpError = 0, 10^{-6}, 10^{-3}.
- Model Size: 16, 32, 64.
- Initial Scale Factor: 1/2, 1/4, 1/8.
- Final Scale Factor: 1/16, 1/64, 1/256.
- Scale Factor Contraction Ratio: 1/2, 1/4, 1/8.
- Function Evaluation Limit: 1, 3, 6, 12 (i.e. 1, $n/2$, n, $2n$).
- Pattern Update Method: 1, 2, 3, 4, 5

Every combination of parameter values was tested. A total of 6480 test runs were performed.

The problem domain is bounded by $[0:1]^6$. The starting point for all the runs was the midpoint of the domain: $X = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$.

The 6D Hartman problem is non-convex with four local minimizers. Therefore the first measure of performance to investigate is which runs found the global minimizer. Figure 4.6 plots the l_2 norm distance of each solution from the global minimizer for all the test runs. The error in the final objective function value increases along the horizontal axis.

The three models are built from three independent sets of points. Each is an orthogonal array-based Latin hypercube sample. The larger sets do not necessarily contain the points of the smaller sets. Each is designed to cover the space as uniformly as possible with the given number of points.

At least two distinct groups of results are evident in this plot. The solutions with values of $||X - X^*|| < 0.5$ are all close to the global minimizer. The solutions with $||X - X^*|| > 1.0$ are closer to one of the other three local minimizers. Figure 4.7 plots the same results with the normalized error in the final objective $(|f - f^*|/|f*|)$ on the vertical axis. This clearly shows the clustering of results.

Of the 6480 runs, 3385 (52%) are in the cluster close to the global minimizer. This is good performance considering this method has no algorithmic features specifically designed to solve the global optimization problem. The behavior arises as a side-effect of the use of a pattern search method in the `SEARCH` strategy (see §3.3.1).

Removing from consideration the results in the basin of the global minimizer and computing the distance of the remaining results from the second best minimizer produces the data plotted in Figure 4.8. Referring back to Figure 4.6, it seems fair to conclude that essentially all of the solutions that are not close to the global minimizer are close to the second best local minimizer. Figure 4.9 plots the relative difference in the objective function from the second local minimum value for these solutions. It shows that most are within 10% of the minimum value. The other solutions typically terminated because the limit on the number of function evaluations was reached.

These results show that the algorithm works correctly over a wide range of input parameter settings. The efficiency of the algorithm (measured in number of function

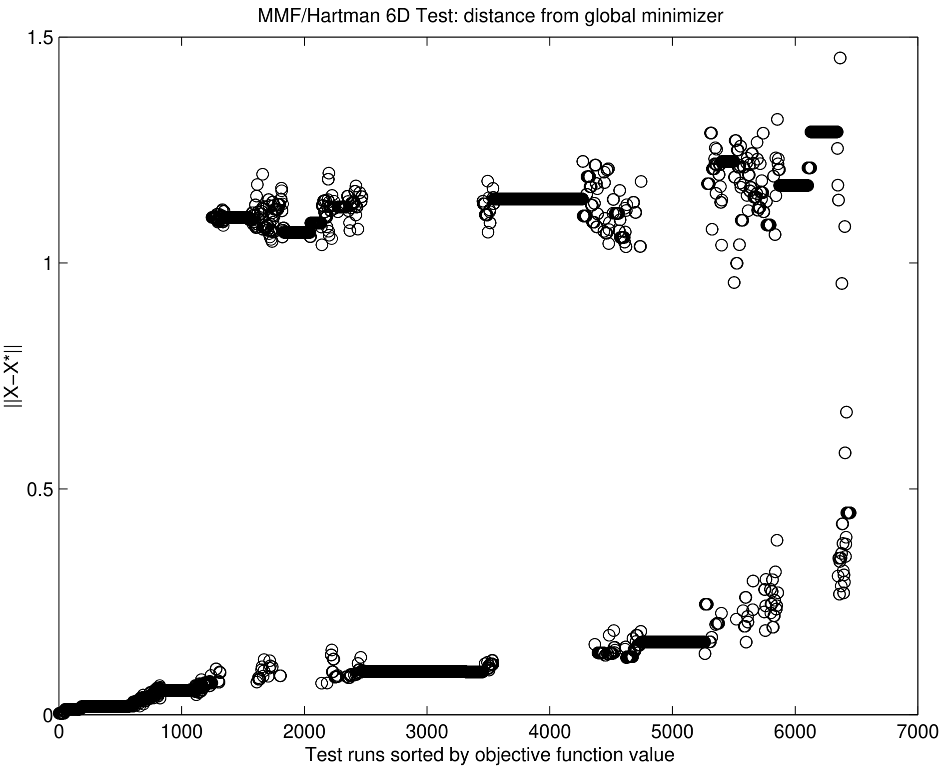


Figure 4.6 Plot of distance from global minimizer for each solution for test problem 2. Solutions are sorted by increasing objective value.

evaluations) varies a great deal, but even the runs that were inefficient came fairly close to one of the local minimizers.

A very small number of runs failed. These failures were all due to the inability to compute a new model that fits all the data values. There were 31 runs (0.5%) that terminated without producing a result. All used interpolatory models (6 runs used polynomial models and 25 used DACE models with zero interpolation error). The software that constructs the models (both polynomial and DACE) can fail if the data it is trying to fit is too ill-conditioned. This ill-conditioning can arise when some data points are much closer together than others. Also, the DACE software is sensitive to co-linearity in the data points. Unfortunately, these situations both tend to arise

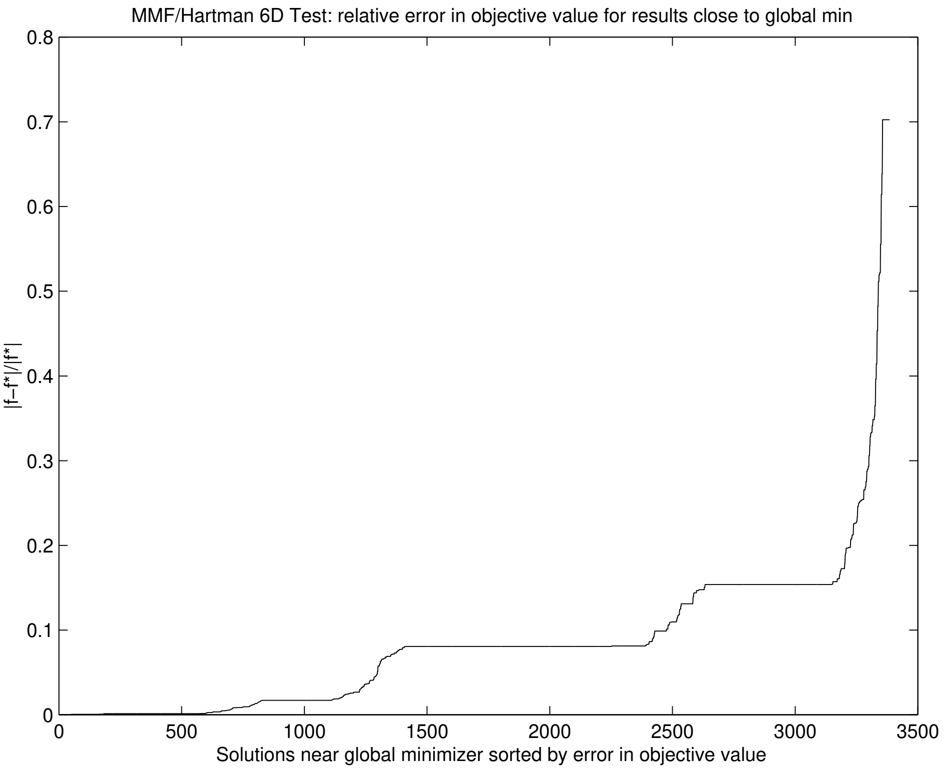


Figure 4.7 Plot of distance from the global minimizer versus the relative error in the final objective value for each solution for test problem 2.

in MMF runs. The grid spacing decreases as the algorithm progresses so the data points tend to get closer together. Also, the data points are taken from a regular grid, increasing the odds that points will be co-linear. It follows that this is likely to be a recurring problem in MMF methods. Considerable effort was expended on making the model fitting software (both DACE and polynomial) as robust as possible. More experimentation is needed to determine whether the low failure rate (0.5%) seen in this test is indicative of realistic applications.

Of the 6449 successful runs, 2257 (35%) terminated in less than the maximum of 100 objective function evaluations. This suggests that termination criteria are an important consideration in the Model Management Framework, as they are in most

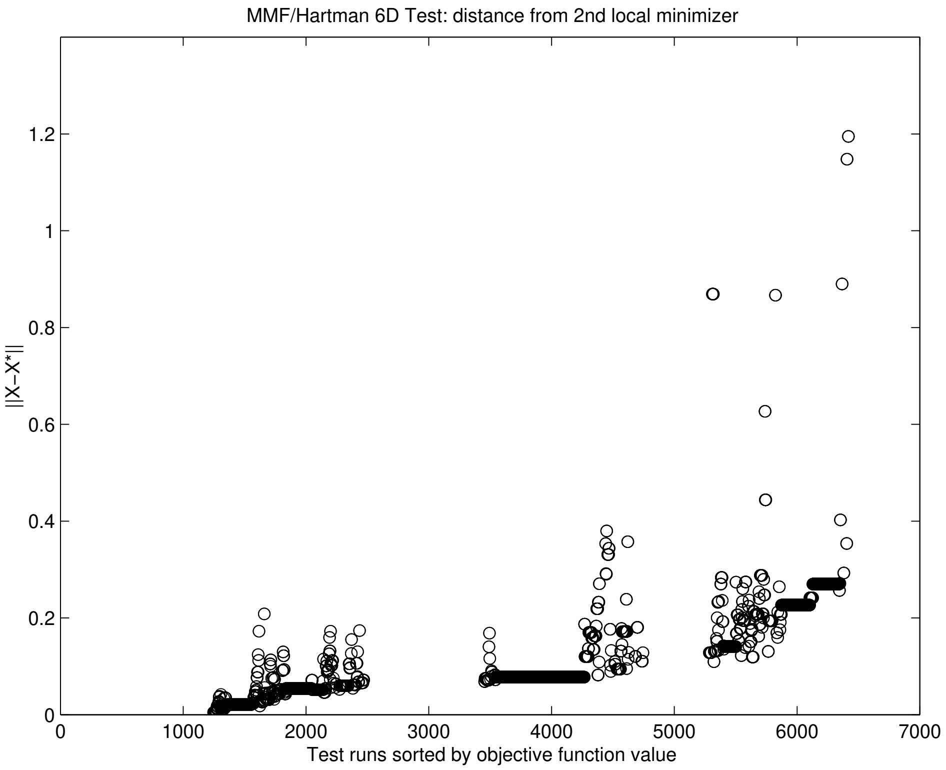


Figure 4.8 Distance from the second minimizer for each solution for test problem 2. Solutions are sorted by increasing objective value.

optimization methods. This is not surprising since the performance of pattern search methods in general is sensitive to termination criteria because they converge linearly once they are in the neighborhood of a solution.

Terminating on the number of function evaluations is a somewhat arbitrary way to separate the solutions. Since we are interested in which parameter settings produced the most efficient results, any method of filtering the more efficient solutions from the less efficient would be valid. In particular, the number of MMF iterations is another reasonable criterion, especially in an implementation of MMF on a parallel computer where the effective cost of the function evaluations will vary depending on the number of processors used.

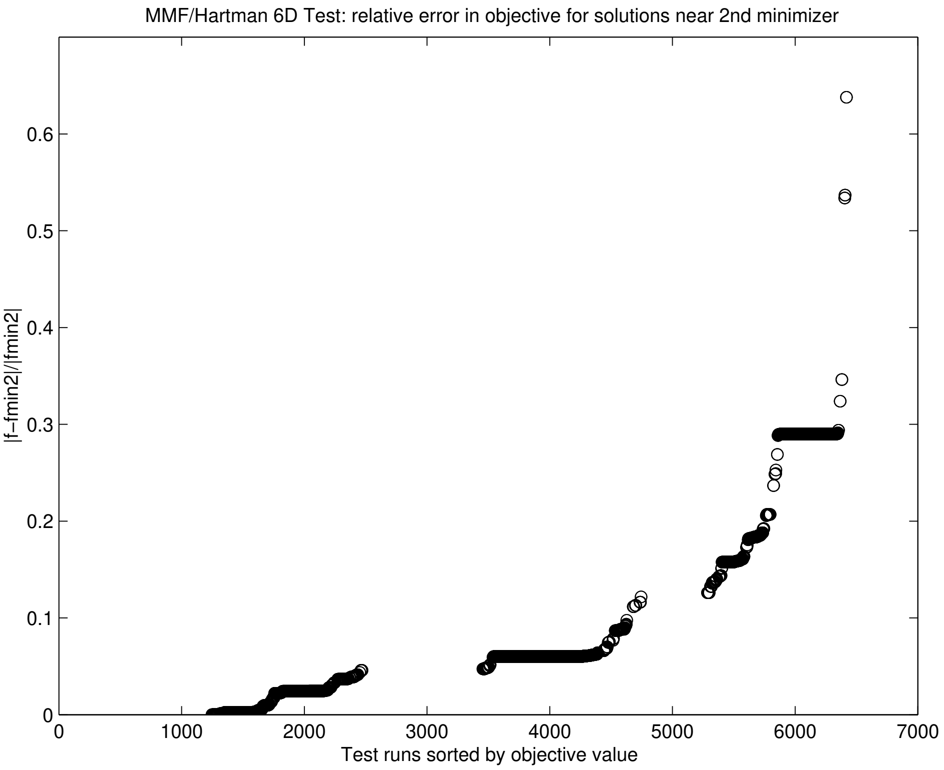


Figure 4.9 Relative difference in objective function value for solutions of test problem 2 that are near the second (best non-global) minimizer.

In the rest of this discussion only those runs that passed the evaluation limit criterion will be considered.

The runs that terminated in less than 100 function evaluations were distributed fairly evenly across the four types of model (interpolating polynomial models, interpolating DACE models and DACE models with 10^{-6} and 10^{-3} interpolation error). The interpolating DACE models appear more often (29%) than the polynomial models (18%), while the non-interpolating DACE models did about equally well (27% and 25% respectively). This data does not provide compelling evidence of the superiority of either of the models.

Two-thirds (69%) of the runs had objective function values within 10% of the global minimum, and 18% were within 2%. We will call these the "best" runs, since not only did the algorithm produce a good approximation of the global solution, it also terminated in a reasonable amount of time.

Not surprisingly, the models built with 64 initial data points produced most (66%) of the best results. Furthermore, these models produced almost half of all runs that met the 100 function evaluation criterion although they only account for one third of the runs. This indicates that this algorithm will be more likely to converge and produce a better answer when a better initial model is used.

The surprise is that the 16 point initial models produced more of the best results than the 32 point models (20% vs. 14%). This gap widens if you account for the extra 16 function evaluations required to build the 32 point model by selecting only the runs with the 32 point model that terminated within 84 function evaluations. The 32 point model runs that meet this more stringent standard account for less than 11% of the best results. In other words, based on total function evaluations, the 16 point models get a best result almost twice as often as the 32 point models.

Applying this more stringent standard to the 64 point model results changes the picture completely: only 2.4% of the best runs use 64 point models *and* converge within 52 evaluations (the 100 evaluation limit minus the extra 48 evaluations used in building the model).

These results suggest that it is more effective to use function evaluations inside the model management algorithm than to use them to produce a better initial model. Results in the "real-world" test case discussed in the next section also exhibit this behavior. Although we cannot draw any general conclusions from just two test cases, this is suggestive enough to warrant future research.

The caveat to this observation is that the number of function evaluations is not always an accurate measure of cost. The function evaluations used to construct a model may not all cost the same (e.g., if they can be computed in parallel) or may be required for other reasons (e.g., to validate the model or to identify important variables). Each application must use an appropriate measure of cost to decide what initial model is best.

4.4 Helicopter rotor blade design problem

4.4.1 Problem description

This problem comes from the Helicopters division of The Boeing Company [28]. The problem is to optimize the performance of a helicopter rotor blade while minimizing vibration. This is a multidisciplinary problem involving structures, fluid dynamics and propulsion disciplines. This problem is of significant interest to Boeing and has been investigated as part of the Boeing-IBM-Rice collaboration.

This problem is a good test for the Model Management Framework because it is typical of large scale multidisciplinary engineering design problems. It has a moderate number of variables (31); large enough to present significant challenges but not so large as to make testing impractical. The computational expense of the objective evaluation can be varied without qualitatively changing the characteristics of the problem, allowing some level of control over the resources required to solve the problem and allowing relatively cheap experiments (hours rather than days of computer time) to be performed. The accuracy of the objective function (with respect to the "real" rotor blade performance) increases with the computational expense. The computer model of the objective function is non-convex, has many local minima, is not defined everywhere (i.e. does not compute results for all inputs), and may not be smooth everywhere it is defined.

The objective function is computed using a program developed at Boeing called `Tech01` [62]. This program has features common to many large-scale engineering analysis codes. It is a complex program, integrating code from several disciplines developed over many years by multiple programmers. The program is fragile, and it will not converge for some sets of inputs, including inputs that are feasible with respect to the explicit constraints given for the problem as well as inputs that are relatively close to inputs for which the program does converge. The output of the program is not smooth with respect to the inputs. The program source is proprietary to Boeing (only object code was available for testing), so the problem has been integrated into the Model Management Framework without changing any of the `Tech01` code. All these characteristics are shared with many other computer programs used in the type of applications which the Model Management Framework targets.

The `Tech01` program can take from several minutes to several hours to evaluate a single objective function value, depending on the level of fidelity selected. The tests reported here used a low level of fidelity, resulting in run times of several minutes per evaluation. The designs computed are less realistic as a result, but the computational cost of testing is greatly reduced.

The variant of the test problem on which we report has 31 independent variables that specify various design parameters at a set of positions along the span of the rotor blade, such as mass, stiffness and center of gravity. The problem has bounds on all the variables and a single linear inequality constraint on the sum of the mass variables. The Model Management Framework implements the bound constraints by rejecting any point that does not satisfy the constraint. The convergence theory guarantees convergence in this case. The same approach is used to satisfy the linear inequality constraints. The convergence theory as presented does not apply to this case.[‡‡] The constraint was included in the problem to see if the method would succeed in practice. The computational cost of enforcing the linear constraint is negligible.

4.4.2 Details of the model management algorithm

The pattern search-based component algorithms described in §3.3.1 were used with the Model Management Framework for this problem. The pattern used in `SEARCH` contained approximately 30,000 points. The `SEARCH` algorithm from §3.3.1 evaluates the model at every feasible point in the pattern. For this problem, in any particular execution of `SEARCH` as many as 5000 of the points were feasible[*] and as many as 2000 produced decrease in the model, although this latter number decreased rapidly as the algorithm progressed; the number of points that produced decrease in the model value was typically a few hundred or less. A large pattern was used because the models are very cheap to evaluate and the `Tech01` program is known to produce function values with many local minima so extensive sampling of the model was considered necessary to produce better trial steps.

This reflects an important issue in `SEARCH` algorithm design: compared to the cost of evaluating the actual function just about everything else that can be done in the algorithm is cheap. This gives tremendous flexibility in designing algorithms, up to and including using "brute force" approaches that would be impractical if applied directly to the actual function.

The initial model of the objective function was constructed at Boeing by Andrew Booker using his DACEPAC software package [3, 4]. Fifty-nine function evaluations

---

‡‡A recent extension to the underlying pattern search convergence theory [49] could be used to prove convergence in the presence of linear inequality constraints.

*The scaling of the initial pattern put most of the pattern points outside the bounds.

were used for the initial model. This represents the number of successful evaluations of the `Tech01` code. An additional 97 unsuccessful evaluations were performed for which the code did not converge. We do not include these evaluations in the cost of building the model because we want the cost to reflect the amount of data used to build the model. Furthermore, the author's experience with many engineering analysis codes suggests this is an unusually high failure rate for such codes and does not reflect the general case. Also, it might be possible to reduce this failure rate by tuning the parameters that control the `Tech01` code or by running the code for more iterations. On the down side, this might increase the run time of each function evaluation. This tradeoff was not investigated. The parameters used were as specified by the Boeing engineers.

The 59 data points in the initial model is a relatively small number for a problem of this dimension. Consequently, the model captures very little of the behavior of the actual objective function. This represents a difficult environment for any model management algorithm, and so it is an appropriate test case.

Four MMF results are presented. These represent different settings of the limit on the number of trial steps returned by `SEARCH`. This limit determines how many times the actual function will be evaluated before the Framework chooses to set the trial step to zero. This limit trades off the cost of function evaluations against the ability of the model to predict decrease. The better the model at predicting decrease, the fewer the number of trial steps that should need to be evaluated to find one which decreases the actual objective. The lower the limit, the more likely the algorithm is to execute `POLL`, with its potentially high cost.

4.4.3 Other methods

Several other approaches were used to solve this problem. The results are presented for comparison. One approach applied a pattern search method directly to the actual objective function. Another used a model management method with polynomial models. Another approach used an iterative statistical sampling technique to choose points at which to evaluate the actual objective, taking the lowest value computed as the result. In a fourth approach the sampling results were used to construct models to which an optimization procedure was applied. Another approach applied a genetic algorithm directly to the actual objective function. Each method is described briefly below.

The same starting point was used for all methods for which a starting point is meaningful. The same initial model was used for all the model-based methods. The initial point and initial model were provided by Boeing.

Pattern search method

The pattern search method used is the Parallel Direct Search method of Dennis and Torczon [23] as implemented for unconstrained problems in the PDS program by Torczon [71]. PDS has been modified by Serafini to support constraints and the standard Message Passing Interface (MPI) parallel communications library [64]. This modified version was applied directly to the actual objective function. No model was used.

Two different starting points were used. One was the same starting point used for the other methods and the second was the point with the lowest objective value from the 59 points used to construct the model. These two cases are called "PDS1" and "PDS2", respectively. The second case is of interest because it helps characterize the behavior of PDS on this problem. Since PDS is based on a similar theory to the model management theory, this characterization may prove enlightening. Several tests with different sized patterns were conducted for each starting point and the most efficient runs are presented. The PDS1 run used a 96 point pattern. The PDS2 run used a 64 point pattern. In both cases, larger (more costly) patterns were tested but did not improve the results.

BLGS method

The BLGS method [6] mentioned in §3.2.2 was also used on this problem. The same 59 point DACE model used in the MMF results was used as the initial model. Three model refinements were performed, using 50 function evaluations each time. Paul Frank at Boeing developed the BLGS software and generated the results presented here.

Genetic algorithm method

A solution using a parallel genetic algorithm was also performed. The PGAPack software [46] was used for this solution. The result presented used a steady-state reproductive strategy with a population size of 200 and a replacement rate of 10% of the population per iteration. These parameters were recommended by David Levine, the author of PGAPack.

DFO method

A new derivative-free optimization method developed by Conn, Mints and Toint [14] was also applied to this problem. The method, called DFO, is a form of model management algorithm. The model is a quadratic polynomial approximation. The model is constructed incrementally from the actual function values produced by the search. The search strategy solves a trust-region optimization subproblem with the model to find a trial iterate. The model management strategy works to maintain a good basis for the polynomial. The results presented here are due to Katya Mints at IBM.

Like the Model Management Framework, DFO is a new method still being developed. Results are presented for two variations of the DFO implementation. One scales the variables in the optimization problem and the other does not. The variable values in this problem span 10 orders of magnitude so scaling seems like a good idea. However, the results for the version without scaling are the better of the two.

Sampling method

This method is a straw man. It is not intended to represent the behavior of a realistic method. The approach is simply to construct a set of points in the space (a sample), evaluate the actual objective function at each point in the set and keep the point with the best objective value. Repeat with larger samples until the result stops improving. Note that no model is used. Note also that the samples were generated independently and no attempt was made to adapt the larger samples to the results from the smaller samples.

Results for four iterations of this method are presented. The samples in the iterations contained 59, 201, 381 and 684 points. All the samples were selected from orthogonal array-based Latin hypercube samples of the Bose type [56]. The 59 point sample is the same as was used to build the initial models for the Model Management Framework runs.

Manual model optimization method

A very simple approach to doing model management "by hand" was tested. Like the sampling approach, this approach is also a straw man. The points and objective values from the three smallest samples from the sampling method were used to build DACE models. A global optimization method was applied to each model and the actual objective function was evaluated at each solution. In essence this is very similar to the sampling method except the model is used to find trial points that may have a better value of the actual objective than any point in the sample. The only management of the model is that at each iteration a new model is built from a larger amount of data.

Only the cost of the actual objective evaluations done at the sample points is measured and reported. The computational costs of constructing and optimizing the models are not counted. The results are labeled "ModelOpt" in the table below. All three sets of sample points and all three models were provided by Andrew Booker at Boeing.

4.4.4 Parallel computing issues

All the approaches used are amenable to parallel computations to a greater or lesser degree. There is parallelism available in the Model Management Framework in the function evaluations at the trial steps returned by `SEARCH` and in the evaluation at the core steps in `POLL`. Given the assumption that function evaluations are much more expensive than anything else, it should be possible to utilize at least as many processors as there are variables in the problem. However, the implementation of the Framework used in the results reported here is sequential.

PDS and PGAPack are very similar in their use of parallelism. The function evaluations in each iteration can be executed in parallel in both methods. There is a small sequential component at the end of each iteration that, by Amdahl's law, limits the potential speedup. In practice, both methods can scale to large number of processors by increasing the size of the pattern in PDS or the size of the population in PGAPack.

The sampling approach also has good parallel speedup because all the evaluations are completely independent of each other and there is no other work to do except choose the minimum of all the samples. The ModelOpt approach performs the same work as the sampling approach, but then performs a global optimization to construct each model and another global optimization to find the minimizer of each model. Both of these tasks can be parallelized efficiently.

The implementation of DFO used for the results presented here is sequential. No parallel algorithm for DFO exists at the time of this writing, but because of the non-deterministic nature of the algorithm, it could be advantageous to run the code several times on the same problem and take the best result. These runs are totally independent and can be executed in parallel efficiently. The effective cost of the whole solution then would be the cost of the most expensive run.

There are two measures of parallel efficiency that are relevant to this discussion. One is how much time it takes to compute a solution for a given problem with a given algorithm, as the number of processors is changed. The other is how much time it takes to compute a solution and how good that solution is, as the number of processors and the algorithm are changed together.

In the first case, for relatively small numbers of processors (less than the number of variables), all the methods will have similar efficiency. It is for larger numbers of processors that the differences between the methods become significant. The sampling and model optimization methods are the best in this case, since the number of function evaluations increases as the algorithm progresses so any number of processors can be utilized if the algorithm is run for long enough. The other methods all have limits on the amount of concurrency they possess and thus on the number of parallel processors they can use efficiently.

In the second case, where the algorithm can depend on the amount of parallelism available, the comparison is different. All of the methods can be tailored to take advantage of a particular number of processors, however large. The methods differ in the efficiency with which they utilize the processors.

As in the first case, the sampling and model optimization methods can be scaled with the number of processors very efficiently. The simplest approach is to start with a sample size that is the same as the number of processors and make the size of each successive sample an integral multiple of the number of processors.

The PDS and PGAPack algorithms can also be tailored to fit any given number of processors by increasing the pattern and population sizes, respectively. PGAPack also has another parameter that can be modified to tailor the algorithm. This parameter, the *replacement ratio*, is the percentage of the population replaced at each iteration. It also affects how many function values are computed concurrently and thus how much parallelism can be utilized. A discussion of the tradeoff between these parameters is beyond the scope of this discussion. For details, see the PGAPack documentation [46] and the literature on genetic algorithms (e.g. [33, 18, 52]). The parameter settings used for the results presented here were suggested by David Levine, the author of PGAPack.

Clearly, the issues involved in parallel performance evaluation are complex. The results presented in the next section use the number of function evaluations as the performance metric, as if each code was run serially. This is not necessarily indicative of how the different programs would perform in a realistic parallel computing environment. Any comparisons made between the different methods based on these results must take this into account.

4.4.5 Results

Figure 4.10 plots the objective function value versus number of function evaluations for each of the methods. The lower plot expands the low end of the horizontal axis. Only the symbols in the plots represent actual data. The connecting lines are included only to make the plot more readable.

Table 4.5 below summarizes the data in the plots. The count of function evaluations and the corresponding objective function value are given for each method, except DFO, including some intermediate results. For the ModelOpt method the model values at the model optimizers are also given to show the accuracy of the model.

Table 4.6 summarizes the results for DFO. The DFO algorithm has a random step so no two executions of the program will give exactly the same result in the same number of function evaluations. We present the results from several (10) runs of DFO† for two versions of the algorithm; one in which the independent variables are automatically scaled, and one in which they are not scaled. The independent variables in this problem have values that span 10 orders of magnitude. All the other methods scale the variables, either manually or automatically.

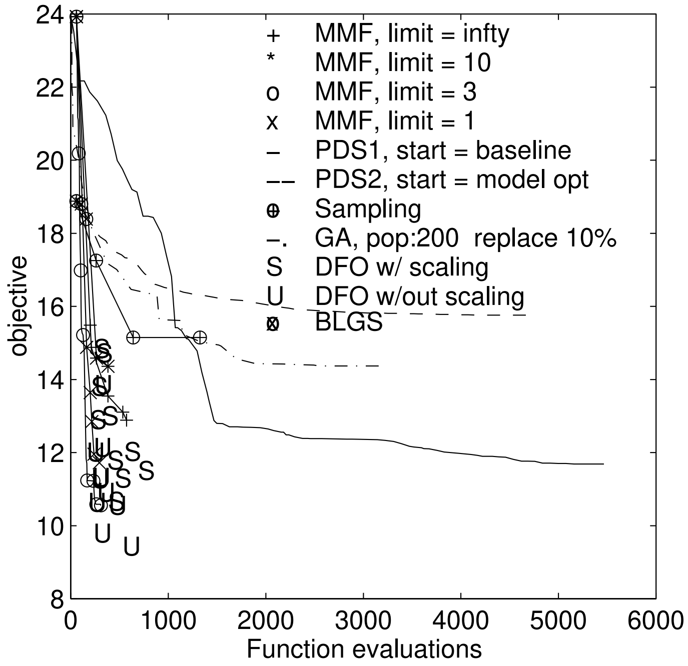
It should be noted that the global minimum for this function is unknown.

4.4.6 Analysis of results

The baseline solution was provided by Boeing and represents a physically reasonable rotor blade design. It is not arbitrary. It is intended to be indicative of the quality of starting point that would usually be encountered in practice.

---

†It should be noted that the PGAPack results are also non-deterministic, but because of the expense of a single solution, multiple runs with these parameters were not performed.



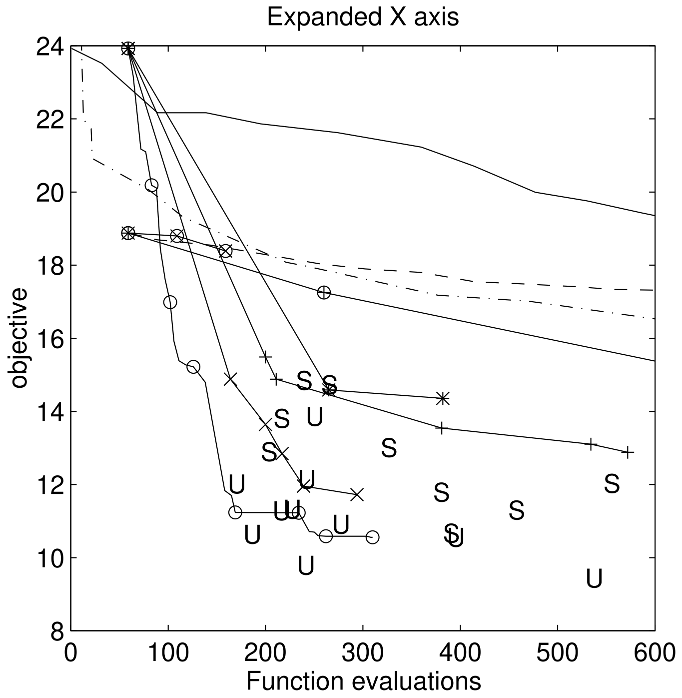


Figure 4.10 Convergence history for all methods on `Tech01` 31 variable problem.

Method	evals	obj	model obj
Baseline	1	23.93	
PDS1	1500	12.80	
	5465	11.68	
PDS2	863	16.60	
	3044	15.82	
MMF/1	294	11.72	
MMF/3	237	11.15	
MMF/10	382	14.36	
MMF/∞	572	12.88	
Sampling	59	18.88	
	260	17.19	
	641	15.15	
	1321	18.59	
ModelOpt	59	20.54	15.89
	260	24.50	8.855
	641	14.87	8.633
PGAPack	221	18.081	
	471	17.011	
	907	15.646	
	3229	14.368	
BLGS	59	26.899	16.271
	109	***	9.678
	159	18.338	11.279

Table 4.5 Results for all methods (except DFO) on the 31 variable TECH01 problem. (Note: "***" indicates the actual objective function did not converge.)

Method	evals	obj
DFO (Scaled)	208	13.815
	257	14.732
	231	14.852
	547	12.034
	688	11.513
	382	10.674
	318	13.013
	195	12.905
	372	11.797
	449	11.307
DFO (Unscaled)	161	12.021
	218	11.329
	268	10.920
	232	9.799
	207	11.285
	241	13.865
	177	10.645
	233	12.147
	386	10.561
	528	9.443

Table 4.6 Results for DFO on the 31 variable TECH01 problem.

Performance results for the PDS program are listed for two cases in Table 4.5. The run labeled "PDS1" starts from the Baseline point. Run "PDS2" starts from the point in the 59 point sample with the best objective value.

The results from the PDS runs will be discussed in more depth than the other methods because the MMF is based on many of the same ideas as PDS. Thus, the behavior of PDS provides insight into the behavior of the Model Management Framework.

For each PDS run, two different numbers of function evaluations are listed in the table. The lower number is at an intermediate stage of the computation and the higher is for the final result. The two stages are shown because the performance of PDS (in terms of decrease in the objective function value per iteration evaluation) changes (dramatically, in run PDS2) at a certain point in each run, as can be seen in Figure 4.10. Each performance curve has a "knee" where the slope decreases significantly. This is indicative of the performance of PDS on many problems. The knee occurs because PDS uses a fixed pattern size throughout the computation. Before the knee, PDS makes rapid progress toward a solution in nearly all iterations, indicating the pattern is finding decrease most of the time. After the knee, PDS makes very slow progress because some iterations find no decrease and most of the successful iterations find very small amounts of decrease. In this second stage PDS has found a basin of a local minimizer and must shrink the pattern to converge to that minimizer. This is where the linear asymptotic convergence rate of pattern search methods dominates their performance. Note that the MMF results show similar behavior.

The results presented are the best results obtained by running PDS with different size patterns for each case. There is a performance tradeoff with pattern size. Typically, larger patterns move to the basin of the solution more quickly so the performance before the knee in the curve is better, but they converge to the solution more slowly (i.e. take more iterations for the same relative decrease in the objective) so the performance after the knee is worse. Typically, the smaller patterns tend to behave in the opposite manner. Actual behavior is problem dependent.

This behavior comes about because larger patterns have more points further away from the current iterate. As the process converges, the likelihood that any of these points will have have a lower function than a point close to the current iterate decreases. Thus the computer time expended to evaluate these points is mostly wasted. Of course this behavior is very problem-dependent; on average, the larger pattern will make more progress toward the solution in each iteration than a smaller pattern, but at higher cost. In essence this is a tradeoff of marginal cost (the extra function evaluations of a larger pattern) against marginal gain (the likelihood that one of these evaluations will produce a better function value). If the marginal cost is small (as it might be in a parallel computing environment) than any marginal gain may be worthwhile.

Determining how to balance this tradeoff would not be too difficult except for the additional consideration that larger patterns are capable of finding better solutions for non-convex functions because they do more extensive searching. Also, as can be clearly seen by comparing the PDS1 and PDS2 results, the starting point can have a significant impact on the quality of the final solution. The PDS2 run improves quickly at first then progresses more slowly. This is indicative of finding a basin (by following a large gradient) and then converging to the minimizer of that basin (by wandering around in the bottom of the basin). The PDS1 run is slower in the first stage but ultimately finds a basin with a lower minimizer. The simplest explanation for this behavior (although difficult to verify) is that the PDS2 run stays in the same basin in which it starts (or a nearby one) and so spends relatively little time "traveling" before it starts contracting the pattern scale factor and converging to the local minimizer.

The PDS1 run, because it starts at a point with a higher function value, spends more time traveling before it settles in a basin and starts converging.

The performance of either run likely could be improved by changing the pattern to reduce its size once the search has chosen a basin and started contracting the pattern scale factor. The advantage of this approach is that fewer function evaluations would be used in the second stage of the run. The disadvantage is that it is difficult to determine when it is appropriate to make this change; the knee in the performance curve usually is not as clear as in the two cases presented here, The payoff can be significant: the PDS1 result used a 96 point pattern for the whole run and reducing the pattern to 64 points at the knee could have saved as many as 1200–1500 evaluations.

Performance of pattern search (and, to an extent, model management) methods is very dependent on the choice of pattern and other starting conditions so some other pattern search method may perform better than PDS on this problem. However, in general PDS performs well on this problem, and we consider these results a fair representation of the performance of parallel pattern search methods.

The Sampling results require some additional explanation. The sample sets in each iteration were independently generated and do not contain the smaller samples as subsets. The value listed in the table is the number of evaluations including those in the smaller sample sets. This represents the cost of iteratively evaluating the samples until a termination condition is reached. The cost of choosing the sample is assumed to be negligible. Figure 4.10 plots the total number of evaluations for each sample set and all its predecessors. The best function value from the largest sample is actually worse that the value from the next smaller sample. Table 4.5 shows this. Figure 4.10 shows the better value.

These results are presented for completeness. They are not intended to represent how one would actually use the sampling idea in an optimization algorithm.

Sequential evaluation of independent sample sets is clearly not a very good method for this problem. The last (and most costly) sample does not improve the result and the best result achieved by the method is significantly worse than the result obtained by the model management methods.

The ModelOpt results are not plotted. They are presented in the Table only to show the ineffectiveness of the simple, but often used, approach of using a model as a surrogate of the actual objective function without some kind of management strategy. The results for the smaller models show a definite failure to approximate accurately the basin of a minimizer of the actual function. The optimizer of the largest model is the only one that manages to produce real decrease from its starting point, and even then only slightly. The key feature to observe in these results is that the values at the model optimizers do not agree well with the true function values at those points in any of the cases. Even if the model has correctly identified a basin of the actual function, it has not helped much in finding the minimizer in that basin. These results also argue strongly for the use of model management strategies.

The MMF results show the effect of varying one of the parameters in the management algorithm, the "function evaluation limit" defined in §4.2. This parameter determines the maximum number of times the Framework will evaluate the actual function on steps returned by `SEARCH` in a single iteration of Steps 2–8 in Figure 2.4. In other words, this is the maximum size of the set of trial steps that `SEARCH` may produce. The curves plotted in Figure 4.10 indicate that as the current iterate approaches the optimizer, the cost of performing the extra function evaluations becomes a burden rather than a benefit. This is very similar to how the performance of pattern search methods varies with the size of the pattern. More function evaluations (like larger patterns) are beneficial early in the search because they sample the space more effectively and allow longer steps. Later in the search, accuracy is more important than coverage, and using fewer function evaluations (smaller patterns) is more efficient.

In the model management context there is the added issue that it is important to improve the accuracy of the model, especially when it fails to correctly predict decrease in the actual function. So a larger set of trial steps will tend to improve the predictive capability of the models (up to a point), though no guarantee of improvement can be made. As in the case of parallel pattern search methods, it is a tradeoff of marginal gain and marginal cost. If the extra function evaluations can be performed cheaply (e.g. computed in parallel), than they may be worth doing even if they produce only a little gain.

The down side is that as the number of data points increases, it may become more difficult to construct a new model. For example, the DACE models discussed here tend to become ill-conditioned if the data points get too close together or too close to co-linear. Both are more likely to occur as the number of data points in the model increases.

The PDS and PGAPack results are more similar to each other than to any of the other results. The relatively slow convergence of both methods is clearly demonstrated. PGAPack performs better than PDS in the early parts of the runs, but in the PDS1 run, PDS does a better job of continuing to find additional decrease in the objective. The PDS2 run starts at a lower objective value, presumably closer to a local minimizer of the model. The convergence history shows that it converges very consistently, but to a point that does not have a particularly good value, presumably in the same basin as the initial point or in a nearby one.

The performance of BLGS on this problem is not necessarily indicative of its performance in general. The initial 59 point model is smaller than Paul Frank, the author of BLGS, would normally recommend [27]. He would prefer to build the initial model for this problem using around 200 function values. Also, the BLGS method is concerned with the problem of global optimization more than the model management algorithm used for these tests. Thus the model management algorithm can tolerate less accuracy from the initial model than can BLGS on this problem, since BLGS will be more likely to find points where the model over-predicts the decrease in the actual objective function.

The DFO runs show a large degree of variability due to the non-deterministic nature of the algorithm. The results for the version of the algorithm without scaling are noticeably better than the results for the version with scaling. This issue is discussed further in [5].

The four MMF runs and the various DFO runs perform similarly, and much better than the other methods that were tested. The common characteristic of all the methods is that they do not require the gradient of the actual objective function. DFO implicitly forms an approximation of the gradient, but it gets it from the polynomial model essentially for free. The pattern search and genetic algorithm results are both from good implementations of the respective methods, and both methods were used with reasonable parameter settings. The author has been the primary developer of PDS for the last three years and the parameter settings used for the PGAPack result were recommended by the developer of that package. The BLGS result may not be a fair representation of that method's performance in general: it might perform better with a better initial model, but any improvement in performance that might be gained must come at the cost of more function evaluations. The other two methods are straw men and their performance is neither impressive nor surprising.

This problem has many of the characteristics of the class of applications for which the Model Management Framework is intended to be used. The results clearly indicate that even with the simplistic search and management strategies that were used, the

Model Management Framework is a viable and competitive method for solving this complex, expensive, engineering design problem. The success on this problem suggests that the Framework approach may work well for similar applications.

Chapter 5

Discussion

5.1 Software design and implementation

This section describes the software implementation of the Model Management Framework.

The theory describes the Framework as a form of pattern search method, and defines four model management component algorithms that are required by the Framework's exploratory moves algorithm. The architecture of the software implementation inherits this decomposition and extends it. The component algorithms are encapsulated in separate routines with specifications based on the definitions of the component algorithms' behavior (Definitions 2–5). The design of the Framework software uses these routines in a manner which allows details of the model management strategy to be changed without changing any of the Framework software (or any of the application software that uses the Framework). The software is implemented in Fortran because this is the predominant language for large scale numerical computing in engineering and the sciences. The Fortran90 version of the language is used in order to take advantage of the capabilities for data structures, dynamic memory management and function overloading that this version provides.

Figure 5.1 presents a graphical representation of the software architecture. The four layers in the graph each represent separate pieces of software. The arrows point from the caller to the callee routines.

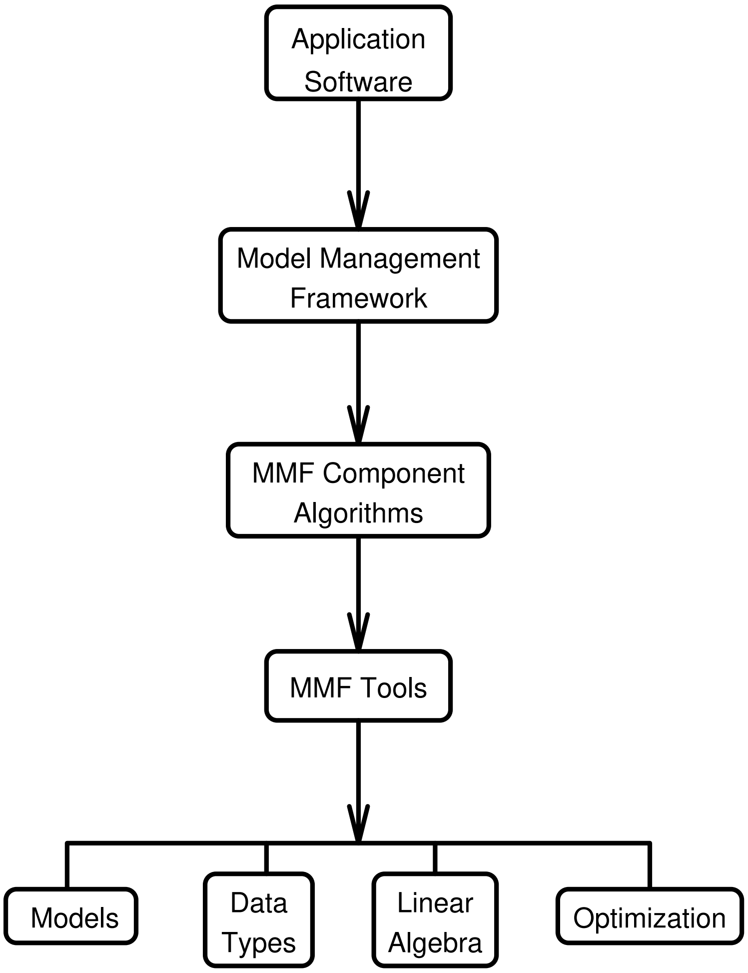


Figure 5.1 Architecture of the Model Management Framework software

The highest layer is the application program. This is the software that uses the Model Management Framework to solve a nonlinear optimization problem. Engineering design software is the canonical example of such an application.

The next layer is the software that implements the Framework itself. Nearly all the functionality of this layer is encapsulated in one procedure, called `MMFPS`. This is the procedure that the application program would invoke to solve the optimization problem. The interface to this procedure is similar to other optimization routines but involves more information than usual because of the use of models.

In addition to the `MMFPS` procedure, the Framework layer also provides various procedures that simplify the task of setting up the data that needs to be input to `MMFPS`. The Framework gets these utility procedures from the lowest layer in the architecture hierarchy (labeled "MMF Tools" in Figure 5.1). These procedures do not depend on the implementation of the Framework, although the Framework may depend on their implementations. Some of the tools have standardized interfaces to allow different implementations to be used transparently. The Framework layer provides these procedures to the application layer rather than having the application layer access them directly to guarantee that the Framework and application software use consistent versions of these tools.

In addition to the Tools procedures the Framework layer uses the procedures that implement the model management component algorithms. These reside in the next lower layer. The design of the interfaces between the Framework and the model management procedures hides the implementations of the model management procedures from the Framework, allowing different management strategies to be implemented without changes to the Framework itself. The only routines in the component layer are `SEARCH`, `MANAGEMDL`, `TERMINATE` and `POLL`. These routines correspond to the component algorithms described in Chapters 2 and 3. The components layer provides no other functionality. The routines in the component layer are not accessed by the application layer directly. All application-specific aspects of a model management algorithm implementation can be restricted to the components layer. No changes to any other part of the software except possibly the Tools (Models and Optimization, for example).

5.2 Discussion of numerical results

The model management algorithm used to produce the results presented in Chapter 4 is not what one would use for a real application. It is presented here only to show the viability of the Model Management Framework as a mechanism for the development and implementation of algorithms for real problems. Even so, the performance of the algorithm is reasonably good. In this section we try to explain the performance results and suggest ways to improve the example algorithm.

The primary observation we can draw from the results of the TECH01 problem is that using a single model of the entire domain is not an adequate approach. As the optimization process progresses, the distances between the trial iterates decrease and the difficulty of fitting the model to all the points increases. The TECH01 results for both the DACE and polynomial models show the usefulness of the model degrading as the optimization progresses. The symptoms are the decreasing frequency of finding decrease in the actual function at the candidate steps returned by `SEARCH` and the increasing frequency of executing `POLL`. The condition number of the correlation matrix in the DACE models and the number of terms in the polynomial models provide quantitative measures of the difficulty of fitting the models to the data. Both numbers increase dramatically: the condition number exceeds 100 million; the number of polynomial terms exceeds 1000.

We assume that an effective model update strategy must capture the behavior of the actual function at the trial iterates as they are generated by the optimization process in order to effectively capture the behavior of the function near the iterates where the search will likely look for the next iterate. The approach used in the test results accomplishes this by forcing the model to interpolate (approximately, in some cases) the actual function values. In general this will not work very well. We assume that the actual functions of interest are inherently more complex in their behavior than the models are. Attempting to force a model to interpolate both the large-scale global behavior as well as the small-scale local behavior is probably too much to ask of any function approximation based modeling strategy. Thus, the difficulties encountered in the TECH01 problem will probably occur frequently with this model management implementation.

One potential approach to avoiding these difficulties is to use a separate model defined on a small neighborhood around the current iterate. This alleviates the difficulty of capturing both the large and small scale variations in the actual function with the same model. It adds some complexity to the model update strategy because the points used to construct the local model may need to be modified as the trial iterate moves. Additionally, the size of the region on which the model is defined may also need to be modified as the optimization converges. The latter is not as significant an issue because we assume the Model Management Framework will only be used to get the first few digits of accuracy in the solution, so more than a few changes in scale are unlikely. The use of a hierarchy of models defined on regions of different sizes and intended to capture behavior on different scales might be preferred in some cases. The drawback is that managing these models will be more difficult.

Another potential approach is to relax the requirement for interpolation at the trial iterates. In the DACE models this can be accomplished by perturbing the values on the diagonal of the correlation matrix by a small amount [16]. The condition number of the matrix can be improved significantly with a perturbation as small as 10^{-6}.

Another similar approach is to address the rank deficiency of the correlation matrix directly and regularize the matrix and solve it as a least squares problem. This gives up the interpolation behavior of the standard DACE model, but avoids the worst of the ill-conditioning.

5.3 Solving constrained problems

The analysis of the Model Management Framework we have presented does not address problems with constraints other than bounds on the variables. This shortcoming is inherited from the convergence theory for pattern search methods, which applies only to unconstrained and bound constrained problems. This does not mean that the Framework cannot be used to solve other types of constrained problems, but some extra effort will be required to design and implement an algorithm and to make sure the result is a good solution to the problem.

An extension to the pattern search theory that defines conditions under which the method will converge for linear constraints has been developed by Lewis and Torczon [49]. It is very likely that such an extension will be directly applicable to the Model Management Framework theory. So in the future the case of linear constraints will be handled by the Framework. This assumes the cost of evaluating the constraints (a matrix-vector multiply) and implementing the strategy required by the convergence theory are acceptable. If not, the approaches suggested for handling expensive nonlinear inequality constraints may be applicable. Convergence of pattern search methods for nonlinear inequality constraints remains an open question.

The simplest approach to handling constraints in the Model Management Framework is to evaluate the constraints directly and enforce feasibility the same way the Framework does for bound constraints. This can work well for inequality constraints that are cheap to evaluate. Equality constraints and expensive inequality constraints require other approaches. However, even if the inequality constraints are cheap this approach has several pitfalls. Extra care must be taken to analyze solutions that are near a constraint boundary because the theory does not guarantee convergence in this case. It is possible that the search might fail to find a descent direction away from the current iterate even though one is present in the function. This can happen when the descent direction is along the constraint boundary.

One approach for the case where the inequality constraints are too expensive to evaluate at every trial step is to construct a model of the constraint function. There are two basic alternatives in this approach: model the outputs of the constraint function (using a vector-valued model or multiple scalar-valued models) or model the feasibility directly (using a boolean-valued model). The latter probably will be simpler if there are many constraints, although the behavior that the model will need to capture may be very complex.

Since there will be no guarantee in general that the model will predict feasibility correctly it will be necessary to add an extra step to the Framework to check feasibility using the actual constraint function whenever the actual objective function is evaluated. This will present additional opportunities for model management algorithms. The accuracy of the constraint model(s) may vary differently than that of the objective model during the computation so the models will have to be managed separately. In some applications it may be advantageous to use different management strategies for the different models.

Most problems with linear equality constraints can be solved by using the constraints to eliminate variables or by projecting onto the constraint manifold. In most cases the cost of solving the linear equations will be small compared to the cost of objective function evaluation. If the system is too large to solve, it may be possible to use a model of the constraints.

5.4 Parallel algorithm design

Another important issue is how parallelism affects the choice of algorithms and algorithm parameters. As mentioned above, both PDS and PGAPack have algorithm parameters (pattern and population size) that can be used to increase the amount of available concurrency in response to an increase in the number of available processors. The key point is that this may also change the solutions that the methods produce. In the cases of PDS and PGAPack, a larger pattern or population will usually (though not always) produce a better result in a fixed number of iterations (but not function evaluations) or the same result in fewer iterations. The Model Management Framework also has this behavior.

The number of available processors can have a significant effect on the choice of model management algorithm. In particular, the size of the set of trial steps generated by the `SEARCH` component can be increased at little or no effective cost as the number of available processors increases, since all the trial steps can be evaluated concurrently. In principle, there is no limit to the number of processors that can be used effectively in this manner. In practice, there is likely to be a point of diminishing returns where the `SEARCH` algorithm cannot produce enough good trial steps. One way of increasing the amount of concurrency is to select some trial steps because they improve the model, regardless of their function values.

In a similar manner, different algorithms for `POLL` can be considered depending on the available parallel resources. Here there is a limit on the available concurrency even in theory, since the size of a positive basis is at most $2n$ points (n being the number of variables). There is also an interplay between the `SEARCH` and `POLL` algorithms that is affected by parallelism and that is the choice of the condition that determines when `SEARCH` gives up and `POLL` is invoked. The worst case cost of `POLL` is fixed in terms of number of function evaluations but the worst case cost of `SEARCH` can be tailored to some extent. When the effective (worst case) cost of `POLL` is high because the number of available processors is low, then the `SEARCH` algorithm can be designed to try harder before giving up. When the effective cost of `POLL` is low because enough processors are available, it may be advantageous to design the SEARCH algorithm so that it gives up easily.

Chapter 6

Conclusions

The use of models in optimization and the ideas of model management have been around for a long time. The sequential linear programming (SLP) and sequential quadratic programming (SQP) methodologies are model management strategies. The quasi-Newton globalization strategies rely on the ability to control the accuracy of the quadratic Taylor series model and to solve an optimization subproblem using the model. Methods that take advantage of the special characteristics of Taylor series models have dominated numerical optimization. Methods that use other models have seen limited use, mostly in engineering optimization.

To use these models in our Framework we must provide a convergence theory to replace the theories for Taylor series-based methods that depend so heavily on the accuracy property of the Taylor series models. The theory we use was developed for pattern search methods. It relies on the discrete nature of the pattern and the positive basis requirement on the core pattern instead of on a bound on the accuracy of the model. This thesis has shown how models can be used with the pattern search method and what algorithmic behavior is required to guarantee convergence.

Interest in non-Taylor series based optimization methods has grown in recent times for two main reasons. The cost of the functions that arise in some applications is too high for traditional methods and the cost or difficulty of constructing useful Taylor series models is too large. The usual solution to the first problem is to get a faster computer. This works in some applications. In others, the complexity of the problems is increasing faster than computer capabilities. One of the forces pushing the use of optimization is the need to solve the problems more quickly, with less (human) effort.

Shorter engineering design cycles is an example: engineering managers want better solutions in less time than ever before. The requirements of global competitiveness make it a necessity to improve both product and process.

Two factors holding back growth in the use of model management methods in optimization have been a lack of generality and a lack of rigor. Model management for Taylor series models is very well developed, but relies heavily on the special accuracy characteristics of these models. Additionally, the nonlinear programming community often focuses on methods with fast asymptotic convergence rates. This is at odds with the needs of many of these expensive applications where the accuracy required from the solution is often only a few significant digits. This requirement plays to one of the strengths of the model management approach in that the transient and asymptotic convergence rates are the same. This makes the model management approach less well suited to problems requiring highly accurate solutions, but that is unavoidable. Fast local convergence (super-linear or quadratic) can only be expected from methods that use gradient and Hessian information.

In some sense, the optimization field is suffering from its own success. As the use of optimization grows, and there are more and more successful applications of optimization to real world problems, it is inevitable that there will be demand for solutions to more and more difficult problems. A typical example is the use of higher fidelity simulation software in computing the objective and constraint functions. In aerodynamics, the problem has progressed from solving the potential flow equations, to the Euler equations, to the Navier-Stokes equations. Each step has cost several orders of magnitude more than the previous, and has been coupled with an increase in the complexity of the design problem: from 2D airfoils to clean 3D wings to full configurations with engines and control surfaces. Inevitably, the software that computes the solutions gets more complex as the problems get more difficult, and complex software usually has more complicated behavior, making for harder optimization problems.

The model management approach addresses both problems. The use of models can reduce the number of expensive function evaluations that are required to reach a solution of a given accuracy. And they can reduce the difficulty of the optimization problem by filtering out the "bad" behavior of the complex analysis software.

The work presented here addresses several of the issues involved in using model management methods on real world problems. The Framework provides an abstraction of a general model management algorithm. This abstraction is not tied to any particular method of modeling the actual function(s) or searching the design space. It provides fairly detailed guidelines for designing methods that are convergent, but it is also flexible. It allows the algorithm to take advantage of any special knowledge of the application or modeling technique that may be available.

The cost of this flexibility is that it leaves much of the burden of achieving efficiency in practice up to the algorithm designer. This is an unavoidable situation. No general-purpose method can be efficient on all problems, especially without even gradient information about the functions. We contend that the Framework accomplishes much of the work of developing a correct algorithm, freeing the algorithm designer to focus solely on the performance issue.

The correctness of the Framework and of the algorithms developed using it depends on the convergence theory for pattern search methods. The Framework builds on this theory by explicitly separating the hypotheses on the exploratory moves algorithm into two distinct parts: searching for trial steps and polling the core pattern. This distinction allows a model to be used in selecting trial steps instead of the usual pattern search strategy of evaluating the objective function at the points in the pattern. The Framework does not limit how the model is used in the search. Polling the core ensures the theoretical requirements for convergence are fulfilled. This separation allows the strategies used to improve the performance of the method to be unconstrained by the convergence requirements.

The numerical test results presented support the argument that algorithms can be developed using the Framework that will be useful for real world problems of significant cost and complexity. The simple test cases provide some insight into the behavior of one, albeit simple, model management algorithm using the Framework. The helicopter design problem shows that a problem with the characteristics of many "real world" problems can be solved without major (and costly) changes to either the engineering analysis code.

These results do not prove that any specific model management algorithm, whether implemented using the Framework or not, will be efficient for any particular application, but they do provide a baseline for the development of more useful algorithms and a performance benchmark for more efficient algorithms.

Chapter 7

Future Work

There are two fronts for future research. One involves using the Framework to develop fast algorithms for practical applications. The other involves improvements to the Framework itself.

7.1 Model management algorithm design

There remains much work to be done in algorithm development. The algorithm that produced the results presented here does not take full advantage of the capabilities of the modeling software it uses. The use of other modeling approaches should also be investigated.

There are several important issues that need to be addressed in the area of using the Framework to develop algorithms. The first is how to overcome the problem of degrading model quality as the actual function values from trial iterates are incorporated into the model. In the DACE and polynomial models this degradation in quality is due to the clustering of the iterates as the optimization process converges as well as to the tendency for the iteration steps to be linearly dependent. Specific approaches to alleviate this problem should be investigated.

A general idea with much potential is to use one model defined on the whole domain and a second model defined only on a close neighborhood of the current iterate. This raises several algorithm design issues including how to choose which data points to add to which model, how and when to change the domain of the local model, what modeling strategy to use for each model.

Another idea that should be investigated is the use of a noninterpolatory model that is not sensitive to spacing of data points. This raises issues in algorithm design for `SEARCH`, since the behavior of the model in the region close to the current iterate will have different characteristics from an interpolating model.

A general problem in using the Framework is what to do when the `SEARCH` strategy fails. The algorithms investigated so far fall back immediately on the `POLL` algorithm. Because the cost of polling can be high, it is desirable to avoid this whenever possible. It may be useful to choose points not suggested by `SEARCH` and evaluate the actual objective function at them in hopes of finding decrease or improving the model. A general purpose speculation strategy is to extrapolate along the path taken by the several most recent trial iterates and evaluate there. This simple idea raises several issues: how many trial iterates to use; how to define the path (e.g., use a curve fit of some kind); how far to project. The Hooke and Jeeves pattern search method [37] exemplifies this idea.

This idea may have particular benefit when applied to bound constrained problems. The strictly feasible nature of the MMF approach makes it difficult to accurately model a region near a boundary when only the points produced by `SEARCH` are used to update the model. If the model does not predict decrease close to the boundary, no points will be chosen there, so if the model is wrong it will not be corrected without executing `POLL`. An algorithm to speculatively evaluate points on the boundary (or beyond it, if the objective function and model are both known to work outside the bounds) could have a significant impact on performance in this case.

The MMF, like all pattern search methods, suffers from a sensitivity to problem scaling. Since there is no gradient information, the scaling must be applied explicitly. A poor choice of scaling can have a large negative impact on the performance of a pattern search method, in the worst case reducing it to nothing more than a sequence of line searches in the coordinate directions. Ideally, the scaling should dynamically adapt to the function during the execution of the program. This issue should be dealt with at the level of the component algorithms since it affects performance but not convergence. A suite of tools for computing and modifying scaling data would be very useful to algorithm designers and implementers.

Another open issue in both model management algorithm design and pattern search algorithm design is the effect the choice of core pattern has on performance. We assume in general that a smaller (fewer points) core pattern is preferable because it costs less to evaluate in the worst case. However, this is balanced by the better sampling achieved by a core pattern with more points. This balance should be investigated.

7.2 MMF extensions

The Model Management Framework implementation should be extended to allow for modeling the constraints. One approach is to add models for each constraint and manage them individually, using the same techniques used to manage the model of the objective function. This has the disadvantage of increasing complexity. Another approach is to treat all the constraints with a single model and define a norm to use in computing actual vs. predicted ratios. The usual issues of constrained optimization arise: feasible vs. infeasible methods, choosing Lagrange multiplier, identifying active sets, etc. An additional concern is maintaining efficiency of the method when the solution is near a constraint and the constraint model is not very accurate.

The implementation should also be extended to allow parallel calculation of the function values needed to check the trial steps produced by `SEARCH` and to poll the core pattern. There are a few algorithm design issues related to this but they are relatively minor. This is mostly a performance issue.

Another extension of major interest in engineering design is the multi-objective optimization problem. This raises some of the same issues for model management as in the constraint modeling problem just described, compounded by the relative immaturity of multi-objective optimization methods in general.

There are several possible extensions to the model management convergence theory that are of interest. Torczon and Lewis are extending the underlying pattern search convergence theory to linear constraints and this extension should be incorporated into the model management theory and implementation. The pattern search theory can likely be extended to allow the scale factor contraction ratio to vary across iterations (currently it must be constant). The most likely conjecture is that the ratio need only be an integer power of some minimum ratio. It is possible that a stronger conjecture can be proven allowing the ratio to take on other values. Allowing the ratio to vary would allow for faster convergence than is currently possible. A extremely useful extension to the theory would be to relax the requirement that all the trial iterates be taken from the lattice, but this currently does not appear possible without adding an explicit sufficient decrease condition. Model management algorithm design would be greatly simplified if any point with decrease could be chosen.

Another useful extension to the Framework would be a mechanism to invoke the DFO algorithm when the current iterate is in the basin of the minimizer (because the DFO algorithm has faster asymptotic convergence). Ideally, the mechanism would allow a “warm start” of DFO using function values computed during the model management process. This requires an extension to the DFO algorithm. This should improve the performance of DFO by allowing a better initial model to be constructed than in the current algorithm. It will also eliminate the randomness currently present in the choice of the first step.

The combination of the rapid initial decrease possible with the Model Management Framework combined with the fast local convergence of the DFO algorithm could provide a potent tool for applications where high solution accuracy is required.

Appendix A

Model Management Framework Programmer's Guide

This document describes the Model Management Framework (MMF) software v2.0, including the data structures and subroutines used within the framework. This information should be useful to programmers using the framework in an application program and implementing algorithms for the framework. It may also be useful to anyone trying to understand the framework methodology itself. Additional details can be found in my thesis, especially the theory chapter.

A.1 Introduction

A computer program that uses the model management framework (MMF) can be divided hierarchically into three levels: the top level application code, the mid-level framework code, and the low level algorithm-specific code. Figure 1 shows a diagram of this hierarchy.

The fundamental concept of the framework is that the middle level code is relatively fixed, as in an ordinary subroutine library, but the low level code is changed in order to implement different model managment algorithms. The interfaces to the low level code stay the same, but the implementations change. The goal is to allow for algorithms to change without modification to the mid-level framework code or the top level application code.

A.1.1 Top level: Application

The application is any program that needs to use the MMF. There is one primary interface between the application and the framework. That interface is the `MMFPS` subroutine. From the application's point of veiw, the framework is like any other optimization subroutine – it takes as input an objective function and an initial point and tries to return a point with a lower objective function value. The unusual feature of the framework compared to most optimization software is that in addition to the routine to evaluate the objective function it also takes a routine to evaluate a model of the objective. (It also takes a constraint routine and a model of it, but we will ignore that for the moment.)

In addition to the usual optimization data, the `MMFPS` subroutine requires some data related to various control parameters in MMF. The framework provides data structures to encapsulate and communicate this data and subroutines to manipulate the structures easily.

A.1.2 Middle level: Framework

The framework code is implemented as a collection of Fortran90 *modules*. The `MMFModule` module is exported to the application. It defines the `MMFPS` subroutine and the data structures used in the interface between the framework and the application. It also defines support routines that manipulate these data structures.

The framework implements a fairly general type of pattern search optimization method. The method is decomposed into a small set of algorithmic steps that are further decomposed into a set of specific routines with precisely defined interfaces and behavior. The behavior specifications come from the convergence theory. The interface specifications allow the low-level algorithm details to be modified without changing the framework code.

A.1.3 Low level: Algorithm and Tools

The framework provides the outline of an optimization method. The low level software fills in the details. There are four subroutines that define a model management algorithm in the framework. The subroutines are contained in the `MMFSubrsModule` module. One routine (`SEARCH`) is used to search for a new point that improves on the current point. The second (`MANAGEMDL`) implements the strategy for updating the model(s). The third (`TERMINATE`) decides when to stop searching. The fourth (`POLL`) implements a fallback strategy for when the search fails. This fallback strategy is required by the theory in order to guarantee convergence. These four subroutines are defined in a module called `MMFSubrsModule`.

There are other low-level subroutines used in the framework. These provide basic functionality that is necessary for many model management algorithms, like model manipulation, solution to optimization subproblems, function evaluation, etc. In general these routines are "black boxes" in relation to the framework code, although in some cases the implementations may impact the low level model management algorithm routines (e.g. the implementation of the modeling routines may affect the `MANAGEMDL` algorithm routine).

A.2 Exception Handling

All procedures in the Model Management Framework have an argument for exception identification information. Fortran90 procedures in the top and middle layers use a variable of type `Status_t` which is defined in `MMFTypesModule`. Lower level F90 procedures and F77 procedures use a scalar integer variable. The `%error` component of `Status_t` has the same meaning as the integer variable used in the other procedures.

By convention, values of the `%error` variable have the meanings shown below.

- $= 0$ indicates the procedure succeeded
- < 0 indicates an unrecoverable exception (error) occurred
- > 0 indicates a recoverable exception (warning) occurred

An unrecoverable exception should only be returned when the output values of the procedure cannot be computed. A recoverable exception is returned when the output values are computed but execution of the procedure did not proceed normally. The value returned should identify the nature of the exception as precisely as possible in order to allow the calling procedure to take appropriate action.

The `Status_t` type has two other components besides `%error`; `%errnum` and `%errmsg`. The `%errnum` component is an integer variable and the `%errmsg` component is a character string. The value of `%errnum` is used to provide additional information about the exception condition. For example, if the `%error` value indicates an I/O error occurred, the `%errnum` value could be the IOSTAT value from the I/O operation. The `%errmsg` should be a message describing the exception that is suitable for a user to read. It should indicate what the problem was and where it occurred.

This is not necessarily the best possible mechanism for exception handling, but it is better than none.

A.3 Descriptions of Module Contents

A.3.1 Module `MMFModule`

This module implements the application interface to the MMF. This is the only module that the application should have to access (via the F90 `USE` statement). The module provides the `MMFPS` subroutine, the data types used in the arguments to `MMFPS`, and subroutines to manipulate the data types.

The arguments to MMFPS can be separated into several groups.

Three of the arguments represent the data in the optimization problem. Two of the arguments are the input initial solution and the output final solution. It is assumed that the application code evaluates the objective and constraint functions at the initial point and ensures it is feasible. MMFPS ensures the final solution is feasible. The third argument in this group is a data type that contains all the constraint information: upper and lower bounds, coefficients of the linear constraints, right-hand sides of the equality constraints, etc. The constraint variable is initialized by the application and its contents are not modified by the MMF. A subroutine is provided to read the constraint data from a file. (The format for this and other MMF files is documented in Section 4.)

Four of the arguments are the subroutines to evaluate the objective and constraint functions and their models. A model can be omitted by passing the same subroutine for the model as for the function itself. This should only be done when the actual function is inexpensive to compute and deterministic. (In MMF v2.0, the objective model **cannot** be omitted.) The interfaces to these subroutines are specified in the MMF Interface Specifications v1.6.

Three of the arguments are control variables for various aspects of the model management algorithm. One defines the stopping criteria for the iterative pattern search method used in MMFPS. Another defines various control parameters for the pattern search. The third defines which functions have their values cached. (Caching is a win if the function is expensive to compute, but can be more costly if the function take only a few CPU seconds.)

A.3.2 Module `MMFSubrsModule`

This module contains four subroutines: `SEARCH`, `MANAGEMDL`, `TERMINATE`, and `POLL`. The behavior of these routines is described in the MMF theory discussion. The `MMFModule` module `USE`s this module. These four subroutines determine the details of the model management algorithm that gets executed by the framework.

`SEARCH` Select one or more points in the pattern that are candidates for the next approximate solution.

`MANAGEMDL` Update the models in whatever manner is appropriate.

`TERMINATE` Decide whether to call `SEARCH` again or stop iterating and call `POLL`.

`POLL` If no improvement in the objective function was found by calling `SEARCH`, evaluate the objective at points in the core pattern until one with improvement is found.

Subroutine `SEARCH`

This subroutine returns a set of trial solutions which may produce decrease in the actual objective. The only requirement is that the points in the trial solutions be on the current grid. The trial solutions are returned in an array of `MMFSoln_t`. The array is dynamic and may have zero elements.

The interface for `SEARCH` is:

call `SEARCH( OriginalSoln, CurrentSoln, Pattern, Constraints, FcnCache,`
`OBJ, CON, OBJMDL, CONMDL, trial_solns, eval_counts, status )`

where the arguments to `SEARCH` are:

`OriginalSoln` (**type**: `MMFSoln_t`, **intent**: in) The solution at the start of the current exploratory moves (before iteration 1). This is the solution the exploratory moves must improve on. It is not strictly necessary for `SEARCH`.

`CurrentSoln` (**type**: `MMFSoln_t`, **intent**: in) The current solution in the exploratory moves. This is the solution that this iteration is attempting to improve on. During iteration 1, `CurrentSoln .EQ. OriginalSoln`.

`Pattern` (**type**: `Pattern_t`, **intent**: in) The definition of the current pattern. This contains the definition of the current grid in the `Pattern%basis` and `Pattern-%scale` variables.

`Constraints` (**type**: `Constraint_t`, **intent**: in) The constraint data (bounds, coefficients, etc.) needed by `MMFEVAL` in order to determine feasibility of a point. This structure is opaque (none of the components should be accessed directly).

`FcnCache` (**type**: `FcnCache_t`, **intent**: in) The cache data for the "truth" functions (`OBJ` and `CON`) needed by `MMFEVAL`. This structure is opaque.

`OBJ,OBJMDL` (**type**: **subroutine**) Subroutines to evaluate the objective function. `OBJ` evaluates the actual function. `OBJMDL` evaluates the corresponding model(s). Both must conform to the interface specification for objective function evaluation routines defined in the *MMF Interface Specification v1.6* document.

`CON,CONMDL` (**type**: **subroutine**) Subroutines to evaluate the nonlinear inequality constraint function. `CON` evaluates the actual function. `CONMDL` evaluates the corresponding model(s). Both must conform to the interface specification for constraint function evaluation routines defined in the *MMF Interface Specification v1.6* document. If there is no model of the constraint function, `CONMDL` may be the same as `CON`. If there are no nonlinear inequality constraints, the subroutine `CONDUMMY` can be used as the actual argument for both `CON` and `CONMDL`.

`trial_solns` (**type**: pointer to `MMFSoln_t(:)`, **intent**: inout) On entry, this pointer is associated with an array of solutions (possibly zero-size). On exit, each solution in the array must correspond to a point on the grid. The array may be returned with zero size. If the size of the array is changed, `SEARCH` must deallocate the pointer before allocating new array space. `SEARCH` must not be implemented to rely on the continued existence of the array space associated with `trial_solns` after `SEARCH` exits. The pointer must not be returned in the undefined or null states.

`eval_counts` (**type**: integer(2), **intent**: out) On exit, returns the number of calls to the `OBJ` and `CON` subroutines in the first and second elements of the array (respectively).

`status` (type: `Status_t`, **intent**: out) On exit, contains exception information. See Section A.2 for details.

Subroutine `MANAGEMDL`

The primary purpose of this subroutine is to modify the model to adapt to the most recent results. There is no theoretical requirement that the model be modified in any particular way. The subroutine is allowed to modify the new solution, but if it does the new solution must fulfill the same theoretical requirements as the result of the `POLL` routine: namely that the solution must correspond to a step from the pattern at the current pattern scale factor `Pattern%scale`, must be feasible, and must decrease the actual object function. The new solution need not be in the array of trial solutions.

The interface for `MANAGEMDL` is:

call `MANAGEMDL( OriginalSoln, CurrentSoln, Pattern, Constraints, FcnCache, OBJ, CON, OBJMDL, CONMDL, TrialSolns, NumTrialSolnsEval, new_soln, eval_counts, status )`

where the arguments to `MANAGEMDL` are:

`OriginalSoln` (**type**: `MMFSoln_t`, **intent**: in) The solution at the start of the current exploratory moves (before iteration 1). This is the solution the exploratory moves must improve on.

`CurrentSoln` (**type**: `MMFSoln_t`, **intent**: in) The current solution in the exploratory moves. This is the solution that this iteration is attempting to improve on. During iteration 1, `CurrentSoln .EQ. OriginalSoln`.

`Pattern` (**type**: `Pattern_t`, **intent**: in) The definition of the current pattern. This contains the definition of the current grid in the `Pattern%basis` and `Pattern-%scale` variables.

`Constraints` (**type**: `Constraint_t`, **intent**: in) The constraint data (bounds, coefficients, etc.) needed by `MMFEVAL` in order to determine feasibility of a point. This structure is opaque (i.e. none of the components should be accessed directly).

`FcnCache` (**type**: `FcnCache_t`, **intent**: in) The cache data for the "truth" functions (`OBJ` and `CON`) needed by `MMFEVAL`. This structure is opaque.

`OBJ,OBJMDL` (**type**: **subroutine**) Subroutines to evaluate the objective function. `OBJ` evaluates the actual function. `OBJMDL` evaluates the corresponding model(s). Both must conform to the interface specification for objective function evaluation routines defined in the *MMF Interface Specification v1.6* document.

`CON,CONMDL` (**type**: **subroutine**) Subroutines to evaluate the nonlinear inequality constraint function. `CON` evaluates the actual function. `CONMDL` evaluates the corresponding model(s). Both must conform to the interface specification for constraint function evaluation routines defined in the *MMF Interface Specification v1.6* document. If there is no model of the constraint function, `CONMDL` may be the same as `CON`. If there are no nonlinear inequality constraints, the subroutine `CONDUMMY` can be used as the actual argument for both `CON` and `CONMDL`.

`TrialSolns` (**type**: `MMFSoln_t(:)`, **intent**: in) Array of solutions (possibly zero-size) returned by `SEARCH`.

`NumTrialSolnsEval` (**type**: integer, **intent**: in) The number of elements of `TrialSolns` for which the actual functions have been evaluated. `NumTrialSolnsEval .LE.` **SIZE**`(TrialSolns)`. The elements of `TrialSolns` are evaluated in order starting with the first element.

`new_soln` (**type**: MMFSoln_t, **intent**: inout) On entry, contains the current best solution. If different from `CurrentSoln` then `new_soln` .EQ. `TrialSolns(NumTrialSolnsEval)`. If not, no decrease was found in the current `MMFEM` iteration. On exit, `new_soln` may be different if the returned value satisfies all the conditions on the result of `POLL()`.

`eval_counts` (**type**: integer(2), **intent**: out) On exit, returns the number of calls to the `OBJ` and `CON` subroutines in the first and second elements of the array (respectively). Should be explicitly set to zero if no calls are executed.

`status` (**type**: `Status_t`, **intent**: out) On exit, contains exception information. See Section A.2 for details.

Subroutine `TERMINATE`

The sole purpose of this subroutine is to decide when to stop the loop in `MMFEM` that calls `SEARCH` and `MANAGEMDL`. The implementation of `TERMINATE` is not allowed to modify the current solution. The implementation must guarantee that the value `.TRUE.` is returned for some finite value of the iteration counter `NIters`.

The interface for `TERMINATE` is:

call `TERMINATE(` `Niters`, `OriginalSoln`, `PreviousSoln`, `CurrentSoln`,
`Pattern`, `FcnCache`,
`OBJ`, `CON`, `OBJMDL`, `CONMDL`, `TrialSolns`, `NumTrialSolnsEval`,
`tc`, `status`)

where the arguments to `TERMINATE` are:

`NIters` (**type**: integer, **intent**: in) The number of `MMFEM` iterations executed so far.

`OriginalSoln` (**type**: `MMFSoln_t`, **intent**: in) The solution at the start of the current exploratory moves (before iteration 1). This is the solution the exploratory moves must improve on.

`PreviousSoln` (**type**: `MMFSoln_t`, **intent**: in) The solution from the previous `MMFEM` iteration. This is the solution that this iteration is attempting to improve on. During iteration 1, `PreviousSoln .EQ. OriginalSoln`.

`CurrentSoln` (**type**: `MMFSoln_t`, **intent**: in) The solution in the current `MMFEM` iteration. If the iteration failed to find a step with decrease, this will be the same point as `PreviousSoln`.

`Pattern` (**type**: `Pattern_t`, **intent**: in) The definition of the current pattern. This contains the definition of the current grid in the `Pattern%basis` and `Pattern%scale` variables.

`Constraints` (**type**: `Constraint_t`, **intent**: in) The constraint data (bounds, coefficients, etc.) needed by `MMFEVAL` in order to determine feasibility of a point. This structure is opaque (i.e. none of the components should be accessed directly).

`FcnCache` (**type**: `FcnCache_t`, **intent**: in) The cache data for the "truth" functions (`OBJ` and `CON`) needed by `MMFEVAL`. This structure is opaque.

`OBJ,OBJMDL` (**type**: **subroutine**) Subroutines to evaluate the objective function. `OBJ` evaluates the actual function. `OBJMDL` evaluates the corresponding model(s). Both must conform to the interface specification for objective function evaluation routines defined in the *MMF Interface Specification v1.6* document.

`CON,CONMDL` (**type**: **subroutine**) Subroutines to evaluate the nonlinear inequality constraint function. `CON` evaluates the actual function. `CONMDL` evaluates the corresponding model(s). Both must conform to the interface specification for constraint function evaluation routines defined in the *MMF Interface Specification v1.6* document. If there is no model of the constraint function, `CONMDL` may be the same as `CON`. If there are no nonlinear inequality constraints, the subroutine `CONDUMMY` can be used as the actual argument for both `CON` and `CONMDL`.

`TrialSolns` (**type**: `MMFSoln_t(:)`, **intent**: in) Array of solutions (possibly zero-size) returned by `SEARCH`.

`NumTrialSolnsEval` (**type**: integer, **intent**: in) The number of elements of `TrialSolns` for which the actual functions have been evaluated by `MMFEM`. `NumTrialSolnsEval .LE.` **SIZE**`(TrialSolns)`. The elements of `TrialSolns` are evaluated in order starting with the first element.

`tc` (**type**: TermCond_t, **intent**: inout) The termination conditions data. The only components of this structure that may be modified safely are the function evaluation counts (`tc%nobjevals` and `tc%nconevals`).

`status` (**type**: `Status_t`, **intent**: out) On exit, contains exception information. See Section A.2 for details.

Subroutine `POLL`

This subroutine is only called when the `SEARCH/MANAGEMDL` loop in `MMFEM` fails to find a new solution which decreases the actual objective function. The convergence theory requires that the actual objective function must be evaluated at all the steps in the core pattern before this routine is allowed to return with a zero step (i.e. return with a new solution the same as the old solution). If a point corresponding to a step in the core pattern is found that decreases the objective, it can be returned without evaluating the objective at the rest of the points. Only points corresponding to steps in the core pattern may be returned as the new solution.

The core pattern is represented as a 2D array of integer values. Each column represents steps in each dimension away from the current iterate. The formula to compute the points corresponding to the core steps is:

$$x_{c_i} = \texttt{OldSoln\%x} + \texttt{Pattern\%scale} * \texttt{Pattern\%basis} * \texttt{Pattern\%core}(:, \texttt{i}).$$

The interface for `POLL` is:

$$\textbf{call}\ \texttt{POLL(}\ \begin{array}{l}\texttt{OldSoln}, \texttt{Pattern}, \texttt{Constraints}, \texttt{FcnCache}, \\ \texttt{OBJ}, \texttt{CON}, \texttt{OBJMDL}, \texttt{CONMDL}, \\ \texttt{new\_soln}, \texttt{tc}, \texttt{status}\ )\end{array}$$

where the arguments to `POLL` are:

`OldSoln` (**type**: `MMFSoln_t`, **intent**: in) The current best solution. This is the value that was input to `MMFEM`. It is the solution that `POLL` needs to improve.

`Pattern` (**type**: `Pattern_t`, **intent**: in) The definition of the current pattern. This contains the definition of the current grid in the `Pattern%basis` and `Pattern%scale` variables.

`Constraints` (**type**: `Constraint_t`, **intent**: in) The constraint data (bounds, coefficients, etc.) needed by `MMFEVAL` in order to determine feasibility of a point. This structure is opaque (i.e. none of the components should be accessed directly).

`OBJ,OBJMDL` (**type**: **subroutine**) Subroutines to evaluate the objective function. `OBJ` evaluates the actual function. `OBJMDL` evaluates the corresponding model(s). Both must conform to the interface specification for objective function evaluation routines defined in the *MMF Interface Specification v1.6* document.

`CON,CONMDL` (**type**: **subroutine**) Subroutines to evaluate the nonlinear inequality constraint function. `CON` evaluates the actual function. `CONMDL` evaluates the corresponding model(s). Both must conform to the interface specification for constraint function evaluation routines defined in the *MMF Interface Specification v1.6* document. If there is no model of the constraint function, `CONMDL` may be the same as `CON`. If there are no nonlinear inequality constraints, the subroutine `CONDUMMY` can be used as the actual argument for both `CON` and `CONMDL`.

`new_soln` (**type**: MMFSoln_t, **intent**: inout) On entry, must be `INITIALIZE`d. The input values are irrelevant. On exit, `new_soln` must either be the same as `OldSoln` or must produce decrease in the objective function `OBJ` and must be feasible and must be from the current core pattern (`Pattern%core`). If any of the points in the core pattern produce decrease, then `new_soln` *must* also produce decrease.

`eval_counts` (**type**: integer(2), **intent**: out) On exit, returns the number of calls to the `OBJ` and `CON` subroutines in the first and second elements of the array (respectively). Should be explicitly set to zero if no calls are executed.

`status` (**type**: `Status_t`, **intent**: out) On exit, contains exception information. See Section A.2 for details.

A.3.3 Module `ConstraintModule`

This module defines the `Contraint_t` data type and the `INITIALIZE`, `RELEASE`, `READ`, `WRITE` and `CHECKCON` generic subroutines. A set of F77 routines are defined in the same file that all operate on a shared constraint variable stored in the module. For most applications only one set of constraint data is needed so this should suffice.

Constraint handling in the MMF is straightforward. All solutions must be feasible, so the only important operation that uses the constraint data is checking for feasibility. To do this may require evaluating the nonlinear inequality constraint function if the bound and linear inequality constraints are satisfied. The `CHECKCON` routine (or the `CHECKCONSTR` F77 routine) determines the feasibility of a given solution. The `MMFEVAL` routine defined in `MMFTypesModule` calls `CHECKCON` as part of evaluating the solution at a point.

A.3.4 Module `MMFTypesModule`

This module defines most of the defined types that are specific to the MMF, and subroutines to manipulate them. The types are:

`MMFSoln_t` An approximate solution to the optimization problem. The `MMFEVAL` routine is used to set the components in this structure.

- `%feasible` (logical) $.TRUE.$ when the point `%x` is feasible. $.FALSE.$ otherwise.
- `%isconeval` (logical) $.TRUE.$ when the nonlinear constraint function has been evaluated since the type was initialized. $.FALSE.$ otherwise.
- `%isobjeval` (logical) $.TRUE.$ when the objective function has been evaluated since the type was initialized. $.FALSE.$ otherwise.
- `%x` (real, pointer[:]) contains the coordinates of the solution point.
- `%con` (real, pointer[:]) when `%isconeval` is $.TRUE.$, contains the values of the nonlinear constraint function at `%x`. Otherwise undefined.
- `%obj` (real) when `%isobjeval` is $.TRUE.$, contains the value of the objective function at `%x`. Otherwise undefined.

`Pattern_t` Data for the pattern used in the pattern search method in `MMFPS`.

`TermCond_t` Termination condition data for `MMFPS`.

`FcnCache_t` Function cache identifiers for the objective and constraint functions.

`Status_t` Describes the exception, if any, that occurred during the execution of a procedure.

The module implements several generic routines for some or all of the types. These are:

`INITIALIZE` Set initial values in a data structure, allocating array space if necessary.

`RELEASE` Unset values in a data structure, deallocating array space.

`READ` Initialize a data structure by reading values from a file.

`PRINT` Write the values in a data structure in human-readable (pretty-printed) form.

`WRITE` Write the values in a data structure into a file that is acceptable to `READ`.

The module also defines the `MMFEVAL` subroutine, which evaluates the objective and constraint functions at a given point and determines feasibility. It also handles the function caches. `MMFEVAL` can be used with the actual functions or the models thereof.

A.3.5 Module `MMFModelModule`

This module implements the standardized interfaces to the objective and constraint model manipulation routines. Since the interfaces are defined at the Fortran77 level, the actual code for these routines is not part of the module. This enables the routines to be called by Fortran77 code that can't `USE` the module. The module also contains the F90 `INTERFACE` definitions for the F77 routines.

Two instantiations of this module are provided with the MMF. One implements polynomial models using the spline library written by Thomas Grandine. The other implements the DACE models described by Welch et al. in [74] and developed by Andrew Booker. In both cases, not all the functionality of the underlying software libraries is available through the module routines. Only the functionality needed by the MMF is implemented. Specifically, routines to initialize, evaluate, and calibrate a model with scalar or vector values are provided. Both versions of the module implement the same subroutine interfaces. In fact, the code in the two versions is almost identical, as each relies on a lower level module that provides a structure that encapsulates the information in the model and generic routines to manipulate the structure. The routines in `MMFModelModule` simply instantiate separate models for the objective and constraint functions and hide the data structures behind subroutine calls that match the MMF interface specifications.

The subroutines provided by `MMFModelModule` are:

`EVALOBJMDL` Evaluate the model of the objective function.

`EVALCONMDL` Evaluate the model of the constraint function.

`CALOBJMDL` Calibrate the model of the objective function at a single point.

`CALCONMDL` Calibrate the model of the constraint function at a single point.

`SETOBJMDLFILE` Specify the name of the file that defines the objective function model.

`SETCONMDLFILE` Specify the name of the file that defines the constraint function model.

Each model is initialized the first time it is used by reading the model data from an external file. It is possible to initialize the model directly using the underlying F90 code, but no F77 interface to this functionality is provided. The `SET*MDLFILE` routines allow the names of the input files to be set before the first use of the model. The default file names are `''objective.mdl''` and `''constraint.mdl''`.

Bibliography

[1] Natalia Alexandrov, J. E. Dennis, Jr., Robert Michael Lewis, and Virginia Torczon. A trust region framework for managing the use of approximation models in optimization. *Structural Optimization*, 15(1):16–23, February 1998.

[2] Jean-Francois M. Barthelemy and Raphael T. Haftka. Approximation concepts for optimum structural design – a review. *Structural Optimization*, 5:129–144, 1993.

[3] A. J. Booker. DOE for computer output. Technical Report BCSTECH–94–052, Boeing Computer Services, Research and Technology, M/S 7L–68, Seattle, Washington 98124, December 1994.

[4] A. J. Booker. DACEPAC User's Manual. Informal report, July 1996.

[5] A. J. Booker, J. E. Dennis, Paul D. Frank, David B. Serafini, Virginia Torczon, and Michael W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. To appear in *Structural Optimization.*, 1998.

[6] Andrew J. Booker, Andrew R. Conn, John E. Dennis, Paul D. Frank, Michael Trosset, and Virginia Torczon. Global modeling for optimization: Boeing/IBM/Rice collaborative project 1995 final report. Informal report, 1995.

[7] Andrew J. Booker, Paul D. Frank, Andrew R. Conn, John E. Dennis, David Serafini, Virginia Torczon, and Michael Trosset. Multi-level design optimization: Boeing/IBM/Rice collaborative project 1996 final report. Technical Report BISSTECH–96–031, Boeing Computer Services, Research and Technology, Box 3707, M/S 7L–68, Seattle, Washington 98124, December 1996.

[8] D. M. Bortz and C. T. Kelley. The simplex gradient and noisy optimization problems. Technical Report CRSC-TR97-27, Center for Research in Scientific Computation, North Carolina State University, Raleigh, NC 27695, September 1997. To appear in *Proceedings of AFOSR Workshop on Optimal Design and Control, Arlington, VA, September 30 – October 3, 1997.* <http://www4.ncsu.edu/eos/users/c/ctkelley/www/pubs.html<¿.

[9] G. E. P. Box and N. R. Draper. *Empirical Model Building and Response Surfaces.* New York: Wiley, 1987.

[10] George E. P. Box. Evolutionary operation: A method for increasing industrial productivity. *Applied Statistics*, 6(2):81–101, June 1957.

[11] S. L. Burgee, A. A. Giunta, V. Balabanov, B. Grossman, W. H. Mason, R. Narducci, R. T. Haftka, and L. T. Watson. A coarse grained parallel variable–complexity multidisciplinary optimization program. Technical Report 95–20, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, October 1995.

[12] Richard G. Carter. Numerical experience with a class of algorithms for nonlinear optimization using inexact function and gradient information. *SIAM J. of Scientific Computing*, 14(2):368–388, 1993.

[13] C. K. Chui. *Multivariate Splines.* CBMS-NSF Regional Conference Series in Applied Mathematics. Society of Industrial and Applied Mathematics, 1988.

[14] A. R. Conn, K. Scheinberg, and Ph. L. Toint. On the convergence of derivative-free methods for unconstrained optimization. Technical Report 96/10, Department of Mathematics, Faculté Universitaires, B-5000 Namur, Belgium, December 1996.

[15] A. R. Conn and Ph. L. Toint. An algorithm using quadratic interpolation for unconstrained derivative free optimization. In Gianni Di Pillo and Franco Giannessi, editors, *Nonlinear Optimization and Applications.* Plenum Publishing, 1995.

[16] Noel A. C. Cressie. *Statistics for spatial data.* J. Wiley, New York, 1991.

[17] C. Davis. Theory of positive linear dependence. *American Journal of Mathematics*, pages 448–474, 1954.

[18] Lawrence Davis, editor. *Handbook of genetic algorithms.* Van Nostrand Reinhold, New York, 1991.

[19] Carl de Boor and A. Ron. On multivariate polynomial interpolation. *Constructive Approximation*, 6:287–302, 1990.

[20] Carl deBoor and A. Ron. Computational aspects of polynomial interpolation in several variables. *Mathematics of Computation*, 58(198):705–727, 1992.

[21] J. E. Dennis and Virginia Torczon. Managing approximation models in optimization. In Natalia Alexandrov and M.Y. Hussaini, editors, *Multidisciplinary Design Optimization: State-of-the-Art*, pages 330–347. SIAM, Philadelphia, 1997. Also available as Rice U. CAAM Tech. Report 95-19.

[22] J. E. Dennis, Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, volume 16 of *Classics in Applied Mathematics.* SIAM, Philadelphia, 1996.

[23] J. E. Dennis, Jr. and Virginia Torczon. Direct search methods on parallel machines. *SIAM J. Optimization*, 1(4):448–474, November 1991.

[24] L. C. W. Dixon and G. P. Szegö, editors. *Towards Global Optimization 2.* North-Holland Pub. Co., 1978.

[25] R.A. Fisher. *The Design of Experiments.* Oliver and Boyd, Edinburgh, 1935.

[26] C. A. Floudas and P.M. Pardalos, editors. *State of the Art in Global Optimization: Computational Methods and Applications.* Kluwer Academic Publishers, 1996.

[27] P. D. Frank. Private communication. February 1998.

[28] P. D. Frank. Brief description of the helicopter rotor blade design project. Informal report, September 1994.

[29] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization.* Academic Press, London, 1981.

[30] A. A. Giunta, V. Balabanov, D. Haim, B. Grossman, W. H. Mason, L. T. Watson, and R. T. Haftka. Wing design for a high-speed civil transport using a design of experiments methodology. In *Proceedings of the Sixth AIAA/NASA/USAF Symposium on Multidisciplinary Analysis and Optimization, Bellevue, WA, 1996.* American Institute of Aeronautics and Astronautics, 1996. AIAA Paper 96-4001.

[31] A. A. Giunta, J. M. Dudley, R. Narducci, B. Grossman, R. T. Haftka, W. H. Mason, and L. T. Watson. Noisy aerodynamic response and smooth approximation in HSCT design. In *Proceedings of the Fifth AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, September 7–9, 1994*, pages 1117–1128. American Institute of Aeronautics and Astronautics, 1994.

[32] Torkel Glad and Allen Goldstein. Optimization of functions whose values are subject to small errors. *BIT*, 17(2):160–169, 1977.

[33] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley Pub. Co., Reading, MA., 1989.

[34] Thomas Grandine. Private communication. The Multi-Variate Polynomial (MVP) library is not publicly available. We are grateful to Dr. Grandine and to Boeing Information Services for the use of this software.

[35] William E. Hart. A generalized stationary point convergence theory for evolutionary algorithms. In Thomas Baeck, editor, *Proc 7th Intl Conf on Genetic Algorithms (ICGA97)*, pages 127–134, San Francisco, CA, 1997. Morgan Kaufmann.

[36] William E. Hart. A stationary point convergence theory for evolutionary algorithms. In Richard K. Belew and Michael D. Vose, editors, *Foundations of Genetic Algorithms 4*, pages 325–342, San Fransico, CA, 1997. Morgan Kaufmann Publishers, Inc.

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

[38] R. Horst and P.M. Pardalos, editors. *Handbook of Global Optimization.* Kluwer Academic Publishers, 1995.

[39] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Application*, 79(1), October 1993.

[40] M. Kaufman, V. Balabanov, B. Grossman, W. H. Mason, L. T. Watson, and R. T. Haftka. Multidisciplinary optimization via response surface techniques. In *Proceedings of the 36th Israel Conference on Aerospace Sciences, Israel*, pages A–57–A–67, February 21–22, 1996.

[41] C. T. Kelley. Design of automotive valve trains with implicit filtering. presented at the Fifth SIAM Conference on Optimization, Victoria, B.C., May 20–22, 1996; Web address http://www4.ncsu.edu/eos/users/c/ctkelley/www/iffco.html.

[42] C.T. Kelley. Detect and remediation of stagnation in the nelder-mead algorithm using a sufficient decrease condition. Technical Report CRSC-TR97-2, Center for Research in Scientific Computation, North Carolina State University, Raleigh, NC 27695, January 1997. Submitted for publication.

[43] A. I. Khuri and J. A. Cornell. *Response Surfaces: Designs and Analyses.* Marcel Dekker, New York, 1987.

[44] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the Nelder-Mead simplex algorithm in low dimensions. To appear in *SIAM Journal on Optimization*.

[45] Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems.* SIAM, 1995. Classics in Applied Mathematics 15.

[46] David Levine. Users guide to the PGAPack parallel genetic algorithm library. Technical Report ANL-95/18, Argonne National Laboratory, January 1996. Available from URL ftp://info.mcs.anl.gov/pub/pgapack/pgapack.tar.Z.

[47] Robert Michael Lewis and Virginia Torczon. Pattern search algorithms for bound constrained minimization. Technical Report 96–20, ICASE, NASA Langley Research Center, Hampton, VA 23681–0001, March 1996. To appear in *SIAM Journal on Optimization*.

[48] Robert Michael Lewis and Virginia Torczon. Rank ordering and positive bases in pattern search algorithms. Technical Report 96–71, ICASE, NASA Langley Research Center, Hampton, VA 23681–0001, December 1996. To appear in *Mathematical Programming*.

[49] Robert Michael Lewis and Virginia Torczon. Pattern search methods for linearly constrained minimization. Technical Report 98–3, ICASE, NASA Langley Research Center, Hampton, VA 23681–0001, January 1998. Submitted to *SIAM Journal on Optimization*.

[50] J. H. May. *Linearly Constrained Nonlinear Programming: A Solution Method That Does Not Require Analytic Derivatives.* PhD thesis, Yale University, December 1974.

[51] K. I. M. McKinnon. Convergence of the Nelder-Mead simplex method to a nonstationary point. Technical Report MS 96–006, Department of Mathematics and Statistics, University of Edinburgh, May 1996. To appear in *SIAM Journal on Optimization*.

[52] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer–Verlag, 1992.

[53] R. Mifflin. A superlinearly convergent algorithm for minimization without evaluating derivatives. *Mathematical Programming*, 9:100–117, 1975.

[54] J. Mockus. *Bayesian Approach to Global Optimization.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1989.

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

[56] A. B. Owen. Orthogonal arrays for computer experiments, integration and visualization. *Statistica Sinica*, 2:439–452, 1992.

[57] P.M. Pardalos and J.B. Rosen. *Constrained Global Optimization: Algorithms and Applications.* Springer–Verlag, 1987.

[58] Elijah Polak. *Computational Methods in Optimization: A Unified Approach.* Academic Press, New York, 1977.

[59] M. J. D. Powell. Trust region methods that employ quadratic interpolation of the objective function. Presented at the Fifth SIAM Conference on Optimization, Victoria, B.C., May 20–22, 1996, May 1996.

[60] Jose F. Rodriguez, John E. Renaud, and Layne T. Watson. Trust region augmented lagrangian methods for sequential response surface approximation and optimization. In *Proceedings of the 1997 ASME Design Engineering Technical Conferenes, Sacramento, California.* American Society of Mechanical Engineers, 1997.

[61] Matthias Schonlau, William J. Welch, and Donald R. Jones. A data analytic approach to Bayesian global optimization. presented at 1997 ASA conference, 1997.

[62] L. A. Schulz, B. Panda, F. J. Tarzanin, R. C. Derham, B. K. Oh, and L. Dadone. Interdisciplinary analysis for advanced rotors – approach, capabilities and status. presented at the AHS Aeromechanics Specialists Conference, January 1994.

[63] R. S. Sellar, J. E. Renaud, and S. M. Batill. Optimization of mixed discrete/continuous design variable systems using neural networks. In *Proceedings of the Fifth AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, September 7–9, 1994.* American Institute of Aeronautics and Astronautics, 1994.

[64] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, Cambridge, Massachusetts, 1995. The unannotated standard is online at <http://www.mcs.anl.gov/mpi/index.html>.

[65] J. Sobieszczanski-Sobieski and R. T. Haftka. Multidisciplinary aerospace design optimization: survey of recent developments. *Structural Optimization*, 14(1):1–23, 1997.

[66] W. Spendley, G. R. Hext, and F. R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. *Technometrics*, 4(4):441–461, November 1962. call number QA 276 .T4.

[67] Jimmy C. Tai, D. N. Mavris, and D. P. Schrage. Application of a response surface method to the design of tipjet driven stopped rotor/wing concepts. In *Proceedings of the First AIAA Aircraft Engineering, Technology, and Operations Congress, Los Angeles, CA*, September 1995.

[68] B. Tang. Orthogonal array-based latin hypercubes. *Journal American Statistical Association*, 88(424):1392–1397, 1993.

[69] Virginia Torczon. Multi-directional search: a direct search algorithm for parallel machines. Technical Report 90–7, Department of Computational and Applied Mathematics, Rice University, Houston, TX 77251–1892, 1990. Ph.D. Thesis.

[70] Virginia Torczon. On the convergence of the multidirectional search algorithm. *SIAM J. Optimization*, 1(1):123–145, February 1991.

[71] Virginia Torczon. PDS: Direct search methods for unconstrained optimization on either sequential or parallel machines. Technical Report 92–9, Department of Mathematical Sciences, Rice University, Houston, TX 77251–1892, 1992. Submitted to *ACM Transactions on Mathematical Software*.

[72] Virginia Torczon. On the convergence of pattern search methods. *SIAM J. Optimization*, 7:1–25, February 1997.

[73] Michael W. Trosset and Virginia Torczon. Numerical optimization using computer experiments. Technical Report 97–38, ICASE, NASA Langley Research Center, Hampton, VA 23681–0001, August 1998. Submitted to *Technometrics*.

[74] William J. Welch, Robert J. Buck, Jerome Sacks, Henry P. Wynn, Toby J. Mitchell, and Max D. Morris. Screening, predicting, and computer experiments. *Technometrics*, 34(1):15–25, February 1992.

[75] William J. Welch. A mean squared error criterion for the design of experiments. *Biometrika*, 70(1):205–213, 1983.