3

Elements of Numerical Optimization

Having outlined the design processes common in much of the aerospace sector and highlighted some of the issues that arise when trying to bring together multiple disciplines in a single computational environment, including the subject of design search and optimization (DSO), attention is next focused on the subject of formal optimization methods and the theory underlying them. These increasingly lie at the heart of much computationally based design, not because designers necessarily believe that such methods can yield truly optimal designs, or even very good designs, but rather because they provide a systematic framework for considering some of the many decisions designers are charged with taking. Nonetheless, it is helpful when looking at computational approaches to design to have a thorough understanding of the various classes of optimization methods, where and how they work and, more importantly, where they are likely to fail.

As has already been noted in the introduction, optimization methods may be classified according to the types of problems they are designed to deal with. Problems can be classified according to the nature of the variables the designer is able to control (nonnumeric, discrete, real valued), the number and type of objectives (one, many, deterministic, probabilistic, static, dynamic), the presence of constraints (none, variable bounds, inequalities, equalities) and the types of functional relationships involved (linear, nonlinear, discontinuous). Moreover, optimizers can be subclassified according to their general form of approach. They can be gradient based, rule based (heuristic) or stochastic – often, they will be a subtle blend of all three. They can work with a single design point at a time or try to advance a group or "population" of designs. They can be aiming to meet a single objective or to find a set of designs that together satisfy multiple objectives. In all cases, however, optimization may be defined as the search for a set of inputs $\mathbf{x}$ that minimize (or maximize) the outputs of a function $f(\mathbf{x})$ subject to constraints $g_i(\mathbf{x}) \geq 0$ and $h_j(\mathbf{x}) = 0$.

In essence, this involves trying to identify those regions of the design space (sets of designer chosen variables) that give the best performance and then in accurately finding the minimum in any given region. If one thinks of trying to find the lowest point in a geographical landscape, then this amounts to identifying different river basins and then

Computational Approaches for Aerospace Design: The Pursuit of Excellence. A. J. Keane and P. B. Nair

tracking down to the river in the valley bottom and then on to its final destination – in optimization texts, it is common to refer to such regions as "basins of attraction" since, if one dropped a ball into such a region, the action of gravity would lead it to the same point wherever in the basin it started. The search for the best basins is here termed *global optimization*, while accurately locating the lowest point within a single basin is referred to as *local optimization*. Without wishing to stretch the analogy too far, constraints then act on the landscape rather like national borders that must not be crossed. It will be obvious that many searches will thus end where the "river" crosses the border and not at the lowest point in the basin. Of course, in design, we are often dealing with many more than two variables, but the analogy holds and we can still define the idea of a basin of attraction in this way. The aim of optimization is to try and accomplish such searches as quickly as possible in as robust a fashion as possible, while essentially blindfolded – if we had a map of the design space showing contours, we would not need to search at all, of course.

Notice that even when dealing with multiple design goals it is usually possible to specify a single objective in this way, either by weighting together the separate goals in some way, or by transforming the problem into the search for a Pareto optimal set where the goal is then formally the production of an appropriately spanning Pareto set (usually one that contains designs that represent a wide and evenly spread set of alternatives). It is also the case that suitable goals can often be specified by setting a target performance for some quantity and then using an optimizer to minimize any deviations from the target. This is often termed *inverse design* and has been applied with some success to airfoil optimization where target pressure distributions are used to drive the optimization process. Note that throughout this part of the book we will tackle the problem of minimization, since maximization simply involves a negation of the objective function.

3.1 Single Variable Optimizers – Line Search

We begin this more detailed examination of optimization methods by first considering the problem of finding an optimal solution to a problem where there is just one real-valued variable – so-called line search problems. Although rarely of direct interest in design, such problems illustrate many of the issues involved in search and serve as a useful starting point for introducing more complex ideas. Moreover, many more sophisticated search methods use line search techniques as part of their overall strategy. We will leave the issue of dealing with constraints, other than simple bounds, until we come to multivariable problems.

3.1.1 Unconstrained Optimization with a Single Real Variable

The first problem that arises when carrying out line search of an unbounded domain is to try and identify if a minimum exists at all. If we have a closed form mathematical expression for the required objective, this can of course be established by the methods of differential calculus – the first differential is set equal to zero to identify turning points and the sign of the second differential checked at these to see if it is positive there (consider $f\{x\} = x^2 - 4x - 1$; then, $\mathrm{d}f/\mathrm{d}x = 2x - 4$ and $\mathrm{d}^2 f/\mathrm{d}x^2 = 2$, so there is a minimum at $x = 2$). Clearly, in most practical engineering problems, the functional relationship between variables and goals is not known in closed form and so numerical schemes must be adopted. In such circumstances, the simplest approach to finding a minimum is to make a simple step in either positive or negative direction (the choice is arbitrary) and see if the function

reduces. If it does, one keeps on making steps in the same direction but each time of twice the length until such time as the function starts to increase, a bound on the variable is reached or the variable becomes effectively infinite (by virtue of running out of machine precision). If the initial step results in the function increasing, one simply reverses direction and keeps searching the other way. Given the doubling of step length at each step, one can begin with quite small initial moves and yet still reach machine precision remarkably quickly. Many alternative methods for changing the step size have been proposed and that adopted in practice will depend on the presence or absence of bounds. If the problem is bounded, then continuously doubling the step length is clumsy, since it is likely to lead to unnecessarily coarse steps near the bounds. At the other extreme, using fixed steps may mean the search is tediously slow. An alternative is to fit the last three points evaluated to a parabola and then to take a step to the location predicted as the turning point (as indicated by the parabola's differential and assuming that it has a minimum in the direction of travel).

As with all numerical approaches to optimization, one must next consider when this kind of strategy will fail. Simply stepping along a function is only guaranteed to bracket a minimum if the domain being searched contains a single turning point, as Figure 3.1 makes clear. If there are multiple minima, or even just one minimum, but also an adjacent maximum (as in the figure), the strategy can fail. This brings us immediately to the idea of local search versus global search. Local search can be defined as searching a domain in which there is just a single minimum and no maxima (and its converse when seeking a maximum) – anything else leads to global search. Thus, we may say that the strategy of trying to bracket a minimum by taking steps of doubling lengths is a form of *local* search since it seeks to locate a single turning point. Starting with a smaller initial step size, or at a location closer to the minimum, will, of course, tend to bracket the optimum more accurately. This will also better cope with situations where there are multiple turning values, although there are still no guarantees that any minima will be found. It turns out that, even with only one variable, no search method can guarantee to find a minimum in

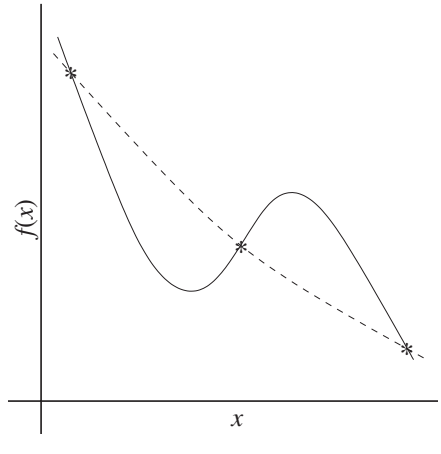


Figure 3.1 Attempting to bracket a function with two turning points (solid line) – the *'s mark the sampled points and the function appears to reduce smoothly to the right (dashed line).

the presence of multiple minima and maxima. The best that can be hoped for when dealing with *global* search problems is that a method is *likely* to find a good optimum value.

Assuming, however, that it is possible to bracket the desired minimum (by finding a triple x_1, x_2, x_3 where $f(x_2)$ is less than both $f(x_1)$ and $f(x_3)$), the next requirement in line search is that the bounds enclosing the turning point be squeezed toward the actual optimum as quickly as possible. This immediately raises issues of accuracy – if a highly accurate solution is required, it is likely that more steps will be needed, although this is not always so. Note also, that in the limit, all smooth, continuous functions behave quadratically at a minimum, and therefore the accuracy with which a minimum can generally be located is fixed by the square root of the machine precision in use, that is, to about 2×10^{-4} in single precision and 2×10^{-8} in double precision on 32-bit machines. Next, in keeping with our classification of optimizers as being gradient based, rule based or stochastic, we describe methods for homing in on the optimum of all three types.

The simplest scheme that can be adopted is to sample the bracketed interval using a series of random locations, keeping the best result found at each stage until as many trials as can be afforded have been made. This approach is easy to program and extremely robust, but is slow and inefficient – such characteristics are often true of stochastic schemes. It can be readily improved on by using successive iterations to tighten the bounds for random sampling on the assumption that there is a single local minimum in the initial bracket (although this might be argued as shifting the search into the category of heuristic methods). Random one-dimensional searches should ideally be avoided wherever possible – certainly, they should not be used if it is believed that the function being dealt with is smooth in the sense of having a continuous first derivative and it is possible to bracket the desired minimum. Sometimes, however, even in one dimension, the function being studied may have many optima and the aim of search is to locate the best of these, that is, global search. In such cases, it turns out that searches with random elements have an important role to play. We leave discussion of these methods until dealing with problems in multiple dimensions.

The next simplest class of methods, the rule based or heuristic methods, lead to a more structured form of sampling that directly seeks to move the bounds together as rapidly as possible. There are a number of schemes for doing this and all aim to replace one or other (or both) of the outer bounds with ones closer to the optimum and thus produce a new, tighter bracketing triplet. The two most important of them are the golden section search and the Fibonacci search. The fundamental difference between the two is that to carry out a Fibonacci search one must know the number of iterations to be performed at the outset, while the golden section search can be carried on until sufficient accuracy in the solution is achieved. If it can be used, the Fibonacci is the faster of the two but is also the less certain to yield an answer of given accuracy. This is another common distinction that arises in search methods – does one have to limit the search by the available time or can one keep on going until the answer is good enough. In much practical design work, searches will be limited by time, particularly if the functions being searched are computationally expensive to deal with. In such circumstances, the ability to use a method that will efficiently complete in a fixed number of steps can be a real advantage.

The golden section search places the next point to evaluate $(3 - \sqrt{5})/2 (\approx 0.38197)$ into the larger of the two intervals formed by the current triplet (measured from the central point of the triplet as a fraction of the width of the larger interval). Provided we start with a triplet where the internal point is 0.38197 times the triplet width from one end, this will mean that whichever endpoint we give up when forming the new triplet, the remaining three points

will still have this ratio of spacing. Moreover, the width of the triplet is reduced by the same amount whichever endpoint is removed, and thus the search progresses at a uniform (linear) rate. To see this, consider points spaced out at $x_1 = 0.0$, $x_2 = 0.38197$ and $x_3 = 1.0$. With these spacings, x_4 must be placed at 0.61804 ($= 0.38197 \times (1.0 - 0.38197) + 0.38197$) and the resulting triplet will then be either $x_1 = 0.0$, $x_2 = 0.38197$ and $x_4 = 0.61804$ or $x_2 = 0.38197$, $x_4 = 0.61804$ and $x_3 = 1.0$. In either case, the triplet is 0.61804 long and the central point is 0.23607 ($= 0.38197 \times 0.61804$) from one end. Even if the initial triplet is not spaced out in this fashion, the search rapidly converges to this ratio, which has been known since ancient times as the "golden section" or "golden mean", which leads to the method's name.

The Fibonacci search proceeds in a similar fashion but operates with a series due to Leonardo of Pisa (who took the pseudonym Fibonacci). This series is defined as $f_0 = f_1 = 1$ and $f_k = f_{k-1} + f_{k-2}$ for $k \geq 2$ (i.e., 1, 1, 2, 3, 5, 8, 13, 21, ...). To apply the search, the bounds containing the minimum are specified along with the intended number of divisions, N. Then the current triplet width is reduced by moving both endpoints toward each other by the same amount, given by multiplying the starting triplet width by a reduction factor t_k at each step k, that is, the two endpoints are each moved inward by $t_k \times (x_2 - x_1)/2$. The reduction factor is controlled by the Fibonacci series such that $t_k = f_{N+1-k}/f_{N+1}$, which ensures that (except for the first division) when moving in each endpoint, one or other or both will lie at previously calculated interior points; see Figure 3.2 (note that when both endpoints lie at previously calculated locations this amounts to a null step and the search immediately proceeds to the next level). The process continues until $k = N$ and it may be shown that this approach is optimal in the sense that it minimizes the maximum uncertainty in the location of the minimum being sought. To achieve a given accuracy ε, N should

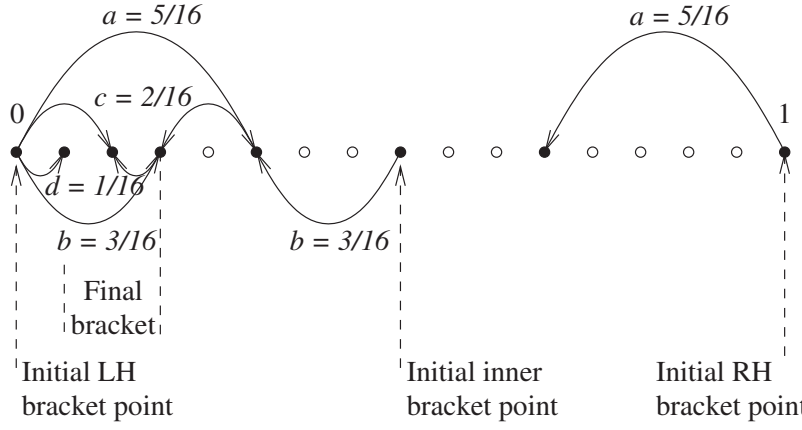


Figure 3.2 Steps in a Fibonacci line search of four moves, labeled a, b, c and d, with triplets of (0/16, 8/16, 16/16), (0/16, 5/16, 8/16), (0/16, 3/16, 5/16) and (1/16, 2/16, 3/16) – that is, there are four moves, so that $N = 4$ and the corresponding reductions factors are $t_1 = f_4/f_5 = 5/8$, $t_2 = (f_3/f_5)/(8/16) = (3/8)/(8/16) = 3/4$, $t_3 = (f_2/f_5)/(5/16) = (2/8)/(5/16) = 4/5$, $t_4 = (f_1/f_5)/(3/16) = (1/8)/(3/16) = 2/3$ – therefore, only five new points are sampled before the final bracket is established – those shown filled on the figure.

be chosen such that $1/f_{N+1} > \varepsilon/(x_2 - x_1) \geq 1/f_{N+2}$. As already noted for most functions (certainly those that are smooth in the vicinity of the optimum) the accuracy with which a turning point can be trapped is limited by the *square root* of the machine precision (i.e., it is limited to 2×10^{-4} in single precision) – attempting to converge a solution more closely than this, by any means, is unlikely to be useful.

An alternative to these two rule-based systems is to fit a simple quadratic polynomial to the bracketing triple at each step, and then, by taking its differential, calculate the location of its turning point and place the next point there. This is called *inverse parabolic interpolation* and the location of the next point x_4 is given by

$$x_4 = x_2 - \frac{(x_2 - x_1)^2(f(x_2) - f(x_3)) - (x_2 - x_3)^2(f(x_2) - f(x_1))}{2(x_2 - x_1)(f(x_2) - f(x_3)) - (x_2 - x_3)(f(x_2) - f(x_1))}. \tag{3.1}$$

This process can easily be mislead if the bracketing interval is wide, and so a number of schemes have been proposed that seek to combine this formula and the more robust process of the golden section search – that by Brent is perhaps the most popular (Brent 1973; Press et al. 1986). Brent's scheme alternates between a golden section search and parabolic convergence depending on the progress the algorithm is making, and is often used within multidimensional search methods.

The most sophisticated line searches are gradient-based methods, which work, of course, by explicitly using the gradients of the function when calculating the next step to take. Sometimes, the gradients can be found directly (for example, when dealing with closed form expressions, numerically differentiated simulation codes or adjoint codes) but, more commonly, they must be found by finite differencing. All such methods are sensitive to the accuracy of the available gradient information, but where this is of good quality, and particularly where the underlying function is approximately polynomial in form, they offer the fastest methods for carrying out line searches. Where they work, they are superlinear in performance. The degree to which they outperform the linear golden section and Fibonacci approaches depends on the order of the scheme adopted. The order that can be successfully adopted, in turn, depends on the quality of the available gradients and the degree to which the underlying function can be represented by a polynomial – again, we see the trade-off between robustness and speed.

The most aggressive gradient approach to take is to simply fit a polynomial through all the available data with appropriate gradients at the points where these are available. Then, differential calculus may be applied to locate any turning values and to distinguish between minima, maxima and points of inflexion. Assuming a minimum exists within the current triple, the next point evaluated is placed at the predicted location. This process can be iterated with the new points and gradients being added in turn. If all three gradients are known for the initial bracketing triple, then a fifth-order polynomial can be fitted and its turning values used. It will be immediately obvious that there are some serious flaws with such an approach. First, even if all the data is noise free and the quintic is a good model, the roots of its first differential must be obtained by a root search; secondly, high-order polynomials are notoriously sensitive to any noise and can become highly oscillatory and, thirdly, there is no guarantee that a minimum value will exist within the initial bracket when it is modeled by a quintic. Finally, as extra points are added, the order of the complete polynomial rapidly rises and all these issues become ever more problematic. It is therefore normal to take a more conservative approach.

Rather than fit the highest-order polynomial available, one can use a cubic instead. If the chosen polynomial is to be an interpolator, this requires that a subset of the available

data be used from the bracketing triple. For a cubic model, one might use the three data values and the gradient of the central point of the triple to fix the four coefficients. It is then simple to solve for the roots of the gradient equation (which is of course second order) and jump to the minimum that must lie within the bracket. This will lead to a new triple and the process can be repeated, with the gradient being calculated at the central point each time (or to save time, at one or other end if the gradient is already known there). Note also, that if one always has gradients available, there is no need to seek a triple to bracket the minimum – provided the gradients of two adjacent points point toward each other, this suffices to bracket the turning point. Moreover, a cubic (or Hermitian) interpolant can be constructed from the two endpoints and their gradients and an iterative scheme constructed that will be very efficient if the function is smooth and polynomial-like.

An alternative gradient-based scheme is to use the Newton–Raphson approach for finding the zeros of a function to find the zeros of its derivative – this is commonly termed the *secant method*. It requires the gradient at the current point and an estimate of the second derivative. Again, if the minimum is bracketed by a pair of points where both derivatives are known, then the next location to chose can be based on a forward finite difference of the derivatives, that is, $x_3 = x_1 - (x_2 - x_1)\mathrm{d}f(x_1)/\mathrm{d}x/\{\mathrm{d}f(x_2)/\mathrm{d}x - \mathrm{d}f(x_1)/\mathrm{d}x\}$, which, since $\mathrm{d}f(x_2)/\mathrm{d}x$ is positive and $\mathrm{d}f(x_1)/\mathrm{d}x$ is negative, always yields a new location within the initial bracket. Of course, if the second derivative is also known at each point, this information can be used instead.

It is possible to use gradients in an even more conservative fashion, however. If gradients are available, then bracketing the minimum does not require that these be known very accurately; rather, one merely searches for a pair of adjacent points where the sign of the gradient goes from negative to positive. If one then carries out a series of interval bisections, keeping the pair of points at each time based only on the signs of the gradients, a more robust, if slower, search results. This can be particularly advantageous if the derivative information is noisy. Notice, however, that if the gradients are being found by finite differencing it hardly makes sense to evaluate four function values (i.e., two closely spaced pairs) in order to locate the next place to jump to – it is far better to use the golden section or Fibonacci methods. A slightly more adventurous approach based on triplets is to use a simple straight line (or secant) extrapolation of the gradient from that at the central point and the lower of the two edge points and see if this indicates a zero within the triplet. If the zero is within the triplet, this point is adopted, while if not, a simple bisection of the best pairing is used, again followed by iteration. This is the approach recommended by Press et al. (1992) for line searches with gradients.

3.1.2 Optimization with a Single Discrete Variable

Sometimes, when carrying out optimization problems, the free variables that the optimizer controls are discrete as opposed to being continuous. Such problems obviously arise when dealing with integer quantities. But they also arise when working with stock sizes of materials, fittings, screw sizes, and so on. In practice, since when dealing with such discrete variables no gradient information exists, all these problems are equivalent and optimizers that work with integers can solve all problems in this class by a simple scaling of the variables (i.e., we use the integers as indices in a look-up table of the discrete values being studied). The important point with such discrete problems is that the series of design

variables be *ordered* in some way. Therefore, only the problem of optimization with integer variables is considered further here.

There are two fundamentally different approaches to the problem of integer optimization. The first is to work directly in integer arithmetic while the second is to assume that the design variables are actually continuous and simply to round to the nearest integer when evaluating the objective function (or constraints). Methods that work with integers can be constructed from those already described in a number of cases, albeit they may work rather less efficiently. If a global search in a multimodal space is being performed, some form of stochastic or exhaustive search over the integers being dealt with will be required – these are dealt with later. Conversely, if a local search is being conducted then, as before, the first goal is to find a bracketing triple that captures the optimum being sought. For example, it is easy to devise a one-dimensional bracketing routine equivalent to that already described for continuous variables: all that is required is that the first step taken be an integer one – when this is doubled, this will still of course result in an integer value and so such a search will naturally work with integers.

Having bracketed the desired minimum with a triplet, methods must then be devised to reduce the bracket to trap the optimum. Here, in some sense, the integer problem is easier than the real-valued one as no issues of accuracy and round-off arise. The simplest, wholly integer strategy that can be proposed is to bisect the larger gap of the bracket (allowing for the fact that this region will from time to time contain an even number of points and thus cannot be exactly bisected and either the point to the right or left of the center must be taken, usually alternately). Then, the bracket triple is reduced by rejecting one or other endpoints. This process is carried out until there are no more points left in the bracket at which time the value with lowest objective becomes the optimum value. This search can be refined by not simply bisecting the larger gap in the bracket at each step, but by instead using the point closest to the golden section search value. This is equivalent to using a true golden section search and rounding the new value to the nearest integer quantity at each iteration.

The Fibonacci search is rather better suited to integer problems than both of these methods as, with a little adaptation, it may be directly applied without rounding; see Figure 3.3. We first identify the pair of Fibonacci numbers that span the total integer gap between the upper and lower limits of the triplet to be searched (i.e., the width of the triplet) – in the figure, the initial bracket is 18-units wide and so the pair of Fibonacci numbers that span this range are 13 and 21. We then move the limit farthest from the initial inner point of the triple inward so that the remaining enclosed interval width to be searched is an exact Fibonacci number (i.e., here, we move the point at 18 inward to 13) – if this triplet does not bracket the minimum, we take the triplet formed from the original inner point, the new endpoint and the new endpoint's previous position and repeat the process of reducing to the nearest Fibonacci number until we do have a triplet that brackets the minimum and has a length equal to one of the Fibonacci numbers. Once the triplet is an exact Fibonacci number in width, we simply move both bounds inward in a sequence of Fibonacci numbers until we reach unity and hence identify the location of the minimum – here, 5, 3, 2 and 1 (we start with largest number that will fit twice into the interval remaining after making the first move).

Polynomial-based and gradient-based searches can only be applied to integer problems if the fiction of a continuous model is assumed as part of the search, i.e., that variable values between the integers *can* be accommodated by the search. As already noted, the

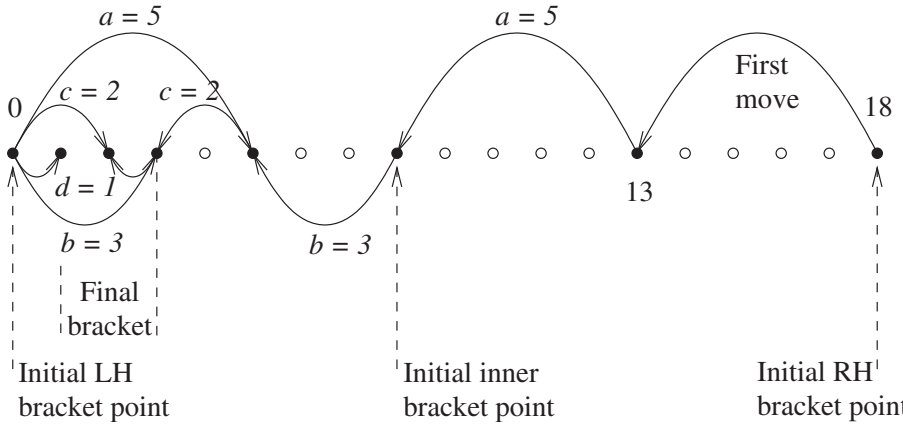


Figure 3.3 Steps in a Fibonacci integer search of five moves, an initial move to reduce the interval to a Fibonacci number followed by those labeled a, b, c and d, with triplets of (0, 8, 18), (0, 8, 13), (0, 5, 8), (0, 3, 5) and (1, 2, 3) – therefore, only six new points are sampled before the final bracket is established – those shown filled on the figure.

simplest way to do this is to proceed as for a continuous variable and then to round to the nearest integer value when calculating the objective and constraints. It is also sometimes possible to treat the whole problem as being continuous, including the calculation of the objective and any constraints and just to round the final design to the nearest whole number – such approaches can be effective when dealing with stock sizes of materials. For example, if the search is to find the thickness of a plate that minimizes a stress, this can be treated as a continuous problem. Then, the final design can be selected from the available stock sizes by rounding up. In fact, in many engineering design processes, the designer is often required to work with available materials and fittings, and so, even if most of this work is treated as continuous, it is quite common to then round to the nearest acceptable component or stock size. If rounding is used for each change of variable, however, care must be taken with gradient-based routines as they can be misled by the fictional gradients being used – this can often make them no faster than the Fibonacci approach or, worse, sometimes this means they become trapped in loops that fail to converge. Again a trade-off between speed and robustness must be made.

3.1.3 Optimization with a Single Nonnumeric Variable

Sometimes, searches must be carried out over variables that are not only discontinuous but not even numeric in nature. The most common such case arises in material selection. For example, when choosing between metals for a particular role, the designer will be concerned with weight, strength, ductility, fatigue resistance, weldability, cost, and so on. There may be a number of alloys that could be used, each with its own particular set of qualities, and the objective being considered may be calculated by the use of some complex nonlinear CSM code. The most obvious way to find the optimum material is to try each one in turn and select the best – such a blind search is bound to locate the optimum in the end. If a complete search cannot be afforded, then some subset must be examined and the

search stopped after an appropriate number have been considered – very often, this is done manually, with the designer running through a range of likely materials until satisfied that a good choice has been identified. The difficulty here arises in ordering the choice of material to consider next. Clearly, each could be given an index number and the search performed as over integers for a finite number of steps, or even using one of the search methods outlined in the previous section. It is, of course, the lack of any natural ordering between index number and performance that makes such a search difficult, and even somewhat random. Unless the designer has a good scheme for ordering the choices to be made when indexing them, it is likely that an exhaustive search will be as efficient as any other method. Such observations apply to all problems where simple choices have to be made and where there is no natural mapping of the choices to a numerical sequence.

3.2 Multivariable Optimizers

When dealing with problems with more than one variable, it is, of course, a perfectly viable solution to simply apply methods designed for one variable to each dimension in turn, repeating the searches over the individual variables, until some form of convergence is reached. Such methods can be effective, particularly in problems with little interaction between the variables and a single optimum value, although they tend to suffer from convergence problems and find some forms of objective landscapes time consuming to deal with (narrow valleys that are not aligned with any one variable, for example). Unfortunately, most of the problems encountered in engineering design exhibit strong coupling between the variables and multiple optima and so more sophisticated approaches are called for. This has led to the development of a rich literature of methods aimed at optimization in multiple dimensions. Here, we will introduce a number of the more established methods and also a number of hybrid strategies for combining them.

Most of the classical search methods work in a sequential fashion by seeking to find a series of designs that improve on each other until they reach the nearest optimum design. This is not true of all methods, however – a pure random search, for example, does not care in which order points are sampled or if results are improved and is capable of searching a landscape with multiple optima. Here, we begin by examining those that work in an essentially sequential fashion, before moving on to those that can search entire landscapes (i.e., local and global methods, respectively). Note that in some of the classical optimization literature the term “global convergence” is used to indicate a search that will find its way to an optimal design wherever it starts from, as opposed to one that will search over multiple optima.

Local searches have to deal with two fundamental issues: first, they must establish the direction in which to move and, secondly, the step size to take. These two considerations are by no means as trivial to deal with as they might seem and very many different ways of trying to produce robust optimizers that converge efficiently have been produced. Almost all local search methods seek to move downhill, beginning with large steps and reducing these as the optimum is approached, so as to avoid overshoot and to ensure convergence. Global search methods must additionally provide a strategy that enables multiple optima to be found, usually by being able to jump out of the basin of attraction of one optimal design and into that of another, even at the expense of an initial worsening of the design.

We will illustrate these various issues by carrying out searches on a simple test function proposed some years ago by one of the authors – the "bump" problem. This multidimensional problem is usually defined as a maximization problem but can be set up for minimization as follows:

$$\begin{aligned} \textbf{Minimize}: \quad & -\text{abs}\left(\sum_{i=1}^{n}\cos^4(x_i) - 2\prod_{i=1}^{n}\cos^2(x_i)\right) \Big/ \sqrt{\sum_{i=1}^{n} i x_i^2} \\ \textbf{Subject to}: \quad & \prod_{i=1}^{n} x_i > 0.75 \quad \text{and} \quad \sum_{i=1}^{n} x_i < 15n/2 \\ & 0 \le x_i \le 10, \quad i = 1, 2, \ldots, n, \end{aligned} \tag{3.2}$$

where the x_i are the variables (expressed in radians) and n is the number of dimensions. This function gives a highly bumpy surface where the true global optimum is usually defined by the product constraint. Figure 3.4 illustrates this problem in two dimensions. When dealing with optimizers designed to find the nearest optimal design, we start the search at (3.5, 2.5) on this landscape, and see if they can locate the nearest optimum, which is at (3.0871, 1.5173) where the function has a value of -0.26290. When applying global methods, we start at (5, 5) and try to find the locations of multiple optima. Finally, when using constrained methods, we try and locate the global constrained optimum at (1.5903, 0.4716) where the optimum design is defined by the intersection of the objective function surface and the

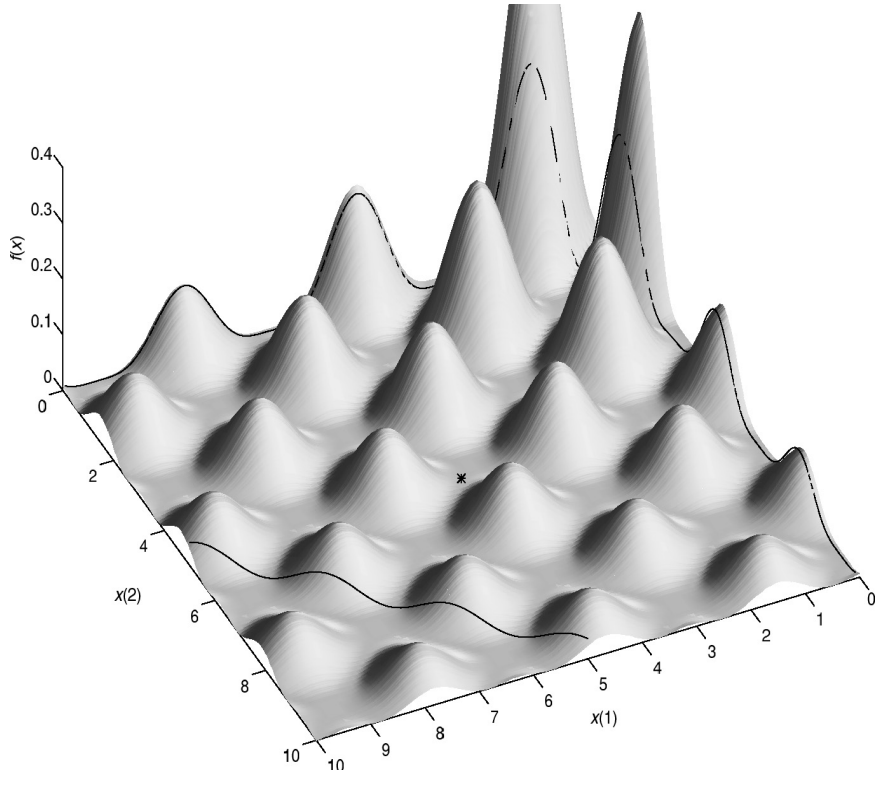


Figure 3.4 Constrained bump function in two dimensions (plotted in its original maximization form).

product constraint and the function value is 0.36485. All of the search methods discussed in this section are unconstrained in their basic operation, and so mechanisms have to be added to them to deal with any constraints in design problems being studied. As many of these are common to a variety of search methods, they are dealt with in a subsequent section.

Before examining the search methods in detail, we briefly digress, however, by discussing a key aspect of the way that searches are actually implemented in practice.

3.2.1 Population versus Single-point Methods

The single variable methods that have been outlined above are all serial in nature, in that new design points are derived and processed one at a time. This is natural when working with one variable and it is also the easiest way to understand many search tools but, increasingly, it is not the most efficient way to proceed in practice. Given that so many of the analysis tools used in design can be time consuming to run, there is always pressure to speed up any search method that is running large computational analyses. The most obvious and simple way of doing this is to consider multiple design points in parallel (we note also that many large analysis codes are themselves capable of parallel operation, and this fact too can often be exploited). The idea of searching multiple solutions in parallel leads to the idea of population-based search. Such searches can be very readily mapped onto clusters of computers with relatively low bandwidth requirements between them, and computing clusters suitable for this kind of working are increasingly common in design offices, formed either by linking together the machines of individual designers when they are not otherwise in use (so-called cycle harvesting) or in the form of dedicated compute clusters.

The populations of possible designs being examined by a search method can be stepped forward in many ways: in a gradient-based calculation, they may represent small steps in each variable direction and be used to establish local slopes by finite differencing; in a heuristic pattern-based search, multiple searches may be conducted in different areas of a fitness landscape at the same time; in an evolutionary search, the population may form a pool of material that is recombined to form the next generation of new designs. Sometimes, the use of populations fits very naturally into the search and no obvious bottlenecks or difficulties arise (evolutionary searches fall into this category, as do multiple start hill climbers, and design of experiment based methods), sometimes they can only be exploited for part of the time (as in the finite differencing of gradients before a pattern move takes place in a gradient-based search) and sometimes it is barely possible to accommodate simultaneous calculations at all (such as in simulated annealing). In fact, when choosing between alternatives, it turns out that the raw efficiency of a search engine may well be less important than its ability to be naturally mapped to cluster-based computing systems.

It goes without saying that to be of practical use in parallel calculations a population-enabled search must have a mechanism for deploying its activities over the available computing facilities in a workable fashion – and this often means that issues of job management form an intrinsic part of the implementation of such methods. In what follows, however, we will assume that, where needed, a population-based method can deploy its calculations in an appropriate fashion. Finally, it should be noted that what mostly matters in design is the elapsed time required to find a worthwhile improvement, including the time

needed to define and set up the search, that needed to carry it out and that involved in analyzing results as they become available so as to be able to steer, terminate or restart the search. Methods that fit naturally with a designer's way of working and desire to be involved in the product improvement will obviously be more popular than those that do not.

3.2.2 Gradient-based Methods

We begin our study of multivariable search methods with those that exploit the objective function gradient since the fundamental difference between multiple variable and single variable searches is, of course, the need to establish a direction for the next move rather than just the length of the step to take, that is, one must know which way is down. Given a suitable direction of search, optimization in multiple dimensions is then, at least conceptually, reduced to a single variable problem: once the direction of move is known, the relative magnitudes of the input variables can be fixed with respect to each other. An obvious approach to optimization (and one due to Cauchy) is to use gradient information to identify the direction of steepest descent and go in that direction.[1] Then, a single move or entire line search is made in that direction. Following this, the direction of steepest descent is again established and the process repeated. If a line search is used, the new direction of descent will be at right angles to the previous one (the search will, after all, have reached a minimum in the desired direction) so that a series of repeated steepest descent line searches leads to a zigzag path; see Figure 3.5. If a line search is not used, issues then arise as to how long each step should be in a particular direction – if they are too long, then the best path may be overshot; conversely, if they are too small, the search will be very slow.

Although steepest descent optimization methods will clearly work, there are rather better schemes for exploiting gradients: all are based on the idea that, near a minimum, most functions can be approximated by a quadratic shape, that is, in two dimensions, the objective

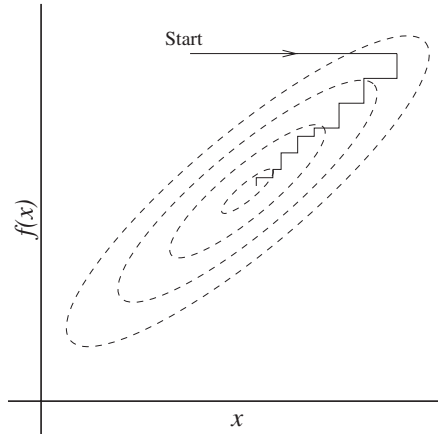


Figure 3.5 A steepest descent search in a narrow valley.

[1] We assume that suitable steps are taken to make the required gradients available – in fact, the calculation of such gradients by finite differencing schemes is by no means trivial and is also often prohibitively expensive if many variables are involved; conversely, if good quality gradients can be found from efficient adjoint or direct numerical differentiation methods, this will not be a problem.

function surface is an ellipsoid bowl and the objective function contours are ellipses. For a continuous smooth function, this is always true sufficiently close to any minimum (consider a Taylor series expansion taken about the optimum). Unfortunately, many of the optima being sought in engineering design are either defined by constraint boundaries or by discontinuities in their objective functions – this means that many of the gradient-based methods that give very good performance in theory do much less well in practice. Nonetheless, it is useful to review their underlying mechanisms as many effective search methods draw on an understanding of this basic behavior. We leave to Chapter 4 a detailed analysis of methods for computing the gradients of functions.

Newton's Method

The simplest way to exploit the structure of a quadratic form is via Newton's method. In this, we assume that $f(\mathbf{x}) \approx {}^1\!/_2\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} + \mathbf{b}^{\mathrm{T}}\mathbf{x} + c$ where now $\mathbf{x}$ is the vector of variables being searched over and $f(\mathbf{x})$ remains the objective function. $\mathbf{A}$, $\mathbf{b}$ and c are unknown coefficients in the quadratic approximation of the function, $\mathbf{A}$ being a positive definite square matrix known as the Hessian, $\mathbf{b}$ a vector and c a constant (it is the positive definiteness of $\mathbf{A}$ that ensures that the function being searched is quadratic and convex, and thus does contain a minimum – clearly we cannot expect such methods to work where there is no minimum in the function or where pathologically it is not quadratic, even in the limit as we approach the optimum). Notice that simple calculus tells us that the elements of the Hessian $\mathbf{A}$ are given by the second differentials of the objective function, that is, $A_{ij} = \partial^2 f/\partial x_i \partial x_j$ and that the gradient vector is given by $\mathbf{A}\mathbf{x} + \mathbf{b}$. At a minimum of $f(\mathbf{x})$, the gradient vector is of course zero.

Newton's search method is very similar to that for finding the zero of a function – here, we are seeking the zero of the gradient, of course. If our function is quadratic and we move from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$, then we can write

$$f(\mathbf{x}_{i+1}) \approx f(\mathbf{x}_i) + (\mathbf{x}_{i+1} - \mathbf{x}_i)^{\mathrm{T}} \nabla f(\mathbf{x}_i) + 1/2(\mathbf{x}_{i+1} - \mathbf{x}_i)^{\mathrm{T}} \mathbf{A}(\mathbf{x}_{i+1} - \mathbf{x}_i) \tag{3.3}$$

$$\text{or} \quad \nabla f(\mathbf{x}_{i+1}) \approx \nabla f(\mathbf{x}_i) + \mathbf{A}(\mathbf{x}_{i+1} - \mathbf{x}_i). \tag{3.4}$$

Newton's method simply involves setting $\nabla f(\mathbf{x}_{i+1})$ to zero so that the required step becomes $(\mathbf{x}_{i+1} - \mathbf{x}_i) \approx -\mathbf{A}^{-1} \nabla f(\mathbf{x}_i)$.

In this method, we assume that the Hessian is known everywhere (or can be computed) and so we can then simply move directly to the minimum of the approximating quadratic form in one step. If the function is actually quadratic, this will complete the search, and if not, we simply repeat the process with a new Hessian. This process converges very fast (quadratically) but relies on the availability of the Hessian. In most practical optimization work, the Hessian is not available directly and its computation is both expensive and prone to difficulties. Therefore, a range of methods have been developed that exploit gradient information and work without the Hessian, or with a series of approximations that converge toward it.

Conjugate Gradient Methods

Conjugate gradient searches work without the Hessian but try to improve on the behavior of simple steepest descent methods. The key problem with the steepest descent type approach is that the direction of steepest gradient rarely points toward the desired minimum, even

if the function being searched is quadratic (in fact, only if the contours are circular does this direction always point to the minimum – an extremely unlikely circumstance in real problems). Conjugate gradient methods are based on the observation that a line through the minimum of quadratic function cuts all the contours of the objective function at the same angle. If we restrict our searches to run along such directions, we can, in theory, minimize an arbitrary quadratic form in the same number of line searches as there are dimensions in the problem, provided the line search is accurate. In practice, badly scaled problems can impact on this and, moreover, few real problems are actually quadratic in this way. Nonetheless, they usually provide a worthwhile improvement on simple steepest descent methods.

To design a conjugate gradient search, we must set out a means to decide on the conjugate directions and then simply line search down them. The conjugate gradient method starts in the current direction of steepest descent $\mathbf{v}_0 = -\nabla f(\mathbf{x}_0)$ and searches until the minimum is found in that direction. Thereafter, it moves in conjugate directions such that each subsequent direction vector $\mathbf{v}_i$ obeys the conjugacy condition $\mathbf{v}_i \mathbf{A} \mathbf{v}_j = 0$. These are found without direct knowledge of $\mathbf{A}$ by use of the recursion formula $\mathbf{v}_{i+1} = \lambda \mathbf{v}_i - \nabla f(\mathbf{x}_{i+1})$, where

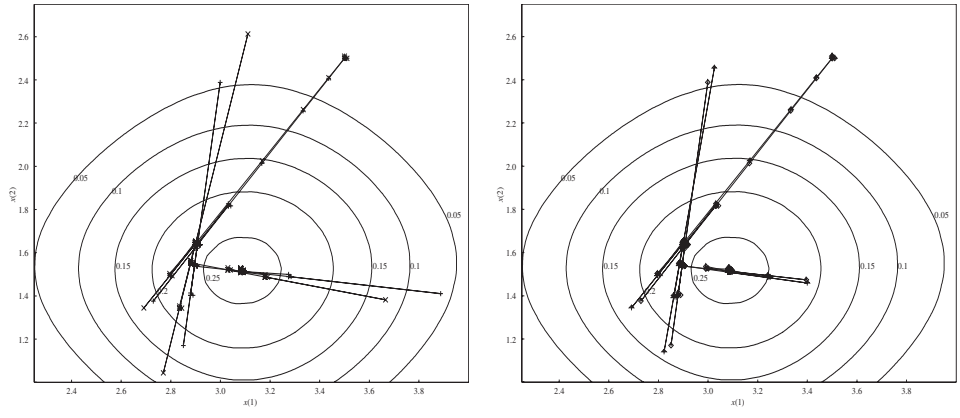
$$\lambda = \nabla f(\mathbf{x}_{i+1})^{\mathrm{T}} \nabla f(\mathbf{x}_{i+1}) / \nabla f(\mathbf{x}_i)^{\mathrm{T}} \nabla f(\mathbf{x}_i) \quad \text{or} \tag{3.5}$$

$$\lambda = (\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i))^{\mathrm{T}} \nabla f(\mathbf{x}_{i+1}) / \nabla f(\mathbf{x}_i)^{\mathrm{T}} \nabla f(\mathbf{x}_i). \tag{3.6}$$

The first of these two expressions for the factor λ is due to Fletcher and Reeves and is the original version of the method, while the later is due to Polak and Ribiere and helps the process deal with functions that are not precisely quadratic in nature. Notice that in this process there is no attempt to build up a model of the matrix $\mathbf{A}$ (requiring storage of the dimension of the problem size squared) – rather the idea is to be able to identify the next conjugate direction whenever a line minimization is complete using storage proportional to the dimension of the problem. The behavior of this kind of search is directly dependent on the quality of the gradient information being used and the accuracy of the line searches carried out in each of the conjugate directions.

Conjugate gradient searches are illustrated in Figures 3.6(a) and (b) for both forms of the recursion formula and with gradients either found by finite differencing or explicitly from the formulation of the problem itself. In all cases, Brent's line search is used along each conjugate direction, the code being taken from the work of Press et al. (1992). Rather coarse steps are used for the finite differencing schemes to illustrate the effect that inaccuracies in gradient information can have on the search process, but in fact, the consequences are here minor, amounting to around 5% more function calls and less than 0.1% error in the final solution (although this does help distinguish the traces in the figures). Table 3.1 details the results of these four searches. It is clear from the figures and the table that even though the function being dealt with here is very smooth and its contours quite close to elliptic, the recursion formula due to Polak and Ribiere leads to a much more efficient identification of the true conjugate directions. Further, the use of exact gradient information not only speeds up the process, but also leads to more accurate answers.

It should be noted in passing that the code implementations due to Press et al. have been criticized in some quarters as being rather inefficient and sometimes not completely reliable. There are faster implementations but these are usually more complex to code and not always as robust. To illustrate the speed differences, Table 3.1 also contains results from the version described by Gilbert and Nocedal (1992), which are available from the web-based



(a) Fletcher–Reeves (b) Polak–Ribiere

Figure 3.6 Conjugate gradient searches.

Table 3.1 Results of conjugate gradient searches on local optimum in the bump function of (3.2)

Method	**Gradient Calculation**	**Source of Code**	**Final Objective Function Value (% of true optimal objective)**	**Number of Function Calls (gradient calls)**
Fletcher–Reeves	Direct	Press et al.	−0.26201 (99.7)	215 (84)
		Gilbert and Nocedal	−0.26289 (100)	99 (99)
	Finite difference	Press et al.	−0.26106 (99.3)	385 (0)
		Gilbert and Nocedal	−0.26267 (99.9)	282 (0)
Polak–Ribiere	Direct	Press et al.	−0.26290 (100)	52 (19)
		Gilbert and Nocedal	−0.26290 (100)	17 (17)
	Finite difference	Press et al.	−0.26289 (100)	113 (0)
		Gilbert and Nocedal	−0.26286 (100)	45 (0)

NEOS service.[2] These routines are substantially quicker (particularly when direct gradient information is available), but also, in our experience, slightly more sensitive to noise and rounding errors – the main differences stemming from the use of Mor'e and Thuente's line

[2] `http://www-neos.mcs.anl.gov/neos/`

search (Mor'e and Thuente 1994). The routine also seems not to like the function discontinuities introduced by the one pass external penalty functions often used when constrained problems are being tackled (see section 3.3.4). In all cases, similar tolerances on the final solution were specified. Interestingly, the Press et al. code does not invoke the gradient calculation at every step whereas the Gilbert and Nocedal version does. Choice in these areas is often a matter of personal preference and experience and we make no particular recommendation – we would also note that different workers' implementations of the same basic approaches often differ in this way.

Quasi-Newton or Variable Metric Methods

Quasi-Newton or variable metric methods also aim to find a local minimum with a minimal sequence of line searches. They differ from conjugate gradient methods in that they slowly build up a model of the Hessian matrix $\mathbf{A}$. The extra storage required for most problems encountered in engineering design is of no great consequence (modern PCs can happily deal with problems where the Hessian has dimensions in the hundreds with no great difficulty). There is some evidence that they are faster than conjugate gradient methods but this is problem specific and also not so overwhelming that it renders conjugate gradient methods completely obsolete. Again, choice between such approaches often comes down to personal preference and experience.

The basic idea of quasi-Newton methods is to construct a sequence of matrices that slowly approximate the inverse of the Hessian ($\mathbf{A}^{-1}$) in the region of the local minimum with ever greater accuracy. Ideally, this sequence converges to $\mathbf{A}^{-1}$ in as many line searches as there are dimensions in the problem. Given an approximation to the inverse Hessian $\mathbf{A}^{-1}$, it is possible to compute the next step in the sequence in the same fashion as a Newton search, although since we know it is only an approximation, line searches are used rather than simple moves directly to the presumed optimal location. A key factor in the practical implementation of this process is that the approximations to the inverse Hessian are limited to being positive definite throughout, so that the search always moves in a downhill direction – recall that far from the local minimum, the function being searched may not behave quadratically and so the actual inverse Hessian at such a point may in fact point uphill. Even so, it is still possible to make an uphill step by overshooting the minimum in the current direction. Therefore, when implementing the algorithm, the step sizes must be limited to avoid overshoot. The Broyden–Fletcher–Goldfarb–Shanno (BFGS) updating strategy achieves this (Press et al. 1992). Figure 3.7 and Table 3.2 illustrate the process on the bump problem, c.f., Figures 3.6(a) and (b) and Table 3.1. The results obtained are essentially identical to those yielded by the Polak–Ribiere conjugate gradient search.

Quasi-Newton routines can suffer from problems if the variables being used are poorly scaled, as the inverse Hessian approximations can then become singular. By working with the Cholesky decomposition of the Hessian rather than updates to its inverse, it is possible to make sure that round-off errors do not lead to such problems and it is also then simple to ensure that the Hessian is always positive definite – a version of this approach is due to Spellucci (1996). An alternative is simply to restart the basic quasi-Newton method once it has finished to see if it can make any further improvement – although untidy, such a simple strategy is very easy to implement, of course. The results from using Spellucci's code are also given in Table 3.2. This works much faster when direct gradients are available but slightly slower when they are not (although it then returns a slightly more accurate answer).

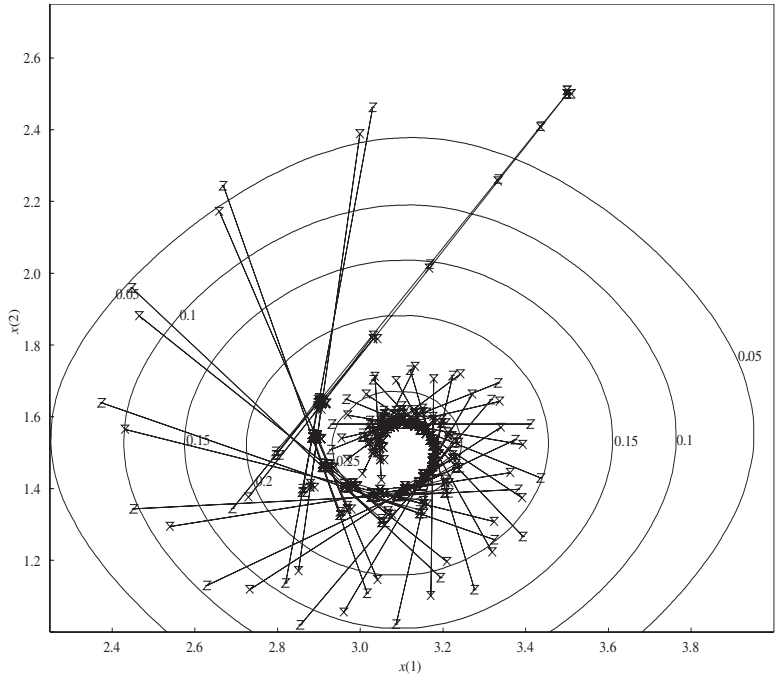


Figure 3.7 Broyden–Fletcher–Goldfarb–Shanno quasi-Newton searches.

Table 3.2 Results of quasi-Newton searches on local optimum in the bump function of (3.2)

Method	**Gradient Calculation**	**Source of Code**	**Final Objective Function Value (% of true optimal objective)**	**Number of Function Calls (gradient calls)**
Broyden–	Direct	Press et al.	−0.26290 (100)	54 (20)
Fletcher–		Spellucci	−0.26290 (100)	13 (12)
Goldfarb–	Finite	Press et al.	−0.26288 (100)	124 (0)
Shanno–	difference	Spellucci	−0.26290 (100)	157 (0)

This is because it uses a rather complex and accurate six-point method to compute the gradients numerically when they are not available analytically.

The impact of the large finite difference steps we have used with the Press et al. code (and thus the less accurate gradients being made available) is more noticeable here – adopting a more normal step size reduces the function call count by around 30%, although again the impact on the final results is still less than 0.1%. This sensitivity explains why the Spellucci code uses the sophisticated approach that it does – since a model of the Hessian is being slowly constructed in quasi-Newton methods, it is important for efficiency that this is not corrupted by inaccurate information. Note also that the Spellucci quasi-Newton search with

direct gradient calculations is clearly the fastest gradient-based method demonstrated of all those considered here.

It is clear from a study of these methods that computation of the gradients is key to how well they perform. Effort can be expended to do this highly accurately or, alternatively, more robust line searches can be used. In either case, the resulting searches are much more costly than where direct gradient information is available. It is these observations that have lead to the considerable interest in the development of adjoint and direct numerical differentiation schemes in many areas of analysis (and which are discussed in Chapter 4).

3.2.3 Noisy/Approximate Function Values

It will be clear from the previous subsections that gradient-based searches can rapidly identify optima in a fitness landscape with high precision. Although there is some variation between methods and also some subtleties in the fine workings of these searches, there is much experience on their use and readily available public domain codes. Of course, these methods only seek to follow the local gradient to the nearest optimum in the function, and so, if they are to be used where there are multiple optima, they must be restarted from several different starting points to try and find these alternative solutions. One great weakness in gradient-based searches, however, is their rather poor ability to deal with any noise in the objective function landscape.

Noise in computational methods is rather different from the kind found in ordinary experimental or in-service testing of products because it is repeatable in nature. When one carries out multiple wind tunnel tests on the same wing, each result will differ slightly from the last – this is a commonplace experience, and all practical experimenters will take steps to deal with the noise, usually by some form of averaging over multiple samples. With computational models, a rather different scenario arises: repeating a computation multiple times will (at least should) lead to the same results being returned each time. This does not mean that such results are free of noise, however: if the solver being used is iterative in nature or if it requires a discretization scheme, then both of these aspects will influence any results. When a small adjustment is made to the geometry being considered, then the convergence process or the discretization may well differ sufficiently for any changes in the results to be as much due to these sources of inaccuracy as to the altered geometry itself. If one then systematically plots the variation of some objective function for a fine grained series of geometry changes, the resulting curves appear to be contaminated with noise; see Figure 3.8 – the fact that the detail of such a noise landscape is repeatable does not mitigate the impact it can have on optimization processes, particularly those that make use of gradients, and most severely, if those gradients are obtained by finite differencing over short length scales.

To illustrate this fact, the results from the previous two tables may be compared with those in Table 3.3. Here, the same objective function has been used except that in each case a small (1%) random element has been added to the function (and where used, the gradient). The results in the table are averages taken over 100 runs of each search method with statistically independent noise landscapes in each case. It may be seen that there are two consequences: first, the methods take many more steps to converge (if they do converge), and secondly, the locations returned by the optimizers are often significantly in error compared to the actual optimum. These tests reveal that quasi-Newton methods have greater difficulty dealing with noise, in that not only do they return poorer results but also sometimes the

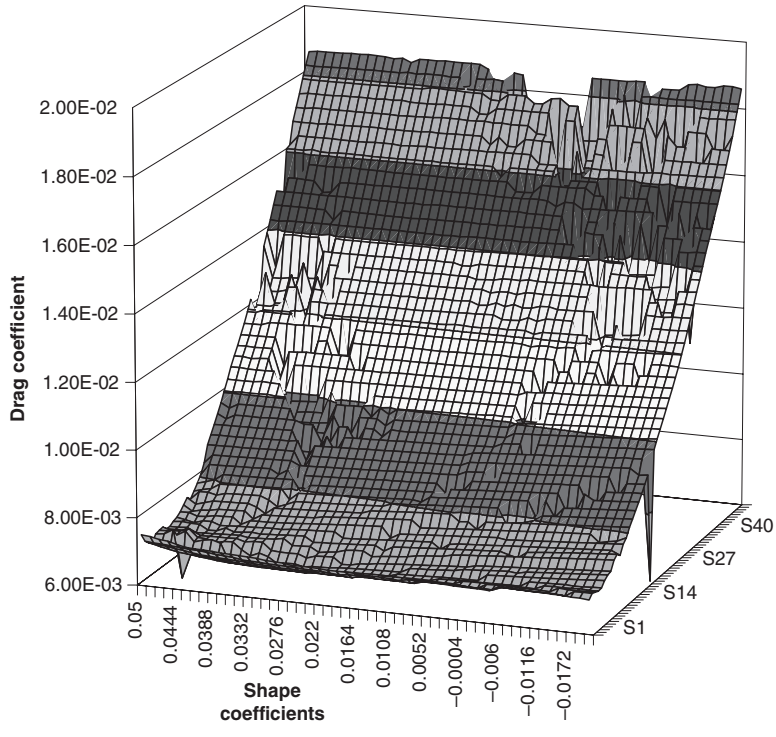


Figure 3.8 An objective function contaminated by discretization noise and the occasional failed calculation – airfoil section drag computed using a full potential code for variations in orthogonal shape functions (M = 0.80, t/c = 0.06, Cl = 0.4).

calculations become singular and fail. Note also that the standard deviations in the results in the table tend to be worse when using finite differences than for the directly evaluated gradients, although, interestingly, the mean value of the Polak–Ribiere conjugate gradient search due to Press et al. is in fact slightly better when using finite differences. Recall, however, that a rather coarse finite difference step length is being used here (0.01) and this in fact helps when dealing with a noisy function since then the chances of extreme gradients being calculated are reduced. If a much smaller (and more normal) step length of 1×10^{-7} is used, the results become much worse for the finite differencing schemes – that for the Press et al. Polak–Ribiere search drops to an average objective function value of -0.17595 (66.9% of the true value) with a standard deviation of 0.0886 (an order of magnitude worse) and an average search length of 1,083 calls. Clearly, when dealing with a function that is not strictly smooth, any gradient-based method must be used with extreme caution. If finite differencing is to be used, then step lengths must be adopted that are comparable with the noise content. Note that, again here, we also give results from the Gilbert and Nocedal conjugate gradient and Spellucci BFGS searches for comparison – they fare less well as they often fail to converge and give rather poorer results, though they retain their undoubted speed advantage.

Table 3.3 Average results of conjugate gradient and quasi-Newton searches on local optimum in the bump function of (3.2) with 1% noise

Method	Gradient Calculation	Source of Code	Final Objective Function Value (% of true optimal objective)	Standard Deviation in Final Objective Function Value	Number of Function Calls (gradient calls)
Fletcher–Reeves	Direct	Press et al.	−0.25564 (97.2)	0.0078	282 (128)
		Gilbert and Nocedal	−0.24292 (92.4) fails sometimes	0.0084	13 (13)
	Finite difference	Press et al.	−0.25265 (96.1)	0.0096	691 (0)
		Gilbert and Nocedal	−0.23763 (90.4) fails sometimes	0.0087	32 (0)
Polak–Ribiere	Direct	Press et al.	−0.25742 (97.9)	0.0077	455 (142)
		Gilbert and Nocedal	−0.24259 (92.3) fails sometimes	0.0088	13 (13)
	Finite difference	Press et al.	−0.26092 (99.2)	0.0100	628 (0)
		Gilbert and Nocedal	−0.23890 (90.9) fails sometimes	0.0090	34 (0)
Broyden–Fletcher–Goldfarb–Shanno	Direct	Press et al.	−0.24924 (94.8) fails sometimes	0.0095	143 (57)
		Spellucci	−0.07267 (27.6) fails most times	0.1016	32 (8)
	Finite difference	Press et al.	−0.24990 (95.1) fails sometimes	0.0116	269 (0)
		Spellucci	−0.01935 (7.4) fails always	0.0000	29 (0)

It is also the case that none of the codes used here has been specifically adapted for working in noisy landscapes. If the designer knows that the analysis methods in use have some kind of noise content, then great care should be taken when selecting and using search engines. Certainly, the methods that are fastest on clean data will rarely prove so effective when noise is present – they can however sometimes be adapted to cope in these circumstances (as here, for example, by using larger than normal step sizes when finite differencing).

3.2.4 Nongradient Algorithms

Because establishing the gradient of the objective function in all directions during a search can prove difficult, time consuming and sensitive to noise, a variety of search methods have been developed that work without explicit knowledge of gradients. These methods may be grouped in a variety of ways and are known by a range of names. Perhaps, the best term for them collectively is "zeroth order methods" (as opposed to first-order methods, which draw on the first differential of the function, etc.) although the term "direct search" is also often used. The most common distinction within these methods is between pattern/direct searches and stochastic/evolutionary algorithms. This distinction is both historical, in that pattern/direct search methods were developed first, and also functional since the stochastic and evolutionary methods are generally aimed at finding multiple optima in a landscape, while pattern/direct methods tend simply to be trying to find the location of the nearest optimum without recourse to gradient calculations. In either case, it will be clear that the problems of obtaining gradients by finite differencing in a noisy landscape are immediately overcome by simply not carrying out this step. It is also the case that these methods are often capable of working directly with discrete variable problems since at most stages all that they need to be able to do is to rank alternative designs. If, however, the problem at hand is not noisy, gradients can be found relatively easily and only a local search is required, it is always better to use gradient-based methods, such as the conjugate or quasi-Newton-based approaches already described. When there are modest amounts of noise or the gradients are modestly expensive to compute, no clear-cut guidance can be offered and experimentation with the various alternatives is then well worthwhile.

Pattern or Direct Search

Pattern (or direct) searches all involve some form of algorithm that seeks to improve a design based on samples in the vicinity of the current point. They work by making a trial step in some direction and seeing if the resulting design is better than that at the base point. This comparison does not require knowledge of the gradient or distance between the points, but simply that the two designs can be correctly ranked. Then, if the new design is an improvement, the step is captured and a further trial made, usually in the same direction but perhaps with a different step length. If the trial move does not improve the design, an alternative direction is chosen and a further trial made. If steps in all directions make the design worse, the step size is reduced and a further set of explorations made. Sometimes, the local trials are augmented with longer-range moves in an attempt to speed up the search process. Usually, these are based on analysis of recent successful samples. In this way, a series of gradually converging steps is made that tries to greedily capture improvements on the current situation. This process continues until some minimum length of step fails to produce an improvement in any direction and it is then assumed that the local optimum has

been reached. A common feature of these searches is that the directions used in the local trial *span* the space being searched, that is, they comprise a set of directions that can be used to create a vector in any direction. In this way, they offer some guarantees of convergence, though often these are not as robust as for gradient-based methods.

In general, pattern searches have no way of accepting designs poorer than the current base point, even if moves in such a direction may ultimately prove to be profitable. It is this fact that makes these methods suitable only for finding the location of the nearest optimum and not for global searches over multimodal landscapes. If multiple optima are to be found with pattern searches, then they must be restarted from various locations so that multiple basins of attraction can be sampled. Usually, stochastic or evolutionary methods are better for such tasks, at least to begin with.

To see how pattern searches work, it is useful to review two well-accepted methods. The most well known methods are probably those due to Hooke and Jeeves (1960) and Nelder and Mead (1965). This latter search is also commonly referred to as the Simplex algorithm – not to be confused with the simplex method of linear programming. Both date from the 1960s and their continued use is testament to their inherent robustness in dealing with real problems and relative simplicity of programming. The approach due to Hooke and Jeeves is the earlier, so we begin with a description of their method.

The pattern search of Hooke and Jeeves begins at whatever point the user supplies and carries out an initial exploratory search by making a step of the given initial step size in the direction of increasing the first variable. If this improves the design, the base point is updated and the second variable is tested. If it fails, a negative step in the direction of the first variable is taken and again this is kept if it helps. If both fail, the second variable is tested, with any gains being kept, and so on, until all steps of this size have been tested and any gains made. This sequence of explorations gives a characteristic staircase pattern, aligned with the coordinate system; see the clusters of points in Figure 3.9.

When all coordinate directions have been tested, a pattern move is made. This involves changing all variables by the same amount as cumulated over the previous exploratory search (the diagonal moves in Figure 3.9). If this works, it is kept, while if not, it is abandoned (this happens twice in the figure, where it overshoots the downhill direction). In either case, a new exploratory search is made. At the end of the exploration, another pattern move is made; however, this time if the previous pattern move was successful, the next pattern move is built from a combination of the just completed exploratory steps and the previous pattern move, that is, the pattern move becomes bigger after each successful exploratory search/pattern move combination and slowly aligns itself to the most profitable direction of move. This allows the search to make larger and larger steps. Conversely, once a pattern move fails, the search resets and has to start accumulating pattern moves from scratch. Finally, if none of the exploratory steps yields an improvement, the step size is reduced by an arbitrary factor and the whole process restarted. This goes on until the minimum step size is reached. The only parameters needed are the initial and minimum step sizes, the reduction factor when exploratory moves fail and the total number of steps allowed before aborting the search. Clearly, a smaller final step size gives a more refined answer, while a smaller reduction factor tends to slow the search into a more careful process that is more capable of teasing out fine detail.

It is immediately apparent from studying the figure that the Hooke and Jeeves search will not initially align itself with the prevailing downhill direction. Rather, a number of pattern moves have to be made before the method adapts to the local topography. If, while

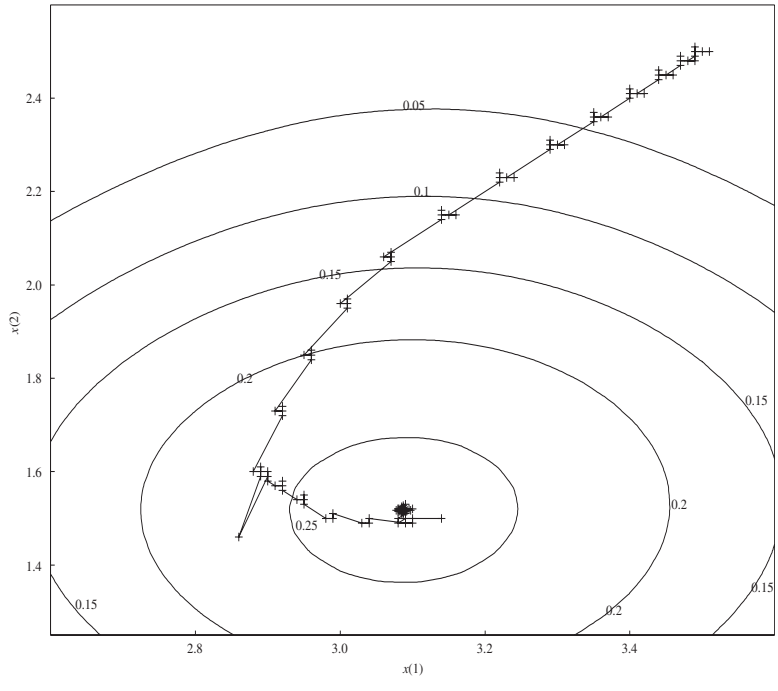


Figure 3.9 The Hooke and Jeeves pattern search in two dimensions.

Table 3.4 Results of pattern searches on local optimum in the bump function of (3.2) without and with 1% noise (using the code provided by Siddall in both cases)

Method	**Noise**	**Final Objective Function Value (% of true optimal objective)**	**Number of Function Calls**
Hooke and Jeeves	none	−0.26290 (100)	167
	1%	−0.22757 (86.6)	375
Nelder and Mead	none	−0.26290 (100)	181
	1%	−0.25409 (96.6)	149

these moves are made, the local landscape seen by the optimizer changes so that the ideal search direction swings around, as here, eventually the method starts to run uphill and a new set of explorations becomes necessary. Thus, it is almost inevitable that such a search will be slower than those that build models of the gradient landscape. Nonetheless, the search is robust and quite efficient when compared to other schemes, particularly on noisy landscapes or those with discontinuities; see Table 3.4, which shows results achieved from the version of the code provided in Siddall (1982).

The Nelder and Mead simplex search is based around the idea of the simplex, a geometrical shape that has $n + 1$ vertices in n dimensions (i.e., a triangle in two dimensions and a tetrahedron in three, etc). Provided a simplex encloses some volume (or area in two

dimensions), then if one vertex is taken as the origin, the vectors connecting it to the other vertices span the vector space being searched (i.e., a combination of these vectors, suitably scaled, will reach any other point in the domain). The basic moves of the algorithm are as follows:

1. reflection – the vertex where the function is worst is reflected through the face opposite, but the volume of the simplex is maintained;
2. reflection and expansion – the worst vertex is again reflected through the face opposite but the point is placed further from the face being used for reflection than where it started with the volume of the simplex being increased;
3. contraction – a vertex is simply pulled toward the face opposite and the volume of the simplex reduced;
4. multiple contraction – a face is pulled toward the vertex opposite, again reducing the volume of the simplex.

To decide which step to take next, the algorithm compares the best, next to best and worst vertices of the simplex to a point opposite the worst point and outside the simplex (the trial point), labeled b, n, w and r, respectively in Figure 3.10, which illustrates the search in two dimensions (and where the simplexes are then triangles). The distance the trial point lies outside the simplex is controlled by a scaling parameter that can be user defined, but is commonly unity. The following tests are then applied in series:

1. if the objective function at the trial point lies between the best and the next to best, the trial point is accepted in lieu of the worst, that is, the simplex is reflected;

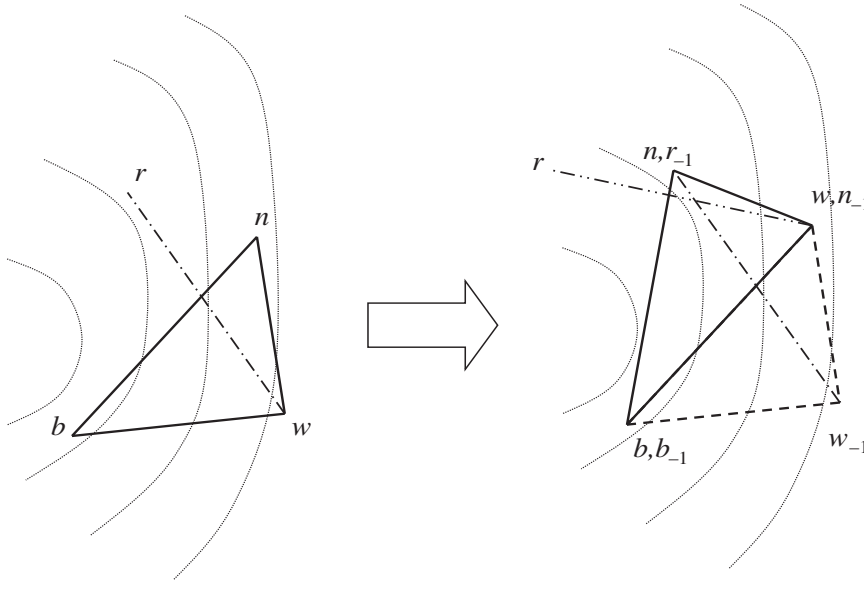


Figure 3.10 The Nelder and Mead simplex method.

2. if the trial point is better than the current best point, not only is the worst point replaced by the best but the simplex is expanded in this direction, that is, reflection and expansion – if the expansion fails, a simple reflection is used instead;

3. if the trial point is poorer than the current worst point, the simplex is contracted such that a new trial point is found along the line joining the current worst point and the existing trial point, if this new trial point is better than the worst simplex point, it is accepted, that is, contraction, while if it is not, all points bar the best are contracted, that is, multiple contraction;

4. otherwise, the trial point is at least better than the worst, so a new simplex is constructed on the basis of this point replacing the worst and in addition a contraction is also carried out.

This rather complex series of steps is set out in the original reference and many subsequent books. It is studied in some detail by Lagarias et al. (1998). The process terminates when the simplex is reduced to a user set minimum size or the average difference between the vertex objective function values is below a user-supplied tolerance. To apply the method, the user must supply the coordinates of the first simplex (usually by making a series of small deflections to a single starting point) and decide the parameters controlling the expansion and contraction ratios. Table 3.4 also shows results for using this routine on the bump function both with and without noise. Figure 3.11 illustrates the search path for the noise-free case.

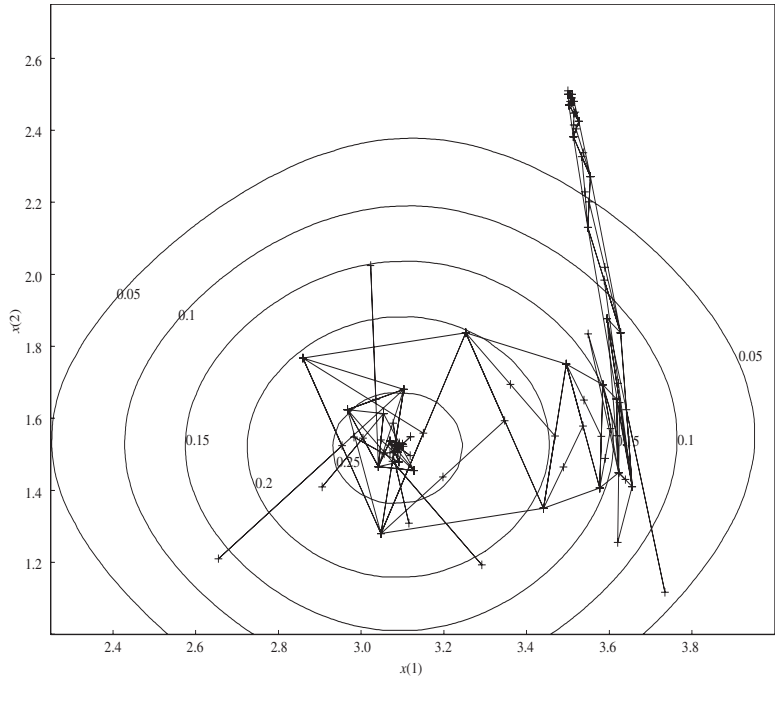


Figure 3.11 The Nelder and Mead simplex search in two dimensions.

The method is reasonably efficient and robust and not surprisingly therefore remains very popular.

This brief description of two of the oldest pattern or direct searches illustrates two aspects that are common to all such methods: first, there is a process for making cautious downhill movements by searching exhaustively across a set of directions that span the search space, typically with quite modest steps (sometimes referred to as "polling" the search space); and, secondly, there is some form of acceleration process that seeks to take larger steps when the previous few moves seem to justify this. The first of these is aimed at ensuring some form of convergence while the second tries to make the process more efficient by covering the ground with fewer steps. When designing a pattern search, these two issues must be borne in mind and suitable strategies implemented to allow the search to switch between them when needed.

Perhaps, the greatest area for innovation lies in the way that accelerated moves are made: in this phase, the algorithm is seeking to guess what the surface will be like some distance beyond what may be justified by a simple linear model using the current local exploration. A number of ideas have been proposed and some researchers refer to this part of pattern searches as "the oracle", indicating the attempt to see into the future! There are a number of themes that emerge in such exploration systems. One approach is to use as much of the previous history of the search as possible to build up some approximation that can used as a surrogate for the real function – this can then be analyzed to locate interesting places to try. It has one significant drawback – how far back should data be used in constructing this approximation? If the data used is too far from the current area of search, it may be misleading, and if it is too recent, it may add very little. The most common alternative is to scatter a small number of points forward from the current location and build an approximation that reaches into the search – clearly, for this to differ from the basic polling search process, it must use larger steps and fewer of them. Again, this is the drawback – if too few steps are used, the approximation will be poor, and if they are to close to the current design, the gains will be limited. Formal design of experiment (DoE) methods can help in planning such moves in that they help maximize the return on investment in such speculative calculations.

In any work on constructing oracle-like approximations, choice must also be made as to whether the approximation in use will interpolate the data used or alternatively a lower order regression model will be adopted. Adaptive step lengths can also be programmed in so that the algorithm becomes greedier the more successful it is. Provided the search is always able to fall back on the small step, exhaustive local polling searches using a basis that spans the space, it should be able to get (near) to the optimal location in the end. It is, of course, also the case that in much design work an improvement in design is the main goal, given reasonable amounts of computing effort, rather than a fully converged optimal design, which may be oversensitive to some of the uncertainties always present in computationally based performance prediction.

Although pattern searches remain popular, they do have drawbacks, often in terms of final convergence. For example, it has been demonstrated that the original Nelder and Mead method can be fooled by some search problems into collapsing its simplexes into degenerate forms that stall the search. In fact, all pattern search methods can be fooled by suitably chosen functions that are either discontinuous or have discontinuous gradients into terminating before finding a local optimum. There are a number of refinements now available that correct many of these issues, but the basic limitation remains. The work of

Kolda et al. (2003) shows how far provable convergence can be taken in various forms of pattern search, for example. Despite such limitations, pattern search methods often work better in practice than might be expected in theory. Moreover, a simple restart of a pattern search is often the easiest way to check for convergence. If the method is truly at an optimal location, the cost of such a restart is typically small compared to the cost of arriving at the optimum in the first place. Another approach is to take a small random walk in the vicinity of the final design and see if this helps at all – again, this is easy to program and is usually of low cost.

It will be apparent from all this that many variants of pattern/direct search exist and some effort has been directed toward establishing a taxonomy of methods (Kolda et al. 2003). One variant that the authors have found useful for problems with a small amount of multimodality is the Dynamic Hill Climbing (DHC) method of Yuret and de la Maza (1993) – this combines pattern search with an automated restart capability that seeks to identify fresh basins of attraction. Since these restarts are stochastic in nature, it bridges the division between pattern and stochastic methods that we consider next (and is thus a form of hybrid search; see later).

Stochastic and Evolutionary Algorithms

The terms stochastic and evolutionary are used to refer to a range of search algorithms that have two fundamental features in common. First, the methods all make use of random number sequences in some fashion, so that if a search is repeated from the same starting point with all parameters set the same, but with a different random number sequence, it will follow a different trajectory over the fitness landscape in the attempt to locate an optimal design. Secondly, the methods all seek to find globally, as opposed to locally, optimal designs. Thus, they are all capable of rejecting a design that is at some minimum (or maximum) of the objective function in the search for better designs, that is, they are capable of accepting poorer designs in the search for the best design.

These methods have a number of distinct origins and development histories but can, nevertheless, be broadly classified by a single taxonomy. The most well known methods are simulated annealing (SA), genetic algorithms (GA), evolution strategies (ES) and evolutionary programming (EP). We will outline each of these basic approaches in turn before drawing out their common threads and exploring their relative merits and weaknesses.

Simulated annealing optimization draws its inspiration from the process whereby crystalline structures take up minimum energy configurations if cooled sufficiently slowly from some high temperature. The basic idea is to allow the design to "vibrate" wildly to start with so that moves can be made in arbitrary directions and then to slowly "cool" the design so that the moves becomes less arbitrary and more downhill in focus until the design "solidifies", hopefully into some minimum energy state. The basic elements of this process are:

1. a scheme for generating (usually random) perturbations to the current design;
2. a temperature schedule that is monitored when taking decisions;
3. a model of the so-called Boltzmann probability distribution $P(E) = \exp(-\mathrm{d}E/kT)$, where T is the temperature, $\mathrm{d}E$ is the change in energy and k is Boltzmann's constant.

At any point in the algorithm, the current objective function value is mapped to an energy level and compared to the energy level of a proposed change in the design produced using

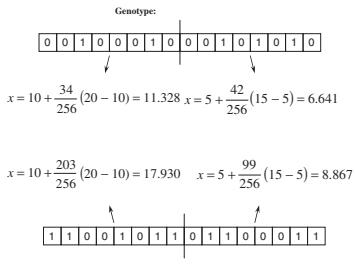
the perturbation scheme, that is, dE is calculated. If the perturbed design is an improvement on the current design, this move is accepted. If it is worse, it may still be accepted but this decision is made according to the Boltzmann probability, that is, a random number uniformly distributed between zero and one is drawn and, if $\exp(-dE/kT)$ is greater than this number, the design is still accepted. To begin with, the temperature T is set high so that almost all changes are kept. Then, as T is reduced, designs have increasingly to be an improvement to be used. In this way, the process starts as a simple random walk across the search space and ends as a greedy downhill search with random moves. This basic Metropolis algorithm is described by Kirkpatrick et al. (1983).

It is nowadays a commonly accepted practice that the temperature should be reduced in a logarithmic fashion though this can be done continuously or in a series of steps (the authors tend to use a fixed number of steps). Also, it is necessary to establish the value for the "Boltzmann" constant for any particular mapping of objective function to E. This is most readily achieved by making a few random samples across the search space to establish the kinds of variation seen when changes are made according to the scheme in use. The aim is that the initial value of kT should be significantly larger than the typical values of dE seen (typically, ten times more).

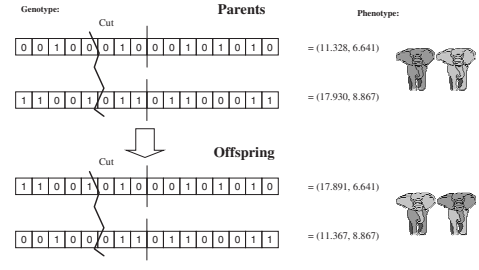
The most difficult aspect of setting up a SA scheme is to decide how to make the perturbations between designs. These need to be able to span the search space being used and can come from a variety of mechanisms. One very simple scheme is just to make random changes to the design variables. Many references describe much more elaborate approaches that have been tuned to problem-specific features of the landscape.

Genetic Algorithms are based on models of Darwinian evolution, that is, survival of the fittest. They have been discussed for at least 35 years in the literature but perhaps the most well known book on the subject is that by Goldberg (1989). The basic idea is that a pool of solutions, known as the population, is built and analyzed and then used to construct a new, and hopefully improved, pool by methods that mimic those of natural selection. Key among these are the ideas of fitness and crossover, that is, designs that are better than the average are more likely to contribute to the next generation than those that are below average (which provides the pressure for improvement) and that new designs are produced by combining ideas from multiple (usually two) "parents" in the population (which provides a mechanism for exploring the search space and taking successful ideas forward). This process is meant to mimic the natural process of crossover in the DNA of creatures that are successful in their environment. Here, the vector of design variables is treated as the genetic material of the design and it is this that is used when creating new designs. Various forms of encoding this information are used, of which the most popular are a simple binary mapping and the use of the real variables directly. There are numerous ways of modeling the selection of the fittest and for carrying out the "breeding" process of crossover. The simplest would be roulette wheel sampling whereby a design's chances of being used for breeding are random but directly proportional to fitness and one point binary crossover whereby a new design is created by taking a random length string of information from the leading end of parent one and simply filling the remaining gaps from the back end of parent two.

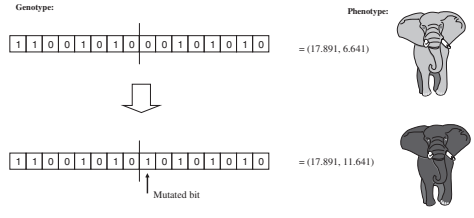
There are a number of further features that are found in GAs: mutation is commonly adopted, whereby small random changes are introduced into designs so that, in theory, all regions of the search space may be reached by the algorithm. Elitism is also normally used, whereby the best design encountered so far is guaranteed to survive to the next generation unaltered, so that the process cannot lose any gains that is has made (this, of course,



(a) Binary encoding of two designs – phenotype uses real numbers, for example, variables between 10.0 and 20.0 and 5.0 and 15.0 using eight binary digits for each variable.



(b) Cut points and results for crossover – here, crossover acts a single point in the genotype and all material before the cut point is exchanged in the offspring.



(c) Bit-flip mutation – a single bit in the genotype is flipped.

Figure 3.12 Genetic algorithm model and operators.

has no natural equivalent – it amounts to immortality until a creature is killed by a better one). Inversion or some form of positional mapping is also sometimes used to ensure that when crossover takes place the location of specific pieces of information in the genetic makeup can be matched to other, related aspects. The population pool in the method can vary in size during search and it can be updated gradually or all at one time. In addition, various methods have been proposed to prevent premature convergence (dominance) of the population by mediocre designs that arrive early in the search. Most common are various niche and resource sharing methods that make overly popular designs work harder for their place in the population. The literature in this field is vast but the interested reader can find most of the ideas that have proved useful by studying the first five sets of proceedings of the International Conference on Genetic Algorithms (Belew and Booker 1991; Forrest 1993; Grefenstette 1985, 1987; Schaffer 1989).

The basic processes involved can be illustrated by considering a simple "vanilla" GA applied to a two-dimensional problem that uses eight-bit binary encoding. In such a model each design is mapped into a binary string of 16 digits; see Figure 3.12a. The first eight bits

are mapped to variable one such that 00000000 equates to the lower bound and 11111111 to the upper, with 256 values in total; and similarly for variable two. If we take two designs, these may be "crossed" using single-point crossover by randomly selecting a cut point between 1 and 15, say 5; see the upper part of Figure 3.12(b). Then, we create two "children" from the two "parents" by forming child one from bits 1–5 of parent one and 6–16 of parent two while child two is formed from the remaining material, that is, bits 1–5 of parent two and bits 6–16 of parent one; see the lower part of Figure 3.12(b). Mutation can then be added by simply flipping an arbitrary bit in arbitrary children according to some random number scheme, usually at low probability; see Figure 3.12(c). Provided that selection for parenthood is linked in some way to the quality of the designs being studied, this seemingly trivial process yields a powerful search engine capable of locating good designs in difficult multimodal search spaces. Nonetheless, a vanilla GA such as that just described is very far from representing the state of the art and will perform poorly when compared to the much more sophisticated methods now commonly available – its description here is given just to illustrate the basic process.

Evolution Strategies use evolving populations of solutions in much the same way as GAs. Their origins lie with a group of German researchers in the 1960s and 1970s (Back et al. 1991) (i.e., at about the same time as GAs). However, as initially conceived, they worked without the process of crossover and instead focused on a sophisticated evolving vector of mutation parameters. Also, they always use a real-valued encoding. The basic idea was to take a single parent design and make mutations to it that differed in the differing dimensions of the search. Then, the resulting child would either be adopted irrespective of any improvements or, alternatively, be compared to the parent and the best taken. In either case, the mutations adopted in producing the child were encoded into the child along with the design variables. These were stored as a list of standard deviations and also, sometimes, as correlations and were then themselves mutated in further operations, that is, the process evolved a mutation control vector that suited the landscape being searched at the same time as it evolved a solution to the problem itself. In later developments, a population-based formulation was introduced that does use a form of crossover and it is in this form that ESs are normally encountered today. They still maintain relatively complex mutation strategies as their hallmark, however. The resulting approach still takes one of two basic forms: with the mutated children automatically replacing their parents, a so-called (μ, λ)-ES, or with the next generation being selected from the combined parents and children, a $(\mu + \lambda)$-ES. In the latter case, if the best parent is better than the best child, it is commonly preserved unchanged (elitism). The best of the solutions is remembered at each pass to ensure that the final result is the best solution of all, irrespective of the selection mechanism.

Crossover can be discrete or intermediate on both the design vector and the mutation control vector. In some versions, the mutations introduced are correlated between the different search dimensions so that the search is better able to traverse long narrow valleys that are inclined to the search axes. The breeding of new children can be discrete and then individual design vector values are taken from either parent randomly or intermediate and the values are then the average of those of the parents. Separate controls for the breeding of the design vector and the mutation standard deviations are commonly available so that these can use different approaches.

In their simplest form, the mutations are controlled by vectors of standard deviations which evolve with the various population members. Each element in the design vector is changed by the addition of a random number with this standard deviation scaled to the range of the variable. The rate at which the standard deviations change is controlled by a parameter of the method, $\Delta\sigma$ such that $\sigma_i^{\text{NEXT}} = \sigma_i^{\text{PREV}} \exp(N(0, \Delta\sigma))$, where the σ_is are the individual standard deviations and $N(0, \Delta\sigma)$ is a random number with zero mean and $\Delta\sigma$ standard deviation. The σ_is are normally restricted to be less than some threshold (say 0.1) so that the changes to the design vector (in a nondimensional unit hypercube) do not simply push the vector to its extreme values. The initial values of the σ_is are user set, typically at 0.025. Since successful children will owe their success, at least in part, to their own vector of mutation controls, the process produces an adaptive mutation mechanism capable of rapidly traversing the design space where needed and also of teasing out fine detail near optimal locations. Because a population of designs is used, the method is capable of searching multiple areas of a landscape in parallel.

In EP algorithms (Fogel 1993), evolution is carried out by forming a mutated child from each parent in a population with mutation related to the objective function so that successful ideas are mutated less. The objective functions of the children are then calculated and a stochastic ranking process used to select the next parents from the combined set of parents and children. The best of the solutions is kept unchanged at each pass to ensure that only improvements are allowed. The mutation is controlled by a parameter that sets the order of the variation in the amount of mutation with ranking. A value of one gives a linear change, two a quadratic, and so on (only positive values being allowed). In all cases, the best parent is not mutated and almost all bits in the worst are changed.

The stochastic process for deciding which elements survive involves jousting each member against a tournament of other members selected at random (including possibly itself) with a score being allocated for the number of members in the tournament worse than the case being examined. Having scored all members of both parent and child generations, the best half are kept to form the next set of parents. The average number of solutions in the tournament is set by a second parameter. An elitist trap ensures that the best of all always survives.

In EP, the variables in the problem are typically discretized in binary arithmetic so that all ones (1111...) again would represent an upper bound and all zeros (0000...) the lower bound. The total number of binary digits used to model a given design vector (the word length) is the sum over each variable of this array – the longer the word length, the greater the possible precision but the larger the search space. The algorithm works by forming a given number of random guesses and then attempting to improve on them, maintaining a population of best guesses as the process continues.

A number of common threads can be drawn out from these four methods:

1. a population of parent designs is used to produce a population of children that inherit some of their characteristics (these populations may comprise only a single member as in SA);

2. the mechanisms by which new designs are produced are stochastic in nature so that multiple statistically independent runs of the search will give different results and therefore averaging will be needed when assessing a method;

3. if there are multiple parents, they may be combined to produce children that inherit some of their characteristics from each parent;

4. all children have some finite probability of undergoing a random mutation that allows the search to reach all parts of the search space;

5. the probability rates that control the process by which changes are made may be fixed, vary according to some preordained schedule or be adapted by the search itself depending on the local landscape;

6. a method is incorporated to record the best design seen so far, as the search can both cause the designs to improve and deteriorate;

7. some form of encoding may be used to provide the material that the search actually works with; and

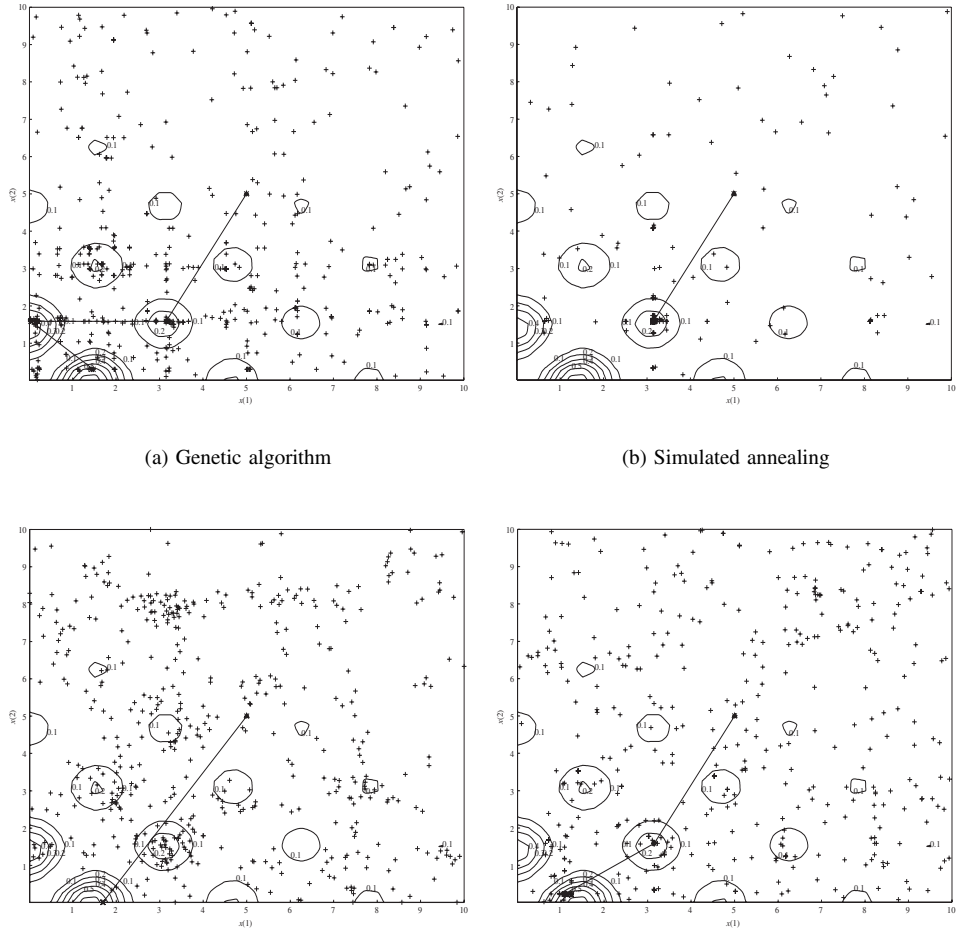
8. a number of user set parameters must be controlled to use the methods.

Finally, it should be noted that no guarantees of ultimate convergence can be provided with evolutionary algorithms (EAs) and they are commonly slow at resolving precise optima, particularly when compared to gradient-based methods.

Figures 3.13(a–d) show the points evaluated by a variant of each search method on the bump test function with the best of generation/annealing temperature points linked by a line (the codes are from the Options search package[3]). In all cases, 500 evaluations have been allowed – such methods traditionally continue until told to stop rather than examining the current best solution, since they are always trying to find new basins of attraction that have yet to be explored. It is immediately apparent from the figures that all the methods search across the whole space and none get bogged down on the nearest local minimum, instead moving from local optimum to local optimum in the search for the globally best design. All correctly identify the best basin of attraction, and Table 3.5 shows how good the final values are when compared to the true optimum of -0.68002 in this unconstrained space, when averaged over 100 independent runs.

There is little to choose between the methods except that the ES method occasionally seems to get trapped in a slightly suboptimal basin. The performance of these methods on harder problems with more dimensions and more local optima is of course not easy to predict from such a simple trial. In almost all cases, careful tuning of a method to a landscape will significantly improve performance. Moreover, when dealing with global search methods like these, a trade-off must be made between speed of improvement in designs, quality of final design returned and robustness against varying random number sequences. Usually, a robust and fast improving process is most useful in the context of aerospace design, rather than one that occasionally does very well but often fails or a very slow and painstaking search that yields the best performance but only at very great computational expense. Another important feature is the extent to which parallel computation can be invoked. Clearly, with a population-based method that builds new generations in a stepwise fashion, an entire

[3] `http://www.soton.ac.uk/~ajk/options/welcome.html`



(a) Genetic algorithm

(b) Simulated annealing

(c) Evolution strategy

(d) Evolutionary programming

Figure 3.13 Evolutionary algorithm search traces on bump problem, (3.2), without constraints.

generation of calculations may be carried out simultaneously – such calculations can often readily be mapped onto the individual processors of a large cluster. The fact that SA does not readily allow this is one of the key drawbacks of the method.

One important common feature of the stochastic methods that make use of encoding schemes is that these can very easily be mapped onto discrete or integer optimization problems. For example, it is very easy to arrange for a binary encoding to map onto a list of items rather than to real numbers – in fact, this is probably a more natural use of such schemes. Even simulated annealing can be used in this way if a binary encoding scheme is used to generate the perturbations needed to move from design to design.

Table 3.5 Results of evolutionary searches on global optimization of the bump function of (3.2) averaged over 100 independent runs using 500 evaluations in each case (using the code provided by Keane in all cases)

Method	Average Result	% of True Optimum (−0.68002)	Standard Deviation in Final Objective Function Value
GA	−0.61606	90.6	0.0751
SA	−0.61896	91.0	0.1128
ES	−0.52065	76.6	0.1117
EP	−0.58528	86.1	0.0831

3.2.5 Termination and Convergence Aspects

In all search methods, some mechanism must be put in place to terminate the process. When engaged on local search, it is normal to terminate when the current design cannot be improved upon in the local vicinity, that is, an optimum has been located. For global search, no such simple test can be made, since there is always the possibility that a basin remains to be found that, if searched, will contain a better design. In the first case, the search is said to be converged, while in the second, it must be terminated using some more complex criterion. These two issues are, of course, related. Moreover, if the cost of evaluations is high, then it may well be necessary to terminate even a local search well before it is fully converged. We are thus always concerned with the rate of improvement of a search with each new evaluation.

In the classical gradient search literature, this rate of improvement is often described as being linear, quadratic or superlinear, depending on an analysis of the underlying algorithm. Formally, a search is said to be linear if, in the limit as the iteration count goes to infinity, the rate of improvement tends to some fixed constant that is less than unity, that is, if $|f(x_{i+1}) - f(x^*)|/|f(x_i) - f(x^*)| = \delta < 1$, where $f(x^*)$ is the final optimal objective function value. If the ratio tends in the limit to zero, the search is said to be superlinear, while if $|f(x_{i+1}) - f(x^*)|/|f(x_i) - f(x^*)|^2 = \delta$, the search is said to be quadratic. For example, the series 0.1, 0.01, 0.001, ... exhibits linear convergence, $1/i! = 1$, 1/2, 1/6, 1/24, 1/125, ... is superlinear and 0.1, 0.01, 0.0001, 0.000000001, ... is quadratic.

Ideally, we would wish all searches to be superlinear or quadratic if possible. Newton's method is quadratically convergent if started sufficiently close to the optimum location. Conjugate gradient and quasi-Newton searches are always at least linear and can approach quadratic convergence. It is much more difficult to classify the speed of pattern and stochastic searches. Fortunately, when carrying out real design work, a steady series of improvements is usually considered enough! Nonetheless, faster searches are generally better than slower ones and so we normally seek a steep gradient in the search history. Note also that the accuracy with which an optimal design can be identified is limited to the square root of the machine precision (and not the full precision), so it is never worth searching beyond this limit – again, in real design work, such accuracy is rarely warranted since manufacturing constraints will limit the precision that can be achieved in any final product.

When carrying out global searches, where there are presumed to be several local optima, a range of other features must be considered when comparing search methods. Consider a graph that shows the variation of objective function with iteration count for a collection of

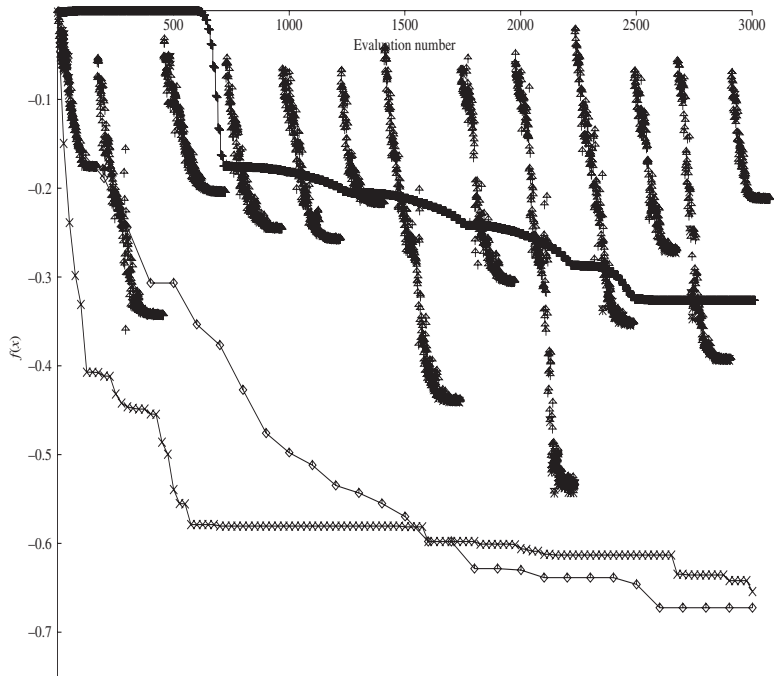


Figure 3.14 Search histories for a range of different search methods (+ – repeated Hooke and Jeeves pattern search with evolving penalty function, × – simulated annealing, arrows – dynamic hill climbing (Yuret and de la Maza 1993) and diamonds – genetic algorithm).

different search methods, such as Figure 3.14. A number of features can be observed in this plot. Obviously, the gradient tells us something about the convergence rate of the search at any point. However, this is nonuniform in these traces and clearly goes through various phases – this is to be expected if some of the time a search is attempting to identify new basins of attraction and sometimes it is rapidly converging to the best point in a newly located basin. Moreover, even if new basins can be regularly located, there will be no guarantee that each new basin will contain a design that improves on the best so far. Sometimes, by chance, or because the starting design is quite good, no basins beyond that first searched will contain better designs. Whatever the circumstances, the designer must make some decision on when to terminate a search and this cannot be based on local convergence criteria alone. The four most common approaches are:

1. to terminate the search after a fixed number of iterations;
2. to terminate the search after a fixed elapsed time;
3. to terminate the search after a fixed number of iterations have produced no further improvement in the global optimum; or
4. to terminate the search after a fixed number of distinct basins of attraction have been located and searched.

The last of these is only viable if the designer has some good understanding of the likely number of basins in the problem being dealt with, though this is often possible when dealing with variants on a familiar problem.

Given a termination criterion, it is then possible to try and rank different global search methods; see again Figure 3.14. Here, we see that one search exhibits very good initial speed but then slows down and is finally overtaken (simulated annealing), one that is slower but continues to steadily improve throughout (genetic algorithm), a third that works in cycles, with significant periods where it is searching inferior basins (dynamic hill climbing – a multistart method) and a fourth that appears to make no progress to begin with but which gets going some distance into the search (a Hooke and Jeeves pattern search with evolving penalty). The methods can be ranked in a number of ways:

1. by the steepness of the initial gradient (useful if the function evaluation cost is high and a simple improvement in design is all that is needed);
2. whether the ultimate design has a very good objective function (good if the function cost is low or if a specific and demanding performance target must be met); or
3. if the area below the trace curve is low (important when one is trying to balance time and quality in some way – N.B., it should be high for maximization, of course).

In addition, a number of other factors may be of interest, such as the volume of the search space traversed. Consider a narrow cylindrical pipe placed around the search path as it traverses the design space or small spheres placed around each new trial point – these represent the volume explored in some sense. The repeatability of the search will also be of interest if it is started from alternative initial designs or if used with differing random number sequences – this is particularly important for stochastic search methods. Ideally, a global search should explore the design space reasonably well within the limits of the available computing budget and also be reasonably robust to starting conditions and random number sequences. Depending on the circumstances, however, a search that is usually not very good at getting high-quality designs but occasionally does really well may be useful – this may be thought equivalent to having a temperamental star player on the team who can sometimes score in unexpected circumstances. Such players can be very useful at times even if they spend many games on the substitutes' bench! Again, we offer no clear-cut recommendation on how to rank search methods – we merely observe that many approaches are possible and the designer should keep an open mind as to how to make rational choices.

3.3 Constrained Optimization

In almost all practical design work, the problems being dealt with are subject to a variety of constraints. Such constraints are defined in many ways and restrict the freedom of the designer to arbitrarily choose dimensions, materials, and so on, that might otherwise lead to the best performance, lightest weight, lowest cost, and so on. It is also the case that many classical optimization problems are similarly constrained and this has led to the development of many ways of trying to satisfy constraints while carrying out optimization. More formally, we can recall our definition of the optimization problem as the requirement to minimize the function $f(\mathbf{x})$ subject to multiple inequality constraints $g_i(\mathbf{x}) \geq 0$ and equality constraints $h_j(\mathbf{x}) = 0$. Note that in many real design problems the number of constraints may be very

large, for example, in stress analysis we may require that the Von Mises stress be less than some design stress at *every* node in a finite element mesh containing hundreds of thousands of elements.

A key issue in dealing with constraints during design is the tolerance with which they must be met. When examining the mathematical subtleties of a method's convergence, we may wish to satisfy constraints to within machine precision, but this is never required in practical design work. In many cases, slight violation of a constraint may be permissible if some other benefit is achieved by doing so – it is rare for anything in design work to be so completely black and white that a change in the last significant digit in a quantity renders a design unacceptable. Moreover, it is also arguable if equality constraints ever occur for continuous variables in design – rather, such constraints can usually be regarded as a tightly bound pair of inequality constraints and this is often how they must be treated in practice. Nonetheless, in this section on constraints, we deal with the formal problem and the methods that have been devised to deal with them – we simply urge the reader to keep in mind that in practical work things are never so simple.

3.3.1 Problem Transformations

The most common way of dealing with constraints is to attempt to recast the problem into one without constraints by some suitable transformation. The simplest example of this is where there is an algebraic equality constraint that may be used to eliminate one of the variables in the design vector. So, for example, consider trying to minimize the surface area of a rectangular box where the volume is to be held fixed. If the sides have lengths L, B and H, then it is clear that the total surface area A is $2(L \times B + L \times H + B \times H)$, while the volume V is just $L \times B \times H$. If the volume is given, then we may use this equality constraint to eliminate the variable H by setting it equal to $V/(L \times B)$ and so the objective function then becomes $2(L \times B + (L + B) \times V/(L \times B))$, that is, a function of two variables and the constraint limit (the value V), rather than three variables. Whenever such changes may be made, this should automatically be done before starting to carry out any optimization. Moreover, even when dealing with inequality constraints, such a transformation may still be possible: if an inequality is not active at the location of the optimum being sought, it can be ignored; if it is active, it can be replaced with an equality. Thus, if knowledge of which constraints will be active can be found, these constraints can be eliminated similarly, if simple enough.

Usually, however, the constraints being dealt with will result from some complex PDE solver that cannot be so easily substituted for. Even when this is the case, equality constraints can sometimes still usefully be removed by identifying a single variable that dominates the constraint and then solving an "inner loop" that attempts to find the value of the key design variable that satisfies the constraint by direct root searching. A classical example of this occurs in wing design – it is common in such problems to require that the wing produce a specified amount of lift while the angle of attack is one of the variables open to the designer. If the overall aim is to change wing shape so as to minimize drag while holding lift fixed, this can best be done by using two or three solves to identify the desired angle of attack for any given geometry rather than simply leaving the angle of attack as one of a number of design variables and attempting to impose a lift equality constraint on the search process. This substitution works because we know, *a priori*, that angle of attack directly controls lift and, moreover, that the relationship between the two is often very simple (approximately linear).

3.3.2 Lagrangian Multipliers

Normally, it is not possible to simply remove constraints by direct substitution; other methods must be adopted. The classical approach to the problem is to use Lagrangian multipliers. If we initially restrict our analysis to equality constraints, then we may substitute $f(\mathbf{x}) - \sum \lambda_j h_j(\mathbf{x})$ for the original constrained function. Here, the λ_j are the so-called Lagrangian multipliers and the modified function is referred to as the Lagrangian function (or simply the "Lagrangian" in much of the work in this field). If we now find a solution such that the gradients of the Lagrangian all go to zero at the same time as the constraints are satisfied, the final result will minimize the original problem (there are a number of further requirements such as the functions being differentiable, linear independence of constraint gradients, the Hessian of the Lagrangian being positive definite, and so on, the so-called Karush–Kuhn–Tucker (KKT) conditions, but we omit discussions of these here for clarity – see Boggs and Tolle (1989, 1995) for more complete details). The inclusion of the multipliers increases the problem dimensions, since they must be chosen as well as the design variables, but it also turns the optimization problem into the solution of a set of coupled (nonlinear) equations.

A simple example will illustrate the multiplier idea. Let the objective function being minimized $f(\mathbf{x})$ be just x^2 and the equality constraint be $x = 1$. It is obvious that the function has a minimum of 1 at $x = 1$ (no other value of x satisfies the constraint). The equivalent Lagrangian function with multiplier, $f'(\mathbf{x})$, is $x^2 - \lambda\ (x - 1)$, where the equality constraint has been set up such that it equals zero, as required in our formulation. Figure 3.15 shows

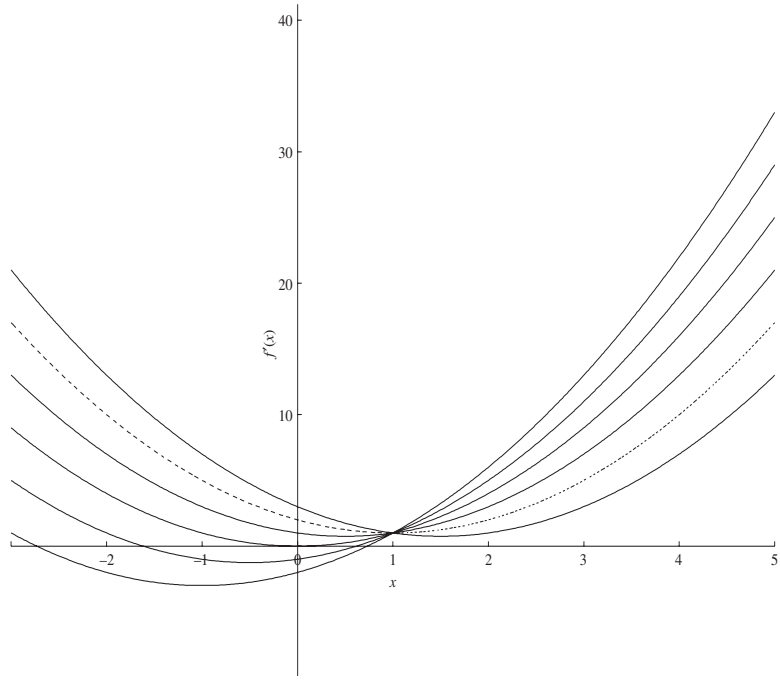


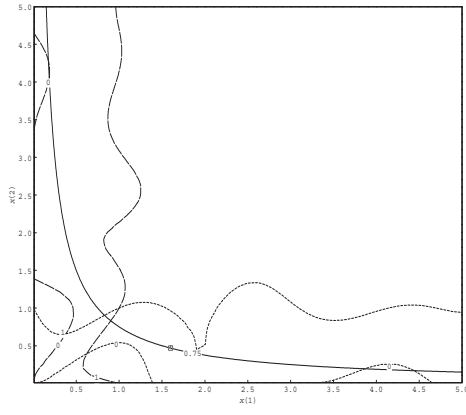
Figure 3.15 Plots of the Lagrangian function for various values of the Lagrangian multiplier (that shown dotted is for $\lambda = 2$).

a series of curves of this function for differing values of λ. Of these, the curve for $\lambda = 2$ is the one that has zero gradient (its minimum) at $x = 1$ and thus satisfies the constraint, that is, minimizing $x^2 - 2(x - 1)$ minimizes the original objective function and satisfies the equality constraint. When dealing with arbitrary functions, one must simultaneously seek zeros of all the gradients of $f'(\mathbf{x})$ and search over all the λs for values that meet the constraints.

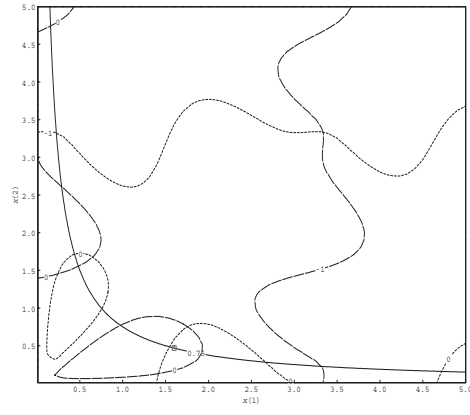
Now, consider the situation where we have a single inequality constraint, for example, $x \geq 1$. It will be obvious that the solution to this problem is also at $x = 1$ and again the objective function is just 1. To deal with this problem using Lagrangian multipliers, we must introduce the idea of slack variables to convert the inequality constraint to an equality constraint. Thus, if we have constraints $g_i(\mathbf{x}) \geq 0$, we can write them as $g_i(\mathbf{x}) - a_i^2 = 0$, where a_i is a slack variable and then, proceeding as before, the Lagrangian becomes $f(\mathbf{x}) - \sum \lambda_j h_j(\mathbf{x}) - \sum \lambda_i [g_i(\mathbf{x}) - a_i^2]$. This of course introduces a further set of variables and so now we must, in principle, search over the xs, λs *and* the as. To obtain equations for the as, we just differentiate the Lagrangian with respect to these as and equate to zero, yielding $2\lambda_i a_i = 0$, so that at the minima either $\lambda_i = 0$ or $a_i = 0$. In the first case, the constraint has no effect (because it is not active at the minima), while in the second it has been reduced to a simple equality constraint. This is perhaps best seen as a rather involved mathematical way of saying that the inactive inequality constraints should be ignored and the active ones treated simply as equalities – hardly an astounding revelation. Nonetheless, in practice, it means that while searching for the Lagrangian multipliers that satisfy the constraints one must also work out which inequality constraints are active and only satisfy those that are – again something that is more difficult to do in practice than to write down. In our example, the inequality is always active and so the solution is as before and the curves of the Lagrangian remain the same.

Consider next our two-dimensional bumpy test function. Here, we have $(\cos^4 x_1 + \cos^4 x_2 - 2\cos^2 x_1 \cos^2 x_2)/(x_1^2 + 2x_2^2)$ subject to $x_1 x_2 > 0.75$ and $x_1 + x_2 < 15$. So, adding slack variables a_1 and a_2 and multipliers λ_1 and λ_2, the Lagrangian becomes $(\cos^4 x_1 + \cos^4 x_2 - 2\cos^2 x_1 \cos^2 x_2)/(x_1^2 + 2\,x_2^2) - \lambda_1\,(x_1 x_2 - 0.75 - a_1^2) - \lambda_2\,(x_1 + x_2 - 15 - a_2^2)$. We can then study this new function for arbitrary values of the multipliers and slack variables. To use it to solve our optimization problem, we must identify which of the two inequality constraints will be active at the minimum being sought – let us arbitrarily deal with the product constraint and assume that it is active: remember in the interior space neither constraint is active and so the problem reverts to an unconstrained one as far as local search is concerned. To set the product constraint alone as active, we set a_1 and a_2 both to zero along with λ_2 and just search for λ_1, x and y. Remember that our requirement for a minimum is that the gradients of the Lagrangian with respect to x and y both be zero and that simultaneously the product constraint be satisfied as an equality, that is, three conditions for the three unknowns.

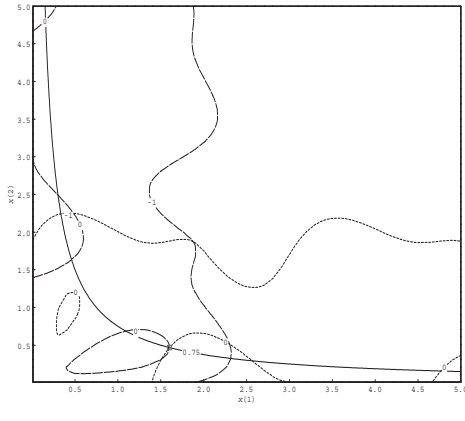
Figures 3.16(a–c) show plots of the gradients and the constraint line for three different values of λ_1 and it is clear that in the third of these, when λ_1 is equal to 0.5332, a solution of (1.60086, 0.46850) solves the system of equations (in the first figure the contours of zero gradient do not cross, while in the second they cross multiple times but away from the constraint line). Figures 3.17 and 3.18 show a plot of the original and Lagrangian functions, respectively, with this optimum position marked. It is clear that the minimum is correctly located and that in Lagrangian space it lies at a saddle point and *not* a minimum (i.e., a simple minimization of the augmented function will not locate the desired optimum – we



(a) $\lambda_1 = -1.0$



(b) $\lambda_1 = 0.3$



(c) $\lambda_1 = 0.5332$

Figure 3.16 Contour lines for gradients of the Lagrangian (dotted with respect to x_1, dashed with respect to x_2) and the equality constraint $x_1 x_2 = 0.75$. The square marker indicates the position of the constrained local minimum.

require only that the Hessian of the Lagrangian be positive definite, that is, a stationary point in the Lagrangian – this means that minimization of the Lagrangian will not solve the original problem and is a key weakness of methods that exploit the KKT conditions directly in this way). This process can be repeated for the other local optima on the product constraint and, by setting λ_1 to be zero and λ_2 nonzero, those on the summation constraint can also be found. The only other possibility is where the constraint lines cross so that both might be active at a minimum. Here, this lies outside the range of interest and so is not considered further.

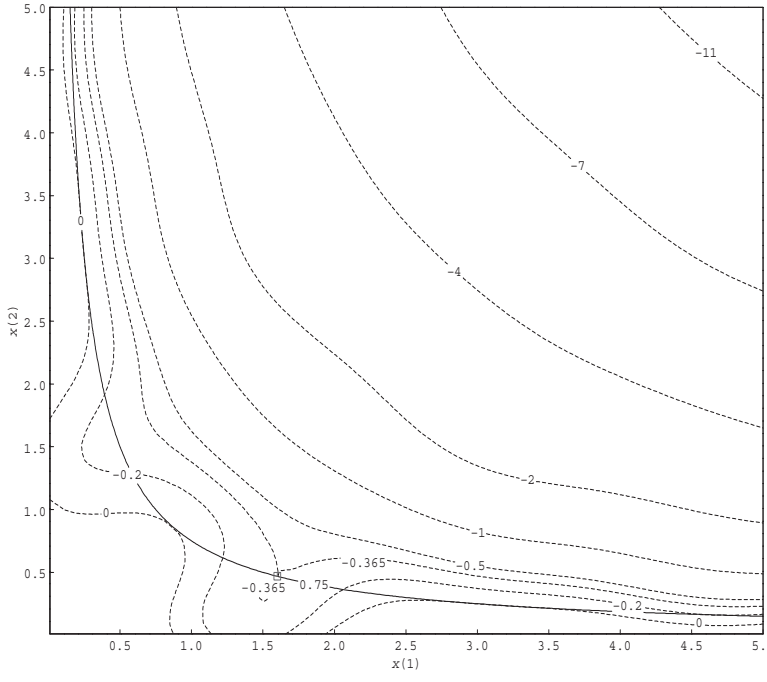


Figure 3.17 Contour lines of the Lagrangian (dotted) for $\lambda_1 = 0.5332$ and the equality constraint $x_1 x_2 = 0.75$. The square marker indicates the position of the constrained local minimum.

3.3.3 Feasible Directions Method

It will be clear that methods based on Lagrangian multipliers are designed to explore along active constraint boundaries. By contrast, the feasible directions method (Zoutendijk 1960) aims to stay as far away as possible from the boundaries formed by constraints. Clearly, this approach is only applicable to inequality constraints, but as has already been noted, these are usually in the majority in design work. The basic philosophy is to start at an active boundary and follow a "best" direction into the feasible region of the search space. There are two criteria used to select the best feasible direction. First is the obvious desire to improve the objective function as rapidly as possible, that is, to follow steepest descent downhill. This is, however, balanced with a desire to keep as far away as possible from the constraint boundaries. To do this, the quantity β is maximized subject to $-\mathbf{s}^{\mathrm{T}}\mathrm{D}g_i(\{\mathbf{x}\}) + \theta_i\beta \leq 0$ for each active constraint, $\mathbf{s}^{\mathrm{T}}\mathrm{D}f(\mathbf{x}) + \beta \leq 0$ and $|s_i| \leq 1$, where $\mathbf{s}$ is the vector that defines the direction to be taken and the θ_i are positive so-called push-off factors. The push-off factors determine how far the search will move away from the constraint boundaries. Again, these are problem dependent and are generally larger the more highly nonlinear the constraint is. The key to this approach is that the maximization problem thus defined is linear and so can be solved using linear programming methods which are extremely rapid and robust.

Once the search direction $\mathbf{s}$ has been fixed, we carry out a one-dimensional search in this direction either until a minimum has been found or a constraint has been violated. In the first case, we then revert to an unconstrained search as we are in the interior of the

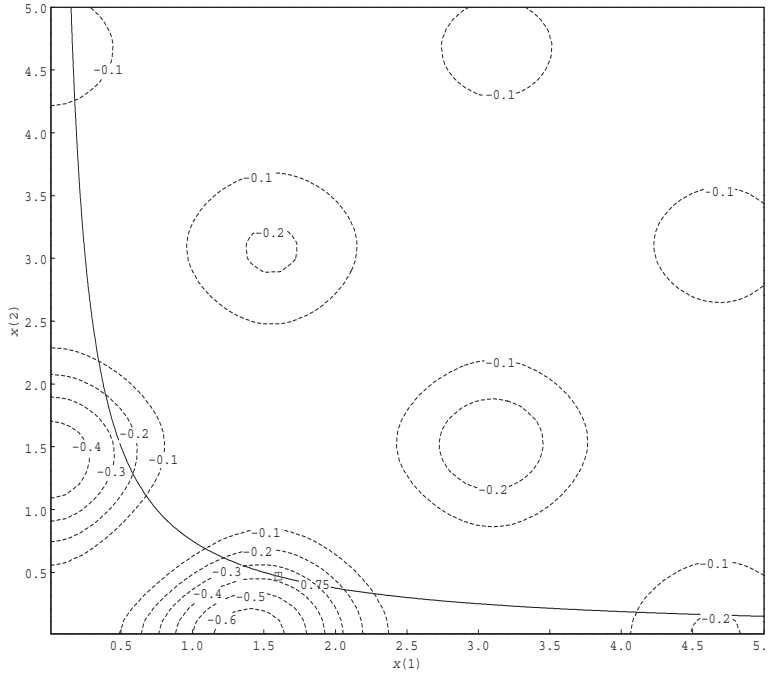


Figure 3.18 Contour lines of the original function and the constraint $x_1 x_2 = 0.75$. The square marker indicates the position of the constrained local minimum.

feasible region. In the second, we repeat the calculation of the feasible direction and iterate the procedure. This basic approach is used in the popular Conmin program.[4] In its basic form, it can become inefficient when searching into a shallow bowl on a constraint surface, since moving away from the boundary is then counterproductive.

3.3.4 Penalty Function Methods

It will be clear by now that there are no universally efficient and robust schemes for dealing with constraints in design search problems. Perhaps, not surprisingly, a large number of essentially heuristic approaches have therefore been developed that can be called upon depending on the case in hand. One such class revolves around the concept of penalty functions, which are somewhat related to Lagrangian multipliers. The simplest pure penalty approach that can be applied, the so-called one pass external function, is to just add a very large constant to the objective function value whenever any constraint is violated (or if we are maximizing to just subtract this): that is, $f_p(\mathbf{x}) = f(\mathbf{x}) + P$ if any constraint is violated, otherwise $f_p(\mathbf{x}) = f(\mathbf{x})$. Then, the penalized function is searched instead of the original objective. Provided the penalty added (P) is very much larger than the function being dealt with, this will create a severe cliff in the objective function landscape that will tend to make search methods reject infeasible designs. This approach is "external" because it is only applied in the infeasible regions of the search space and "one pass" because it is

[4] http://www.vrand.com

immediately severe enough to ensure rejection of any infeasible designs. Although simple to apply, the approach suffers from a number of drawbacks:

1. The slope of the objective function surface will not in general point toward feasible space in the infeasible region so that if the search starts from, or falls into, the infeasible region it will be unlikely to recover from this.

2. There is a severe discontinuity in the shape of the penalized objective at the constraint boundary and so any final design that lies on the boundary is very hard to converge to with precision, especially using efficient gradient-descent methods such as those already described (and, commonly, many optima in design are defined by constraint boundaries).

3. It takes no account of the number of constraint violations at any infeasible point.

Because of these limitations, a number of modifications have been proposed. First, a separate penalty may be applied for each violated constraint. Additionally, the penalty may be multiplied by the degree of violation of the constraint, and thirdly, some modification of the penalty may be made in the feasible region of the search near the boundary (the so-called interior space). None of these changes is as simple as might at first be supposed.

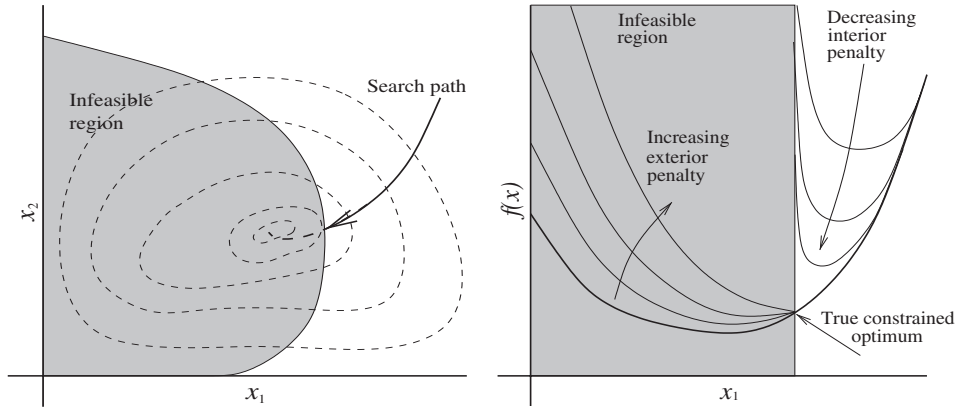
Consider first adding a penalty for each violated constraint: $f_p(\mathbf{x}) = f(\mathbf{x}) + mP$, where m is the number of violated constraints – this has the benefit of making the penalty more severe when multiple constraints are violated. Care must be taken, however, to ensure that the combined effect does not cause machine overflow. More importantly, it will be clear that this approach adds more cliffs to the landscape – now, there are cliffs along each constraint boundary so that the infeasible region may be riddled with them. The more discontinuities present in the objective function space, the harder it is to search, particularly with gradient-based methods.

Scaling the total penalty by multiplying by the degree of infeasibility is another possibility but can again lead to machine overflow. Now, $f_p(\mathbf{x}) = f(\mathbf{x}) + \sum P\langle|g_i(\mathbf{x})|\rangle + \sum P\langle|h_j(\mathbf{x})|\rangle$, where the angle brackets $\langle.\rangle$ are taken to be zero if the constraint is satisfied but return the argument value otherwise. In addition, if the desire is to cause the objective function surface to point back toward feasibility, then knowledge is required of how rapidly the constraint function varies in the infeasible region as compared to the objective function. If multiple infeasible constraints are to be dealt with in this fashion, they will need normalizing together so that their relative scales are appropriate. Consider dealing with a stress infeasibility in Pascals and a weight limit in metric tonnes – such elements will commonly be six orders of magnitude different before scaling. If individual constraint scaling is to be carried out, we need a separate P_i and P_j for each inequality and equality constraint, respectively: $f_p(\mathbf{x}) = f(\mathbf{x}) + \sum P_i\langle|g_i(\mathbf{x})|\rangle + \sum P_j\langle|h_j(\mathbf{x})|\rangle$. Finding appropriate values for all these penalties requires knowledge of the problem being dealt with, which may not be immediately obvious – in such cases, much trial and error may be needed before appropriate values are found.

Providing an interior component for a penalty function is even harder. The aim of such a function is, in some sense, to "warn" the search engine of an approaching constraint boundary so that action can be taken before the search stumbles over the cliff. Typically, this requires yet a further set of scaled penalties S_i, so that $f_p(\mathbf{x}) = f(\mathbf{x}) + \sum P_i\langle|g_i(\mathbf{x})|\rangle + \sum P_j\langle|h_j(\mathbf{x})|\rangle + \sum S_i/g_i^s(\mathbf{x})$, where the superscript s indicates a satisfied inequality con-

straint. Since the interior inequality constraint penalty goes to infinity at the boundary (where $g_i(\mathbf{x})$ goes to zero) and decreases as the constraint is increasingly satisfied (positive), this provides a shoulder on the feasible side of the function. However, in common with all interior penalties, this potentially changes the location of the true objective away from the boundary into the feasible region. Now, this may be desirable in design contexts, where a design that is on the brink of violating a constraint is normally highly undesirable, but again it introduces another complexity.

Thus far, all of the penalties mentioned have been defined only in terms of the design variables in use. It is also possible to make any penalty vary depending on the stage of the search, the idea being that penalties may usefully be weakened at the beginning of an optimization run when the search engine is exploring the landscape, provided that suitably severe penalties are applied at the end before the final optimum is returned. This leads to the idea of Sequential Unconstrained Minimization Techniques (SUMT), where a series of searches is carried out with ever more severe penalties, that is, the P_i, P_j and S_i all become functions of the search progress. Typically, the P values start small and increase while the S values start large and tend to zero. To see this, consider Figure 3.19 (after Siddall (1982) Fig. 6.27), where a single inequality constraint is shown in a two-dimensional problem and the final optimal design is defined by the constraint location. It is clear from the cross section that the penalized objective evolves as the constraints change so that initially a search approaching the boundary from either side sees only a gentle distortion to the search surface but finally this ends up looking identical to the cliff of a simple one pass exterior penalty. A number of variants on these themes have been proposed, and they are more fully explained by Siddall (1982) but it remains the case that the best choice will be problem specific and often therefore a matter of taste and experience.



(a) Contour map

(b) Cross section at fixed x_2 showing evolving penalized function

Figure 3.19 Evolution of a penalty function in a Sequential Unconstrained Minimization Technique (SUMT).

3.3.5 Combined Lagrangian and Penalty Function Methods

Since many penalty functions introduce discontinuities which make search with efficient gradient descent methods difficult, a potentially attractive approach is to combine a penalty with the method of Lagrange multipliers. Again, we consider just the case of active constraints (i.e., equalities). Now, our modified function becomes $f(\mathbf{x}) - \sum \lambda_j h_j(\mathbf{x}) + c \sum h_j(\mathbf{x})^2$, where c is some large, but not infinite penalty – this model is therefore amenable to search by gradient-based methods provided the original objective and constraint functions are themselves smooth, that is, we have the smoothness of the Lagrangian approach combined with the applicability of unconstrained search tools of simple penalty methods. To proceed, we minimize this augmented function with arbitrary initial multipliers but then update the multipliers as $\lambda_j^{k+1} = \lambda_j^k - 2c_k h_j(\mathbf{x})^k$, where the superscript or subscript k indicates the major iteration count. Moreover, we also double the penalty c_k at every major iteration, although, it in fact only needs to be large enough to ensure that the Lagrangian has a minimum at the desired solution rather than a stationary point (N.B., if c_k becomes too large, the problem may become ill-conditioned). If this process is started sufficiently close to an optimum with sufficiently large initial penalty, a sequence of searches rapidly converges to the desired minimum using only unconstrained (potentially gradient based) methods. Now, while this still requires some experimentation, it can be simply used alongside any of the unconstrained searches already outlined. Table 3.6 shows the results achieved if we follow this approach on our bumpy test function with a Polak–Ribiere conjugate gradient method.

3.3.6 Sequential Quadratic Programming

As has already been noted, the minimum of our constrained function does not lie at a minimum of the Lagrangian, only at a stationary point where its Hessian is positive definite. The most obvious approach to exploiting this fact is simply to apply Newton's method to

Table 3.6 Results of a search with combined Lagrangian multipliers and penalty function on the bump function of (3.2)

Major Iteration (k)	x_1	x_2	$h(x)$
1	2.0	1.0	1.25
2	1.599091	0.4656781	−0.0053383
3	1.600862	0.4685018	0.0000067
4	1.600861	0.4684972	−0.0000011
5	1.600861	0.4684978	−0.0000001

Major Iteration (k)	λ	c	**Penalized Lagrangian**	$f(x)$
1	0.0	50	78.11924	0.005757
2	0.00028665	100	−0.36640314	0.36782957
3	0.5341084	200	−0.36497975	0.36497616
4	0.5327684	400	−0.36497975	0.36498033
5	0.5332084		−0.36497975	0.36497982

search for the appropriate solution to the equations of the multipliers and gradients, but this fails if the search is started too far from the optimum being sought. The minimum does, however, lie at a minimum of the Lagrangian in the subspace of vectors orthogonal to the gradients of the active constraints. This result is the basis for a class of methods known as Sequential Quadratic Programming (SQP) (Boggs and Tolle 1995). In SQP, a quadratic (convex) approximation to the Lagrangian with linear constraints is used, in some form, in an iterative search for the multiplier and variable values that satisfy the KKT conditions. At each step of this search, the quadratic optimization subproblem is efficiently solved using methods like those already discussed (typically quasi-Newton) and the values of the xs and λs updated until the correct solution is obtained. Ongoing research has provided many sophisticated codes in this class. These are all somewhat complex, and so a detailed exposition of their workings lies outside the scope of this text. Nonetheless, it will be clear that such methods have to allow for the fact that as the design variables change it may well be the case that the list of active inequality constraints also changes, so that some of λs then need setting to zero while others are made nonzero. SQP methods also assume that the functions and constraints being searched have continuous values and derivatives, and so on, and so will not work well when there are function discontinuities – a situation that is encountered all too often in design. Nonetheless, they remain among the most powerful tools for solving local constrained optimization problems that are continuous; see for example Lawrence and Tits (2001).

3.3.7 Chromosome Repair

So far, all of the constraint-handling mechanisms that have been described have been designed with traditional sequential optimizers in mind. Moreover, all have aimed at transforming the original objective function, either by eliminating constraints and thus changing the input vector to the objective calculation or by adding some terms to the function actually being searched. It is, however, possible to tackle constraints via the mechanism of repair – a solution is said to be repaired when the resulting objective function value is replaced by that of a nearby feasible solution. In the context of evolutionary computing, where such mechanisms are popular and there are some biological analogies, this is commonly referred to as "chromosome repair" since it is the chromosome that encodes the infeasible point that is being repaired. Of course, repairing a design can be a difficult and costly business and one that involves searching for a feasible (if not optimal) solution. The methods used to do this are essentially exactly the same as in any other search process except that now the objective is the feasibility of the solution rather than its quality and such a search is, by definition, not constrained further. Here, we assume that some mechanism is available to make the repair and restrict our discussion to what a search algorithm might do with a repaired solution. This depends very much on how the search itself works.

After repair, we have a modified objective function value and also the coordinates of the nearby feasible location where this applies. The simplest thing to do with this modified function is to use it to replace the original objective and continue searching. It will be immediately clear, however, that as one traverses down a vector normal to a constraint boundary the most likely neighboring repair location will not change and so the updated objective function would then become constant in some sense, that is, the objective function surface would be flat in the infeasible region and the search would have no way of knowing that a constraint had been violated. Now, this may not matter if a stochastic or population-based

method is being used, since the next move will not be looking at the local gradient. Conversely, any downhill method is likely to become confused in such circumstances. In the evolutionary search literature, such an approach is termed *Baldwinian learning* (Baldwin 1896; Hinton and Nowlan 1987).

The alternative approach is to use both the function and variable values of the nearby feasible solution. This modifies the current design vector and so is not suitable for use in any sequential search process where the search is trying to unfold some strategy that guides the sequence of moves being taken: it would amount to having two people trying to steer a vehicle at the same time. This does not matter, however, in evolutionary or stochastic methods where the choice of the next design vector is anyway subject to random moves – the repair process can be seen as merely adding a further mutation to the design over and above that caused by the search algorithm itself. In the evolutionary search literature, this is termed *Lamarckian learning* after the early naturalist, Jean-Baptiste Lamarck (Gould 1980).

Since Baldwinian learning can confuse a gradient-based search and Lamarckian learning is incompatible with any form of directed search, it is no surprise that neither method finds much popularity in sequential search applications. When it comes to evolutionary search, they are more useful but still not simple to implement. Moreover, since the repair process may itself be expensive, most workers tend to prefer the Lamarckian approach, which keeps hold of all the information gleaned in making the repair. There is some evidence to show that such a process helps evolutionary methods when applied to any local improvement process (repair is just one of many ways that a design might be locally improved during an evolutionary search) (Ong and Keane 2004). Even so, the evidence is thus far not compelling enough to make the idea widely popular.

3.4 Metamodels and Response Surface Methods

One of the key factors in any optimization process is the cost of evaluating the objective function(s) and any constraints. In test work dealing with simple mathematical expressions this is not an issue, but in design activities, the function evaluation costs may well be measured in hours or even days and so steps must always be taken to minimize the number of function calls being used. Ideally, any mitigating strategies should work *irrespective* of the dimensions of the problem. One very important way of trying to do this is via the concept of response surface, surrogate or metamodels. In these, a small number of full function (or constraint) evaluations are used to seed a database/curve-fitting process, which is then interrogated in lieu of the full function whenever further information about the nature of the problem is needed by the optimization process.

At its simplest, such metamodeling is just the use of linear regression through existing calculations when results are needed for which the full simulations have not been run. Of course, it is unlikely that the original set of calculations will be sufficiently close to the desired optimal designs by chance, especially when working in many dimensions, so some kind of updating or refinement scheme is commonly used with such models. Updating enables the database and curve fit to be improved in an iterative way by selective addition of further full calculations. Figure 3.20 illustrates such a scheme. In this section, we introduce the basic ideas behind this way of working since they are key to many practical design

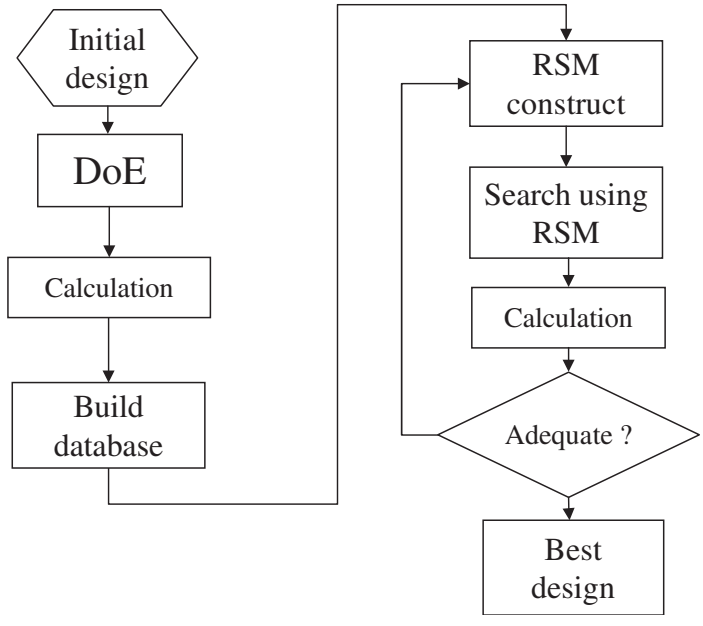


Figure 3.20 Simple serial response surface method with updating.

approaches – a much more extensive treatment is provided in Chapters 5, 6 and 7. We would note in passing that the idea of slowly building an internal model of the unknown function is similar in spirit to the gradient-based methods that build local approximations to the Hessian of a function – the main difference here is the explicit treatment of the model and the fact that in many cases they are built to span much wider regions of the search space.

There are a significant number of important decisions that a user must make when following a metamodel-based search procedure:

1. How should the initial database be seeded with design calculations – will this be a regular grid, random or should some more formal DoE approach be used?
2. Should the initial calculations be restricted to some local region of the search space surrounding the current best design or should they attempt to span a much larger range of possible configurations?
3. Can the initial set of calculations be run in parallel?
4. What kind of curve fit, metamodel or response surface should be used to represent the data found by expensive calculation?
5. What will we do if some of the calculations fail, either because of computing issues or because the underlying PDE cannot be solved for certain combinations of parameters?

6. Is it possible to obtain gradient information directly at reasonable cost via use of perturbation techniques or adjoint solvers and, if so, will these be used in building the metamodel?
7. Is one model enough or should multiple models be built with one for each function or constraint being dealt with?
8. If multiple models are to be built can they be correlated in some sense?
9. Should the model(s) strictly interpolate the data or is regression better?
10. Will the model(s) need some extensive tuning or training process to get the best from the data?
11. What methods should be used to search the resulting surface(s)?
12. Having searched the metamodel, should a fresh full calculation be performed and the database and curve fit updated?
13. If updates are to be performed, should any search on the surface(s) aim simply to find the best predicted design or should some allowance be made for improving the coverage of the database to reduce modeling errors, and so on?
14. If updates are to be performed, could multiple updates be run in parallel and, if so, how should these multiple points be selected?
15. If an update scheme is to be used, what criteria will be used to terminate the process?

Answering all these questions for any particular design problem so as to provide a robust and efficient search process requires considerable experience and insight.

3.4.1 Global versus Local Metamodels

At the time of writing, there are two very popular general metamodeling approaches. The first is to begin with a formal DoE spanning the whole search space, carry out these runs in parallel and then to build and tune a global response surface model using radial basis function or Kriging methods. This model is then updated at the most promising locations found by extensive searches on the metamodel, often using some formal criterion such as expectation of improvement to balance improvements and model uncertainty – this approach is illustrated in Figure 3.21 (Jones et al. 1998).

In the second approach, a local model is constructed around the current design location, probably using an orthogonal array or part-factorial design. The results are then fitted with a simple second-order (or even first-order) regression equation. This is searched to move in a downhill direction up to some amount determined by a trust region limit. The model is then updated with additional full calculations, discarding those results that lie furthest from the current location. The process is repeated until it converges on an optimum (Alexandrov et al. 1998b; Conn et al. 2000; Toropov et al. 1999). Such an approach can be provably convergent, at least to a local optimum of the full function, something which is not possible with global models.

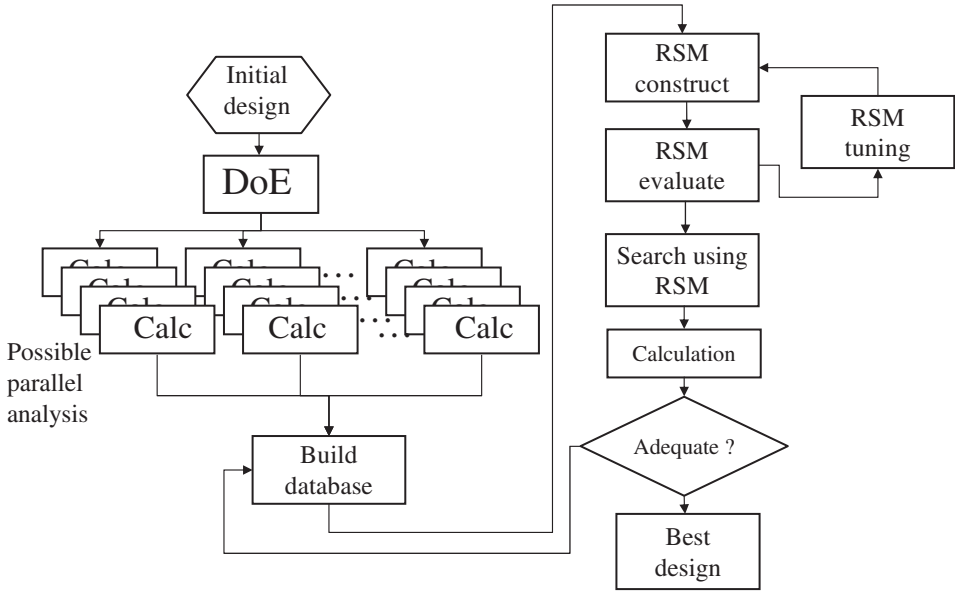


Figure 3.21 Improved response surface method calculation, allowing for parallel DoE-based calculations, RSM tuning and database update.

The choice between these two types of strategy depends largely on the goals of the designer. If an existing design is to be exploited and tuned to yield maximum performance without radical redesign, the trust region approach has much to commend it. Conversely, if the team is unsure where they might go and radical alternatives are required, the global strategy can be very effective. Moreover, as problem dimensions rise, the ability to construct any kind of accurate surrogate rapidly diminishes – one need only consider how the number of vertices of the search space bounding box rises with the power of the number of dimensions to realize just how little data a few hundred points represent when searching even a ten-dimensional search space.

3.4.2 Metamodeling Tools

Assuming that a surrogate approach is to be used, we next set out some of the most popular tools needed to construct a good-quality search process. We begin by noting that there are very many DoE methods available in the literature and the suitability of any particular one depends on both the problem to be modeled and also the class of curve fit to be used. Among the most popular are

1. pure random sequences (completely unbiased but fail to exploit the fact that we expect limited curvature in response surfaces, since we are commonly modeling physical systems);
2. full factorial designs (usually far too expensive in more than a few dimensions, also fail to study the interior of the design space well as they tend to focus on the extremes);

3. face centered cubic or central composite designs (cover more of the interior than full factorial but are even more expensive);
4. part-factorial designs (allow for limited computational budget but again do not cover the design space well);
5. orthogonal array designs, including Plackett and Burman, Box-Benken and Taguchi designs (reasonable coverage but designs tend to fix the number of trials to be used and also each variable is often restricted in the number of values it can take);
6. Sobol sequence and LPτ designs (allow arbitrary and extensible designs but have some limitations in coverage); and
7. Latin hypercubes of various kinds (not easily extensible but easy to design with good coverage over arbitrary dimensions).

Having run the DoE calculations, almost always in parallel, a curve fit of some kind must next be applied. In the majority of cases, the data available is not spread out on some regular grid and so the procedure used must allow for arbitrary point spacing. Those currently popular in the literature include

1. linear, quadratic and higher-order forms of polynomial regression;
2. cubic (or other forms of) splines, commonly assembled as a patchwork that spans the design space and having some form of smoothness between patches;
3. Shepard interpolating functions;
4. radial basis function models with various kernels (which are most commonly used to interpolate the data, but which can be set up to carry out regression in various ways and also can have secondary tuning parameters);
5. Kriging and its derivatives (which generally need the tuning of multiple hyperparameters that control curvature and possibly the degree of regression – this can be very expensive on large data sets in many dimensions);
6. support vector machines (which allow a number of differing models to be combined with key subsets of the data – the so-called support vectors – so that problems with many dimensions and large amounts of data can be handled with reasonable efficiency);
7. neural networks (which have a long history in function matching and prediction but which need extensive training and validation and which are also generally difficult to interpret); and
8. genetic programming (which can be used to design arbitrary functional forms to approximate given data, but which are again expensive to tune and validate).

Constructing such models always requires some computational effort and several of them require that the curve fit be tuned to match the data. This involves selecting various control or hyperparameters that govern things like the degree of regression and the amount of curvature. This can represent a real bottleneck in the search process since this is difficult to parallelize and can involve significant computing resources as the number of data points

and dimensions rises – this is why the use of support vector approaches is of interest since they offer the prospect of affordable training on large data sets.

Typically, tuning (training) involves the use of a metric that measures the goodness of fit of the model combined with its ability to predict results at points where no data is available. For example, in Kriging, the so-called likelihood is maximized, while in neural networks it is common to reserve part of the data outside of the main training process to provide a validation set. Perhaps the most generally applicable metric is leave-out-one cross-validation, where a single element is removed from the training set, and its results predicted from the rest with the current hyperparameters. This is then repeated for all elements in the data and the average or variance used to provide a measure of quality. If this is used during training, it can, however, add to the computational expense.

Once an acceptable model is built, it should normally be refined by adding further expensive results in key locations. There is no hard and fast rule as to how much effort should be expended on the initial data set as compared to that on updating but, in the authors experience, a 50:50 split is often a safe basis to work on, although reserving as little as 5% of the available budget for update computations can sometimes work while there are other cases where using more than 75% of the effort on updating yields the best results. There are two main issues that bear on this choice: first, is the underlying function highly multimodal or not? – if it is, then more effort should go into updating; secondly, can updates be done in parallel or only serially? – if updates are only added one at a time while the initial DoE was done in parallel, then few updates will be possible. Interestingly, this latter point is more likely to be driven by the metric being used to judge the metamodel than the available hardware – to update in parallel, one needs to be able to identify groups of points that all merit further investigation at the same time – if the metric in use only offers a simplistic return, it may be difficult to decide where to add the extra points efficiently. This is where metrics such as expected improvement can help as they tend by their nature to be multimodal and so a suitable multistart hill climbing strategy can then often identify tens of update points for subsequent calculation from a single metamodel. The authors find Yuret's Dynamic Hill Climbing method ideal in this role (Yuret and de la Maza 1993), as are population-based searches which have niche forming capabilities (Goldberg 1989).

3.4.3 Simple RSM Examples

Figures 3.22, 3.23 and 3.24 show the evolution of a Kriging model on the bump problem already discussed as the number of data points is increased and the metamodel rebuilt and retuned. In this example, 100 points are used in the initial search and three sets of 10 updates are added to refine the surface. These are simply aimed at finding the best point in the search space and it can be seen that the area containing the global optimum is much better defined in the final result. Note that the area containing the optimum has to be in the original sample for the update policy to refine it – as this cannot usually be guaranteed, it is useful to include some kind of error metric in the update so as to continue sampling regions where little data is available, if such a strongly multimodal problem is to be dealt with in this way.

Figure 3.25 shows a trust region search carried on the bump function. In this search, a small local DoE of nine points is placed in the square region between (2,1) and (4,3) and then used to construct a linear plus squared term regression model. This is then searched within a unit square trust region based around the best point in the DoE. The best result

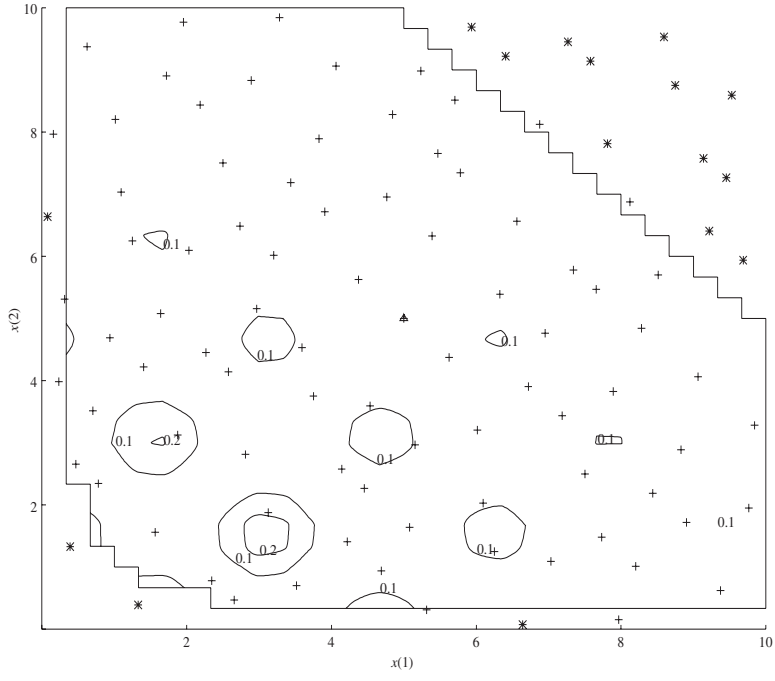


Figure 3.22 Bump function showing location of initial DoE of 100 LPτ points (points marked with an "*" are infeasible).

found in the search is then reevaluated using the true function and used to replace the oldest point in the DoE. This update process is carried out 15 times with a slowly decreasing trust region size, each time centered on the endpoint of the last search and, as can be seen from the figure, this converges to the nearby local optimum, using 24 $(= 9 + 15)$ calls of the expensive function in all. Obviously, such an approach can become very confused if the trust region used encompasses many local optima and the search will then meander around aimlessly until the trust region size aligns with the local basin of attraction size. In many cases, particularly where there is little multimodality but significant noise, this proves to be a very effective means of proceeding.

3.5 Combined Approaches – Hybrid Searches, Metaheuristics

Given all the preceding ideas on how to perform optimization, it will be no surprise that many researchers have tried to combine the best aspects of these various methods into ever more powerful search engines. Before discussing such combined or "hybrid" searches, it is worth pointing out the "no free lunch" theorems of search (Wolpert and Macready 1997). These theorems essentially state that if a search method is improved for any one class of problem it will equivalently perform worse on some other class of problem and that, averaged over all classes of problems, it is impossible to outperform a random walk. Now,

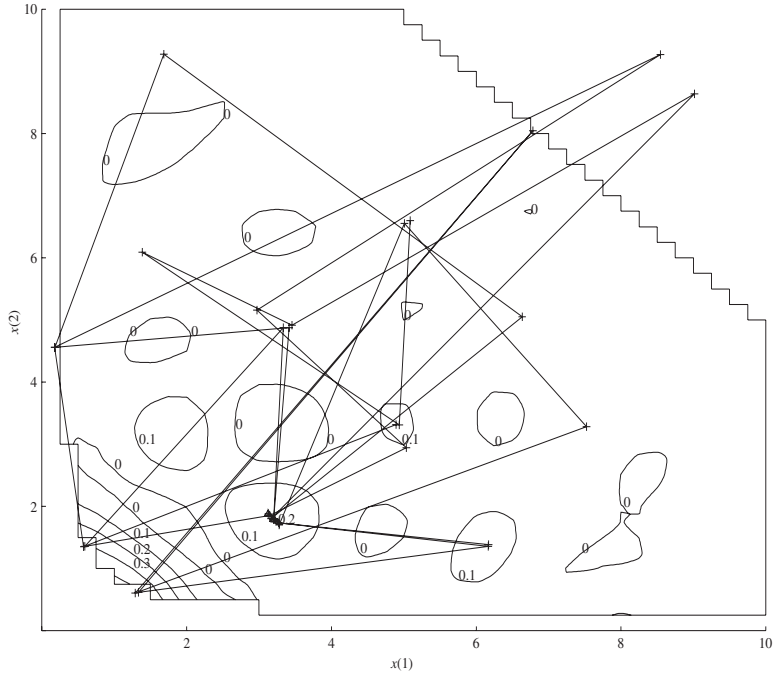


Figure 3.23 RSM of bump function showing search traces for three searches using Dynamic Hill Climbing (DHC) with parallel updates between them (straight lines between crosses link the minima located by each major iteration of the DHC search).

although this result might seem to make pointless much of what has gone before, it should be recalled that the problems being tackled in aerospace design all tend to share some common features and it is the exploitation of this key fact that allows systematic search to be performed. We know, for example, that small changes in the parameters of a design, in general, do not lead to random and violent changes in performance, i.e., that the models we use in design exhibit certain properties of smoothness and consistency, at least in broad terms. It is these features that systematic search exploit. Nonetheless, the no free lunch idea should caution us against any idea that we can make a perfect search engine – all we can in fact do is use our knowledge of the problem in hand to better adapt or tune our search to a particular problem, accepting that this will inevitably make the search less general purpose in nature, or, conversely, that by attempting to make a search treat a wider range of problems it will tend to work less well on any one task.

Despite this fundamental limit to what may be achieved, researchers continue to build interesting new methods and to try and ascertain what problems they may tackle well. Hybrid searches fit naturally into this approach to search. If one knows that certain features of a gradient-based method work well, but that its limitations for global search are an issue, it is natural to try and combine such methods with a global method such as a genetic algorithm. This leads to an approach currently popular in the evolutionary search community. Since evolutionary methods typically are built around a number of operators that are based on natural selection (such as mutation, crossover, selection and inversion), it is relatively easy

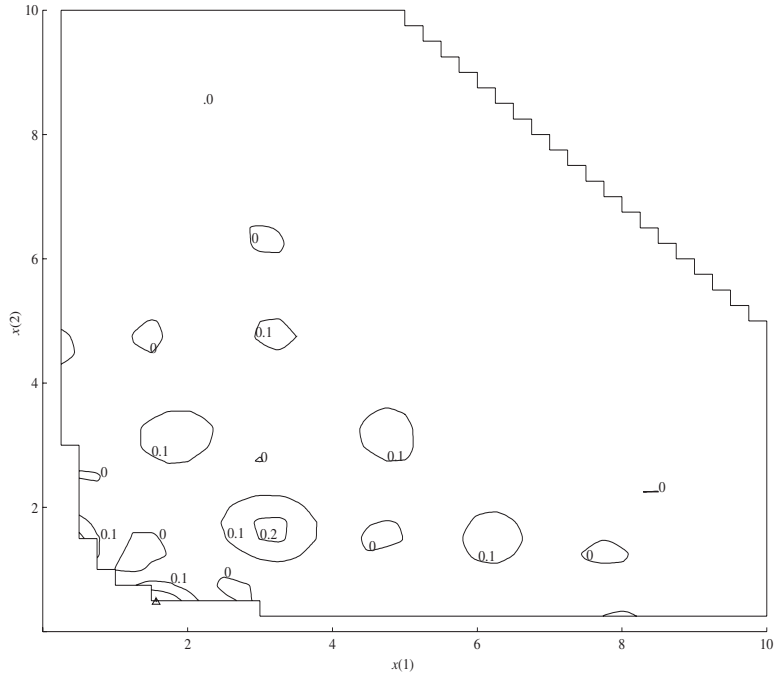


Figure 3.24 Final updated RSM of bump function after 30 updates – the best result found is marked by a triangle at (1.5659, 0.4792) where the estimated function value is 0.3602 and the true value is 0.3633, compare the exact solution of 0.3650 at (1.6009, 0.4685).

to add to this mix, ideas coming from the classical search literature such as gradient-based approaches. In such schemes, some (possibly all) members of a population are selected for improvement using, for example, a quasi-Newton method (which might possibly not be fully converged) and any gains used as another operator in the search mixture.

If another search is added into the inner workings of a main search engine, we think of the result as a hybrid search. Alternatively, we might simply choose to interleave, possibly partially converged, searches with one another by having some kind of control script that the combined process follows. This we would term a metaheuristic, especially if the logic followed makes decisions based on how the process has proceeded thus far. Of course, there are no real hard and fast boundaries between these two logics – the main difference is really in how they appear to the user – hybrid searches usually come boxed up and appear as a new search type to the designer while metaheuristics tend to appear as a scripting process where the user is fully exposed to, and can alter, the logic at play. In fact, the response surface methods already set out may be regarded as a special case of metaheuristics, spanning DoE-based search and whatever method is used to search the resulting metamodel.

To illustrate these ideas, we describe two approaches familiar to the authors – we would stress, however, that it is extremely straightforward to create new combinations of methods and the no free lunch theorems do not allow one to say if any one is better than another

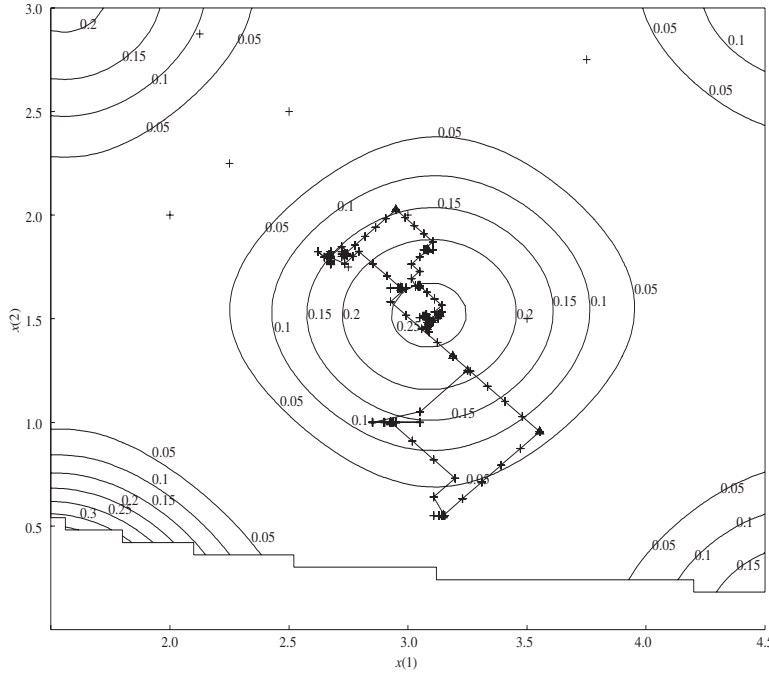


Figure 3.25 Trust region search with linear plus squared term regression model taken over nine points updated 15 times.

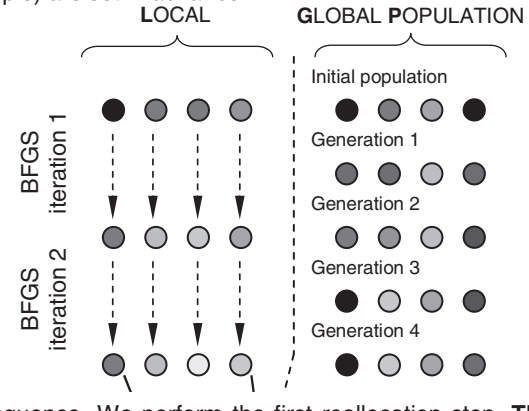
without first stating the problem class being tackled – here, we describe these combinations just to give a taste of what may be done. We would further note that combinations of optimizers can also be effective when dealing with multidisciplinary problems, a topic we return to in Chapter 9.

3.5.1 Glossy – A Hybrid Search Template

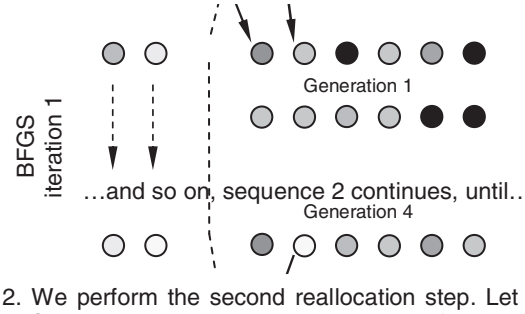
Glossy is a hybrid search template that allows for search time division between exploration and exploitation (Sóbester and Keane 2002). The basic idea is illustrated in Figure 3.26 – an initial population is divided between a global optimizer and a local one. Typically, the global method is a genetic algorithm and the local method is a gradient-based search such as the BFGS quasi-Newton method. The GA subpopulation is then iterated in the normal way for a set number of generations. At the same time, each individual in the BFGS population is improved using a traditional BFGS algorithm but only for as many downhill steps as the number of generations used in the GA. It is further assumed that each downhill search and each member of the GA is evaluated on its own dedicated processor so that all these process run concurrently and finish at the same time. Then, an exchange is carried out between the two subpopulations and at the same time the relative sizes of the subpopulations are updated. The searches are then started again and this procedure cycled

GLOSSY (GA-BFGS) We start with a **randomly generated population** (each circle represents one individual, the color of the circle indicates the objective value of the individual – the lighter the color, the better the objective value)…

…and **allocate the individuals** into two populations where they will undergo a sequence of local exploitation and a sequence of global exploration respectively. The population sizes (4 and 4 in this case) and the sequence lengths (*SL*=2 and *SG*=4 in our example) are set in advance.



End of the first sequence. We perform the first reallocation step. **The populations will be resized according to the average objective value improvement per evaluation achieved during the last sequence.**
In this case the improvement calculated for the GA, was, say, three times higher than that achieved by BFGS. Thus, the ratio of the sizes of the two populations for the next sequence will be three. Therefore, the **L**ocal **P**opulation has to relinquish two individuals to the global one. Those two that **have achieved the least improvement** during the last sequence (individuals 1 and 4 in this case) will migrate. Those that have improved well locally are allowed to continue in the local population.



End of sequence 2. We perform the second reallocation step. Let us suppose that the efficiency of the GA has diminished slightly, so the performance ratio (and thus the population size ratio for the next sequence) is 5/3. Therefore, the **G**lobal **P**opulation now has to concede one individual to the global one. The individual with **the highest objective value** will migrate (the rationale being that the best individuals are likely to be in promising basins of attraction, i.e., they are worth improving locally)

…and sequence 3 begins.

Figure 3.26 The Glossy template (from Sóbester and Keane; reprinted by permission of the American Institute of Aeronautics and Astronautics, Inc.).

through until a suitable termination criterion is met. This strategy allows the GA to carry out global search and the BFGS to locally improve some designs, which can then be fed back to the GA to seed further global searches. At the same time, other globally improved points are handed back for local BFGS refinement. The changing size of the subpopulations allows the search to adapt to the progress made, becoming either more global or more local as the situation demands. Note that since the locally improved designs are returned directly to the GA population this is termed *meta-Lamarckian learning* in some parts of the literature.

3.5.2 Metaheuristics – Search Workflows

As already noted, metaheuristics are typically sequences of commands that invoke various search methods in an adaptive way depending on the progress of the search. They are typically written or adapted by the designer to suit the problem in hand using a general purpose workflow/scripting language such as Matlab® or Python, or the language provided in a commercial search tool such as the multidisciplinary optimization language (MDOL) system in iSIGHT® or even using operating system shell scripts, and so on. The key distinction over hybrid searches being the great flexibility this affords and its control by the designers themselves rather than via reliance on a programming team, etc.

The Geodise[5] toolkit is an example of this way of working. Geodise provides a series of commands in the Matlab environment that may be combined together however the user chooses to create a search workflow. It offers a range of individual search engines, DoE methods and metamodeling capabilities that can be used in concert with those already provided by the main Matlab libraries (or, indeed, any other third party library). Figure 3.27 provides a snippet of Matlab code that invokes this toolkit to carry out a sequence of gradient-based searches on a quadratic metamodel constructed by repeated sampling of the problem space. In this way, a simplistic trust region search can be constructed and the code given here is, in fact, that used to provide Figure 3.25 in the previous section.

It will be clear from examining this example that there is no easy distinction between a workflow language and the direct programming of an optimizer – the characteristic we look for when making the division is the use of existing multidimensional searches as building blocks *by the end users themselves*. Of course, it is a relatively trivial matter to subsequently box up such metacode to form new black-box methods for others to use. In fact, it is often the case that the distinction between classifications says as much about the expertise of the user as it does about the code – a skilled user who is happy to open up and change a Fortran or C code library might justifiably claim that the library provided the ingredients of a metaheuristic approach, while those less skilled would view any combinations worked in such a way as hybrids. We do not wish to labor this point – just to make clear that it is often worthwhile combining the search ingredients to hand in new ways to exploit any known features of the current problem, and that this will be most easily done using a metaheuristic scheme. This approach underpins much of the authors' current work in developing searches for the new problems that they encounter.

[5] `http://www.geodise.org`

```
% trust region search
%
% DoE_struct holds information for the actual function
% meta_struct holds information for the meta-model
% DoE_rslt holds the results from calls to the actual function
% meta_rslt holds the result from searches on the meta-model
% update_pt is used to hold the update points as they are calculated
% trust_sc is the scaling factor for reducing the trust region on each loop
% trust_wd is the current semi-width of the trust region
%
% define initial structures to define problem to be searched
%
DoE_struct=createbumpstruct;
meta_struct=createbumpstruct;
%
% initial DoE details
%
DoE_struct.OMETHD=2.8; DoE_struct.OLEVEL=0;
DoE_struct.VARS=[2,2]; DoE_struct.LVARS=[2,1];
DoE_struct.UVARS=[4,3]; DoE_struct.MC_TYPE=2;
%
% build initial DoE
%
DoE_struct.NITERS=9;
DoE_rslt = OptionsMatlab(DoE_struct);
n_data_pts = DoE_rslt.OBJTRC.NCALLS;
%
% set up trust region controls and initial trust region meta model
%
trust_sc=0.9; trust_wd=0.5;
meta_struct.OLEVEL=0; meta_struct.VARS=DoE_rslt.VARS;
meta_struct.LVARS=[2,1]; meta_struct.UVARS=[4,3];
%
% loop through trust region update as many times as desired
%
for (ii = 1:15)
%
% build and search regression meta model from the current data set
%
meta_struct.OBJMOD=3.3; meta_struct.OMETHD=1.8;
meta_struct.OVR_MAND=1; meta_struct.NITERS=500;
meta_rslt=OptionsMatlab(meta_struct,DoE_rslt);
%
% update trust region around best point but of reduced size
%
trust_wd=trust_wd*trust_sc;
meta_struct.LVARS=meta_rslt.VARS-trust_wd;
```

Figure 3.27 Matlab workflow for a trust region search using the Geodise toolkit (http://www.geodise.org).

```
meta_struct.UVARS=meta_rslt.VARS+trust_wd;
%
% evaluate best point
%
DoE_struct.OMETHD=0.0;
DoE_struct.VARS=meta_rslt.VARS;
update_pt = OptionsMatlab(DoE_struct);
%
% add result to data set, eliminating the oldest point
%
DoE_rslt.OBJTRC.VARS(:,1)=[];
DoE_rslt.OBJTRC.VARS(:,n_data_pts)=update_pt.VARS;
DoE_rslt.OBJTRC.OBJFUN(1)=[];
DoE_rslt.OBJTRC.OBJFUN(n_data_pts)=update_pt.OBJFN;
%
%echo latest result and loop
%
update_pt.VARS update_pt.OBJFN
meta_struct.VARS=update_pt.VARS;
end
```

Figure 3.27 *Continued*

3.6 Multiobjective Optimization

Having set out a series of methods that may be used to identify single objectives, albeit ones limited by the action of constraints, we next turn to methods that directly address problems where there are multiple and conflicting goals. This is very common in design work, since engineering design almost always tensions cost against performance, even if there is only a single performance metric to be optimized. Usually, the design team will have a considerable number of goals and also some flexibility and/or uncertainty in how to interpret these.

Formally, when a search process has multiple conflicting goals, it is no longer possible to objectively define an optimal design without at the same time also specifying some form of weighting or preference scheme between them: recall that our fundamental definition of search requires the ability to rank competing designs. If we have two goals and some designs are better when measured against the first than the second, while others are the reverse, then it will not be possible to say which is best overall without making further choices. Sometimes, it is possible to directly specify a weighting (function) between the goals and so combine the figures of merit from the various objectives into a single number. When this is possible, our problem of course reduces directly to single value optimization and all the techniques already set out may be applied. Unfortunately, in many cases, such a weighting cannot be derived until such time as the best competing designs have been arrived at – this leads to the requirement for methods of identifying the possible candidates from which to select the so-called Pareto set, or front, already mentioned in earlier sections. We recall that this is the set of designs whose members are such that improving any one goal function of a member will cause some other goal function to deteriorate. However, before considering methods that work explicitly with multiple goals, we first consider formal schemes for identifying suitable weights that return our problem to that of single objective optimization.

We note also that, as when dealing with constraints, it is always wise to normalize all the goals being dealt with so that they have similar numerical magnitudes before starting any comparative work. This leads to better conditioned problems and prevents large numbers driving out good ones. We remember also that many design preferences are specified in the form of constraints, i.e., that characteristic A be greater than some limit or perhaps less than characteristic B, for example.

3.6.1 Multiobjective Weight Assignment Techniques

When a design team has multiple goals and the aim is to try and derive a single objective function that adequately represents all their intentions, the problem being tackled is often rather subjective. This is, however, an issue that is familiar to anyone who has ever purchased a house, an automobile, a boat, a computer, and so on: there will be a variety of different designs to choose from and a range of things that make up the ideal. These will often be in tension – the location of a house and its price are an obvious example – a property in a good location will command a high price. It is thus a very natural human activity to try and weigh up the pros and cons of any such purchase – we implicitly carry out some kind of balancing process. Moreover, our weightings often change as we learn more about the alternatives on offer. Multiobjective weight assignment techniques aim to provide a framework that renders this kind of process more repeatable and justifiable – they are, however, not amenable to very rigorous scientific analysis in the way that the equations of flow over a body are. Nonetheless, there is some evidence that a formalized approach leads to better designs (Sen and Yang 1998).

The basis of all weight assignment methods is an attempt to elicit information from the design team about their preferences. The most basic approach is direct weight assignment. In direct assignment, the team chooses a numerical scale, typically containing five or ten points, and then places adjective descriptions on this scale such as “extremely important”, “important”, “average”, “unimportant” and “extremely unimportant”. Each goal is then considered in turn against the adjectives and the appropriate weight chosen from the scale. This can be done by the team collectively, or each member asked to make a ranking and then average values taken. Often, the weighted goals, or perhaps their squares, are then simply added together to generate the required combined goal function. Although this may sound trivial, the fact that an explicit process is used tends to ensure that all team members views' are canvassed and that added thought is given to the ranking process. This approach tends to perform best when there are relatively few goals to be satisfied. It does not, however, give any guidance on what the function used to combine the individually weighted goals together should be, an issue we return to later on.

A slightly more complex way of eliciting and scoring preference information is the eigenvector method. In this approach, each pair of goals is ranked by stating a preference ratio, that is, if goal i is three times more important than goal j, we set the preference p_{ij} equal to three. Then, if goal j is twice as important as goal k, we get p_{jk} equal to two. Note that for consistency this would lead us to assume that p_{ik} was six, that is, three times two. However, if all pair-wise comparisons are made and there are many goals, such consistency is rarely achieved. In any case, having made all the pair-wise comparisons, a preference matrix $\mathbf{P}$ with a leading diagonal of ones can then be assembled. In the eigenvector scheme, we then seek the eigenvector $\mathbf{w}$ that satisfies $\mathbf{Pw} = \lambda_{\max}\mathbf{w}$, where $\lambda_{\max}$ is the maximum eigenvalue of the preference matrix. If the matrix is not completely consistent, it is still possible to seek a solution to this equation and also to gain a measure of any inconsistency

by comparing the eigenvalue to the number of goals, since, for a self consistent matrix, the largest eigenvalue is always equal to the number of goals; see, for example, Saaty (1988). The eigenvector resulting from solution of this equation provides the weighting we seek between the goals. By way of example, consider a case where there are three goals and we assign scores to the matrix **P** as follows:

$$\mathbf{P} = \begin{bmatrix} 1 & 3 & 6 \\ 1/3 & 1 & 2 \\ 1/6 & 1/2 & 1 \end{bmatrix}$$

that is, goal one is three times as important as goal two and goal two is twice as important as goal three. It is easy to show that the largest eigenvalue of this matrix is three and the equivalent eigenvector, which produces the desired weighting, is $\{0.667, 0.222, 0.111\}^{\mathrm{T}}$. The fact that the eigenvector is here equal to the number of goals indicates that the preference matrix is fully consistent. This approach works well for moderate numbers of goals (less than say 10) and copes well with moderate levels of inconsistency.

When the number of goals becomes large, the rapidly expanding number of pair-wise comparisons needed makes the eigenvector approach difficult to support. It is also difficult to maintain sufficient self consistency with very many pair-wise inputs. Consequently, a number of alternative schemes have been developed that permit incomplete information input and also iteration to improve consistency; see Sen and Yang (1998) for further details. There are also schemes that allow progressive articulation of preference information; see, for example, Benayoun et al. (1971).

3.6.2 Methods for Combining Goal Functions, Fuzzy Logic and Physical Programming

Even when the design team can assign suitable weights to each competing objective, this still leaves the issue of how the weighted terms should be combined. Most often, it is assumed that a simple sum is appropriate. However, there are many problems where such an approach would not lead to all possible Pareto designs being identified, even if every linear combination of weights were tried in turn. This is because, for some problems, the Pareto front is nonconvex; see, for example, Figure 3.28. The region in this figure between points A and B is such that, for any point on this part of the front, a simple weighted sum of the two goals can always be improved on by designs at either end *whatever* the chosen weights. To see this, we note that for any relative weighting between two design goals w all designs lying on the line of constant $wx + y$ in Figure 3.28 have the same combined objective when using a simple sum. For one particular design to be best for a given weight, the Pareto front at this point must be tangent to the equivalent weighting line – in the convex region, we cannot construct a tangent line that lies wholly to the left and below the Pareto front, that is, there is always some other design that, for that combination of weights, will lie outside of the convex region.

Since we know that all points on the Pareto front may be of interest to the designer (since each represents a solution that is nondominated), this fundamental limitation of a weighted sum of goal functions has led a number of researchers to consider more complex schemes. These range from the use of simple positive powers of each objective through to fuzzy logic and physical programming models. The basic idea is this: by combining our goals using some more complex function, the lines of constant combined objective in plots like

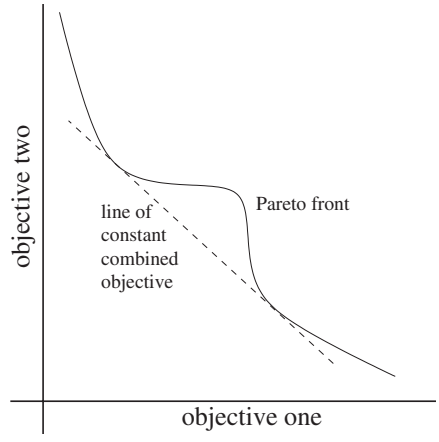


Figure 3.28 A Pareto front with a convex region and line indicating designs with equal combined objective under simple summation of weighted goals.

Figure 3.28 become curves. Provided the functions we use have sufficient curvature in this plot to touch the Pareto front tangentially at all points, given the correct choice of weights, the function in use can then overcome the problems of convexity. So, if we take the sum of squares for example, a contour of constant combined objective is then an ellipse – it will be obvious that such a shape will satisfy our tangency requirement for many convex Pareto fronts, but by no means all such possibilities. One can respond to this problem by using ever more complex functions, of course. However, they rapidly lose any physical meaning to the designer and this has led to the fuzzy logic and physical programming approaches, among others. In these, an attempt is made to capture the design team's subjective information and combine it in a nonlinear fashion *at the same time*, hopefully overcoming convexity while still retaining understanding of the formulation.

In the fuzzy logic approach (Ross 2004), a series of membership functions are generated that quantify the designer's linguistic expressions of preference. For example, if the designer believes that a goal must be at least five and preferably ten, but values over ten offer no further benefit, then a function such as that in Figure 3.29 might be used. Note that the shape of the function that translates the goal value to its utility is here piecewise linear but this is an arbitrary choice: a sigmoid function would just as well suffice. Similar functions are provided for all goals (and usually also all constraints), always mapping these into a scale from zero to one. The resulting utilities are then combined in some way. Typically, they might be summed and divided by the total number of functions, a product taken or simply the lowest overall used. In all cases, the resulting overall objective function still runs between zero and one and an optimization process can be used on the result. Note that this function will have many sharp ridges if piecewise linear membership functions are used or if a simple minimum selection process is used to combine them. Although it allows a more complex definition of the combined objective than simple weights, there is still no guarantee that all parts of the Pareto front will be reached by any subsequent search.

Physical programming (Messac 1996) is another method that seeks to use designer preferences to construct an aggregate goal function. It is again based on a series of linguistic statements that the design team may use to rank their preferences. Goals and constraints

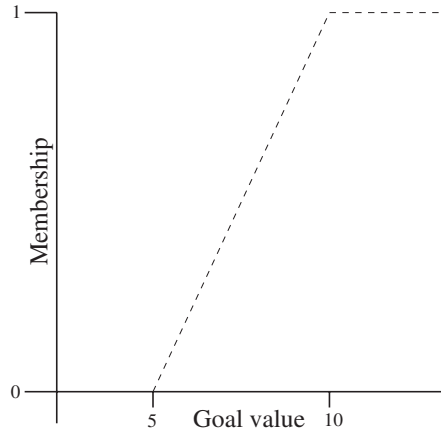


Figure 3.29 Simple fuzzy logic membership function.

are defined to be either hard (with rigid boundaries) or soft. When soft, the degree of acceptability is broken down between highly desirable, desirable, tolerable, undesirable, highly undesirable and unacceptable. Moreover, moving one goal across a given region is considered better than moving all the remaining goals across the next better region, for example, moving one across the tolerable region is better than moving all across the desirable region. In this way, the scheme aims to eliminate the worst features in a design. A principal motivation of the approach is to reduce the amount of time designers use studying aggregate goals with different sets of weightings, although designers may still wish to reassign what they mean by each of the linguistic definitions, of course.

In all cases, once a weighting and form of combined goal function has been decided, a single objective problem can then be tackled. If the outcome of a search with the chosen weights and function leads to designs that the team find inconsistent with their intuitive preferences, either the weight assignment process can then be iterated in light of these views or a different approach to constructing the combined goal function used. Sometimes, this may involve adding or removing preference information or goals. Commonly, such iteration is reduced when the more complex methods of weight assignment have been adopted, since the design team's preferences should then have been satisfied in as far as they have been articulated in the modeling.

3.6.3 Pareto Set Algorithms

Often, it is not possible or desirable to assign weights and combine functions before seeking promising designs – the designers may wish to "know what is out there" and actively seek a solution that is counterintuitive in the hope that this may lead to a gain in market share or a step forward in performance. In such circumstances, the Pareto solution set, or at least some approximation to it, must be sought.

The most obvious way to construct a Pareto set is to simply use a direct search to work through the various combinations of weights in a linear sum over the given objectives. Such an approach suffers from the obvious drawback that, when there are many objectives, there will be many weights to set and so building a reasonable Pareto set may be slow.

Also, it will not be clear, *a priori*, whether identifying designs that are optimal with one set of weights will be much harder than designs with other combinations – we already know that convex regions of the front cannot be found in this way – if a thorough survey over all possible weights is desired, this will further increase the cost of search. The desire to be able to efficiently construct Pareto sets, including designs lying in convex regions of the front, has lead to an entire subspecialization in the field of optimization research. It is also an area where much progress has been made using population-based methods, for the obvious reason that such methods work with sets of designs in the first place.

Perhaps, the best way to understand methods that aim to construct a Pareto set is to observe that, when constructing the front, the real goal is a well-balanced set of solutions, that is, one characterized by the members of the set being uniformly distributed along the front. Then, given that the designer chooses the number of members in the set, i.e., the granularity of the solution, it is the function of the search to place these points evenly along the front. Therefore, building Pareto fronts usually consists of two phases: first, a set of nondominated points must be found, and secondly, these must be spaced out evenly. One effective way of doing this is via an archive of nondominated points (Knowles and Corne 1999). As the search proceeds, if any design is identified that is currently nondominated, it is placed in an archive of fixed size – the size being defined *a priori* to reflect the granularity required. Then, as new nondominated points are found, these are added to the archive until such time as it is full. Existing points in the archive are pruned if the new points dominate them, of course. Thereafter, all new points are compared to the archived set and only added if either a) they dominate an existing point or b) they increase the uniformity of coverage of the set, that is, they reduce crowding and so spread out the solutions along the emerging Pareto front – with the crowded members being the target for pruning to maintain the set size. A number of crowding measures can be used in this role – for example, if e is the normalized Euclidean distance in the objective function space between members of the archive, then the authors find a useful crowding metric to be $1/\sum e$, where the sum is limited to the nearest 5% of solutions in the front (the normalization is achieved by dividing each objective function by the maximum range of function values seen in that direction).

The archive approach to locating the Pareto front additionally requires a way of generating new nondominated points as efficiently as possible. Since the archive represents a useful store of material about good designs, it is quite common in Pareto set construction to try to extract information from this set when seeking new points. The genetic algorithm operation of crossover forms a good way of doing this as does the mutation of existing nondominated points to form the starting points for downhill or other forms of searches. Another approach is to construct a pseudoobjective function and employ a single objective function optimizer to improve it. The aim of the pseudoobjective is to score the latest design indirectly by whether or not it is feasible, dominates the previous solution or is less crowded than the previous solution. Figure 3.30 is a chunk of pseudocode that implements such an approach and which can be used to identify Pareto fronts in the Options search package in conjunction with an evolutionary strategy single objective optimizer.[6]

An alternative, more direct solution is to treat the generation used in an evolutionary algorithm as the evolving Pareto set itself and then to rank each member in the set by the degree to which it dominates other members. By introducing some form of crowding or niche forming mechanism, such a search will tend to evolve an approximation to the Pareto front with reasonable coverage. In the nondominated sorting GA of Srinivas and Deb (1995),

[6] `http://www.soton.ac.uk/~ajk/options/welcome.html`

```
If(current solution is feasible)then
  If(current solution dominates previous solution)then
    Q=Q-1
  Else
    If(current solution is less crowded than previous solution)then
      Q=Q-1
    Else
      Q=Q+1
    Endif
  Endif
Else
  Q=Q+1
Endif
```

Figure 3.30 Pseudocode for simple objective function when constructing Pareto fronts (Q is the function minimized by the optimizer – this is set to zero at the beginning of a search and is nonstationary, and so, this code is only suitable for methods capable of searching such functions, for example, the evolution strategy approach; the crowding metric is $1/\sum e$, where the sum is limited to the nearest 5% of solutions).

first the nondominated individuals are collected, given an equal score and placed to one side. Then, the remaining individuals are processed to find those that dominate this new set to give a second tier of domination – these are again scored equally but now at reduced value and again placed to one side, and so on, until the entire population is sorted into a series of fronts of decreasing rank. Each rank has its score reduced according to a sharing mechanism that encourages diversity. The final score is then used to drive the selection of new points by the GA in the normal way. The Multi-Objective Genetic Algorithm (MOGA) method of Fonseca and Fleming (1995) is another variant on this approach although now the degree of dominance is used rather than a series of ranks assembled of equal dominance. Its performance is dependent on exactly how the sharing process that spaces points out along the front is implemented and how well this maps to the difficulty of finding points on any particular region of the front. In all cases, if a region is difficult to populate with nondominated designs, it is important that the weighting used when assembling new generations is sufficient to preserve these types of solutions in the population – something the archive approach does implicitly.

3.6.4 Nash Equilibria

An alternative way to trade off the goals in a multiobjective problem is via the mechanism of Nash equilibria (Habbal et al. 2003; Sefrioui and Periaux 2000). These are based on a game player model: each goal in the design problem is given over to a dedicated "player", who also has control of some subset of the design variables. Then, each player in turn tries to optimize their goal using their variables, but subject to the other variables in the problem being held fixed at the last values determined by the other players. As the players take turns to improve their own goals, the problem settles into an equilibrium state such that none can improve on their own target by unilaterally changing their own position. Usually, the

variables assigned to a particular player are chosen to be those that most strongly correlate with their goal, although when a single variable strongly affects two goals, an arbitrary choice has to be made. In addition, a set of initial values has to be chosen to start the game off.

If carried out to complete convergence, playing a Nash game in this way will yield a result that lies on the Pareto front. Moreover, convergence to a nondominated point is usually very robust and quite rapid. The process can also be easily parallelized, since each player's search process can be handed to a dedicated processor and communication between players is only needed when each has found its latest optimal solution. Unfortunately, in many problems, different initial values will lead to different equilibria (i.e., different locations on the Pareto front), as will different combinations of assignments of the variables to different players. Nonetheless, such methods have found a good deal of use in economics and they are beginning to be adopted in aerospace design because of their speed and robustness.

3.7 Robustness

Thus far in this chapter, we have been concerned with tools that may be used to locate the optima in deterministic functions – the assumption being that such locations gave the ideal combinations of parameters that we seek in design. In fact, this is very often not so – as will by now have become clear, many such optima are defined by the actions of constraint boundaries. To develop a design that sits exactly on the edge of a hard constraint boundary is rarely what designers wish to do – this would lead to a degree of risk in the performance of the product that can rarely be justified by any marginal gain that might be attained. Rather, designers almost always wish their designs to exhibit robustness. Such robustness must cope with uncertainties coming from many sources: even today, our ability to predict product behavior at the design stage is by no means exact. Moreover, as-manufactured products always differ, sometimes substantially, from the nominal specifications. Finally, the in-service conditions experienced by products can rarely be foreseen with complete clarity. In consequence, some margin for uncertainty must be made. Thus, we say that a design whose nominal performance degrades only slightly in the face of uncertainty is robust, while one that has dramatic falloff in performance lacks robustness and is fragile. We return to this important topic in much more detail in Chapter 8 – here, we make some of the more obvious observations.

Given the relationship of robustness to changes in design performance, it will be obvious that robust designs will lie in smooth, flat regions of the search space. It is therefore often desirable that an optimizer can not only identify regions of high performance in the search space but also ones where plateaus of robustness may be found; see Figure 3.31. If the best design lies on the edge of the search space or where constraints are active, this will often mean that the way the objective function surface approaches the boundary will be important – a steep intersection with such a boundary will usually imply a fragile design (e.g., point (0,80) in Figure 3.31).

Some search methods are intrinsically better able to locate robust designs than others, while some may be adapted, albeit at extra cost, to yield robust designs. It is also worth noting that the way that targets are specified can also be used to influence this quality in design – if a constraint violation is embedded in the objective function rather than treated explicitly as a hard boundary, this will impact on the way the search proceeds – this can be an important way of managing the trade-off between nominal performance and robustness. In many senses, the desire for robustness can be seen as yet a further goal that designers must

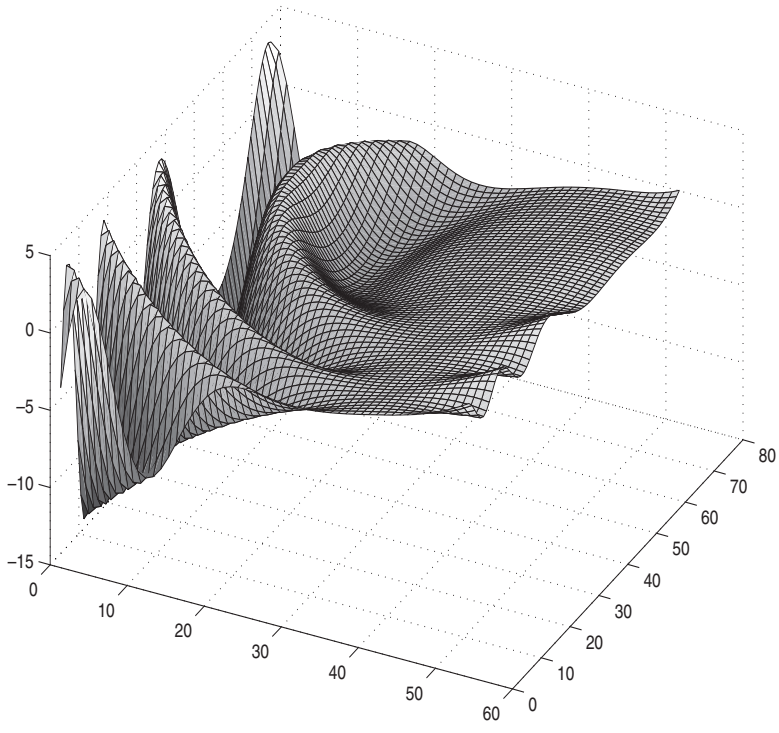


Figure 3.31 A fitness (maximization) landscape the exhibits fragile and robust regions, after Parmee (1996).

strive for and treated along with any others in a multiobjective framework. It is, however, often useful to treat robustness and uncertainty separately when carrying out search.

Perhaps, the most obvious way of dealing with robustness is to add artificial noise to the design vector or the resulting objective or constraints, or all of these, when calculating the performance of the design. This noise can be set to simulate uncertainty in the design so that the search engine must directly cope with this when attempting to find good configurations. Of course, this will mean that repeated calls by the search to the functions being dealt with will yield different responses – few search methods are able to cope with this directly, but it is perhaps the most direct way to simulate what actually happens in practice. Noise added to the design vector represents uncertainty in the as-manufactured product, whereas noise in the objective and constraint functions represents uncertainty in the operating conditions and the design team's representation of the relevant physics.

Some search engines, notably the stochastic search methods (usually with elitism switched off), can easily be set up to work with this kind of dynamic environment and, if direct searches are affordable, this way of working has much to commend it. Uncertainty generated by added noise can also be tolerated by many of the metamodeling approaches, provided that they include an element of regression in their model building capabilities. In either case, issues of search termination become more subtle and complex when the functions being searched are random. Still, using random functions in this way adds only modestly to the expense of the underlying search. Of course, to do this requires knowledge

of how large any random perturbations in the design vector and modeling functions should be and this should be carefully assessed when specifying the problem being tackled – if the random elements included are unjustifiably large, then very pessimistic designs can result.

A more expensive but more exact way of proceeding is to attempt to sample the uncertainty statistics themselves at each point to be evaluated in the search space. To do this, the nominal design being evaluated is used to initiate a small random walk. Then the average, or perhaps worst design found during this walk is returned in place of the nominal design point. In this way, it is possible to make the search work on a pseudo mean or 95% confidence limit design so that any final optimal point will exhibit the desired degree of robustness. However, we have now replaced a single design evaluation with a random walk of perhaps 10–20 points (this will depend on the number of variables in play and the degree of accuracy required) – if this can be afforded it will allow significantly greater confidence to be placed in the final designs produced. An example of this kind of approach is given by Anthony et al. (2000). A more affordable approach is to combine these ideas – the functions and variables being treated as random during the search with random walks being added periodically so that a more precise understanding of the uncertainty is used from time to time.

The most expensive, but at the same time, most accurate way to deal with uncertainty in search is to use fully stochastic performance models throughout. Then, instead of working with deterministic quantities, the whole problem is dealt with in terms of the probability distribution functions (PDFs) of the inputs and outputs. This way of working is in its infancy at the time of writing, since stochastic PDE solvers are commonly orders of magnitude more expensive than their deterministic equivalents (at its most extreme each evaluation is replaced by a full, multidimensional, large scale Monte Carlo evaluation). It is also the case that only rarely do designers have sufficient information to construct the input variable PDFs for working in this way, especially if correlations arise between inputs (and they commonly do – and are even more commonly ignored in such work). Nonetheless, as stochastic PDE solvers become more mature it is inevitable that they will be increasingly used in optimization.