1

Introduction to Optimization in Electromagnetics

As in other areas of engineering and science, research efforts in electromagnetics have concentrated on finding a solution to an integral or differential equation with boundary conditions. An example is calculating the radar cross section (RCS) of an airplane. First, the problem is formulated for the size, shape, and material properties associated with the airplane. Next, an appropriate mathematical description that exactly or approximately models the airplane and electromagnetic waves is applied. Finally, numerical methods are used for the solution. One problem has one solution. Finding such a solution has proved quite difficult, even with powerful computers.

Designing the aircraft with the lowest RCS over a given frequency range is an example of an optimization problem. Rather than finding a single solution, optimization implies finding many solutions then selecting the best one. Optimization is an inherently slow, difficult procedure, but it is extremely useful when well done. The difficult problem of optimizing an electromagnetics design has only recently received extensive attention.

This book concentrates on the genetic algorithm (GA) approach to optimization that has proved very successful in applications in electromagnetics. We do not think that the GA is the best optimization algorithm for all problems. It has proved quite successful, though, when many other algorithms have failed. In order to appreciate the power of the GA, a background on the most common numerical optimization algorithms is given in this chapter to familiarize the reader with several optimization algorithms that can be applied to

Genetic Algorithms in Electromagnetics, by Randy L. Haupt and Douglas H. Werner

electromagnetics problems. The antenna array has historically been one of the most popular optimization targets in electromagnetics, so we continue that tradition as well.

The first optimum antenna array distribution is the binomial distribution proposed by Stone [1]. As is now well known, the amplitude weights of the elements in the array correspond to the binomial coefficients, and the resulting array factor has no sidelobes. In a later paper, Dolph mapped the Chebyshev polynomial onto the array factor polynomial to get all the sidelobes at an equal level [2]. The resulting array factor polynomial coefficients represent the Dolph–Chebyshev amplitude distribution. This amplitude taper is optimum in that specifying the maximum sidelobe level results in the smallest beamwidth, or specifying the beamwidth results in the lowest possible maximum sidelobe level. Taylor developed a method to optimize the sidelobe levels and beamwidth of a line source [3]. Elliot extended Taylor's work to new horizons, including Taylor-based tapers with asymmetric sidelobe levels, arbitrary sidelobe level designs, and null-free patterns [4]. It should be noted that Elliot's methods result in complex array weights, requiring both an amplitude and phase variation across the array aperture. Since the Taylor taper optimized continuous line sources, Villeneuve extended the technique to discrete arrays [5]. Bayliss used a method similar to Taylor's amplitude taper but applied to a monopulse difference pattern [6]. The first optimized phase taper was developed for the endfire array. Hansen and Woodyard showed that the array directivity is increased through a simple formula for phase shifts [7].

Iterative numerical methods became popular for finding optimum array tapers beginning in the 1970s. Analytical methods for linear array synthesis were well developed. Numerical methods were used to iteratively shape the mainbeam while constraining sidelobe levels for planar arrays [8–10]. The Fletcher–Powell method [11] was applied to optimizing the footprint pattern of a satellite planar array antenna. An iterative method has been proposed to optimize the directivity of an array via phase tapering [12] and a steepest-descent algorithm used to optimize array sidelobe levels [13]. Considerable interest in the design of nonuniformly spaced arrays began in the late 1950s and early 1960s. Numerical optimization attracted attention because analytical synthesis methods could not be found. A spotty sampling of some of the techniques employed include linear programming [14], dynamic programming [15], and steepest descent [16]. Many statistical methods have been used as well [17].

1.1 OPTIMIZING A FUNCTION OF ONE VARIABLE

Most practical optimization problems have many variables. It's usually best to learn to walk before learning to run, so this section starts with optimizing one variable; then the next section covers multiple variable optimization. After describing a couple of single-variable functions to be optimized, several

single variable optimization routines are introduced. Many of the multidimensional optimization routines rely on some version of the one-dimensional optimization algorithms described here.

Optimization implies finding either the minimum or maximum of an objective function, the mathematical function that is to be optimized. A variable is passed to the objective function and a value returned. The goal of optimization is to find the combination of variables that causes the objective function to return the highest or lowest possible value.

Consider the example of minimizing the output of a four-element array when the signal is incident at an angle ϕ. The array has equally spaced elements ($d = \lambda/2$) along the x axis (Fig. 1.1). If the end elements have the same variable amplitude (a), then the objective function is written as

$$\mathrm{AF}_1(a) = 0.25\left|a + e^{j\Psi} + e^{j2\Psi} + ae^{j3\Psi}\right| \tag{1.1}$$

where $\Psi = k\,du$
$k = 2\pi/\lambda$
λ = wavelength
$u = \cos\phi$

A graph of AF_1 for all values of u when $a = 1$ is shown in Figure 1.2. If $u = 0.8$ is the point to be minimized, then the plot of the objective function as a function of a is shown in Figure 1.3. There is only one minimum at $a = 0.382$.

Another objective function is a similar four-element array with uniform amplitude but conjugate phases at the end elements

$$\mathrm{AF}_2(\delta) = 0.25\left|e^{j\delta} + e^{j\Psi} + e^{j2\Psi} + e^{-j\delta}e^{j3\Psi}\right| \tag{1.2}$$

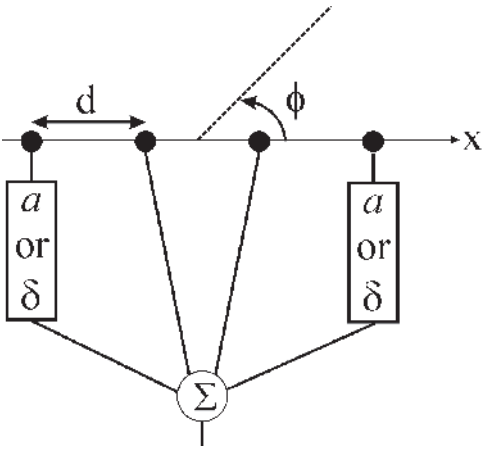

Figure 1.1. *Four-element array with two weights.*

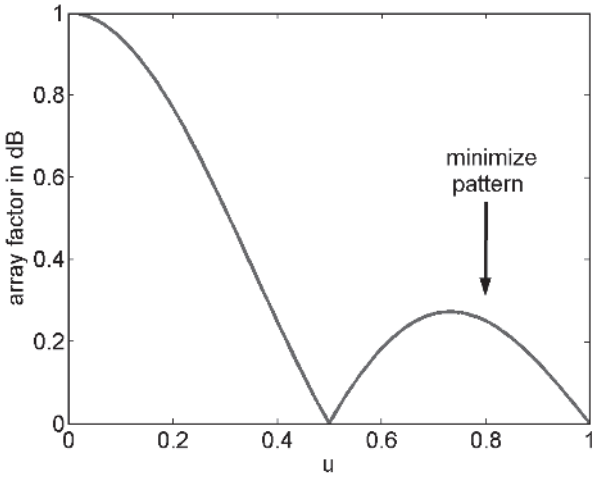

Figure 1.2. *Array factor of a four-element array.*

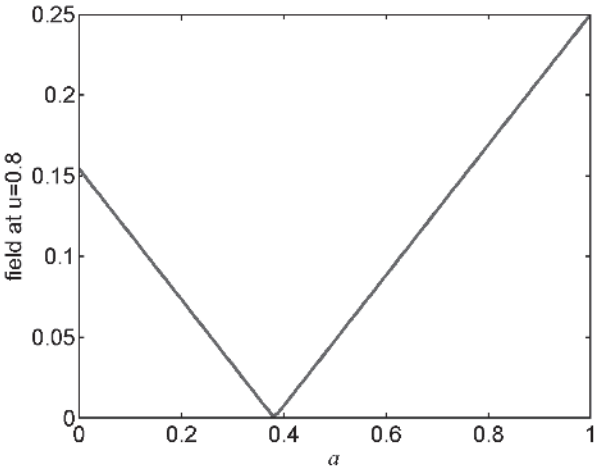

Figure 1.3. *Objective function with input* a *and output the field amplitude at* u = *0.8.*

where the phase range is given by $0 \leq \delta \leq \pi$. If $u = 0.8$ is the point to be minimized, then the plot of the objective function as a function of δ is as shown in Figure 1.4. This function is more complex in that it has two minima. The global or lowest minimum is at $\delta = 1.88$ radians while a local minimum is at $\delta = 0$.

Finding the minimum of (1.1) [Eq. (1.1), above] is straightforward—head downhill from any starting point on the surface. Finding the minimum of (1.2) is a little more tricky. Heading downhill from any point where $\delta < 0.63$ radian (rad) leads to the local minimum or the wrong answer. A different strategy is needed for the successful minimization of (1.2).

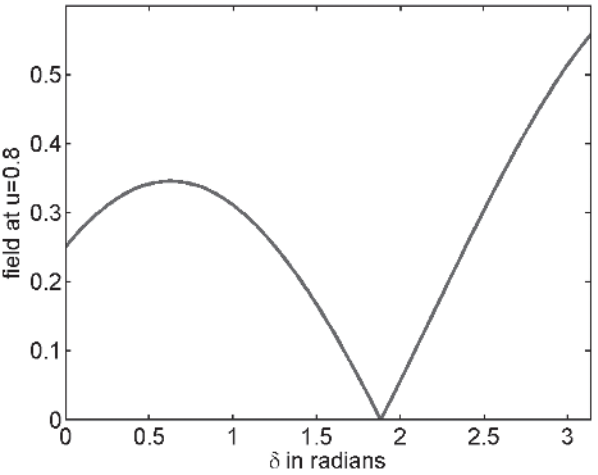

Figure 1.4. *Objective function with input δ and output the field amplitude at* u = *0.8.*

1.1.1 Exhaustive Search

One way to feel comfortable about finding a minimum is to check all possible combinations of input variables. This approach is possible for a small finite number of points. Probably the best example of an exhaustive search is graphing a function and finding the minimum on the graph. When the graph is smooth enough and contains all the important features of the function in sufficient detail, then the exhaustive search is done. Figures 1.3 and 1.4 are good examples of exhaustive search.

1.1.2 Random Search

Checking every possible point for a minimum is time-consuming. Randomly picking points over the interval of interest may find the minimum or at least come reasonably close. Figure 1.5 is a plot of AF_1 with 10 randomly selected points. Two of the points ended up close to the minimum. Figure 1.6 is a plot of AF_2 with 10 randomly selected points. In this case, six of the points have lower values than the local minimum at $\delta = 0$. The random search process can be refined by narrowing the region of guessing around the best few function evaluations found so far and guessing again in the new region. The odds of all 10 points appearing at $\delta < 0.63$ for AF_2 is $(0.63/\pi)^{10} = 1.02 \times 10^{-7}$, so it is unlikely that the random search would get stuck in this local minimum with 10 guesses. A quick random search could also prove worthwhile before starting a downhill search algorithm.

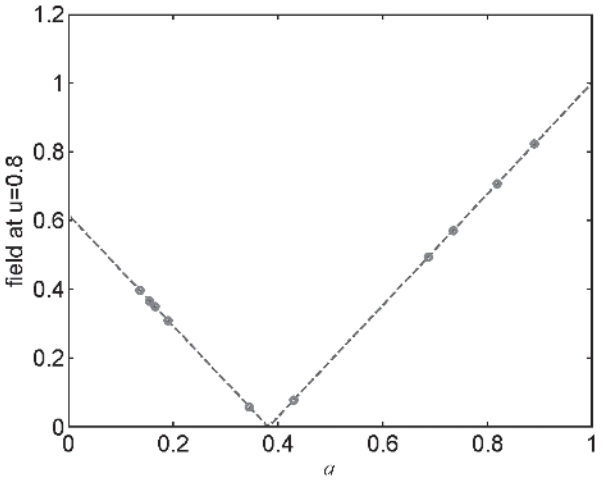

Figure 1.5. *Ten random guesses (circles) superimposed on a plot of AF_1.*

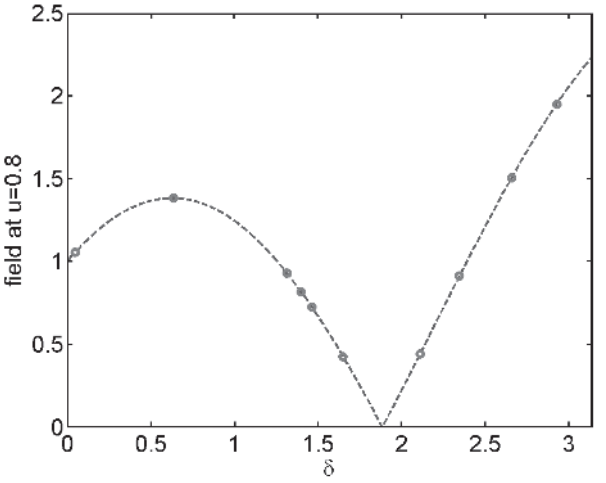

Figure 1.6. *Ten random guesses (circles) superimposed on a plot of AF_2.*

1.1.3 Golden Search

Assume that a minimum lies between two points a and b. Three points are needed to detect a minimum in the interval: two to bound the interval and one in between that is lower than the bounds. The goal is to shrink the interval by picking a point (c) in between the two endpoints (a and b) at a distance Δ_1 from a (see Fig. 1.7). Now, the interval is divided into one large interval and one small interval. Next, another point (d) is selected in the larger of the two subintervals. This new point is placed at a distance Δ_2 from c. If the new point

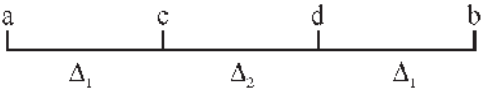

Figure 1.7. *Golden search interval.*

on the reduced interval ($\Delta_1 + \Delta_2$) is always placed at the same proportional distance from the left endpoint, then

$$\frac{\Delta_1}{\Delta_1 + \Delta_2 + \Delta_1} = \frac{\Delta_2}{\Delta_1 + \Delta_2} \tag{1.3}$$

If the interval is normalized, the length of the interval is

$$\Delta_1 + \Delta_2 + \Delta_1 = 1 \tag{1.4}$$

Combining (1.3) and (1.4) yields the equation

$$\Delta_1^2 - 3\Delta_1 + 1 = 0 \tag{1.5}$$

which has the root

$$\Delta_1 = \frac{\sqrt{5} - 1}{2} = 0.38197\ldots \tag{1.6}$$

This value is known as the "golden mean" [18].

The procedure above described is easy to put into an algorithm to find the minimum of AF_2. As stated, the algorithm begins with four points (labeled 1–4 in Fig. 1.8). Each iteration adds another point. After six iterations, point 8 is reached, which is getting very close to the minimum. In this case the golden search did not get stuck in the local minimum. If the algorithm started with points 1 and 4 as the bound, then the algorithm would have converged on the local minimum rather than the global minimum.

1.1.4 Newton's Method

Newton's method is a downhill sliding technique that is derived from the Taylor's series expansion for the derivative of a function of one variable. The derivative of a function evaluated at a point x_{n+1} can be written in terms of the function derivatives at a point x_n

$$f'(x_{n+1}) = f'(x_n) + f''(x_n)(x_{n+1} - x_n) + \frac{f'''(x_n)}{2!}(x_{n+1} - x_n)^2 + \cdots \tag{1.7}$$

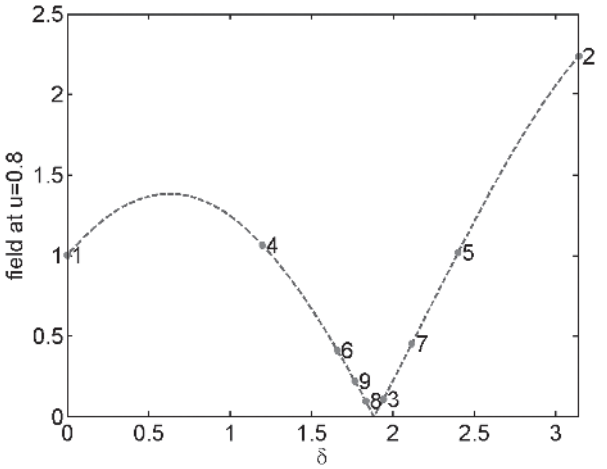

Figure 1.8. *The first eight function evaluations (circles) of the golden search algorithm when minimizing* AF_2.

Keeping only the first and second derivatives and assuming that the next step reaches the minimum or maximum, then (1.7) equals zero, so

$$f'(x_n) + f''(x_n)(x_{n+1} - x_n) = 0 \tag{1.8}$$

Solving for x_{n+1} yields

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{1.9}$$

If no analytical derivatives are available, then the derivatives in (1.9) are approximated by a finite-difference formula

$$x_{n+1} = x_n - \frac{\Delta[f(x_{n+1}) - f(x_{n-1})]}{2[f(x_{n+1}) - 2f(x_n) + f(x_{n-1})]} \tag{1.10}$$

where

$$\Delta = |x_{n+1} - x_n| = |x_n - x_{n-1}| \tag{1.11}$$

This approximation slows the method down but is often the only practical implementation.

Let's try finding the minimum of the two test functions. Since it's not easy to take the derivatives of AF_1 and AF_2, finite-difference approximations will be used instead. Newton's method converges on the minimum of AF_1 for every starting point in the interval. The second function is more interesting,

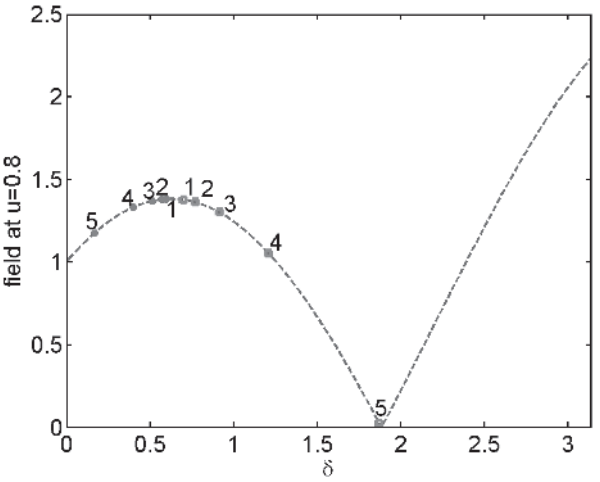

Figure 1.9. *The convergence of Newton's algorithm when starting at two different points.*

though. Figure 1.9 shows the first five points calculated by the algorithm from two different starting points. A starting point at $\delta = 0.6$ radians results in the series of points that heads toward the local minimum on the left. When the starting point is $\delta = 0.7$ rad, then the algorithm converges toward the global minimum. Thus, Newton's method is known as a *local search algorithm*, because it heads toward the bottom of the closest minimum. It is also a nonlinear algorithm, because the outcome can be very sensitive to the initial starting point.

1.1.5 Quadratic Interpolation

The techniques derived from Taylor's series assumed that the function is quadratic near the minimum. If this assumption is valid, then we should be able to approximate the function by a quadratic polynomial near the minimum and find the minimum of that quadratic polynomial interpolation [19]. Given three points on an interval (x_0, x_1, x_2), the extremum of the quadratic interpolating polynomial appears at

$$x_3 = \frac{f(x_0)\left(x_1^2 - x_2^2\right) + f(x_1)(x_2^2 - x_0^2) + f(x_2)(x_0^2 - x_1^2)}{2f(x_0)(x_1 - x_2) + 2f(x_1)(x_2 - x_0) + 2f(x_2)(x_0 - x_1)} \tag{1.12}$$

When the three points are along the same line, the denominator is zero and the interpolation fails. Also, this formula can't differentiate between a minimum and a maximum, so some caution is necessary to insure that it pursues a minimum.

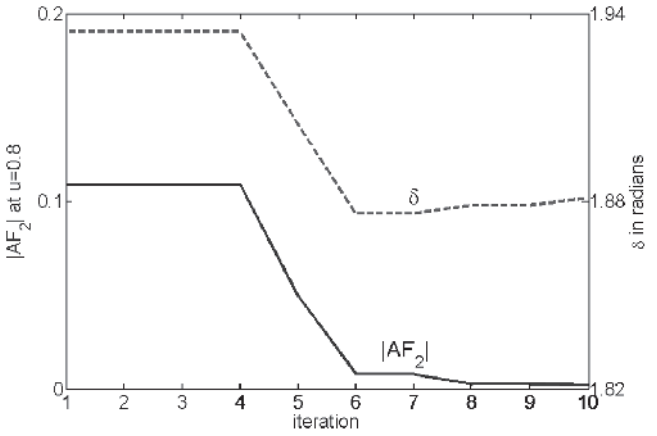

Figure 1.10. *Convergence of the MATLAB quadratic interpolation routine when minimizing AF_2.*

MATLAB uses a combination of golden search and quadratic interpolation in its function *fminbnd.m*. Figure 1.10 shows the convergence curves for the field value on the left-hand vertical axis and the phase in radians on the right-hand vertical axis. This approach converged in just 10 iterations.

1.2 OPTIMIZING A FUNCTION OF MULTIPLE VARIABLES

Usually, arrays have many elements; hence many variables need to be adjusted in order to optimize some aspect of the antenna pattern. To demonstrate the complexity of dealing with multiple dimensions, the objective functions in (1.1) and (1.2) are extended to two variables and three angle evaluations of the array factor.

$$\mathrm{AF}_3(a_1, a_2) = \frac{1}{6}\sum_{m=1}^{3}\left|a_2 + a_1 e^{j\Psi_m} + e^{j2\Psi_m} + e^{j3\Psi_m} + a_1 e^{j4\Psi_m} + a_2 e^{j5\Psi_m}\right| \tag{1.13}$$

$$\mathrm{AF}_4(\delta_1, \delta_2) = \frac{1}{6}\sum_{m=1}^{3}\left|e^{j\delta_2} + e^{j\delta_1} e^{j\Psi_m} + e^{j2\Psi_m} + e^{j3\Psi_m} + e^{j\delta_1} e^{j4\Psi_m} + e^{j\delta_2} e^{j5\Psi_m}\right| \tag{1.14}$$

Figure 1.11 is a diagram of the six-element array with two independent adjustable weights. The objective function returns the sum of the magnitude of the array factor at three angles: ϕ_m = 120°, 69.5°, and 31.8°. The array factor for a uniform six-element array is shown in Figure 1.12. Plots of the objective function for all possible combinations of the amplitude and phase weights appear in Figures 1.13 and 1.14. The amplitude weight objective func-

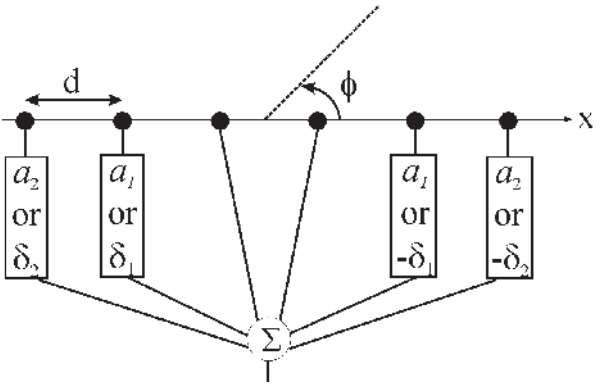

Figure 1.11. *A six-element array with two independent, adjustable weights.*

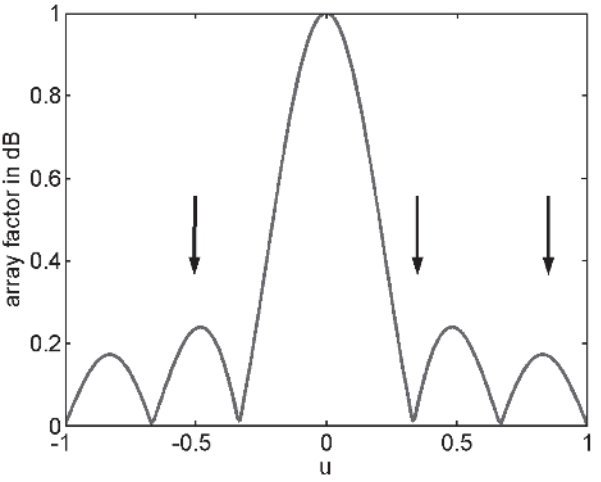

Figure 1.12. *The array factor for a six-element uniform array.*

tion has a single minimum, while the phase weight objective function has several minima.

1.2.1 Random Search

Humans are intrigued by guessing. Most people love to gamble, at least occasionally. Beating the odds is fun. Guessing at the location of the minimum sometimes works. It's at least a very easy-to-understand method for minimization—no Hessians, gradients, simplexes, and so on. It takes only a couple of lines of MATLAB code to get a working program. It's not very elegant, though, and many people have ignored the power of random search in the

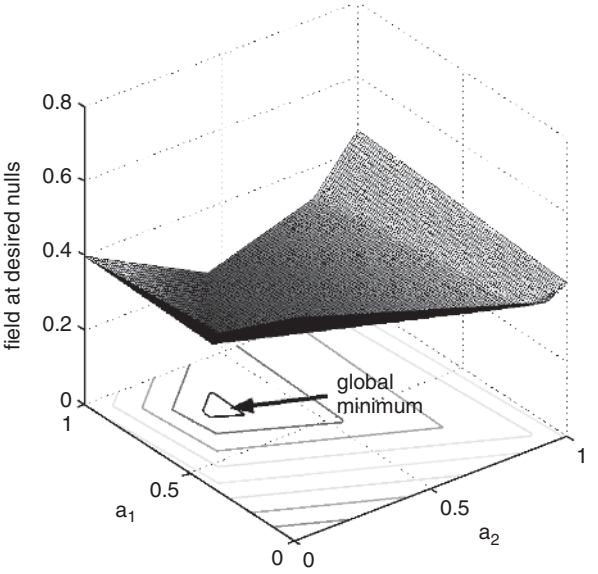

Figure 1.13. *Plot of the objective function for AF_3.*

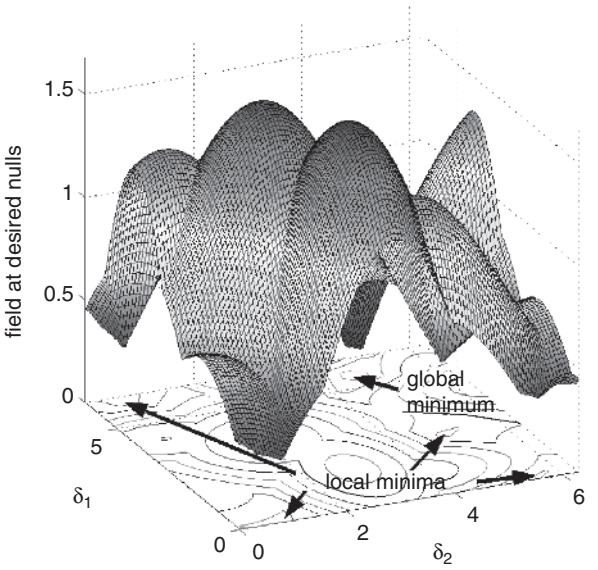

Figure 1.14. *Plot of the objective function for AF_4.*

development of sophisticated minimization algorithms. We often model processes in nature as random events, because we don't understand all the complexities involved. A complex cost function more closely approximates nature's ways, so the more complex the cost function, the more likely that random

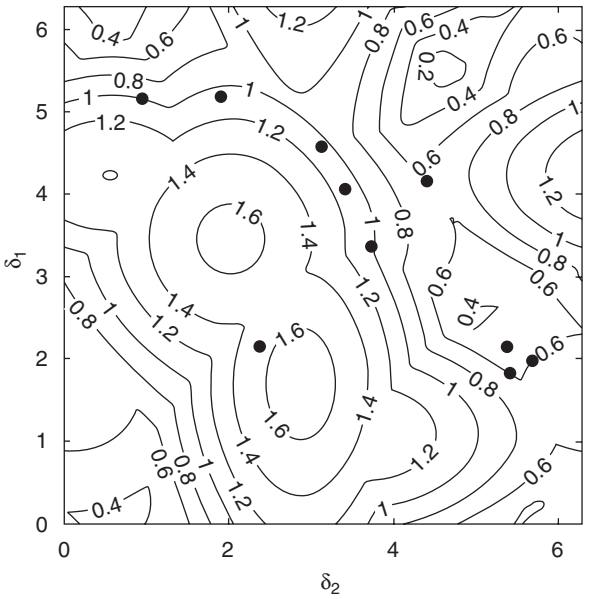

Figure 1.15. *Ten random guesses for AF_4.*

guessing plays an important part in finding the minimum. Even a local optimizer makes use of random starting points. Local optimizers are made more "global" by making repeated starts from several different, usually random, points on the cost surface.

Figure 1.15 shows 10 random guesses on a contour plot of AF_4. This first attempt clearly misses some of the regions with minima. The plot in Figure 1.16 results from adding 20 more guesses. Even after 30 guesses, the lowest value found is not in the basin of the global minimum. Granted, a new set of random guesses could easily land a value near the global minimum. The problem, though, is that the odds decrease as the number of variables increases.

1.2.2 Line Search

A line search begins with an arbitrary starting point on the cost surface. A vector or line is chosen that cuts across the cost surface. Steps are taken along this line until a minimum is reached. Next, another vector is found and the process repeated. A flowchart of the algorithm appears in Figure 1.17. Selecting the vector and the step size has been an area of avid research in numerical optimization. Line search methods work well for finding a minimum of a quadratic function. They tend to fail miserably when searching a cost surface with many minima, because the vectors can totally miss the area where the global minimum exists.

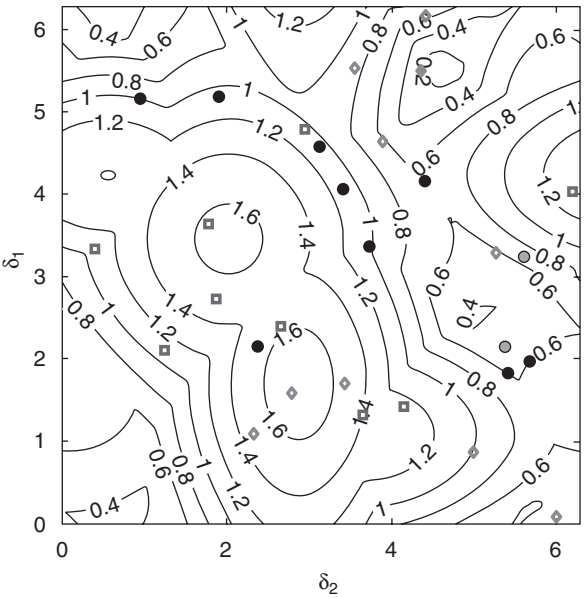

Figure 1.16. *Thirty random guesses for AF_4.*

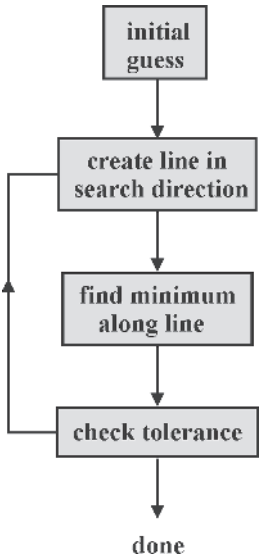

Figure 1.17. *Flowchart of a line search minimization algorithm.*

The easiest line search imaginable is the coordinate search method. If the function has two variables, then the algorithm begins at a random point, holds one variable constant, and searches along the other variable. Once it reaches a minimum, it holds the second variable constant and searches along the first

variable. This process repeats until an acceptable minimum is found. Mathematically, a two-dimensional cost function follows the path given by

$$f(v_1^0, v_2^0) \to f(v_1^1, v_2^0) \to f(v_1^1, v_2^1) \to f(v_1^2, v_2^1) \to \cdots \tag{1.15}$$

where $v_n^{m+1} = v_n^m + \ell_n^{m+1}$ and ℓ_m^{n+1} is the step length calculated using a formula. This approach doesn't work well, because it does not exploit any information about the cost function. Most of the time, the coordinate axes are not the best search directions [20]. Figure 1.18 shows the paths taken by a coordinate search algorithm from three different starting points on AF_4. A different minimum was found from each starting point.

The coordinate search does a lot of unnecessary wiggling to get to a minimum. Following the gradient seems to be a more intuitive natural choice for the direction of search. When water flows down a hillside, it follows the gradient of the surface. Since the gradient points in the direction of maximum increase, the negative of the gradient must be followed to find the minimum. This observation leads to the method of steepest descent given by [19]

$$v^{m+1} = v^m - \alpha^m \nabla f(v^m) \tag{1.16}$$

where α^m is the step size. This formula requires only first-derivative information. Steepest descent is very popular because of its simple form and often excellent results. Problems with slow convergence arise when the cost function

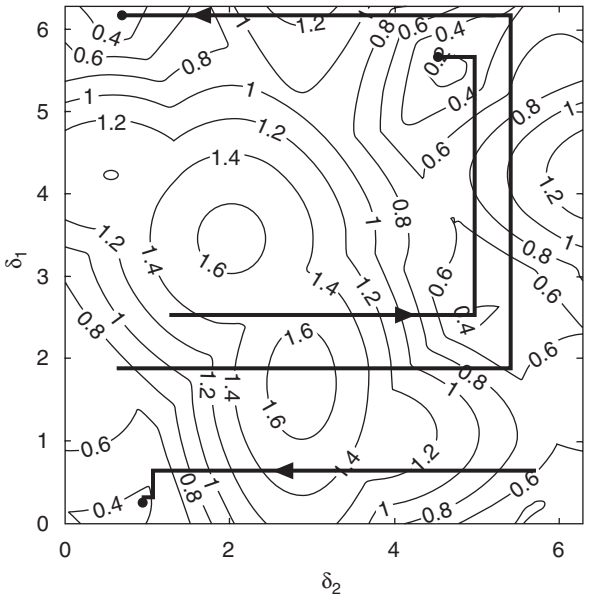

Figure 1.18. *A coordinate search algorithm finds three different minima of AF_4 when starting at three different points.*

has narrow valleys, since the new gradient is always perpendicular to the old gradient at the minimum point on the old gradient.

Even more powerful methods are possible if second-derivative information is used. Starting with the Taylor series expansion of the function

$$f(v^{m+1}) = f(v^m) + \nabla f(v^m)(v^{m+1} - v^m)^T + 0.5(v^{m+1} - v^m)\mathrm{H}(v^{m+1} - v^m)^T + \cdots \tag{1.17}$$

where v^m = point about which Taylor series is expanded
v^{m+1} = point near v^m
^{T} = transpose of vector (in this case row vector becomes column vector)
H = Hessian matrix with elements given by $h_{ij} = \partial^2 f / \partial v_i \partial v_j$

Taking the gradient of (1.17) and setting it equal to zero yields

$$\nabla f(v^{m+1}) = \nabla f(v^m) + (v^{m+1} - v^m)\mathrm{H} = 0 \tag{1.18}$$

which leads to

$$v^{m+1} = v^m - \alpha^m \mathrm{H}^{-1} \nabla f(v^m) \tag{1.19}$$

This formula is known as *Newton's method*. Although Newton's method promises fast convergence, calculating the Hessian matrix and then inverting it is difficult or impossible. Newton's method reduces to steepest descent when the Hessian matrix is the identity matrix. Several iterative methods have been developed to estimate the Hessian matrix with the estimate getting closer after every iteration. The first approach is known as the *Davidon–Fletcher–Powell* (DFP) update formula [21]. It is written here in terms of the mth approximation to the inverse of the Hessian matrix, $Q = H^{-1}$:

$$\begin{aligned} Q^{m+1} = Q^m &+ \frac{(v^{m+1} - v^m)(v^{m+1} - v^m)^T}{(v^{m+1} - v^m)^T(\nabla f(v^{m+1}) - \nabla f(v^m))} \\ &- \frac{Q^m(\nabla f(v^{m+1}) - \nabla f(v^m))(\nabla f(v^{m+1}) - \nabla f(v^m))^T Q^m}{(\nabla f(v^{m+1}) - \nabla f(v^m))^T Q^m (\nabla f(v^{m+1}) - \nabla f(v^m))} \end{aligned} \tag{1.20}$$

A similar formula was developed later and became known as the *Broyden–Fletcher–Goldfarb–Shanno* (BFGS) update [22–25]:

$$\begin{aligned} H^{m+1} = H^m &+ \frac{(\nabla f(v^{m+1}) - \nabla f(v^m))(\nabla f(v^{m+1}) - \nabla f(v^m))^T}{(\nabla f(v^{m+1}) - \nabla f(v^m))^T (v^{m+1} S - v^m)} \\ &- \frac{H^m (v^{m+1} - v^m)(v^{m+1} - v^m)^T H^m}{(v^{m+1} - v^m)^T H^m (v^{m+1} - v^m)} \end{aligned} \tag{1.21}$$

As with the DFP update, the BFGS update can be written for the inverse of the Hessian matrix.

A totally different approach to the line search is possible. If a problem has N dimensions, then it might be possible to pick N orthogonal search directions that could result in finding the minimum in N iterations. Two consecutive search directions, u and v, are orthogonal if their dot product is zero or

$$u \cdot v = uv^T = 0 \tag{1.22}$$

This result is equivalent to

$$uHv^T = 0 \tag{1.23}$$

where H is the identity matrix. If (1.23) is true and H is not the identity matrix, then u and v are known as *conjugate vectors* or *H-orthogonal vectors*. A set of N vectors that have this property is known as a *conjugate set*. It is these vectors that will lead to the minimum in N steps.

Powell developed a method of following these conjugate directions to the minimum of a quadratic function. Start at an arbitrary point and pick a search direction. Next, Gram–Schmidt orthogonalization is used to find the remaining search directions. This process is not very efficient and can result in some search directions that are nearly linearly dependent. Some modifications to Powell's method make it more attractive.

The best implementation of the conjugate directions algorithm is the conjugate gradient algorithm [26]. This approach uses the steepest descent as its first step. At each additional step, the new gradient vector is calculated and added to a combination of previous search vectors to find a new conjugate direction vector

$$v^{m+1} = v^m + \alpha^m \ell^m \tag{1.24}$$

where the step size is given by

$$\alpha^m = -\frac{\ell^{mT} \nabla f(v^m)}{\ell^{mT} H \ell^m} \tag{1.25}$$

Since (1.25) requires calculation of the Hessian, α^m is usually found by minimizing $f(v^m + \alpha^m \ell^m)$. The new search direction is found using

$$\ell^{m+1} = -\nabla f^{m+1} + \beta^{m+1} \ell^m \tag{1.26}$$

The Fletcher–Reeves version of β^m is used for linear problems [18]:

$$\beta^m = \frac{\nabla f^T(v^{m+1}) \nabla f(v^{m+1})}{\nabla f^T(v^m) \nabla f(v^m)} \tag{1.27}$$

This formulation converges when the starting point is sufficiently close to the minimum. A nonlinear conjugate gradient algorithm that uses the Polak–Ribiere version of β^m also exists [18]:

$$\beta^m = \max\left\{\frac{[\nabla f(v^{m+1}) - \nabla f(v^m)]^T \nabla f(v^{m+1})}{\nabla f^T(v^m)\nabla f(v^m)}, 0\right\} \tag{1.28}$$

The nonlinear conjugate gradient algorithm is guaranteed to converge for linear functions but not for nonlinear functions.

The problem with conjugate gradient is that it must be "restarted" every N iterations. Thus, for a nonquadratic problem (most problems of interest), conjugate gradient starts over after N iterations without finding the minimum. Since the BFGS algorithm does not need to be restarted and approaches superlinear convergence close to the solution, it is usually preferred over conjugate gradient. If the Hessian matrix gets too large to be conveniently stored, however, conjugate gradient shines with its minimal storage requirements.

1.2.3 Nelder–Mead Downhill Simplex Algorithm

Derivatives and guessing are not the only way to do a downhill search. The Nelder–Mead downhill simplex algorithm moves a simplex down the slope until it surrounds the minimum [27]. A simplex is the most basic geometric object that can be formed in an N-dimensional space. The simplex has $N + 1$ sides, such as a triangle in two-dimensional space. The downhill simplex method is given a single starting point (v^0). It generates an additional N points to form the initial simplex using the formula

$$v_n^0 = v_1^0 + \mu \ell_n \tag{1.29}$$

where the ℓ_n are unit vectors and μ is a constant. If the simplex surrounds the minimum, then the simplex shrinks in all directions. Otherwise, the point corresponding to the highest objective function is replaced with a new point that has a lower objective function value. The diameter of the simplex eventually gets small enough that it is less than the specified tolerance and the solution is the vertex with the lowest objective function value. A flowchart outlining the steps to this algorithm is shown in Figure 1.19.

Figure 1.20 shows the path taken by the Nelder–Mead algorithm starting with the first triangle and working its way down to the minimum of AF_3. Sometimes the algorithm flips the triangle and at other times it shrinks or expands the triangle in an effort to surround the minimum. Although it can successfully find the minimum of AF_3, it has great difficulty finding the global

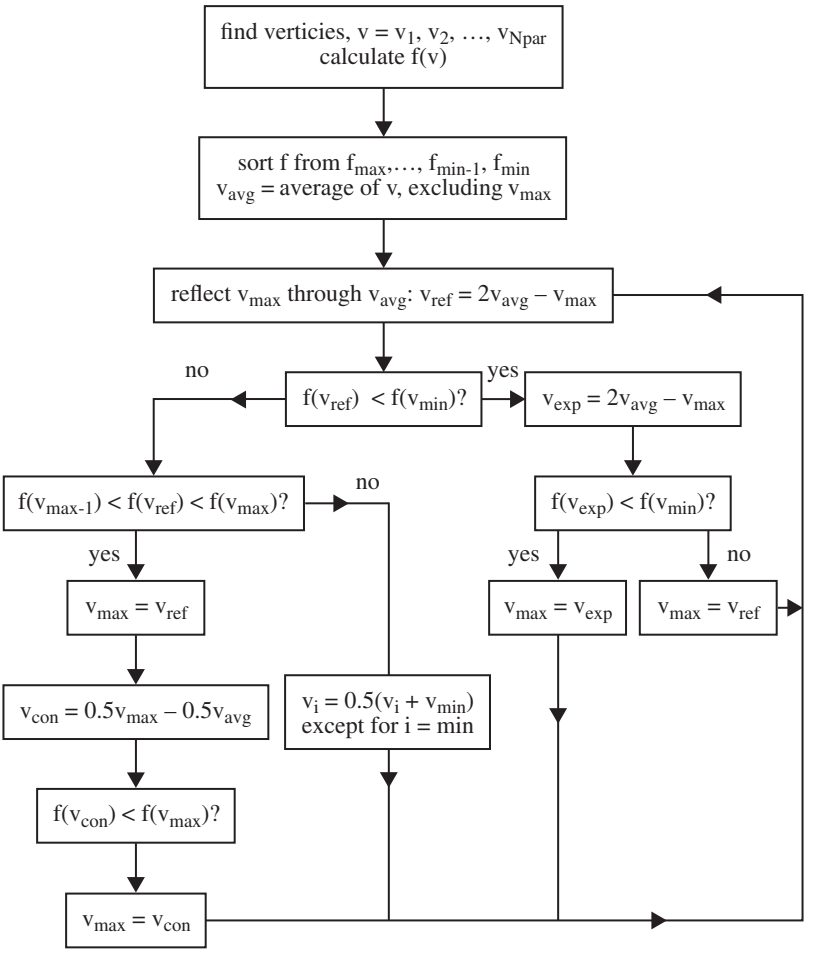

Figure 1.19. *Flowchart for the Nelder–Mead downhill simplex algorithm.*

minimum of AF_4. Its success with AF_4 depends on the starting point. Figure 1.21 shows a plot of the starting points for the Nelder–Mead algorithm that converge to the global minimum. Any other point on the plot converges to one of the local minima. There were 10,201 starting points tried and 2290 or 22.45% converged to the global minimum. That's just slightly better than a one-in-five chance of finding the true minimum. Not very encouraging, especially when the number of dimensions increases. The line search algorithms exhibit the same behavior as the Nelder–Mead algorithm in Figure 1.21.

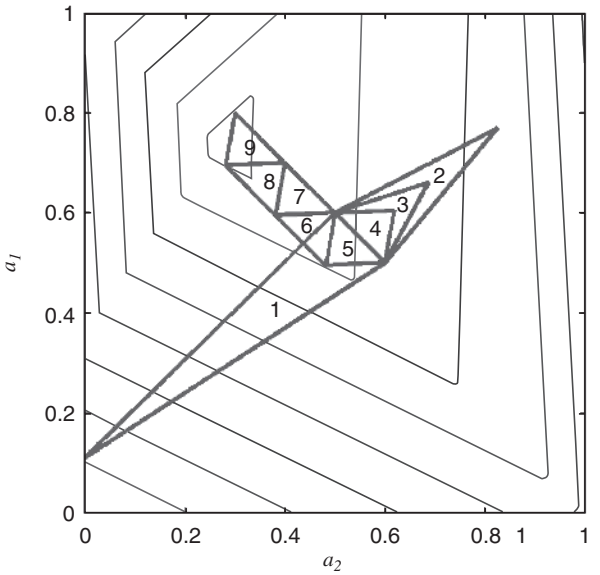

Figure 1.20. *Movements of the simplex in the Nelder–Mead downhill simplex algorithm when finding the minimum of AF_3.*

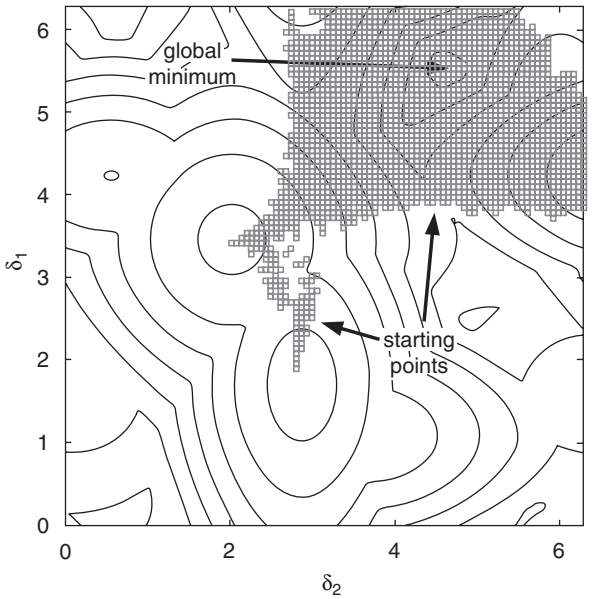

Figure 1.21. *Starting points for the Nelder–Mead algorithm denoted by small squares converge to the global minimum. All other starting points converge to a local minimum.*

1.3 COMPARING LOCAL NUMERICAL OPTIMIZATION ALGORITHMS

This section compares four local optimization approaches against a more formidable problem. Consider a linear array along the x axis with an array factor given by

$$\mathrm{AF}(u, \lambda, w, x, N_{\mathrm{el}}) = \sum_{n=1}^{N_{el}} w_n e^{jkx_n u} \tag{1.30}$$

where N_{el} = number of elements in the array
$w_n = a_n \exp(j\delta_n)$ = complex element weight
x_n = distance of element n from the origin

Some of these variables may be constants. For instance, if all except u are constant, then AF returns an antenna pattern that is a function of angle.

The optimization example here minimizes the maximum relative sidelobe level of a uniformly spaced array using amplitude weighting. Assume that the array weights are symmetric about the center of the array; thus the exponential terms of the symmetric element locations can be combined using Euler's identity. Also assume that the array has an even number of elements. With these assumptions, the objective function is written as a function of the amplitude weights

$$\mathrm{AF}_5(a) = 2\max\left|\sum_{n=1}^{N} a_n \cos[(n-0.5)\Psi_m]\right|, \quad u > u_b \tag{1.31}$$

where u_b defines the extent of the main beam. This function is relatively simple, except for finding the appropriate value for u_b. For a uniform aperture, the first null next to the mainlobe occurs at an angle of about λ/(size of the aperture) $\cong \lambda/[d(N+1)]$. Amplitude tapers decrease the efficiency of the aperture, thus increasing the width of the mainbeam. As a result, u_b depends on the amplitude, making the function difficult to characterize for a given set of input values. In addition, shoulders on the mainbeam may be overlooked by the function that finds the maximum sidelobe level.

The four methods used to optimize AF_5 are

1. Broyden–Fletcher–Goldfarb–Shanno (BFGS)
2. Davidon–Fletcher–Powell (DFP)
3. Nelder–Mead downhill simplex (NMDS)
4. Steepest descent

All of these are available using the MATLAB functions *fminsearch.m* and *fminunc.m*. The analytical solution is simple: $-\infty$ dB. Let's see how the different methods fare.

The four algorithms are quite sensitive to the starting values of the amplitude weights. These algorithms quickly fall into a local minimum, because their theoretical development is based on finding the minimum of a bowl-shaped objective function. The first optimization run randomly generated a starting vector of amplitude weights between 0 and 1. Each algorithm was given 25 different starting values and the results were averaged. Table 1.1 shows the results for a linear array of isotropic point sources spaced 0.5λ apart. A $-\infty$ sidelobe level would ruin a calculation of the mean of the best sidelobe level found over the 25 runs, so the median is as reported in Table 1.1. The mean and median are very close except when a $-\infty$ sidelobe level occurs.

These results are somewhat disappointing, since the known answer is $-\infty$ dB. The local nature of the algorithms limits their ability to find the best or the global solution. In general, a starting point is selected, then the algorithm proceeds downhill from there. When the algorithm encounters too many hills and valleys in the output of the objective function, it can't find the global optimum. Even selecting 25 random starting points for four different algorithms didn't result in finding an output with less than –50 dB sidelobe levels.

In general, as the number of variables increases, the number of function calls needed to find the minimum also increases. Thus, Table 1.1 has a larger median number of function calls for larger arrays. Table 1.2 shows how increas-

TABLE 1.1. Comparison of Optimized Median Sidelobes for Three Different Array Sizes[a]

	22 Elements		42 Elements		62 Elements	
	Median Sidelobe Level (dB)	Median Function Calls	Median Sidelobe Level (dB)	Median Function Calls	Median Sidelobe Level (dB)	Median Function Calls
BFGS	–30.3	1007	–25.3	2008	–26.6	3016
DFP	–27.9	1006	–25.2	2011	–26.6	3015
Nelder Mead	–18.7	956	–17.3	2575	–17.2	3551
Steepest descent	–24.6	1005	–21.6	2009	–21.8	3013

[a]Performance characteristics of four algorithms are averaged over 25 runs with random starting values for the amplitude weights. The isotropic elements were spaced 0.5λ apart.

TABLE 1.2. Algorithm Performance in Terms of Median Maximum Sidelobe Level versus Maximum Number of Function Calls[a]

Algorithm	1000 Function Calls (dB)	3000 Function Calls (dB)	10000 Function Calls (dB)
BFGS	–24.3	–26.6	–28.2
DFP	–24.0	–26.6	–28.3
Nelder–Mead	–17.6	–17.2	–17.5
Steepest descent	–23.3	–21.8	–23.4

[a]Performance characteristics of four algorithms are averaged over 25 runs with random starting values for the amplitude weights. The 42 isotropic elements were spaced 0.5λ apart.

TABLE 1.3. Algorithm Performance in Terms of Median Maximum Sidelobe When the Algorithm Is Restarted Every 2000 Function Calls (5 Times)[a]

Algorithm	10,000 Function Calls (dB)
BFGS	–34.9
DFP	–36.9
Nelder–Mead	–29.1
Steepest descent	–26.1

[a] Performance characteristics of four algorithms are averaged over 25 runs with random starting values for the amplitude weights. The 42 isotropic elements were spaced 0.5λ apart.

ing the maximum number of function calls improves the results found using BFGS and DFP whereas the Nelder–Mead and steepest-descent algorithms show no improvement.

Another idea is warranted. Perhaps taking the set of parameters that produces the lowest objective function output and using them as the initial starting point for the algorithm will produce better results. The step size gets smaller as the algorithm progresses. Starting over may allow the algorithm to take large enough steps to get out of the valley of the local minimum. Thus, the algorithm begins with a random set of amplitude weights, the algorithm optimizes with these weights to produce an "optimal" set of parameters, these new "optimal" set of parameters are used as a new initial starting point for the algorithm, and the process repeats several times. Table 1.3 displays some interesting results when the cycle is repeated 5 times. Again, the algorithms were averaged over 25 different runs. In all cases, the results improved by running the optimization algorithm for 2000 function calls on five separate starts rather than running the optimization algorithm for a total of 10,000 function calls with one start (Table 1.2). The lesson learned here is to use this iterative procedure when attempting an optimization with multiple local minima. The size of the search space collapses as the algorithm converges on the minimum. Thus, restarting the algorithm at the local minimum just expands the search space about the minimum.

An alternative approach known as "seeding" starts the algorithm with a good first guess based on experience, a hunch, or other similar solutions. In general, we know that low-sidelobe amplitude tapers have a maximum amplitude at the center of the array, while decreasing in amplitude toward the edges. The initial first guess is a uniform amplitude taper with a maximum sidelobe level of –13 dB. Table 1.4 shows the results of using this good first guess after 2000 function calls. The Nelder–Mead algorithm capitalized on this good first guess, while the others didn't. Trying a triangle amplitude taper, however, significantly improved the performance of all the algorithms. In fact, the Nelder–Mead and steepest-descent algorithms did better than the BFGS and DFP algorithms. Combining the good first guess with the restarting idea in Figure 1.11 may produce the best results of all.

TABLE 1.4. Algorithm Performance in Terms of Median Maximum Sidelobe after 2000 Function Calls When the Algorithm Seeded with a Uniform or Triangular Amplitude Taper[a]

Algorithm	Uniform Taper Seed (dB)	Triangular Taper Seed (dB)
BFGS	–23.6	–35.9
DFP	–26.0	–35.7
Nelder–Mead	–23.9	–39.1
Steepest descent	–21.2	–39.3

[a]The 42 isotropic elements were spaced 0.5λ apart.

1.4 SIMULATED ANNEALING

Annealing heats a substance above its melting temperature, then gradually cools it to produce a crystalline lattice that has a minimum energy probability distribution. The resulting crystal is an example of nature finding an optimal solution. If the liquid cools too rapidly, then the crystals do not form and the substance becomes an amorphous mass with an energy state above optimal. Nature is seldom in a hurry to find the optimal state.

A numerical optimization algorithm that models the annealing process is known as *simulated annealing* [28,29]. The initial state of the algorithm is a single random guess of the objective function input variables. In order to model the heating process, the values of the variables are randomly modified. Higher heat creates greater random fluctuations. The objective function returns a measure of the energy state or value of the present minimum. The new variable set replaces the old variable set if the output decreases. Otherwise the output is still accepted if

$$r \leq e^{[f(p_{\text{old}})-f(p_{\text{new}})]/T} \tag{1.32}$$

where r is a uniform random number and T is a temperature value. If r is too large, then the variable values are rejected. A new variable set to replace a rejected variable set is found by adding a random step to the old variable set

$$p_{\text{new}} = dp_{\text{old}} \tag{1.33}$$

where d is a random number with either a uniform or normal distribution with a mean of p_{old}. When the minimization process stalls, the value of T and the range of d decrease by a certain percent and the algorithm starts over. The algorithm is finished when T gets close to zero. Some common cooling schedules include (1) linearly decreasing, $T_n = T_0 - n(T_0 - T_n)/N$; (2) geometrically decreasing, $T_n = 0.99T_{n-1}$; and (3) Hayjek optimal, $T_n = c/\log(1 + n)$, where c = smallest variation required to get out of any local minimum, $0 < n \leq N$; T_0 = initial temperature, and T_N = ending temperature.

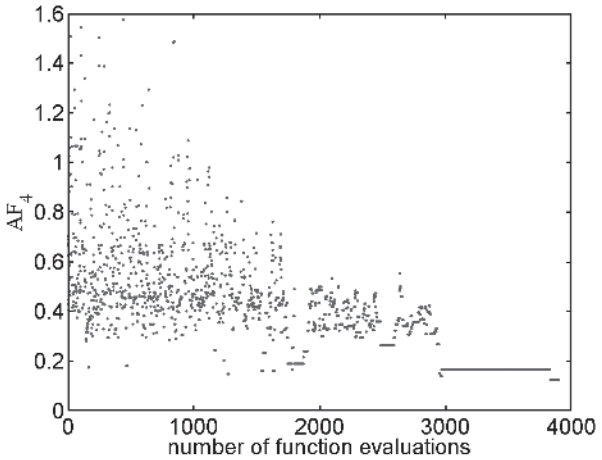

Figure 1.22. *Convergence of the simulated annealing algorithm for AF_4.*

The temperature is lowered slowly, so that the algorithm does not converge too quickly.

Simulated annealing (SA) begins as a random search and ends with little variations about the minimum. Figure 1.22 shows the convergence of the simulated annealing algorithm when minimizing AF_4. The final value was 0.1228 or –33.8 dB. Simulated annealing was first applied to the optimization of antenna arrays in 1988 [30].

SA has proven superior to the local optimizers discussed in this chapter. The random perturbations allow this algorithm to jump out of a local minimum in search of the global minimum. SA is very similar to the GA. It is a random search that has tuning parameters that have tremendous effect on the success and speed of the algorithm. SA starts with a single guess at the solution and works in a serial manner to find the solution. A genetic algorithm starts with many initial guesses and works in a parallel manner to find a list of solutions. The SA algorithm slowly becomes less random as it converges, while the GA may or may not become less random with time. Finally, the GA is more adept at working with continuous, discrete, and integer variables, or a mix of those variables.

1.5 GENETIC ALGORITHM

The rest of this book is devoted to the relatively new optimization technique called the *genetic algorithm* (GA). GAs were introduced by Holland [31] and were applied to many practical problems by Goldberg [32]. A GA has several advantages over the traditional numerical optimization approaches presented in this chapter, including the facts that it

1. Optimizes with continuous or discrete parameters.
2. Doesn't require derivative information.
3. Simultaneously searches from a wide sampling of the cost surface.
4. Works with a large number of variables.
5. Is well suited for parallel computers.
6. Optimizes variables with extremely complex cost surfaces.
7. Provides a list of optimum parameters, not just a single solution.
8. May encode the parameters, and the optimization is done with the encoded parameters.
9. Works with numerically generated data, experimental data, or analytical functions.

These advantages will become clear as the power of the GA is demonstrated in the following chapters. Chapter 2 explains the GA in detail. Chapter 3 gives a step-by-step analysis of finding the minimum of AF_4. Many other more complex examples are presented in Chapters 3–8. For further enlightenment on GAs, please turn to the next chapter.

REFERENCES

1. J. S. Stone, US Patents 1,643,323 and 1,715,433 C.L.
2. C. L. Dolph, A current distribution for broadside arrays which optimizes the relationship between beam width and side-lobe level, *Proc. IRE* **34**:335–348 (June 1946).
3. T. T. Taylor, Design of line source antennas for narrow beamwidth and low side lobes, *IRE AP Trans.* **4**:16–28 (Jan. 1955).
4. R. S. Elliott, Antenna *Theory and Design*, Prentice-Hall, New York, 1981.
5. A. T. Villeneuve, Taylor patterns for discrete arrays, *IEEE AP-S Trans.* **32**(10): 1089–1094 (Oct. 1984).
6. E. T. Bayliss, Design of monopulse antenna difference patterns with low sidelobes, *Bell Syst. Tech. J.* **47**:623–650 (May–June 1968).
7. W. W. Hansen and J. R. Woodyard, A new principle in directional antenna design, *Proc. IRE* **26**:333–345 (March 1938).
8. W. L. Stutzman and E. L. Coffey, Radiation pattern synthesis of planar antennas using the iterative sampling method, *IEEE Trans.* **AP-23**(6):764–769 (Nov. 1975).
9. H. J. Orchard, R. S. Elliot, and G. J. Stern, Optimizing the synthesis of shaped beam antenna patterns, *IEE Proc.* **132**(1):63–68 (Feb. 1985).
10. R. S. Elliot and G. J. Stearn, Shaped patterns from a continuous planar aperture distribution, *IEEE Proc.* **135**(6):366–370 (Dec. 1988).
11. F. Ares, R. S. Elliott, and E. Moreno, Design of planar arrays to obtain efficient footprint patterns with an arbitrary footprint boundary, *IEEE AP-S Trans.* **42**(11):1509–1514 (Nov. 1994).

12. D. K. Cheng, Optimization techniques for antenna arrays, *Proc. IEEE* **59**(12):1664–1674 (Dec. 1971).
13. J. F. DeFord and O. P. Gandhi, Phase-only synthesis of minimum peak sidelobe patterns for linear and planar arrays, *IEEE AP-S Trans.* **36**(2):191–201 (Feb. 1988).
14. J. E. Richie and H. N. Kritikos, Linear program synthesis for direct broadcast satellite phased arrays, *IEEE AP-S Trans.* **36**(3):345–348 (March 1988).
15. M. I. Skolnik, G. Nemhauser, and J. W. Sherman, III, Dynamic programming applied to unequally spaced arrays, *IEEE AP-S Trans.* **12**:35–43 (Jan. 1964).
16. J. Perini, Note on antenna pattern synthesis using numerical iterative methods, *IEEE AP-S Trans.* **12**:791–792 (July 1976).
17. Y. T. Lo, A mathematical theory of antenna arrays with randomly spaced elements, *IEEE AP-S Trans.* **12**(3):257–268 (May 1964).
18. W. H. Press et al., *Numerical Recipes*, Cambridge Univ. Press, New York, 1992.
19. G. Luenberger, *Linear and Nonlinear Programming*, Addison-Wesley, Reading, MA, 1984.
20. H. H. Rosenbrock, An automatic method for finding the greatest or least value of a function, *Comput. J.* **3**:175–184 (1960).
21. M. J. D. Powell, An efficient way for finding the minimum of a function of several variables without calculating derivatives, *Comput. J.* 155–162 (1964).
22. G. C. Broyden, A class of methods for solving nonlinear simultaneous equations, *Math. Comput.* 577–593 (Oct. 1965).
23. R. Fletcher, Generalized inverses for nonlinear equations and optimization, in *Numerical Methods for Non-linear Algebraic Equations*, R. Rabinowitz, ed., Gordon & Breach, London, 1963.
24. D. Goldfarb and B. Lapidus, Conjugate gradient method for nonlinear programming problems with linear constraints, *I&EC Fund.* 142–151 (Feb. 1968).
25. D. F. Shanno, An accelerated gradient projection method for linearly constrained nonlinear estimation, *SIAM J. Appl. Math.* 322–334 (March 1970).
26. J. R. Shewchuk, *An Introduction to the Conjugate Gradient Method without the Agonizing Pain*, Technical Report CMU-CS-94-125, Carnegie Mellon Univ. 1994.
27. J. A. Nelder and R. Mead, *Comput. J.* **7**:308–313 (1965).
28. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, Optimization by simulated annealing, *Science* **220**:671–680 (May 13, 1983).
29. N. Metropolis et al., Equation of state calculations by fast computing machines, *J. Chem. Phys.* **21**:1087–1092 (1953).
30. T. J. Cornwell, A novel principle for optimization of the instantaneous Fourier plane coverage of correction arrays, *IEEE AP-S Trans.* **36**(8):1165–1167 (Aug. 1988).
31. J. H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. Michigan Press, Ann Arbor, 1975.
32. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, New York, 1989.
33. R. L. Haupt and Sue Ellen Haupt, *Practical Genetic Algorithms*, 2nd ed., Wiley, New York, 2004.