8 Optimization Methods

The purpose of this chapter is to bring together a selection of algorithms for optimizing linear and nonlinear functions that have applications in science and engineering. We deal with constrained linear optimization problems and both constrained and unconstrained nonlinear optimization problems.

8.1 Introduction

The major techniques of optimization considered in this chapter are:

1. The solution of linear programming problems by interior point methods
2. The optimization of single-variable nonlinear functions
3. The solution of nonlinear optimization problems and systems of linear equations using conjugate gradient methods
4. The solution of constrained nonlinear optimization problems using the sequential unconstrained minimization technique (SUMT)
5. The solution of nonlinear optimization problems using the genetic algorithm and the method of simulated annealing

It is not our intention to describe fully the theoretical basis for these methods but to give some indication of the ideas that lie behind them. We begin with a discussion of linear programming problems.

8.2 Linear Programming Problems

Linear programming is normally considered to be an operational research (OR) method but has a very wide range of applications. A detailed description of the problem and associated theory is beyond the scope of this text but this information can be obtained from Dantzig (1963) and Sultan (1993). The problem may be expressed in standard form as

$$\begin{aligned} \text{Minimize } f &= \mathbf{c}^\top \mathbf{x} \\ \text{subject to } \mathbf{A}\mathbf{x} &= \mathbf{b} \\ \text{and } \mathbf{x} &\geq \mathbf{0} \end{aligned} \tag{8.1}$$

Numerical Methods Using MATLAB®. DOI: 10.1016/B978-0-12-386942-5.00008-4

where $\mathbf{x}$ is the column vector of n components that we wish to determine. Note that each element of $\mathbf{x}$ is constrained to be greater than zero. This is a common requirement in this type of optimization because most practical optimization problems will require nonnegative values for $\mathbf{x}$. For example, if each element of $\mathbf{x}$ is the number of workers of a particular skill set employed by an organization, the number of workers in any group cannot be negative. The given constants of the system are provided by an m component column vector $\mathbf{b}$, an $m \times n$ matrix $\mathbf{A}$, and an n component column vector $\mathbf{c}$. Clearly all the equations and the function we wish to minimize are linear in form. The problem is an optimization problem and in general it represents the requirement to minimize a linear function, $\mathbf{c}^\top \mathbf{x}$, called the objective function, subject to satisfying a system of linear equalities.

The importance of this type of problem lies in the fact that it corresponds to the general aim of optimizing the use of scarce resources to meet a specific objective. Although we have given the standard form, many other forms of this problem arise that are easily converted to this standard form. For example, the constraints may initially be inequalities and these can be converted to equalities by adding or subtracting additional variables introduced into the problem. The objective may be to maximize the function rather than minimize it. Again this is easily converted by changing the sign of the $\mathbf{c}$ vector.

Some practical examples where linear programming has been applied are

1. The hospital diet problem, requiring food costs to be minimized while dietary constraints are satisfied
2. The problem of minimizing cutting pattern loss
3. The problem of optimizing profit subject to constraints on the availability of specified materials
4. The problem of optimizing the routing of telephone calls

An important numerical algorithm for solving this problem is called the simplex method; see Dantzig (1963). This was applied to wartime problems of troop and material distribution. However, here we consider more recent developments that have provided new algorithms that are theoretically better. These are based on the work of Karmarkar (1984), who produced an algorithm that differed greatly in principle from that of Dantzig. While the theoretical complexity of Dantzig's method is exponential in the number of variables of the problem, some versions of Karmarkar's algorithm have a complexity that is of the order of the cube of the number of variables. It has been reported that for some problems this leads to substantial saving of computational effort. Here we describe an algorithm due to Barnes (1986) that provides an elegant modification of Karmarkar's algorithm but preserves its fundamental principles.

We do not describe the theoretical details of these complex algorithms but it is useful to compare, in broad terms, the nature of the Karmarkar and Dantzig algorithms. The simplex method of Dantzig is best illustrated by considering a simple linear programming problem as follows. In a factory producing electronic components, let x_1 be the number of batches of resistors and x_2 the number of batches of capacitors produced. Each batch of resistors manufactured gains 7 units of profit and each batch of capacitors gains 13 units of profit. Each is manufactured in a two-stage process. Stage 1 is limited to 18 units of time

per week and stage 2 is limited to 54 units of time per week. A batch of resistors requires 1 unit of time in stage 1 and 5 units of time in stage 2. A batch of capacitors requires 3 units of time in the first stage and 6 in the second. The aim of the manufacturer is to maximize profitability while meeting the time constraints; this leads to the following linear programming problem.

$$\text{Maximize } z = 7x_1 + 13x_2 \text{ (where } z \text{ is the profit)}$$

subject to

$$\begin{aligned} x_1 + 3x_2 &\leq 18 \qquad \text{(stage 1 process)} \\ 5x_1 + 6x_2 &\leq 54 \qquad \text{(stage 2 process)} \\ \text{and} \quad x_1, x_2 &\geq 0 \end{aligned}$$

To see how the simplex algorithm works we give a geometric interpretation of this problem in Figure 8.1. In this figure the region lying under the shaded lines and confined by the x_1- and x_2-axes represents the feasible region. This is the region in which all possible solutions to the problem lie. Clearly there is an infinity of such points. Fortunately, it can be shown that the only true candidates for the optimum solution are the points that lie at the vertices of the feasible region. In fact, we can find this optimum using simple geometric principles. The objective function is represented in Figure 8.1 by the dashed line of constant slope and variable intercept proportional to the value of the objective function. If we move this line parallel to itself until it just leaves the feasible region, it leaves at the vertex that gives the maximum value of the objective function. Clearly, beyond this point the values of x_1 and x_2 no longer satisfy the constraints. For this problem the optimum solution is given by $x_1 = 6$, $x_2 = 4$ so that the profit $z = 94$.

Although this provides a solution for this simple two-variable problem, linear programming problems often involve thousands or hundreds of thousands of variables. For practical problems a well-specified numerical algorithm is required. This is provided by

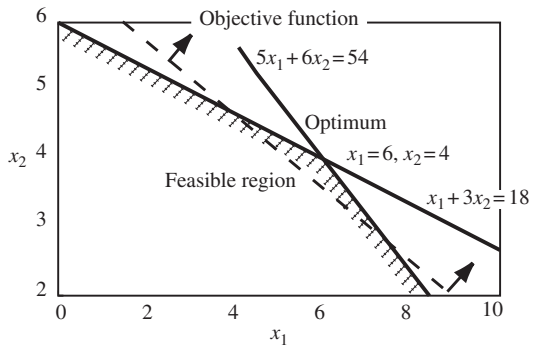

FIGURE 8.1 Graphical representation of an optimization problem. The *dashed line* represents the objective function and the *solid lines* represent the constraints.

Dantzig's simplex algorithm. We do not describe this in detail here but the general principle of its operation is to generate a sequence of points that correspond mathematically to the vertices of the multidimensional feasible region. The algorithm proceeds from one vertex to another, each time improving the value of the objective function, until the optimum is found. These points are all on the surface of the feasible region and for larger problems there may be a huge number of them.

The algorithm proposed by Karmarkar deals with the linear programming problem in a different way. The algorithm was developed at AT&T to solve very large linear programming problems concerned with routing telephone calls in the Pacific Basin. This algorithm transforms the problem to a more convenient form and then searches through the interior of the feasible region using a good direction of search toward its surface. Because this type of algorithm uses interior points, it is often described as an *interior point* method. Since its discovery, many improvements and modifications have been made to this algorithm and here we describe a form which, although conceptually complex, leads to a remarkably simple and elegant linear programming algorithm. This formulation was given by Barnes (1986).

The Barnes algorithm may be applied to any linear programming problem once it is converted to the form of (8.1). However, one important initial modification is required to ensure the algorithm starts at an interior point $\mathbf{x}^0 > \mathbf{0}$. This modification is achieved by introducing an additional column, that is, a new last column, to the $\mathbf{A}$ matrix, the elements of which are the $\mathbf{b}$ vector minus the sum of the columns of the $\mathbf{A}$ matrix. We associate an additional variable with this additional column and, in order that we do not have a superfluous variable in the solution, we introduce an extra element in the vector $\mathbf{c}$. We make the value of this element very large to ensure that the new variable is driven to zero when the optimum is reached. Now we find that $\mathbf{x}^0 = [1\ 1\ 1\ \ldots\ 1]^\top$ satisfies this set of constraints and clearly $\mathbf{x}^0 \geq \mathbf{0}$. We now describe the Barnes algorithm:

- *Step* 0: Assuming n variables in the original problem,

$$\text{set } a(i,n+1) = b(i) - \sum_j a(i,j) \quad \text{and} \quad c(n+1) = 10000$$

$$\mathbf{x}^0 = [1\ 1\ 1\ \ldots\ 1], \qquad k = 0$$

- *Step* 1: Set $\mathbf{D}^k = \text{diag}(\mathbf{x}^k)$ and compute an improved point using the equation

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \frac{s(\mathbf{D}^k)^2(\mathbf{c} - \mathbf{A}^\top \lambda^k)}{\text{norm}(\mathbf{D}^k(\mathbf{c} - \mathbf{A}^\top \lambda^k))}$$

  where the vector λ^k is given by

$$\lambda^k = (\mathbf{A}(\mathbf{D}^k)^2\mathbf{A}^\top)^{-1}\mathbf{A}(\mathbf{D}^k)^2\mathbf{c}$$

The step s is chosen such that

$$s = \min \left\{ \frac{\text{norm}\left(\left(\mathbf{D}^k\right)\left(\mathbf{c} - \mathbf{A}^\top \lambda^k\right)\right)}{x_j^k \left(c_j - \mathbf{A}_j^\top \lambda^k\right)} \right\} - \alpha$$

where $\mathbf{A}_j$ is the jth column of the matrix $\mathbf{A}$ and α is a small preset constant value. Here the minimum is taken for the values

$$\left(c_j - \mathbf{A}_j^\top \lambda^k\right) > 0 \text{ only}$$

Note also that λ^k provides an approximation for the solution of the dual problem (see for example, Problems 8.1 and 8.2).

- *Step* 2: Stop if the *primal* and *dual* values of the objective functions are approximately equal. Else set $k = k + 1$ and repeat from step 1.

Note that in step 2 we use an important result in linear programming. This is that every primal problem (i.e., the original problem) has a corresponding dual problem and if a solution exists, the optimal values of their objective functions are equal. There are several other termination criteria that could be used and Barnes suggested a more complex but more reliable one.

The algorithm provides an iterative improvement starting from the initial point $\mathbf{x}^0$ by taking the maximum step that ensures that $\mathbf{x}^k > \mathbf{0}$ in the normalized direction given by $(\mathbf{D}^k)^2(\mathbf{c} - \mathbf{A}^\top \lambda^k)$. It is this direction that is the crucial element of the algorithm. This direction is a projection of the objective function coefficients into the constraint space. For a proof that this direction reduces the objective function, while ensuring the constraints are satisfied, the reader is referred to Barnes (1986).

The reader should be warned that this algorithm is deceptively simple. In fact, the computation of the direction is very difficult for large problems. This is because the algorithm requires the solution of an extremely ill-conditioned equation system. Many alternatives have been suggested for finding the direction of search, including the use of a conjugate gradient method that is discussed in Section 8.6. The MATLAB function `barnes` provided here solves the ill-conditioned equation system in a direct manner using the MATLAB \ operator. The function `barnes` is easily modified to use the conjugate gradient solver given in Section 8.6.

```
function [xsol,basic,objective] = barnes(A,b,c,tol)
% Barnes' method for solving a linear programming problem
% to minimize c'x subject to Ax = b. Assumes problem is non-degenerate.
% Example call: [xsol,basic]=barnes(A,b,c,tol)
% A is the matrix of coefficients of the constraints.
% b is the right-hand side column vector and c is the row vector of
% cost coefficients. xsol is the solution vector, basic is the
```

```
% list of basic variables.
x2 = [ ];  x = [ ];
[m n] = size(A);
% Set up initial problem
aplus1 = b-sum(A(1:m,:)')';
cplus1 = 1000000;
A = [A aplus1]; c = [c cplus1]; B = [ ];
n = n+1;
x0 = ones(1,n)'; x = x0;
alpha = .0001; lambda = zeros(1,m)';
iter = 0;
% Main step
while abs(c*x-lambda'*b)>tol
    x2 = x.*x;
    D = diag(x); D2 = diag(x2); AD2 = A*D2;
    lambda = (AD2*A')\(AD2*c');
    dualres = c'-A'*lambda;
    normres = norm(D*dualres);
    for i = 1:n
        if dualres(i)>0
            ratio(i) = normres/(x(i)*(c(i)-A(:,i)'*lambda));
        else
            ratio(i)=inf;
        end
    end
    R = min(ratio)-alpha;
    x1 = x-R*D2*dualres/normres;
    x = x1;
    basiscount = 0;
    B = [ ]; basic = [ ];
    cb = [ ];
    for k = 1:n
        if x(k)>tol
            basiscount = basiscount+1;
            basic = [basic k];
        end
    end
    % Only used if problem non-degenerate
    if basiscount==m
        for k = basic
            B = [B A(:,k)];   cb = [cb c(k)];
        end
```

```
            primalsol = b'/B';
            xsol = primalsol;
            break
        end
        iter = iter+1;
    end
    objective = c*x;
```

We now solve the linear programming problem

$$\text{Maximize } z = 2x_1 + x_2 + 4x_3$$

subject to

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 7 \\ x_1 + 2x_2 + 3x_3 &\leq 12 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

The requirements that $x_1, x_2, x_3 \geq 0$ are called nonnegativity constraints. This linear programming problem can be easily transformed to the standard form by adding new positive-valued variables, called slack variables, to the left sides of the inequalities and changing the signs of the coefficients in the objective function so that it is converted to a minimization problem subject to equality constraints as follows:

$$\text{Minimize } -z = -(2x_1 + x_2 + 4x_3)$$

subject to

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &= 7 \\ x_1 + 2x_2 + 3x_3 + x_5 &= 12 \\ x_1, x_2, x_3, x_4, x_5 &\geq 0 \end{aligned}$$

The variables x_4 and x_5 are called the slack variables and they represent the difference between the available resources and the resources used. Note that if the constraints were of the form greater than or equal to zero, we would subtract slack variables to produce equality. These subtracted variables are sometimes called surplus variables. Thus we have

$$\mathbf{c} = \begin{bmatrix} -2 & -1 & -4 & 0 & 0 \end{bmatrix}$$

We use the following script to solve this problem.

```
% e3s801.m
c = [-2 -1 -4 0 0];
A = [1 1 1 1 0;1 2 3 0 1 ]; b = [7 12]';
[xsol,ind,object] = barnes(A,b,c,0.00005);
```

```
fprintf('objective = %8.4f', object)
i = 1;
fprintf('\nSolution is:');
for j = ind
    fprintf('\nx(%1.0f) =%8.4f',j,xsol(i))
    i = i+1;
end;
fprintf('\nAll other variables are zero\n')
```

Running this script provides the result

```
objective = -19.0000
Solution is:
x(1) =  4.5000
x(3) =  2.5000
All other variables are zero
```

Since the original problem was to maximize objective function, its value is 19. This solution illustrates an important theorem of linear programming. The number of nonzero primal variables is at most equal to the number of independent constraints (excluding nonnegativity constraints). In this problem there are only two main constraints. Thus there are only two nonzero variables, x_1 and x_3. The slack variables x_4 and x_5 are zero and so is x_2.

The `lsqnonneg` function, discussed in Section 2.12, provides a method for finding a solution to an equation system in which all components of the solution are nonnegative. This corresponds to a basic feasible solution for the system but it is generally nonoptimum for a specific objective function.

Having examined the process for solving linear optimization problems, we now consider methods that are used to solve nonlinear optimization problems.

8.3 Optimizing Single-Variable Functions

We sometimes need to determine the maximum or minimum value of a one-variable nonlinear function. Throughout this discussion we assume we are seeking the minimum value of the function. If we require the maximum value, then we merely have to change the sign of the original function.

The most obvious way of determining the minimum of a function is to differentiate it and find the value of the independent variable that makes this derivative zero. However, there are situations in which it is not practical to find the derivative directly; see for example (8.4). A method is now described that provides an approximation to the minimum to any required accuracy.

Consider a function $y = f(x)$ and let us assume that in the range $[x_a \;\; x_b]$ there is a single minimum, as shown in Figure 8.2. Two additional points, x_1 and x_2, are chosen arbitrarily so that the range is divided into three intervals. Assuming that $x_a < x_1 < x_2 < x_b$, then

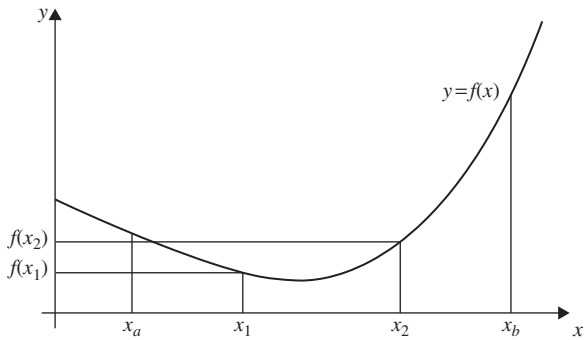

FIGURE 8.2 Graph of a function with a minimum in the range $[x_a\, x_b]$.

If $f(x_1) < f(x_2)$ then the minimum value must lie in the range $[x_a\ x_2]$.

If $f(x_1) > f(x_2)$ then the minimum value must lie in the range $[x_1\ x_b]$.

Either of these ranges must provide a smaller interval than $[x_a\ \ x_b]$ in which the minimum lies. This interval reduction process can be repeated continuously in successively smaller ranges until an acceptably small interval is found for the minimum.

It might be assumed that the most efficient procedure is to select x_1 and x_2 so that the range $[x_a\ x_b]$ is subdivided into three equal intervals. In fact, this is not so and for a more efficient procedure we take

$$x_1 = x_a + r(1-g), \quad x_2 = x_a + rg$$

where $r = x_b - x_a$ and

$$g = \frac{1}{2}\left(-1+\sqrt{5}\right) \approx 0.61803$$

The quantity g is called the *golden ratio*. This quantity has many interesting properties. For example, it is one of the roots of the equation

$$x^2 + x + 1 = 0$$

This golden ratio is also related to the famous Fibonacci series. This series is $1, 1, 2, 3, 5, 8, 13, \ldots$ and it is generated from

$$N_{k+1} = N_k + N_{k-1}, \quad k = 2, 3, 4, \ldots$$

where $N_2 = N_1 = 1$ and N_k is the kth term in the series. As k tends to infinity the ratio N_k/N_{k+1} tends to the golden ratio.

The algorithm is implemented in MATLAB as follows:

```
function [f,a,iter] = golden(func,p,tol)
% Golden search for finding min of one variable nonlinear function.
% Example call: [f,a] = golden(func,p,tol)
% func is the name of the user defined nonlinear function.
% p is a 2 element vector giving the search range.
% tol is the tolerance. a is the optimum value of the function.
% f is the minimum of the function. iter is the number of iterations
if p(1)<p(2)
    a = p(1);  b = p(2);
else
    a = p(2);  b = p(1);
end
g = (-1+sqrt(5))/2;
r = b-a;  iter = 0;
while r>tol
    x = [a+(1-g)*r a+g*r];
    y = feval(func,x);
    if y(1)<y(2)
        b = x(2);
    else
        a = x(1);
    end
    r = b-a; iter = iter+1;
end
f = feval(func,a);
```

We can use the function `golden` to search for the minimum value of the Bessel function of the second kind of order 2. The function `bessely(2,x)` is provided by MATLAB. The following command provides the output shown:

```
>> format long
>> [f,x,iter] = golden(@(x) bessely(2,x),[4 10],0.000001)

f =
  -0.279275263440711

x =
   8.350724427010965

iter =
    33
```

Note that if we had divided the search interval into three equal sections, rather than using the golden ratio, then 39 iterations would have been required.

The search algorithm has been developed assuming that there is only one minimum value of the function in the search range. If there are several minima in the search range, then the procedure locates one, but the one located is not necessarily the global minimum in the range. For example, a Bessel function of the second kind of order 2 has 3 minima in the range 4 to 25, as shown in Figure 8.3.

If we use the function `golden` and search in the ranges 4 to 24, 4 to 25, and 4 to 26, we obtain the results given in Table 8.1. In this table we see that a different minimum has been found when using different search ranges, even though all three minima are in each of the search ranges used. Ideally we require the search to determine the global minimum of the function, something that the function `golden` has failed to accomplish in two out of three tests. Obtaining a *global* solution is a major problem in minimization.

In this particular example we can verify the accuracy of these solutions by calculus. The derivative of the Bessel function of the second kind of order n is given by

$$\frac{d}{dx}\{Y_n(x)\} = \frac{1}{2}\{Y_{n-1}(x) - Y_{n+1}(x)\}$$

where $Y_n(x)$ is the Bessel function of the second kind of order n. (Sometimes $N_n(x)$ is used instead of $Y_n(x)$.)

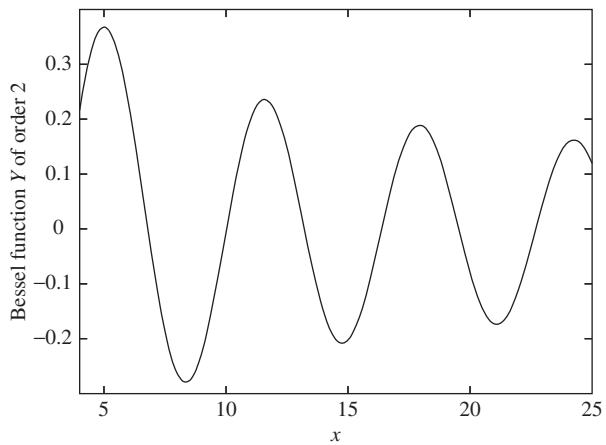

FIGURE 8.3 A plot of the Bessel function of the second kind showing three minima.

Table 8.1 Effect of Different Search Ranges

Search Range	Value of $f(x)$ at Min	Value of x at Min
4 to 24	−0.20844576503764	14.76085144779431
4 to 25	−0.17404548213116	21.09284729991696
4 to 26	−0.27927526323841	8.35068549680869

The minimum (or maximum) value of a function occurs when the derivative of the function is zero. Hence, with $n = 2$ we have a minimum (or maximum) occurring when

$$Y_1(x) - Y_3(x) = 0$$

We cannot escape from the need to use numerical methods because the only way to find the roots of this equation is to use a numerical procedure such as that implemented in the MATLAB function `fzero` (see Chapter 3). Thus, using this function to determine a root near 8 we have

```
>> format long
>> fzero(@(x) bessely(1,x)-bessely(3,x),8)

ans =
   8.350724701413078
```

We can also use `fzero` to find roots at 14.76090930620768 and 21.09289450441274. These results are in good agreement with the minima found using the function `golden`.

8.4 The Conjugate Gradient Method

Here we confine ourselves to solving the problem

$$\text{Minimize } f(\mathbf{x}) \quad \text{for all} \quad \mathbf{x} \in \mathrm{R}^{\mathrm{n}}$$

where $f(\mathbf{x})$ is a nonlinear function of $\mathbf{x}$ and $\mathbf{x}$ is an n-component column vector. This is called a nonlinear unconstrained optimization problem. These problems arise in many applications–for example, in neural network problems where an important aim is to find weights in a network that minimize the difference between the output of the network and the required output.

The standard approach for solving this problem is to assume an initial approximation $\mathbf{x}^0$ and then to proceed to an improved approximation by using an iterative formula of the form

$$\mathbf{x}^{k+1} = \mathbf{x}^k + s\mathbf{d}^k \quad \text{for} \quad k = 0,\ 1,\ 2, \ldots \tag{8.2}$$

Clearly, to use this formula we must determine values for the scalar s and the vector $\mathbf{d}^k$. The vector $\mathbf{d}^k$ represents a direction of search and the scalar s determines how far we should step in this direction. A vast literature has grown up that has examined the problem of choosing the best direction and the best step size to solve this problem efficiently. For example, see Adby and Dempster (1974). A simple choice for a direction of search is to take $\mathbf{d}^k$ as the negative gradient vector at the point $\mathbf{x}^k$. For a sufficiently small step value this can be shown to guarantee a reduction in the function value. This leads to an

algorithm of the form

$$\mathbf{x}^{k+1} = \mathbf{x}^k - s\nabla f\left(\mathbf{x}^k\right) \quad \text{for} \quad k = 0,\ 1,\ 2,\ \ldots \tag{8.3}$$

where $\nabla f(\mathbf{x}) = (\partial f/\partial x_1, \partial f/\partial x_2, \ldots, \partial f/\partial x_n)$ and s is a small constant value. This is called the steepest descent algorithm. The minimum is reached when the gradient is zero, as in the ordinary calculus approach. We also assume that there exists only one local minimum that we wish to find in the range considered. The problem with this method is that although it reduces the function value, the step may be very small and therefore the algorithm is very slow. An alternative approach is to choose the step that gives the maximum reduction in the function value in the current direction. This may be described formally as

$$\text{For each } k \text{ find the value of } s \text{ that minimizes } f(\mathbf{x}^k - s\nabla f(\mathbf{x}^k)) \tag{8.4}$$

This procedure is known as a line search. The reader will note that this is also a minimization problem. However, since $\mathbf{x}^k$ is known, it is a *one-variable* minimization problem in the step size s. Although it is a difficult problem, numerical procedures are available to solve it, one of which is the search method given in Section 8.3. Equations (8.3) and (8.4) provide a workable algorithm but it is still slow. One reason for this poor performance lies in our choice of direction $-\nabla f(\mathbf{x}^k)$.

Consider the function we wish to minimize in (8.4). Clearly the value of s that minimizes $f(\mathbf{x}^k - s\nabla f(\mathbf{x}^k))$ is such that the derivative of $f(\mathbf{x}^k - s\nabla f(\mathbf{x}^k))$ with respect to s is zero. Now, differentiating $f(\mathbf{x}^k - s\nabla f(\mathbf{x}^k))$ with respect to s gives

$$\frac{df\left(\mathbf{x}^k - s\nabla f\left(\mathbf{x}^k\right)\right)}{ds} = -\left(\nabla f\left(\mathbf{x}^{k+1}\right)\right)^\top \nabla f\left(\mathbf{x}^k\right) = 0 \tag{8.5}$$

This shows that the successive directions of search are orthogonal. This is not the best way of getting from our original approximation to the optimum value since the changes in direction are so large.

The conjugate gradient method takes a combination of the previous direction and the new direction to approach the optimum more directly. It uses the same step size choice procedure given by (8.4), so we must now consider how the direction vector is chosen in the conjugate gradient method. Let $\mathbf{g}^{k+1} = \nabla f(\mathbf{x}^{k+1})$ so that the basic formula for the conjugate gradient direction is

$$\mathbf{d}^{k+1} = -\mathbf{g}^{k+1} + \beta\mathbf{d}^k \tag{8.6}$$

Thus the current direction of search is a combination of the current negative gradient plus a scalar β times the previous direction of search. The crucial question is: How is the value of β to be determined? The criterion used is that successive directions of search should be conjugate. This means that $(\mathbf{d}^{k+1})^\top \mathbf{A}\,\mathbf{d}^k = 0$ for some specified matrix $\mathbf{A}$.

This apparently obscure choice of requirement can be shown to lead to desirable convergence properties for the conjugate gradient method. In particular it has the property

that the optimum of a positive definite quadratic function of n variables can be found in n or fewer steps. In the case of a quadratic, $\mathbf{A}$ is the matrix of coefficients of the squared and cross-product terms. It can be shown that the requirement of conjugacy leads to a value for β given by

$$\beta = \frac{\left(\mathbf{g}^{k+1}\right)^{\top}\mathbf{g}^{k+1}}{\left(\mathbf{g}^{k}\right)^{\top}\mathbf{g}^{k}} \tag{8.7}$$

Now (8.2), (8.4), (8.6), and (8.7) lead to the conjugate gradient algorithm given by Fletcher and Reeves (1964), which has the form

- *Step* 0: Input value for $\mathbf{x}^0$ and accuracy ε. Set $k = 0$ and compute $\mathbf{d}^k = -\nabla f(\mathbf{x}^k)$.
- *Step* 1: Determine s_k, which is the value of s that minimizes $f(\mathbf{x}^k + s\mathbf{d}^k)$.
  Calculate $\mathbf{x}^{k+1}$ where $\mathbf{x}^{k+1} = \mathbf{x}^k + s_k\mathbf{d}^k$ and compute $\mathbf{g}^{k+1} = \nabla f(\mathbf{x}^{k+1})$.
  If $\text{norm}(\mathbf{g}^{k+1}) < \varepsilon$, then terminate with solution $\mathbf{x}^{k+1}$, else go to step 2.
- *Step* 2: Calculate new conjugate direction $\mathbf{d}^{k+1}$ where

$$\mathbf{d}^{k+1} = -\mathbf{g}^{k+1} + \beta\mathbf{d}^k \quad \text{and} \quad \beta = (\mathbf{g}^{k+1})^{\top}\mathbf{g}^{k+1}/\{(\mathbf{g}^k)^{\top}\mathbf{g}^k\}$$

- *Step* 3: $k = k + 1$; go to step 1.

Note that in other forms of this algorithm steps 1, 2, and 3 are repeated n times and then restarted with a steepest descent step from step 0. The following is a MATLAB function for this method.

```
function [x1,df,noiter] = mincg(f,derf,ftau,x,tol)
% Finds local min of a multivariable nonlinear function in n variables
% using conjugate gradient method.
% Example call: res = mincg(f,derf,ftau,x,tol)
% f is a user defined multi-variable function,
% derf a user defined function of n first order partial derivatives.
% ftau is the line search function.
% x is a col vector of n starting values, tol gives required accuracy.
% x1 is solution, df is the gradient,
% noiter is the number of iterations required.
% WARNING. Not guaranteed to work with all functions. For difficult
% problems the linear search accuracy may have to be adjusted.
global p1 d1
n = size(x);  noiter = 0;
% Calculate initial gradient
df = feval(derf,x);
% main loop
```

```
while norm(df)>tol
    noiter = noiter+1;
    df = feval(derf,x);
    d1 = -df;
    %Inner loop
    for inner = 1:n
        p1 = x;  tau = fminbnd(ftau,-10,10);
        % calculate new x
        x1 = x+tau*d1;
        % Save previous gradient
        dfp = df;
        % Calculate new gradient
        df = feval(derf,x1);
        % Update x and d
        d = d1; x = x1;
        % Conjugate gradient method
        beta = (df'*df)/(dfp'*dfp);
        d1 = -df+beta*d;
    end
end
```

Notice that the MATLAB function `fminbnd` is used in the function `mincg` to perform the single-variable minimization to find the best step value. It is important to note that the function `mincg` requires three input functions, which must be supplied by the user. They are the function to be minimized, the partial derivatives of this function, and the line-search function. As implemented, the function `mincg` requires the input functions to be user-defined functions, not anonymous functions. An example of the use of `mincg` follows.

The function to be minimized, which is taken from Styblinski and Tang (1990), is

$$f(x_1,x_2) = \left(x_1^4 - 16x_1^2 + 5x_1\right)\Big/2 + \left(x_2^4 - 16x_2^2 + 5x_2\right)\Big/2$$

The function `f01` and the derivative of this function, `f01d`, are defined as follows:

```
function f = f01(x)
f = 0.5*(x(1)^4-16*x(1)^2+5*x(1)) + 0.5*(x(2)^4-16*x(2)^2+5*x(2));

function f = f01d(x)
f = [0.5*(4*x(1)^3-32*x(1)+5); 0.5*(4*x(2)^3-32*x(2)+5)];
```

The MATLAB line-search function `ftau2cg` is defined as

```
function ftauv = ftau2cg(tau);
global p1 d1
q1 = p1+tau*d1;
ftauv = feval('f01',q1);
```

To test the `mincg` function we use the following simple MATLAB commands:

```
>> [sol,grad,iter] = mincg('f01','f01d','ftau2cg',[1 -1]', .000005)
```

The results of executing these statements are

```
sol =
   -2.9035
   -2.9035

grad =
  1.0e-006 *
    0.0156
   -0.2357

iter =
     3
```

Note that

```
>> f = f01(sol)

f =
  -78.3323
```

This is the minimum value of the function determined by `mincg`. It is interesting to see the function that has been optimized and we provide both a three-dimensional and a contour plot of the function in Figures 8.4 and 8.5. The latter includes a plot of the iterates and shows the path taken to reach the optimum solution from a particular starting point. The script used to obtain these graphs is

```
% e3s802.m
clf
[x,y] = meshgrid(-4.0:0.2:4.0,-4.0:0.2:4.0);
z = 0.5*(x.^4-16*x.^2+5*x)+0.5*(y.^4-16*y.^2+5*y);
figure(1)
surfl(x,y,z)
axis([-4 4 -4 4 -80 20])
xlabel('x1'), ylabel('x2'), zlabel('z')
x1=[1 2.8121 -2.8167 -2.9047 -2.9035];
y1=[0.5 -2.0304 -2.0295 -2.9080 -2.9035];
figure(2)
contour(-4.0:0.2:4.0,-4.0:0.2:4.0,z,15);
xlabel('x1'), ylabel('x2')
hold on
```

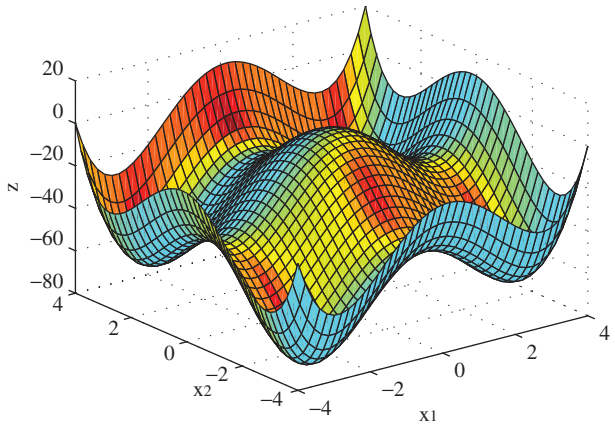

FIGURE 8.4 Three-dimensional plot of $f(x_1,x_2) = \left(x_1^4 - 16x_1^2 + 5x_1\right)/2 + \left(x_2^4 - 16x_2^2 + 5x_2\right)/2$.

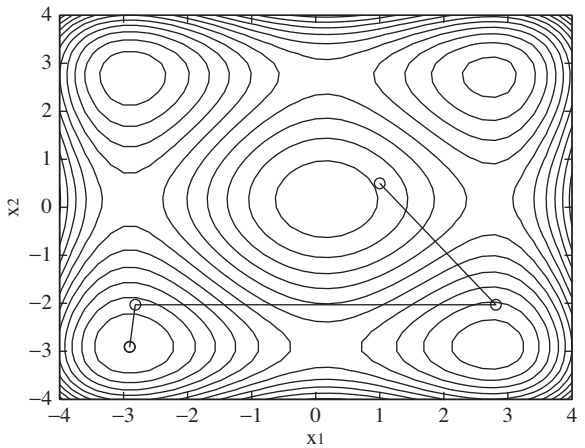

FIGURE 8.5 Contour plot of the function $f(x_1,x_2) = \left(x_1^4 - 16x_1^2 + 5x_1\right)/2 + \left(x_2^4 - 16x_2^2 + 5x_2\right)/2$ showing the location of four local minima. The conjugate gradient algorithm has found the one in the lower left corner. The search path taken by the algorithm is also shown.

```
plot(x1,y1,x1,y1,'o')
xlabel('x1'), ylabel('x2')
hold off
```

In this script the vectors `x1` and `y1` contain the iterates for the conjugate gradient solution of the given function. These values were obtained by running a modified version of the `mincg` function separately. The minimum we have obtained is in fact the smallest of the four local minima that exist for this function. However, this result was fortuitous; all that the conjugate gradient method is able to do is to find one of the four local minima and

even this is not guaranteed for all problems. The conjugate gradient method, because of its small storage requirements, is one of the key algorithms used in neural network problems as part of the back propagation algorithm, but it has many other applications.

It should be noted that a MATLAB optimization toolbox is available and this provides a range of optimization procedures.

8.5 Moller's Scaled Conjugate Gradient Method

In 1993 Moller, when working on optimization methods for neural networks, introduced a much improved version of Fletcher's conjugate gradient method. Fletcher's conjugate gradient method uses a line-search procedure to solve a single-variable minimization problem, which is then used to find the optimum step to take in the chosen direction of search. The procedure used by Fletcher is a fragile, iterative, and computationally intensive process. In addition, the line search depends on a number of parameters that must be estimated by the user. Moller's paper (Moller 1993) introduced a method that allowed the line-search procedure to be replaced by a considerably simplified method for estimating an acceptable step size. However, using a simple estimation of the step size often fails and leads to nonstationary points. Moller noted that a simple approach to the problem fails because it only works for functions with positive definite matrices. Consequently, Moller suggested a method based on a combination of the Levenberg-Marquardt algorithm and the conjugate gradient algorithm. An outline of the algorithm is described in the following; for the details the reader is referred to the original paper.

Consider the n-variable nonlinear function $f(\mathbf{x})$. Moller introduces a scalar parameter λ_k, which is adjusted at each iteration k after considering the sign of δ_k where

$$\delta_k = \mathbf{p}_k^\top \mathbf{H}_k \mathbf{p}_k$$

where $\mathbf{p}_k$ for $k = 1, 2, \ldots, n$ are a set of conjugate directions and $\mathbf{H}_k$ is the Hessian matrix of the function $f(\mathbf{x})$. If $\delta_k \geq 0$ then $\mathbf{H}_k$ is positive definite. However, since only first-order derivitive information is known at each step of the conjugate gradient method, Moller suggests that the Hessian multiplied by $\mathbf{p}_k$ is approximated by

$$\mathbf{s}_k = \frac{f'(\mathbf{x}_k + \sigma_k \mathbf{p}_k) - f'(\mathbf{x}_k)}{\sigma_k} \quad \text{for} \quad 0 < \sigma_k < 1$$

In practice the value of σ_k should be kept as small as possible for a good approximation. This expression in the limit tends to the true Hessian matrix multiplied by $\mathbf{p}_k$. The scalar λ_k is now introduced to regulate the approximation to the Hessian to ensure it is positive definite, specifically by using the equation

$$\mathbf{s}_k = \frac{f'(\mathbf{x}_k + \sigma_k \mathbf{p}_k) - f'(\mathbf{x}_k)}{\sigma_k} + \lambda_k \mathbf{p}_k \quad \text{for} \quad 0 < \sigma_k < 1$$

Thus the value of λ_k is adjusted, and then we check the value of δ_k defined earlier using the approximation to the Hessian; if this is negative then the Hessian is no longer positive definite and the value of λ_k is increased and s_k is checked again. This is repeated until the current estimate of the Hessian is positive definite. The key question is how should λ_k be adjusted to ensure the Hessian estimate becomes positive definite. Let the λ_k be increased to $\bar{\lambda}_k$; then

$$\bar{\mathbf{s}}_k = \mathbf{s}_k + (\bar{\lambda}_k - \lambda_k)\mathbf{p}_k$$

Now at any iteration k a new δ_k, which we can denote as $\bar{\delta}_k$, can be computed from

$$\bar{\delta}_k = \mathbf{p}_k^\top \bar{\mathbf{s}}_k = \mathbf{p}_k^\top (\mathbf{s}_k + (\bar{\lambda}_k - \lambda_k)\mathbf{p}_k) = \mathbf{p}_k^\top \mathbf{s}_k + (\bar{\lambda}_k - \lambda_k)\mathbf{p}_k^\top \mathbf{p}_k$$

But $\mathbf{p}_k^\top \mathbf{s}_k$ is the original value of δ_k before λ_k was increased. So we have

$$\bar{\delta}^k = \delta^k + (\bar{\lambda}_k - \lambda_k)\mathbf{p}_k^\top \mathbf{p}_k$$

Clearly we now require that the new value of $\bar{\delta}^k$ be positive; hence we require that

$$\delta_k + (\bar{\lambda}_k - \lambda_k)\mathbf{p}_k^\top \mathbf{p}_k > 0$$

This will be true if

$$\bar{\lambda}_k > \lambda_k - \frac{\delta_k}{\mathbf{p}_k^\top \mathbf{p}_k}$$

Moller suggests a reasonable choice of $\bar{\lambda}_k$ is

$$\bar{\lambda}_k = 2\left(\lambda_k - \frac{\delta^k}{\mathbf{p}_k^\top \mathbf{p}_k}\right)$$

It is easily verified by back-substitution of this value for $\bar{\lambda}_k$ in our expression for $\bar{\delta}_k$ that

$$\bar{\delta}_k = -\delta_k + \lambda_k \mathbf{p}_k^\top \mathbf{p}_k$$

which, since δ_k is negative, λ_k is positive, and $\mathbf{p}_k^\top \mathbf{p}_k$ is a sum of squares, is clearly positive as required. The step size estimate is based on a quadratic approximation to the function being optimized at the current step and is calculated from

$$\alpha_k = \frac{\mu_k}{\delta_k} = \frac{\mu_k}{\mathbf{p}_k^\top \mathbf{s}_k + \lambda_k \mathbf{p}_k^\top \mathbf{p}_k}$$

Here μ_k is the current negative gradient times the current direction of search $\mathbf{p}_k$. This gives the basis of the algorithm. However, an important issue still to be decided, is how the value of λ_k can be safely and systematically varied. Moller provides a method based on a measure

of how well the current quadratic approximation, defined as f_q, approximates the original function at the point considered. He does this by using the following definition:

$$\Delta_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)}{f(\mathbf{x}_k) - f_q(\alpha_k \mathbf{p}_k)}$$

By virtue of the fact that $f_q(\alpha_k\, \mathbf{p}_k)$ is a quadratic approximation at the current iteration, this can be shown to be equivalent to

$$\Delta_k = \frac{\delta_k^2 (f(\mathbf{x}_k) - f(\mathbf{x}_k + \alpha_k \mathbf{p}_k))}{\mu_k^2}$$

Now if Δ_k is close to 1 then the quadratic approximation $f_q(\alpha_k\, \mathbf{p}_k)$ must be close to $f(x_k + \alpha_k \mathbf{p}_k)$ and hence a good local approximation to the function. This leads to the following steps for the adjustment of λ_k. Use the definition of Δ_k described earlier as the quadratic approximation measure; more details can be found in Moller (1993). Then adjust λ_k as follows:

$$\begin{aligned} &\text{If } \Delta_k > 0.75 \quad \text{then} \quad \lambda_k = \lambda_k / 4 \\ &\text{If } \Delta_k < 0.25 \quad \text{then} \quad \lambda_k = \lambda_k + \frac{\delta_k (1 - \Delta_k)}{\mathbf{p}_k^\top \mathbf{p}_k} \end{aligned}$$

These steps, together with any of the methods for generating conjugate gradient directions of search, provide an algorithm with a simple line-search process. The outline of the major steps in Moller's algorithm are now given:

- *Step* 1: Choose the initial approximation $\mathbf{x}_0$ and initial values for $\sigma_i < 10^{-4}$, $\lambda_i < 10^{-4}$, and $\bar{\lambda}_i = 0$. These values were suggested by Moller. Calculate the initial negative gradient and assign it to $\mathbf{r}_1$ and assign $\mathbf{r}_1$ to the initial direction of search $\mathbf{p}_1$. Set $k = 1$.
- *Step* 2: Calculate second-order information. Specifically, calculate values for σ_k, $\bar{\mathbf{s}}_k$, and δ_k.
- *Step* 3: Scale δ_k using

$$\bar{\delta}^k = \delta^k + (\bar{\lambda}_k - \lambda_k) \mathbf{p}_k^\top \mathbf{p}_k$$

- *Step* 4: If $\delta_k < 0$ then make the Hessian approximation positive definite using

$$\bar{\delta}_k = -\delta_k + \lambda_k \mathbf{p}_k^\top \mathbf{p}_k$$

Set

$$\bar{\lambda}_k = 2 \left(\lambda_k - \frac{\delta^k}{\mathbf{p}_k^\top \mathbf{p}_k} \right)$$

and

$$\bar{\lambda}_k = \lambda_k$$

- *Step* 5: Calculate step size from

$$\alpha_k = \frac{\mu_k}{\delta_k}$$

- *Step* 6: Calculate the factor to test goodness of quadratic fit Δ_k from

$$\Delta_k = \frac{\delta_k^2(f(\mathbf{x}_k) - f(\mathbf{x}_k + \alpha_k \mathbf{p}_k))}{\mu_k^2}$$

- *Step* 7: If $\Delta_k \geq 0$ then the function can be reduced toward the minimum, so use

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

Calculate the new gradient

$$\mathbf{r}_{k+1} = -\nabla f(\mathbf{x}_{k+1})$$

Set $\bar{\lambda}_k = 0$. If $k \bmod N = 0$ then restart algorithm with

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1}$$

else calculate a new conjugate gradient direction.

Use some method to calculate the set of conjugate directions; see the Fletcher-Reeves (1964) for example. A number of other methods are available.

$$\text{If } \Delta_k \geq 0.75 \quad \text{then} \quad \lambda_k = 0.25\lambda_k$$

else

$$\bar{\lambda}_k = \lambda_k$$

- *Step* 8: If $\Delta_k < 0.25$ then increase the scale parameter:

$$\lambda_k = \lambda_k + (\delta_k(1 - \Delta_k)/\mathbf{p}_k^\top \mathbf{p}_k$$

- *Step* 9: If the gradient r_k is still not suffiently close to zero then set $k = k + 1$ and go to step 2; otherwise terminate and return the optimum solution.

The following MATLAB function implements this method.

```
function [res, noiter] = minscg(f,derf,x,tol)
% Conjugate gradient optimization by Moller
% Finds local min of a multivariable nonlinear function in n variables
% Example call: [res, noiter] = minscg(f,derf,x,tol)
% f is a user defined multi-variable function,
```

```
% derf a user defined function of n first order partial derivatives.
% x is a col vector of n starting values, tol gives required accuracy.
% res is solution, noiter is the number of iterations required.
lambda = 1e-8; lambdabar = 0; sigmac = 1e-5; sucess = 1;
deltastep = 0; [n m] = size(x);
% Calculate initial gradient
noiter = 0;
pv = -feval(derf,x); rv = pv;
while norm(rv)>tol
    noiter = noiter+1;
    if deltastep==0
        df = feval(derf,x);
    else
        df = -rv;
    end
    deltastep = 0;
    if sucess==1
        sigma = sigmac/norm(pv);
        dfplus = feval(derf,x+sigma*pv);
        stilda = (dfplus-df)/sigma;
        delta = pv'*stilda;
    end
    % Scale
    delta = delta+(lambda-lambdabar)*norm(pv)^2;
    if delta<=0
        lambdabar = 2*(lambda-delta/norm(pv)^2);
        delta = -delta+lambda*norm(pv)^2;
        lambda = lambdabar;
    end
    % Step size
    mu = pv'*rv; alpha = mu/delta;
    fv = feval(f,x);
    fvplus = feval(f,x+alpha*pv);
    delta1 = 2*delta*(fv-fvplus)/mu^2;
    rvold = rv; pvold = pv;
    if delta1>=0
        deltastep = 1;
        x1 = x+alpha*pv;
        rv = -feval(derf,x1);
        lambdabar = 0; sucess = 1;
        if rem(noiter,n) == 0
            pv = rv;
```

```
        else
            %Alternative conj grad direction generators may be used here
            % beta = (rv'*rv)/(rvold'*rvold);
            rdiff = rv-rvold;
            beta = (rdiff'*rv)/(rvold'*rvold);
            pv = rv+beta*pvold;
        end
        if delta1>=0.75
            lambda = 0.25*lambda;
        end
    else
        lambdabar = lambda;
        sucess = 0;
        x1 = x+alpha*pv;
    end
    if delta1<0.25
        lambda = lambda+delta*(1-delta1)/norm(pvold)^2;
    end
    x = x1;
end
res = x1;
```

We now show the scaled conjugate gradient method applied to two problems:

$$\text{Minimize } f(x_1,x_2) = \left(x_1^4 - 16x_1^2 + 5x_1\right)\Big/2 + \left(x_2^4 - 16x_2^2 + 5x_2\right)\Big/2$$

and

$$\text{Minimize } f(x_1,x_2) = 100(x_2 - x_1^2)^2 + (1-x_1)^2 \text{ (Rosenbrock's function)}$$

The first of these problems has been solved using `mincg`, and the user-defined functions `f01` and `f01d` are given in Section 8.4. Thus we have

```
>> [x, iterns] = minscg('f01','f01d',[1 -1]',.000005)

x =
    2.7468
   -2.9035

iterns =
     8
```

This is not the same solution as that determined by `mincg`. It is a local minimum value of the function but not the global minimum. Other initial values will lead to the global minimum.

To find a minimum of Rosenbrock's function we define the necessary anonymous functions, and solve the problem as follows:

```
>> fr = @(x) 100*(x(2)-x(1).^2).^2+(1-x(1)).^2;
>> frd = @(x) [-400*x(1).*(x(2)-x(1).^2)-2*(1-x(1)); 200*(x(2)-x(1).^2)];
>> [x, iterns] = minscg(fr,frd,[-1.2 1]',.0005)

x =
    1.0000
    1.0000

iterns =
   135
```

Note that a large number of iterations were required to solve this difficult problem.

8.6 Conjugate Gradient Method for Solving Linear Systems

We now apply the conjugate gradient algorithm to minimize a positive definite quadratic function, which has the standard form

$$f(\mathbf{x}) = \left(\mathbf{x}^\top \mathbf{A}\mathbf{x}\right)/2 + \mathbf{p}^\top \mathbf{x} + q \tag{8.8}$$

Here $\mathbf{x}$ and $\mathbf{p}$ are n-component column vectors, $\mathbf{A}$ is an $n \times n$ positive definite symmetric matrix, and q is a scalar. The minimum value of $f(\mathbf{x})$ is such that the gradient of $f(\mathbf{x})$ is zero. However, the gradient is easily found by direct differentiation as

$$\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{p} = \mathbf{0} \tag{8.9}$$

Thus finding the minimum is equivalent to solving this system of linear equations, which becomes, on letting $\mathbf{b} = -\mathbf{p}$,

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{8.10}$$

Since we can use the conjugate gradient method to find the minimum of (8.8), we can use it to solve the equivalent system of linear equations (8.10). The conjugate gradient method provides a powerful method for solving linear equation systems with positive definite symmetric matrices, and it follows quite closely the algorithm we have described for solving nonlinear optimization problems. However, the line search is greatly simplified and the

value of the gradient can be computed within the algorithm in this case. The algorithm takes the form

- *Step* 0: $k = 0$: $\mathbf{x}^k = \mathbf{0}$, $\mathbf{g}^k = \mathbf{b}$, $\mu^k = \mathbf{b}^\top \mathbf{b}$, $\mathbf{d}^k = -\mathbf{g}^k$
- *Step* 1: While system is not satisfied

$$\mathbf{q}^k = \mathbf{A}\mathbf{d}^k, \quad r^k = (\mathbf{d}^k)^\top \mathbf{q}^k, \quad s^k = \mu^k / r^k$$
$$\mathbf{x}^{k+1} = \mathbf{x}^k + s^k \mathbf{d}^k, \quad \mathbf{g}^{k+1} = \mathbf{g}^k + s^k \mathbf{q}^k$$
$$t^k = (\mathbf{g}^{k+1})^\top \mathbf{q}^k, \quad \beta^k = t^k / r^k$$
$$\mathbf{d}^{k+1} = -\mathbf{g}^{k+1} + \beta^k \mathbf{d}^k, \quad \mu^{k+1} = \beta^k \mu^k$$
$$k = k+1, \quad \text{end}$$

Notice that the values of the gradient $\mathbf{g}$ and the step s are calculated directly and no MATLAB function or user-defined function is required.

The MATLAB function `solvercg` implements this algorithm and utilizes the stopping procedure suggested by Karmarkar and Ramakrishnan (1991). See this paper and also Golub and Van Loan (1989) for more details.

```
function xdash = solvercg(a,b,n,tol)
% Solves linear system ax = b using conjugate gradient method.
% Example call: xdash = solvercg(a,b,n,tol)
% a is an n x n positive definite matrix, b is a vector of n
% coefficients. tol is accuracy to which system is satisfied.
% WARNING Large, ill-cond. systems will lead to reduced accuracy.
xdash = [ ];  gdash = [ ];
ddash = [ ];  qdash = [ ];
q=[ ];
mxitr = n*n;
xdash = zeros(n,1);  gdash = -b;
ddash = -gdash; muinit = b'*b;
stop_criterion1 = 1;
k = 0;
mu = muinit;
% main stage
while stop_criterion1==1
    qdash = a*ddash;
    q = qdash; r = ddash'*q;
    if r==0
        error('r=0, divide by 0!!!')
    end
    s = mu/r;
    xdash = xdash+s*ddash;
```

```
    gdash = gdash+s*q;
    t = gdash'*qdash;  beta = t/r;
    ddash = -gdash+beta*ddash;
    mu = beta*mu; k = k+1;
    val = a*xdash;
    if ((1-val'*b/(norm(val)*norm(b)))<=tol) & (mu/muinit<=tol)
        stop_criterion1 = 0;
    end
    if k>mxitr
        stop_criterion1 = 0;
    end
end
```

The following script generates a system of 10 equations with randomly selected elements on which this algorithm can be tested:

```
% e3s803.m
n = 10; tol = 1e-8;
A = 10*rand(n); b = 10*rand(n,1);
ada = A*A';
% To ensure a symmetric positive definite matrix.
sol = solvercg(ada,b,n,tol);
disp('Solution of system is:')
disp(sol)
accuracy = norm(ada*sol-b);
fprintf('Norm of residuals =%12.9f\n',accuracy)
```

Running this script gives the following results:

```
Solution of system is:
    0.2527
   -0.2642
   -0.1706
    0.4284
    0.0017
   -0.1391
   -0.0231
   -0.0109
   -0.2310
    0.2928

Norm of residuals = 0.000000008
```

We note that the norm of the residuals is very small. For ill-conditioned matrices it is necessary to use some kind of preconditioner, which reduces the condition number of

the matrix; otherwise the method becomes too slow. Karmarkar and Ramakrishnan (1991) used a preconditioned conjugate gradient method as part of an interior point algorithm to solve linear programming problems with 5000 rows and 333,000 columns.

MATLAB provides a range of iterative procedures based on conjugate gradient methods for solving $\mathbf{Ax} = \mathbf{b}$. These are the MATLAB functions `pcg`, `bicg`, and `cgs`.

8.7 Genetic Algorithms

In this section we introduce the ideas on which genetic algorithms are based and provide a group of MATLAB functions that implement the key features of a genetic algorithm. These are applied to the solution of some optimization problems. It is beyond the scope of this book to give a detailed account of this rapidly developing field of study and the reader is referred to the excellent text of Goldberg (1989).

Genetic algorithms have been the subject of considerable interest in recent years since they appear to provide a robust search procedure for solving difficult problems. The striking feature of these algorithms is that they are based on ideas from the science of genetics and the process of natural selection. This cross-fertilization from one field of science to another has led to stimulating and fruitful applications in many fields and particularly in computer science.

We will describe the genetic algorithm in the terminology used in the field and then explain how this relates to an optimization problem. The genetic algorithm works with an initial *population*, which may, for example, correspond to numerical values of a particular variable. The size of this population may vary and is generally related to the problem under consideration. The members of this population are usually strings of zeros and ones, that is, binary strings. For example, a small initial or first-generation population may take the form

1000010
1110000
1010101
1111001
1000001

In practice the population may be far larger than this and the strings longer. The strings themselves may be the encoded values of a variable or variables that we are examining. This initial population is generated randomly and we can use the terminology of genetics to characterize it. Each string in the population corresponds to a *chromosome* and each binary element of the string to a *gene*. A new population must now develop from this initial population; to do this we implement the analogue of specific fundamental genetic processes. These are

1. Selection based on fitness
2. Crossover
3. Mutation

A set of chromosomes is selected at the reproduction stage based on natural selection. Thus members of the population are chosen for reproduction on the basis of their fitness defined according to some specified criteria. The fittest are given a greater probability of reproducing in proportion to the value of their fitness.

The actual process of *mating* is implemented using the simple idea of crossover. This means that two members of the population exchange genes. There are many ways of implementing this crossover—for example having a single crossover point or many crossover points. These crossover points are selected randomly. A simple crossover is illustrated in the following for two chromosomes selected according to fitness. Here we have randomly selected a crossover point after the fourth digit.

1110|000
1010|101

After crossover this gives the new chromosomes

1110|101
1010|000

Applying this procedure to our original population, we produce a new generation. The final process is mutation. Here we randomly change a particular gene in a particular chromosome. Thus a 0 may be changed to a 1 or vice versa. The process of mutation in a genetic algorithm occurs very rarely and hence this probability of a change in a string is kept very low.

Having described the basic principles of a genetic algorithm, we now illustrate how it may be applied by considering a simple optimization problem and in so doing fill in some of the details to show how a genetic algorithm may be implemented. A manufacturer wishes to produce a container that consists of a hemisphere surmounted by a cylinder of fixed height. The height of the cylinder is fixed but the common radius of the cylinder and hemisphere may be varied between 2 and 4 units. The manufacturer wishes to find the radius value that maximizes the volume of the container. This is a simple problem and the optimum radius is 4 units. However, it serves to illustrate how the genetic algorithm may be applied.

We can formulate this as an optimization problem by taking r as the common radius of the cylinder and hemisphere and h as the height of the cylinder. Taking $h = 2$ units leads to the formula

$$\text{Maximize } v = 2\pi r^3/3 + 2\pi r^2 \tag{8.11}$$

where $2 \leq r \leq 4$.

The first problem we must consider is how to transform this problem so that the genetic algorithm can be applied directly. First we must generate an initial set of strings to constitute the initial population. The number of bits in each string, that is, the string length, limits the accuracy with which we can find the solution to the problem so it must be chosen with care. In addition, we must select the size of the initial population; again this must be chosen with care since a large initial population increases the time taken to implement the steps of the algorithm. A large population may be unnecessary since the algorithm automatically generates new members of the population in the process of searching the region. The MATLAB function `genbin` is used to generate such an initial population and takes the form

```
function chromosome = genbin(bitl,numchrom)
% Example call: chromosome=genbin(bitl, numchrom)
% Generates numchrom chromosomes of bitlength bitl.
% Called by optga.m.
maxchros = 2^bitl;
if numchrom>=maxchros
  numchrom = maxchros;
end
for k = 1:numchrom
    for bd = 1:bitl
        if rand>=0.5
            chromosome(k,bd) = 1;
        else
            chromosome(k,bd) = 0;
        end
    end
end
```

This function can be defined more succinctly using the MATLAB `round` function as follows:

```
function chromosome = genbin(bitl,numchrom)
% Example call: chromosome = genbin(bitl,numchrom)
% Generates numchrom chromosomes of bitlength bitl.
% Called by optga.m
maxchros=2^bitl;
if numchrom>=maxchros
    numchrom = maxchros;
end
chromosome = round(rand(numchrom,bitl));
```

To generate an initial population of five chromosomes, each with six genes, we call this function as

```
>> chroms = genbin(6,5)

chroms =
     0     1     1     1     0     0     [Population member #1]
     1     1     1     1     0     1     [Population member #2]
     1     0     0     1     0     0     [Population member #3]
     0     0     0     0     1     1     [Population member #4]
     0     1     1     1     0     1     [Population member #5]
```

To aid the reader in the following discussion we have labeled the five members of the population #1 to #5. These labels are not, of course, part of the MATLAB output.

Since we are interested in values of r in the range 2 to 4, we must be able to transform these binary strings to values in the range 2 to 4. This is achieved using the MATLAB function `binvreal`, which converts a binary value to a real value in the required range.

```
function rval = binvreal(chrom,a,b)
% Converts binary string chrom to real value in range a to b.
% Example call rval=binvreal(chrom,a,b)
% Normally called from optga.
[pop bitlength] = size(chrom);
maxchrom = 2^bitlength-1;
realel = chrom.*((2*ones(1,bitlength)).^fliplr([0:bitlength-1]));
tot = sum(realel);
rval = a+tot*(b-a)/maxchrom;
```

We now call this function to convert the previously generated population:

```
>> for i = 1:5, rval(i) = binvreal(chroms(i,:),2,4); end
>> rval

rval =
    2.8889    3.9365    3.1429    2.0952    2.9206
```

As expected, these values are in the range 2 to 4 and provide the initial population of values for r. However, these values tell us nothing about their fitness and to discover this we must judge them against some fitness criterion. In this case the choice is easy since our objective is to maximize the value of the function (8.11). We simply find the values of our objective function (8.11) for these values of r. We must define our function as a MATLAB function and it takes the form

```
>> g = @(x) pi*(0.66667*x+2).*x.^2;
```

Now we use this to evaluate fitness by replacing `x` by the values `rval`:

```
>> fit = g(rval)
```

```
fit =
  102.9330  225.1246  127.0806   46.8480  105.7749
```

Notice at this stage that the total fitness is

```
>> sum(fit)

ans =
  607.7611
```

So the fittest is the value 3.9365, with a fitness value 225.1246, which corresponds to string or population member #2. Fortuitously this is a very good result. The function `fitness` implements the preceding process and is given as follows:

```
function [fit,fitot] = fitness(criteria,chrom,a,b)
% Example call: [fit,fitot] = fitness(criteria,chrom,a,b)
% Calculates fitness of set of chromosomes chrom in range a to b,
% using the fitness criterion given by the parameter criteria.
% Called by optga.
[pop bitl] = size(chrom);
for k = 1:pop
    v(k) = binvreal(chrom(k,:),a,b);
    fit(k) = feval(criteria,v(k));
end
fitot = sum(fit);
```

Thus, repeating the preceding calculations, we have

```
>> [fit, sum_fit] = fitness(g,chroms,2,4)

fit =
  102.9330  225.1246  127.0806   46.8480  105.7749

sum_fit =
  607.7611
```

as before.

The next stage is reproduction when the strings are copied according to their fitness. Thus there is a higher probability of more of the fittest chromosomes in the mating pool. This process of selection is more complex and is based on a process that simulates the use of a roulette wheel. The percentage of the roulette wheel that is allocated to a particular string is directly proportional to the fitness of the string. For the preceding fitness vector `fit` the percentages can be calculated from

```
>> percent = 100*fit/sum_fit

percent =
   16.9364   37.0416   20.9096    7.7083   17.4040

>> sum(percent)

ans =
  100.0000
```

Thus, conceptually, we spin a roulette wheel on which strings 1 to 5 have 16.9364, 37.0416, 20.9096, 7.7083, and 17.4040 percent of the area, respectively. These chromosomes or strings have this chance of being selected. This is implemented by the function `selectga` as follows:

```
function newchrom = selectga(criteria,chrom,a,b)
% Example call: newchrom = selectga(criteria,chrom,a,b)
% Selects best chromosomes from chrom for next generation
% using function criteria in range a to b.
% Called by function optga.
% Selects best chromosomes for next generation using criteria
[pop bitlength] = size(chrom);
fit = [ ];
% calculate fitness
[fit,fitot] = fitness(criteria,chrom,a,b);
for chromnum = 1:pop
    sval(chromnum) = sum(fit(1,1:chromnum));
end
% select according to fitness
parname = [ ];
for i = 1:pop
    rval = floor(fitot*rand);
    if rval<sval(1)
        parname = [parname 1];
    else
        for j = 1:pop-1
            sl = sval(j);  su = sval(j)+fit(j+1);
            if (rval>=sl) & (rval<=su)
                parname = [parname j+1];
            end
        end
    end
end
newchrom(1:pop,:) = chrom(parname,:);
```

We can now use this function to perform the selection stage as follows:

```
>> matepool = selectga(g,chroms,2,4)

matepool =
     1     1     1     1     0     1     [Population member #2]
     1     1     1     1     0     1     [Population member #2]
     0     1     1     1     0     0     [Population member #1]
     0     1     1     1     0     1     [Population member #5]
     0     1     1     1     0     0     [Population member #1]
```

Note that members #1 and #2 have been favored by the selection process and duplicated. Because of the random nature of the selection process, member #3 is not selected, even though it is the second fittest member. We can now use the `fitness` function to obtain the fitness of the new population:

```
>> fitness(g,matepool,2,4)

ans =
  225.1246  225.1246  102.9330  105.7749  102.9330

>> sum(ans)

ans =
  761.8902
```

Notice the substantial increase in overall fitness.

We can now mate the members of this population, but we mate only a proportion of them, in this case 60% or 0.6. In this example the population size is 5 and $0.6 \times 5 = 3$. This number is rounded down to an even number, that is, 2, since only an even number of members of the population can mate. Thus, 2 members of the population are randomly selected for mating. The function that carries this out is `matesome`, defined as follows:

```
function chrom1 = matesome(chrom,matenum)
% Example call: chrom1 = matesome(chrom,matenum)
% Mates a proportion, matenum, of chromosomes, chrom.
mateind = [ ]; chrom1 = chrom;
[pop bitlength] = size(chrom);
ind = 1:pop;
u = floor(pop*matenum);
if floor(u/2)~=u/2
    u = u-1;
end
```

```
% select percentage to mate randomly
while length(mateind)~=u
    i = round(rand*pop);
    if i==0
        i = 1;
    end
    if ind(i)~=-1
        mateind = [mateind i];
        ind(i) = -1;
    end
end
% perform single point crossover
for i = 1:2:u-1
    splitpos = floor(rand*bitlength);
    if splitpos==0
        splitpos = 1;
    end
    i1 = mateind(i); i2 = mateind(i+1);
    tempgene = chrom(i1,splitpos+1:bitlength);
    chrom1(i1,splitpos+1:bitlength) = chrom(i2,splitpos+1:bitlength);
    chrom1(i2,splitpos+1:bitlength) = tempgene;
end
```

We now use this function to mate the strings in the new population `matepool`:

```
>> newgen = matesome(matepool,0.6)

newgen =
     1     1     1     1     0     1     [Population member #2]
     1     1     1     1     0     0     [Created from #2 and #1]
     0     1     1     1     0     1     [Created from #1 and #2]
     0     1     1     1     0     1     [Population member #5]
     0     1     1     1     0     0     [Population member #1]
```

We see that two members of the original population, members #1 and #2, have mated by crossing over after the second digit to create two new members of the population.

Computing the new population fitness we have

```
>> fitness(g,newgen,2,4)

ans =
  225.1246  220.4945  105.7749  105.7749  102.9330

>> sum(ans)

ans =
  760.1018
```

Notice that the total fitness has not improved and indeed, at this stage, we cannot expect improvements every time.

Finally we perform a mutation before repeating this same cycle of steps. This is implemented by the function `mutate` as follows:

```
function chrom = mutate(chrom,mu)
% Example call: chrom = mutate(chrom,mu)
% mutates chrom at rate given by mu
% Called by optga
[pop bitlength] = size(chrom);
for i = 1:pop
    for j = 1:bitlength
        if rand<=mu
            if chrom(i,j)==1
                chrom(i,j) = 0;
            else
                chrom(i,j) = 1;
            end
        end
    end
end
```

This is called with a very small value for `mu` and a population of this size is unlikely to be changed in just one generation. This function is called in the following:

```
>> mutate(newgen,0.05)

ans =
     1     1     0     1     0     1
     1     1     1     1     0     0
     0     1     1     1     0     1
     0     1     1     1     0     1
     0     1     1     1     1     0
```

Notice that in this example two mutations have occurred; the third element of the first chromosome has changed from 1 to 0, and the fifth element of the last chromosome has changed from 0 to 1. Sometimes no mutation will occur. This completes the production of a new generation. The process of selection based on fitness, reproduction, and mutation is now repeated using the new generation and subsequently repeated for many generations.

The function `optga` includes all these steps in one function and is defined as follows:

```
function [xval,maxf] = optga(fun,range,bits,pop,gens,mu,matenum)
% Determines maximum of a function using the Genetic algorithm.
% Example call: [xval,maxf] = optga(fun,range,bits,pop,gens,mu,matenum)
% fun is name of a one variable user defined positive valued function.
```

```
% range is 2 element row vector giving lower and upper limits for x.
% bits is number of bits for the variable, pop is population size.
% gens is number of generations, mu is mutation rate,
% matenum is proportion mated in range 0 to 1.
% WARNING. Method is not guaranteed to find global optima.
newpop = [ ];
a = range(1); b = range(2);
newpop = genbin(bits,pop);
for i = 1:gens
    selpop = selectga(fun,newpop,a,b);
    newgen = matesome(selpop,matenum);
    newgen1 = mutate(newgen,mu);
    newpop = newgen1;
end
[fit,fitot] = fitness(fun,newpop,a,b);
[maxf,mostfit] = max(fit);
xval = binvreal(newpop(mostfit,:),a,b);
```

Now, applying this function to solve our original problem, we specify the range of x from 2 to 4, and use 8-bit chromosomes and an initial population of 10. The process is continued for 20 generations with a mutation probability of 0.005 and a mating proportion of 0.6. Note that `matenum` must be greater than zero and less than or equal to one. Thus

```
>> [x f] = optga(g,[2 4],8,10,20,0.005,0.6)

x =
    3.8980

f =
  219.5219
```

Since the exact solution is $x = 4$, this is a reasonable result. Figure 8.6 gives a graphical representation of the progress of the genetic algorithm. It should be noted that each run of the genetic algorithm can produce a different result because of the random nature of the process. In addition, the number of distinct values in the search space is limited by the chromosome length. In this example the chromosome length is 8 bits, giving 2^8 or 256 divisions. Thus the range of r from 2 to 4 is divided into 256 divisions, each equal to 0.0078125.

We now discuss the philosophy and theory behind this process and the real problems to which genetic algorithms may be applied. The reason why a genetic algorithm differs from a simple direct-search procedure is that it involves two special features: crossover and mutation. Thus, starting from an initial population, the algorithm develops new generations, which rapidly explore the region of interest. This is useful for difficult optimization problems and in particular for those where we wish to find the global

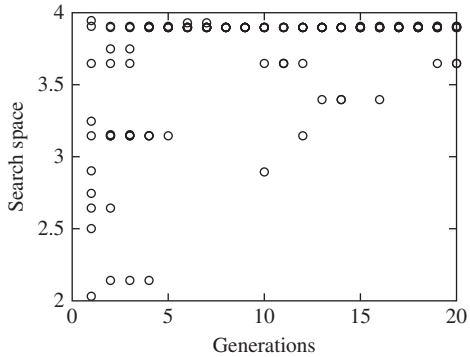

FIGURE 8.6 Each member of the population is represented by ○. Successive generations of the population concentrate toward the value 4 approximately.

maximum or minimum of a function that has many local maxima and minima. In this case standard optimization methods such as the conjugate gradient method of Fletcher and Reeves can locate only the local optimum. However, a genetic algorithm may locate the global optimum, although this is not guaranteed. This is due to the way it explores the region of interest, avoiding getting stuck at a particular local minimum. We do not consider the theoretical justification in detail but describe the key result only.

We first introduce the concept of *schemata*. If we study the structure of the strings produced by a genetic algorithm, certain patterns of behavior begin to emerge. Strings that have high fitness values often have common features, such as a particular combination of binary elements. For example, the fittest strings may have the common feature that they start with 11 and end with 0 or always have the middle three elements 0. We can represent strings with this structure by 11*****0 and ***000** where the asterisks represent "wild card" elements, which may be either 0 or 1. These structures are called schemata and essentially they identify the common features of a set of strings. The reason why a particular schema is interesting is that we wish to study the propagation of such strings that have this structure and are associated with high values of fitness. The length of a schema is the distance between the outermost specified gene values. The order of a schemata is the number of positions specified by 0 or 1. For example,

String	Order	Length
**********1	1	1
******10*1**	3	4
10******	2	2
00******101	5	11
11**00	4	6

It is clear that schemata that are defined by substrings of short length are less likely to be affected by crossover and therefore propagate through the generations unchanged.

We can now state the *fundamental theorem of genetic algorithms*, due to Holland, in terms of these schemata. This states that schemata of short length and low order with above-average fitness are propagated in exponentially increasing numbers throughout the generations. The ones with below-average fitness die away exponentially. This key result explains some of the success of genetic algorithms.

We now provide some further examples that apply the MATLAB genetic algorithm function `optga` to a specific optimization routine.

Example 8.1

Determine the maximum of the following function in the range $x = 0$ to $x = 1$.

$$f(x) = e^x + \sin(3\pi x)$$

Let the function `h` be defined by

```
h = @(x) exp(x)+sin(3*pi*x);
```

Calling `optga` with this function we have

```
>> [x f] = optga(h,[0 1],8,40,50,0.005,0.6)

x =
    0.8627

f =
    3.3315
```

We now apply the supplied MATLAB function `fminsearch` to solve this problem. Note that for use with `fminsearch` the function `h(x)` has been modified by including a minus sign, thereby negating the function since `fminsearch` is performing minimization.

```
>> h1 = @(x) -(exp(x)+sin(3*pi*x));
>> fminsearch(h1,0,1)

ans =
    0.1802

>> h1(ans)

ans =
   -2.1893
```

Here the function `fminsearch` has found a optimal value of the function but it is only a local optimum. The genetic algorithm (GA) has found a good approximation to the global optimum.

Example 8.2

A more demanding problem is to maximize the function

$$f(x) = 10 + \left[\frac{1}{(x-0.16)^2 + 0.1}\right] \sin(1/x)$$

Calling `optga` with this function defined by the anonymous function `phi` gives the following:

```
>> phi = @(x) 10+(1./((x-0.16).^2+0.1)).*sin(1./x);
>> [x f] = optga(phi,[0.001 0.3],8,10,40,0.005,0.6)

x =
    0.1288

f =
   19.8631
```

Figure 8.7 illustrates the difficulty of this problem and shows that the result is a reasonable one.

The changing diversity of the population can be illustrated graphically; see Figures 8.8 and 8.9. These plots show the value of each bit (0 or 1) for each member of the population. If the rectangle corresponding to a particular bit in a particular population member is shaded, it indicates a unit value for the bit; if it is white, it indicates a zero value for the bit. Figure 8.8 shows the initial randomly selected population and it is seen that there is a random selection of white and shade rectangles. Figure 8.9 shows the final population after 50 generations and we see that many of the bits have identical values in each population member.

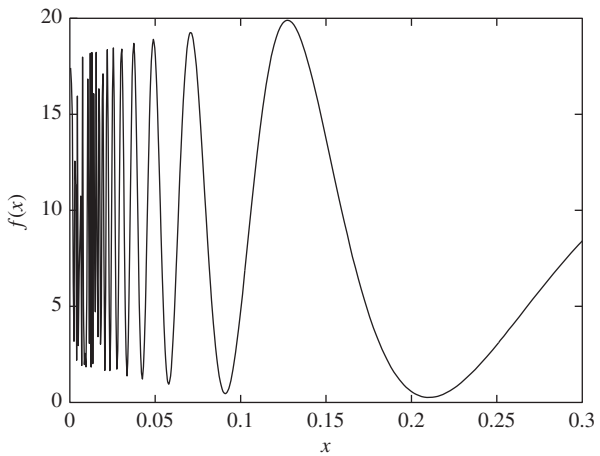

FIGURE 8.7 Plot of the function $10 + [1/\{(x-0.16)^2 + 0.1\}]\sin(1/x)$ showing many local maximum and minimum values.

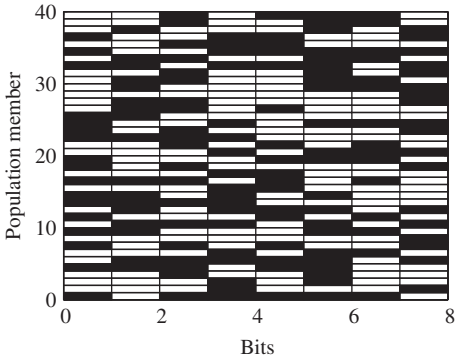

FIGURE 8.8 Initial random distribution of bits.

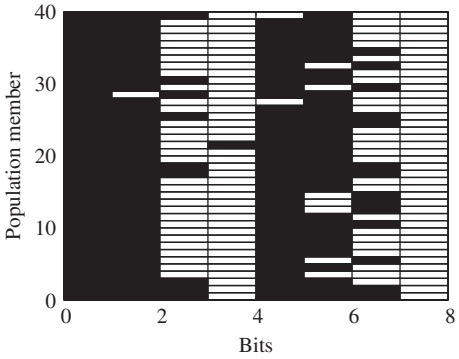

FIGURE 8.9 Distribution of bits after 50 generations.

Genetic algorithms are a developing area of research and many amendments could be made to the functions that we have supplied to implement a genetic algorithm. For example, Gray code rather than binary code can be used; the roulette wheel selection can be implemented in many different ways; crossover can be changed to multipoint crossover or other alternatives. It is often noticed that a genetic algorithm is slow in execution, but it should be remembered that it is best applied to difficult problems, such as those that have multiple optima and where the global optimum is required. Since standard algorithms often fail in these cases, the extra time taken by the genetic algorithm is worthwhile. There are many applications of genetic algorithms that we have not considered and the function `optga` only works for positive-valued functions in one independent variable. It would be easy to extend it to deal with two variable functions and this task is given as exercise for the reader (Problem 8.8)

We now consider using Gray code as an alternative strategy to the standard binary genetic algorithm. Here we interpret each of the strings as a number in Gray code. Gray code is a binary number system where two successive numbers differ by only one bit.

The code was originally developed by Gray to make the operation of systems of mechanical switches more reliable. It is useful at this point to define the Hamming distance. The Hamming distance is the count of the number of bits that are different between two binary vectors. Thus it follows that the Hamming distance between two succesive Gray code numbers is generally smaller than that of two succesive binary code numbers.

The following shows a comparison of the three-bit Gray code and the three-bit binary code

Decimal	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111
Gray	000	001	011	010	110	111	101	100

For the genetic algorithm we must convert from Gray code to decimal. To do this we use two stages: convert Gray code to binary and then convert binary to decimal. To convert from Gray code to binary, the simple algorithm is

$$\begin{aligned}&\text{For bit 1 (the most significant bit), } b(1) = g(1)\\&\text{For bit } i, \text{ where } i = 2,\ldots,n,\\&\quad \text{if } b(i-1) = g(i) \text{ then } b(i) = 0, \text{ else } b(i) = 1\end{aligned}$$

where $b(i)$ is a binary digit and $g(i)$ is the equivalent Gray code digit. This algorithm is implemented in the following function `grayvreal`

```
function rval = grayvreal(gray,a,b)
% Converts gray string to real value in range a to b.
% Example call rval = grayvreal(gray,a,b)
% Normally called from optga.
[pop bitlength] = size(gray);
maxchrom = 2^bitlength-1;
% Converts gray to binary
bin(1) = gray(1);
for i = 2:bitlength
    if bin(i-1) == gray(i)
        bin(i) = 0;
    else
        bin(i) = 1;
    end
end
% Converts binary to real
realel = bin.*((2*ones(1,bitlength)).^fliplr([0:bitlength-1]));
tot = sum(realel);
rval = a+tot*(b-a)/maxchrom;
```

This function can be used instead of `binvreal` in the MATLAB function `fitness` (renamed `fitness_g`) and this function is then used in `selectga` (renamed `selectga_g`). Finally, both

of these functions are used in the function `optga_g`. The following example ilustrates the use of the function `optga_g`.

```
>> g = @(x) exp(x)+sin(3*pi*x);
>> [x f] = optga_g(g,[0 1],8,40,50,0.005,0.6)

x =
    0.8588

f =
    3.3317
```

This provides a result with similar accuracy to that achieved using the ordinary binary algorithm.

While some researchers have found no benefit in using Gray codes, others, such as Caruana and Schaffer (1988), claim that the Gray code GA can sometimes be of significant value.

The genetic algorithm can be used to solve optimization problems in which the solutions are constrained to members of a set of discrete values, rather than being continuous over a defined range. For example, steel sections are rolled in specific sizes, and it would be uneconomical to have special sections rolled. Thus a framework, for example, will be constructed from standard beam sizes and sections. The same applies to electronic circuit components such as resistors, which are manufactured in a set of standard values. Suppose we wish to optimize a design that only includes components that are available in eight sizes. Let us assume the vector of discrete values [10 15 24 36 50 75 90 120] gives some performance values for each of the eight possible components. We can use the binary numbers 000 to 111 to represent the indices of these eight performance values. The GA optimization proceeds in the normal way; however, to calculate the fitness corresponding to a particular binary number, this number is used as the index to the vector of performance values to obtain the corresponding performance value. For example, suppose we require the fitness corresponding to the binary number 100 (i.e., decimal 4). The performance value corresponding to this number is 36 and this number is used in the fitness calculation. A difficulty arises if the number of possible components, and hence the number of members in the set of performance values, is not an integer power of 2. For example, suppose we only have six possible component sizes with performance values of, say, [10 15 24 36 50 75]. To represent the six indices of this vector in binary code requires a minimum of three digits. However, the process of crossover and mutation may generate any of eight binary numbers, but only six have corresponding properties. To overcome this difficulty, two of the performance values are duplicated so that, for example, the properties corresponding to the eight binary numbers 000 to 111 are now [10 10 15 24 36 50 75 75]. This adjustment slightly affects the statistics of the process but it generally works satisfactorily. Although in this discussion we have assumed sets of six or eight component sizes, in most practical problems the component set is likely to have as many as 32 or 64 members.

8.8 Continuous Genetic Algorithm

The continuous genetic algorithm is similar in structure to the binary form of the genetic algorithm we have described in Section 8.7, in that an initial population in the region of interest is generated randomly, pairs are selected from the current population and are mated according to fitness, crossover occurs between chromosomes, and mutation of chromosomes occurs with a specified probabilty. However, these steps have significant differences in their implementation in the continuous GA. Basing our description on an optimization problem we randomly generate a set of chromosomes. This initial population is a set of random real numbers rather than binary digits. The key feature here is that the values can be any of the continuous set of values in the region of interest and not a discrete set of binary values that we have used in the binary form of the algorithm.

Suppose we assume that the function to be optimized has four variables. Then initially each chromosome is a vector of four randomly generated decimal numbers, each lying in the search range for the variable. If we choose to have a population of 20 chromosomes, then each has its fitness assessed according to a fitness criteria and a number of the fittest are chosen for the mating process. For example, the most fit 8 chromosomes from a group of 20 chromosomes may be chosen to constitute the mating pool. From this group, random pairs are chosen for crossover and mating.

The mating process is again broadly similar to that of the binary form of the GA in that a random point is chosen for crossover so that the parental chromosomes are intermixed about this point by simply interchanging the real variable values within the chromosomes. However at this point a crucial difference is introduced since crossover in this form simply interchanges the original set of randomly generated real values without producing new values in the region. So to help explore the region we need to introduce new values. For example, suppose the function to be minimized is a function of four variables, u, v, w, and x, and the two chromosomes to be mated, r_1 and r_2, are given by

$$r_1 = [u_1\ v_1\ w_1\ x_1], \quad r_2 = [u_2\ v_2\ w_2\ x_2]$$

Of course, these two chromosomes are chosen at random from the mating pool according to fitness. At a random point in the chromosome a new value in the region is created by forming two new elements from a linear random combination of the pair of chromosome elements at this point. These new values then replace the original chromosome values at the selected crossover point. The suggested formulae for generating new data values take the form

$$x_a = x_1 - \beta(x_1 - x_2), \quad x_b = x_2 + \beta(x_1 - x_2)$$

Similar equations can be applied to the variables u, v, and w. At each generation, β, the crossover point and the pairing of the fittest four members of the previous population will all be rechosen.

The crossover point can occur at random at points 1, 2, 3, or 4. Depending on which crossover point is chosen, after mating the new chromosomes are

Crossover at 1: $r_1 = [u_a\ v_2\ w_2\ x_2]$, $r_2 = [u_b\ v_1\ w_1\ x_1]$

Crossover at 2: $r_1 = [u_1\ v_a\ w_2\ x_2]$, $r_2 = [u_2\ v_b\ w_1\ x_1]$

Crossover at 3: $r_1 = [u_1\ v_1\ w_a\ x_2]$, $r_2 = [u_2\ v_2\ w_b\ x_1]$

Crossover at 4: $r_1 = [u_1\ v_1\ w_1\ x_a]$, $r_2 = [u_2\ v_2\ w_2\ x_b]$

Other crossover rules may be applied. Note that a one-dimensional problem cannot be solved using this particular mating algorithm.

The process of mutation is implemented in a very similar way to that of the binary genetic algorithm. A mutation rate is chosen; then the number of mutations can be calculated from the number of chromosomes and the number of components in the chromosome. Then positions are randomly selected in the chromosomes and these chromosome values are replaced by random values selected within the region. This is another way of helping the algorithm to explore the region, which increases the chance of finding the global minimum for the whole region. The `contgaf` function implements a continuous genetic algorithm. This particular implementation is arranged to find the *minimum* of a function.

```
function [x,f] = contgaf(func,nv,range,pop,gens,mu,matenum)
% function for continuous genetic algorithm
% func is the multivariable function to be optimised
% nv is the number of variables in the function (minimum = 2)
% range is row vector with 2 elements. i.e [lower bound upper bound]
% pop is the number of chromosomes, gens is the number of generations
% mu is the mutation rate in range 0 to 1.
% matenum is the proportion of the population mated in range 0 to 1.
pops = [ ];  fitv = [ ]; nc = pop;
% Generate chromosomes as uniformly distributed sets of random decimal
% numbers in the range 0 to 1
chrom = rand(nc,nv);
% Generate the initial population in the range a to b
a = range(1); b = range(2);
pops = (b-a)*chrom+a;
for MainIter = 1:gens
    % Calculate fitness values
    for i = 1:nc
        fitv(i) = feval(func, pops(i,:));
    end
    % Sort fitness values
    [sfit,indexf] = sort(fitv);
```

```
% Select only the best matnum values for mating
% ensure an even number of pairs is produced
nb = round(matenum*nc);
if nb/2~=round(nb/2)
    nb = round(matenum*nc)+1;
end
fitbest = sfit(1:nb);
% Choose mating pairs use rank weighting
prob = @(n) (nb-n+1)/sum(1:nb);
rankv = prob([1:nb]);
for i = 1:nb
    cumprob(i) = sum(rankv(1:i));
end
% Choose two sets of mating pairs
mp = round(nb/2);
randpm = rand(1,mp);  randpd = rand(1,mp);
mm = [ ];
for j = 1:mp
    if randpm(j)<cumprob(1)
        mm = [mm,1];
    else
        for i = 1:nb-1
            if (randpm(j)>cumprob(i)) && (randpm(j)<cumprob(i+1))
                mm = [mm i+1];
            end
        end
    end
end
% The remaining elements of nb = [1 2 3,...] are the other ptnrs
md = [ ];
md = setdiff([1:nb],mm);
% Mating between mm and md. Choose crossover
xp = ceil(rand*nv);
addpops = [ ];
for i = 1:mp
    % Generate new value
    pd = pops(indexf(md(i)),:);
    pm = pops(indexf(mm(i)),:);
    % Generate random beta
    beta = rand;
    popm(xp) = pm(xp)-beta*(pm(xp)-pd(xp));
    popd(xp) = pd(xp)+beta*(pm(xp)-pd(xp));
```

```
        if xp==nv
            % Swap only to left
            ch1 = [pm(1:nv-1),pd(nv)];
            ch2 = [pd(1:nv-1),pm(nv)];
        else
            ch1 = [pd(1:xp),pm(xp+1:nv)];
            ch2 = [pm(1:xp),pd(xp+1:nv)];
        end
        % New values introduced
        ch1(xp) = popm(xp);
        ch2(xp) = popd(xp);
        addpops = [addpops;ch1;ch2];
    end
    % Add these ofspring to the best to obtain a new population
    newpops = [ ];   newpops = [pops(indexf(1:nc-nb),:); addpops];
    % Calculate number of mutations, mutation rate mu
    Nmut = ceil(mu*nv*(nc-1));
    % Choose location of variables to mutate
    for k = 1:Nmut
        mui = ceil(rand*nc);  muj = ceil(rand*nv);
        if mui~=indexf(1)
            newpops(mui,muj) = (b-a)*rand+a;
        end
    end
    pops = newpops;
end
f = sfit(1);  x = pops(indexf(1),:);
```

We can test this function on an example already discussed in this chapter: a two-variable function that is taken from Styblinski and Tang (1990). This function is

$$f(x_1,x_2) = \left(x_1^4 - 16x_1^2 + 5x_1\right)\Big/2 + \left(x_2^4 - 16x_2^2 + 5x_2\right)\Big/2$$

We may define this function in MATLAB using an anonymous function as follows:

```
>> tf=@(x) 0.5*(x(1).^4-16*x(1).^2+5*x(1))+0.5*(x(2).^4- ...
16*x(2).^2+5*x(2));
```

This function has several local minima but the global optima is at $(-2.9035,\ -2.9035)$. Here we execute three runs of the continuous genetic algorithm:

```
>> [x,f] = contgaf(tf,2,[-4 4],50,50,0.2,0.6)

x =
   -2.9036   -2.9032
```

```
f =
  -78.3323

>> [x,f] = contgaf(tf,2,[-4 4],50,50,0.2,0.6)

x =
   -2.9035   -2.9037

f =
  -78.3323

>> [x,f] = contgaf(tf,2,[-4 4],50,50,0.2,0.6)

x =
   -2.9035   -2.8996

f =
  -78.3321
```

Notice the difference in the **x** values. The process involves a random element and will not produce the same result every time.

As a further example, determine the minimim value of the function

$$f(x) = \sum_{n=1}^{4} \left[100 \left(x_{n+1} - x_n^2 \right)^2 + (1 - x_n)^2 \right]$$

Obviously, the minimum of this function is zero. Thus we have

```
ff = @(x)(1-x(4))^2+(1-x(3))^2+(1-x(2))^2+(1-x(1))^2+ ...
    100*((x(5)-x(4)^2)^2+(x(4)-x(3)^2)^2+(x(3)-x(2)^2)^2+(x(2)-x(1)^2)^2);
>> [x,f] = contgaf(ff,5,[-5 5],20,100,0.15,0.6)

x =
    0.7617    0.6677    0.6392    0.5876    0.3435

f =
    8.1752
```

This is a good result. The actual minimum is zero at [1 1 1 1 1]. However, the function value at $[-5\ -5\ -5\ -5\ -5]$ is 360,144. If we sought the solution for the minimum value to the nearest integer by evaluating the function in the range -5 to 5 in each dimension, the function would need to be evaluated 161,051 times. To find the solution to an accuracy of 0.1, we would require 1.051×10^{10} function evaluations—not a realistic approach.

For the discussion of the efficiency of the continuous GA, see Chelouah and Siarry (2000). Comparisons between binary and continous genetic algorithms have been carried out by several authors and the continuous GA has been found to have the advantage of greater consistency from run to run and higher precision (Michalewicz, 1996).

8.9 Simulated Annealing

Here we provide a brief introduction to the ideas on which optimization using simulated annealing is based. The technique should be applied to large and difficult problems where we require the global optima and where other techniques are inadequate. Even for relatively simple problems the technique can be slow.

If a metal is allowed to cool sufficiently slowly (metallurgically called postannealing), its metallurgical structure is naturally able to find a minimum energy state for the system. If, however, the metal is cooled quickly, say by quenching in water, then this minimum energy state is not found. This concept of the natural process of finding a minimum energy state can be used to find the global optima of given nonlinear functions. This optimization method is called simulated annealing.

The analogy is not perfect but the fast cooling process may be viewed as equivalent to finding a local minimum of a given nonlinear function corresponding to the energy level, while the slow cooling corresponds to finding the ideal energy state or a global minimum of the function. This slow cooling process may be implemented using the Boltzmann probability distribution of energy states, which plays a prominent part in thermodynamics and has the form

$$P(E) = \exp(-E/kT)$$

where $P(E)$ is the probability of E, a particular energy state, k is Boltzmann's constant, and T is the temperature. This function is used to reflect the cooling process where a change in the energy level, which may be initially unfavorable, ultimately leads to a final minimum global energy state.

This corresponds to the concept of moving out of the region of a local minimum of a nonlinear function in the search for a global solution for the problem. This may require a temporary increase in the value of the objective function, that is, climbing out of the valley of a local minimum, although convergence to the global optimum may still occur if the adjustment to the temperature is slow enough. These ideas lead to an optimization algorithm used by Kirkpatrick et al. (1983), which has the following general structure.

Let $f(\mathbf{x})$ be the nonlinear function to be minimized, where $\mathbf{x}$ is an n-component vector. Then

- *Step* 1: Set $k = 0$, $p = 0$. Chose a starting solution $\mathbf{x}^k$ and an initial, arbitrary temperature T_p.
- *Step* 2: Let a new value of $\mathbf{x}$, $\mathbf{x}^{k+1}$ cause a change, $\Delta f = f(\mathbf{x}^{k+1}) - f(\mathbf{x}^k)$; then

if $\Delta f < 0$, accept the change with probability 1 and $\mathbf{x}^{k+1}$ replaces $\mathbf{x}^k$, $k = k+1$.
If $\Delta f > 0$, accept the change with probability $\exp(-\Delta f/T_p)$ and
$\mathbf{x}^{k+1}$ replaces $\mathbf{x}^k$, $k = k+1$.

- *Step* 3: Repeat from step 2 until there is no significant change of function value.
- *Step* 4: Lower the temperature using an appropriate reduction process $T_{p+1} = g(T_p)$, set $p = p+1$, and repeat from step 2 until there is no further significant change in the function value from temperature reduction.

The key difficulties with this algorithm are choosing an initial temperature and a temperature reduction regime. This has generated many research papers and the details are not discussed here.

The MATLAB function `asaq` is an improved implementation of the preceding algorithm. It is based on a modified and simplified version of an algorithm described by Lester Ingber (1993). This uses a exponential cooling regime with some quenching to accelerate the convergence of the algorithm. The key parameters, such as the values of `qf`, `tinit`, and `maxstep`, and the upper and lower bounds on the variables can be adjusted and may lead to some improvements in the convergence rate. A major change would be to use a different temperature adjustment regime and many alternatives have been suggested. The reader should view these parameter variations as an opportunity to experiment with simulated annealing.

```
function [fnew,xnew] = asaq(func,x,maxstep,qf,lb,ub,tinit)
% Determines optimum of a function using simulated annealing.
% Example call: [fnew,xnew]=asaq(func,x,maxstep,qf,lb,ub,tinit)
% func is the function to be minimized, x the initial approx.
% given as a column vector, maxstep the maximum number of main
% iterations, qf the quenching factor in range 0 to 1.
% Note: small value gives slow convergence, value close to 1 gives
% fast convergence, but may not supply global optimum.
% lb and ub are lower and upper bounds for the variables,
% tinit is the intial temperature value
% Suggested values for maxstep = 200, tinit = 100, qf = 0.9
% Initialisation
xold = x;  fold = feval(func,x);
n = length(x);  lk = n*10;
% Quenching factor q
q = qf*n;
% c values estimated
nv = log(maxstep*ones(n,1));
mv = 2*ones(n,1);
c = mv.*exp(-nv/n);
% Set values for tk
t0 = tinit*ones(n,1);  tk = t0;
```

```
% upper and lower bounds on x variables
% variables assumed to lie between -100 and 100
a = lb*ones(n,1);  b = ub*ones(n,1);
k = 1;
% Main loop
for mloop = 1:maxstep
    for tempkloop = 1:lk
        % Choose xnew as random neighbour
        fold = feval(func,xold);
        u = rand(n,1);
        y = sign(u-0.5).*tk.*((1+ones(n,1)./tk).^(abs((2*u-1))-1));
        xnew = xold+y.*(b-a);
        fnew = feval(func,xnew);
        % Test for improvement
        if fnew <= fold
            xold = xnew;
        elseif exp((fold-fnew)/norm(tk))>rand
            xold = xnew;
        end
    end
    % Update tk values
    tk = t0.*exp(-c.*k^(q/n));
    k = k+1;
end
tf = tk;
```

We will run this script to optimize the following function, which is taken from Styblinski and Tang (1990), and solve it by the conjugate gradient method in Section 8.4:

$$f(x_1, x_2) = \left(x_1^4 - 16x_1^2 + 5x_1\right)\Big/2 + \left(x_2^4 - 16x_2^2 + 5x_2\right)\Big/2$$

The results are as follows:

```
>> fv = @(x) 0.5*(x(1)^4-16*x(1)^2+5*x(1)) +...
          0.5*(x(2)^4-16*x(2)^2+5*x(2));
>> [fnew,xnew] = asaq(fv,[0 0].',200,0.9,-10,10,100)

fnew =
  -78.3323

xnew =
   -2.9018
   -2.9038
```

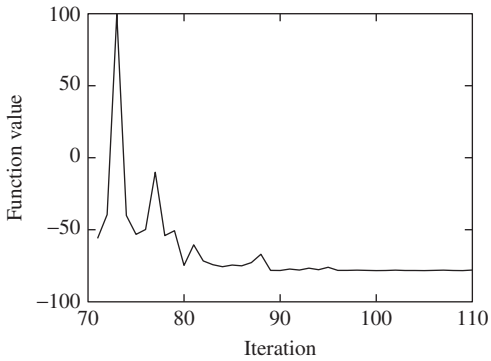

FIGURE 8.10 Graph showing the value of function $f(x_1, x_2) = \left(x_1^4 - 16x_1^2 + 5x_1\right)/2 + \left(x_2^4 - 16x_2^2 + 5x_2\right)/2$ for the final 40 iterations.

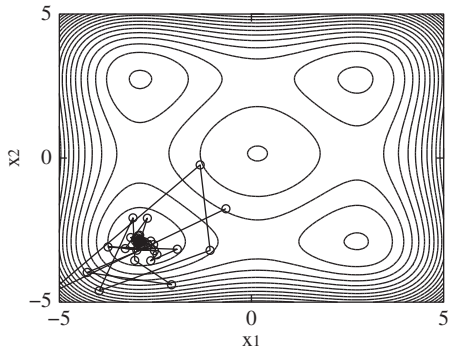

FIGURE 8.11 Contour plot of function $f(x_1, x_2) = \left(x_1^4 - 16x_1^2 + 5x_1\right)/2 + \left(x_2^4 - 16x_2^2 + 5x_2\right)/2$. The final stages in the simulated annealing process are shown. Note how these values are concentrated in the lower left corner, close to the global optimum.

Note that each run provides a different result and is not guaranteed to provide a global optimum unless the parameters are adjusted appropriately for the particular problem. Figure 8.10 provides a plot of the variation in the function value for the final 40 iterations. It illustrates the behavior of the algorithm, which allows both increases and decreases in the function value.

As a further illustration, a contour plot showing only the final stages of the iteration is given in Figure 8.11.

8.10 Constrained Nonlinear Optimization

In this section we consider the problem of optimizing a nonlinear function, subject to one or more nonlinear constraints. This problem can be expressed mathematically as

follows:

$$\text{Minimize } f = f(\mathbf{x}) \quad \text{where} \quad \mathbf{x}^\top = [x_1 \ x_2 \ \dots \ x_n] \tag{8.12}$$

subject to the constraints

$$h_i(\mathbf{x}) = 0 \quad \text{where} \quad i = 1, \dots, p \tag{8.13}$$

Sometimes the minimization problem may have additional or alternative constraints that are of the form

$$g_j(\mathbf{x}) \geq b_j \quad \text{where} \quad j = 1, \dots, q \tag{8.14}$$

To solve this problem we can use the Lagrange multiplier method. This method is not a purely numerical method; it requires the user to apply calculus, and the resulting equations are solved numerically. For large problems this is too onerous on the user, and for this reason it is not a practical method for solving this type of problem. However, it is theoretically important in the development of other, more practical, methods.

If constraints of the form of (8.14) are present, they must be converted to the form of (8.13) as follows. Let $\theta_j^2 = g_j(\mathbf{x}) - b_j$. If the constraint $g_j(\mathbf{x}) \geq b_j$ is violated, then θ_j^2 is negative and θ_j is imaginary. Thus, we have a requirement that for the constraints to be satisfied θ_j must be real. Thus the constraint equation (8.14) becomes

$$\theta_j^2 - g_j(\mathbf{x}) + b_j = 0 \quad \text{where} \quad j = 1, \dots, q \tag{8.15}$$

This constraint equation is of the same general form as the constraint in (8.13).

To solve (8.12) we begin by forming the expression

$$L(\mathbf{x}, \boldsymbol{\theta}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{i=1}^{p} \lambda_i h_i(\mathbf{x}) + \sum_{j=1}^{q} \lambda_{p+j} [\theta_j^2 - g_j(\mathbf{x}) + b_j] = 0 \tag{8.16}$$

The function L is called the Lagrange function and the scalar quantities λ_i are called the Lagrange multipliers. We now minimize this function using calculus; that is, we take the following partial derivatives and set them to zero.

$$\begin{aligned} \partial L/\partial x_k &= 0, \quad k = 1, \dots, n \\ \partial L/\partial \lambda_r &= 0, \quad r = 1, \dots, p+q \\ \partial L/\partial \theta_s &= 0, \quad s = 1, \dots, q \end{aligned}$$

We will find that when we set the differentials with respect to λ_r to zero, we force both $h_i(\mathbf{x})$ $(i = 1, 2, \dots, n)$ and $\theta_j^2 - g_j(\mathbf{x}) + b_j$ $(j = 1, 2, \dots, q)$ to be zero. Thus the constraints to be satisfied. If these terms are zero then minimizing (8.16) is equivalent to minimizing (8.12) subject to (8.13) and (8.14). If we are dealing with a quadratic function with linear constraints then the resulting equations are all linear and relatively easy to solve.

Example 8.3

Consider the solution of a problem with a cubic function and quadratic constraints.

$$\text{Minimize } f = 2x + 3y - x^3 - 2y^2$$

subject to

$$\begin{aligned} &x + 3y - x^2/2 \le 5.5 \\ &5x + 2y + x^2/10 \le 10 \\ &x \ge 0,\ y \ge 0 \end{aligned}$$

To use the Lagrange method we change the form of the constraints to be equality constraints, as follows:

$$\text{Minimize } f = 2x + 3y - x^3 - 2y^2$$

subject to

$$\begin{aligned} &\theta_1^2 + x + 3y - x^2/2 - 5.5 = 0 \\ &\theta_2^2 + 5x + 2y + x^2/10 - 10 = 0 \\ &x \ge 0,\ y \ge 0 \end{aligned}$$

Hence, forming L we have

$$L = 2x + 3y - x^3 - 2y^2 + \lambda_1(\theta_1^2 + x + 3y - x^2/2 - 5.5) + \lambda_2(\theta_2^2 + 5x + 2y + x^2/10 - 10)$$

Taking partial derivaties of L and setting them to zero gives

$$\partial L/\partial x = 2 - 3x^2 + \lambda_1(1 - x) + \lambda_2(5 + x/5) = 0 \tag{8.17}$$

$$\partial L/\partial y = 3 - 4y + 3\lambda_1 + 2\lambda_2 = 0 \tag{8.18}$$

$$\partial L/\partial \lambda_1 = \theta_1^2 + x + 3y - x^2/2 - 5.5 = 0 \tag{8.19}$$

$$\partial L/\partial \lambda_2 = \theta_2^2 + 5x + 2y + x^2/10 - 10 = 0 \tag{8.20}$$

$$\partial L/\partial \theta_1 = 2\lambda_1\theta_1 = 0 \tag{8.21}$$

$$\partial L/\partial \theta_2 = 2\lambda_2\theta_2 = 0 \tag{8.22}$$

If (8.21) and (8.22) are to be satisfied then there are four cases to consider:

Case 1: $\theta_1^2 = \theta_2^2 = 0$. Then (8.17) through (8.20) become, with some rearrangement,

$$\begin{aligned} 2 - 3x^2 + \lambda_1(1 - x) + \lambda_2(5 + x/5) &= 0 \\ 3 - 4y + 3\lambda_1 + 2\lambda_2 &= 0 \\ x + 3y - x^2/2 - 5.5 &= 0 \\ 5x + 2y + x^2/10 - 10 &= 0 \end{aligned}$$

Case 2: $\lambda_1 = \theta_2^2 = 0$. Then (8.17) through (8.20) become, with some rearrangement,

$$\begin{aligned} 2 - 3x^2 + \lambda_2(5 + x/5) &= 0 \\ 3 - 4y + 2\lambda_2 &= 0 \\ \theta_1^2 + x + 3y - x^2/2 - 5.5 &= 0 \\ 5x + 2y + x^2/10 - 10 &= 0 \end{aligned}$$

Case 3: $\theta_1^2 = \lambda_2 = 0$. Then (8.17) through (8.20) become, with some rearrangement,

$$\begin{aligned} 2 - 3x^2 + \lambda_1(1 - x) &= 0 \\ 3 - 4y + 3\lambda_1 &= 0 \\ x + 3y - x^2/2 - 5.5 &= 0 \\ \theta_2^2 + 5x + 2y + x^2/10 - 10 &= 0 \end{aligned}$$

Case 4: $\lambda_1 = \lambda_2 = 0$. Then (8.17) through (8.20) become, with some rearrangement,

$$\begin{aligned} 2 - 3x^2 &= 0 \\ 3 - 4y &= 0 \\ \theta_1^2 + x + 3y - x^2/2 - 5.5 &= 0 \\ \theta_2^2 + 5x + 2y + x^2/10 - 10 &= 0 \end{aligned}$$

The solution of these sets of nonlinear equations requires some iterative procedure. The MATLAB function `fminsearch` finds the minimum of a scalar function of several variables, given an initial estimate. The application of this function to this problem is illustrated for *Case* 1 in the following MATLAB script. Since the right side of each equation is zero, when the solution is found, the function

$$\begin{aligned} &[2 - 3x^2 + \lambda_1(1 - x) + \lambda_2(5 + x/5)]^2 + [3 - 4y + 3\lambda_1 + 2\lambda_2]^2 + \cdots \\ &[x + 3y - x^2/2 - 5.5]^2 + [5x + 2y + x^2/10 - 10]^2 \end{aligned}$$

should equal zero. The function `fminsearch` will choose values of x, y, λ_1, and λ_2 to minimize this expression and bring it very close to zero. This is generally referred to as unconstrained nonlinear optimization. Thus we have converted a constrained optimization to an unconstrained one.

```
% e3s820.m
g = @(X) sqrt((2-3*X(1).^2+X(3).*(1-X(1))+X(4).*(5+X(1)/5)).^2 ...
    +(3-4*X(2)+3*X(3)+2*X(4)).^2+(X(1)+3*X(2)-X(1).^2/2-5.5).^2 ...
    +(5*X(1)+2*X(2)+X(1).^2/10-10).^2);
X = fminsearch(g, [1 1 1 1]);
x = X(1); y = X(2); f = 2*x+3*y-x^3-2*y^2;
lambda_1 = X(3); lambda_2 = X(4);
```

```
disp('Case 1')
disp(['x = ' num2str(x) ', y = ' num2str(y) ', f = ' num2str(f)])
disp(['lambda_1 = ' num2str(lambda_1) ...
    ', lambda_2 = ' num2str(lambda_2)])
```

This script can be extended to include the solution of Cases 2, 3 and 4. The method of analysis is similar to Case 1. Executing this script gives

```
Case 1
x = 1.2941, y = 1.6811, f = -0.18773
lambda_1 = 0.82718, lambda_2 = 0.62128
```

Comparing the values of f computed for each case (see Table 8.2), it is clear that Case 1 gives the minimum solution. The function is shown in Figure 8.12. This graph shows the function,

Table 8.2 Possible Solutions of a Minimization Problem

Case	θ_1^2	θ_1^2	λ_1	λ_2	x	y	f
1	0	0	0.8272	0.6213	1.2941	1.6811	−0.1877
2	1.5654	0	0	0.8674	1.4826	1.1837	0.4552
3	0	2.3236	1.2270	0	0.8526	1.6703	0.5166
4	2.7669	4.3508	0	0	0.8165	0.7500	2.2137

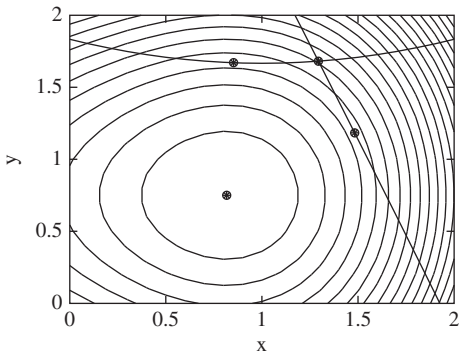

FIGURE 8.12 Function and constraints. The four solutions are also indicated.

the constraints, and the four possible solutions. The optimal solution is not necessarily at the intersection of the constraint boundaries as it is in a linear system. All solutions are feasible but only Case 1 is the global minimum. Case 2 and Case 3 are local minima at the constraint boundary and Case 4 is a local maximum. It should be noted that often one or more solutions will not be feasible; that is, they will not satisfy the constraints.

8.11 The Sequential Unconstrained Minimization Technique

We now give a brief introduction to a standard method for constrained optimization. The sequential unconstrained minimization technique (SUMT) for constrained optimization converts the solution of a constrained optimization problem to the solution of a sequence of unconstrained problems. This method was developed by Fiacco and McCormicks and others in the 1960s. See Fiacco and McCormicks (1964, 1990).

Consider the following optimization problem:

$$\begin{gathered}\text{Minimize } f(\mathbf{x}), \text{ subject to} \\ g_i(\mathbf{x}) \geq 0 \quad \text{for} \quad i = 1,2,\ldots,p \\ h_j(\mathbf{x}) = 0 \quad \text{for} \quad j = 1,2,\ldots,s\end{gathered}$$

where $\mathbf{x}$ is a component vector. By using barrier and penalty functions the requirements of the constraints can be included with the function to be minimized so that the problem is converted to the unconstrained problem:

$$\text{Minimize } f(\mathbf{x}) - r_k \sum_{i=1}^{p} \log_e(g_i(\mathbf{x})) + \frac{1}{r_k} \sum_{j=1}^{s} h_j(\mathbf{x})^2$$

Notice the effect of the added terms. The first term imposes a barrier at zero on the inequality constraints in that as the $g_i(\mathbf{x})$ approaches zero the function approaches minus infinity, thus imposing a substantial penalty. Figure 8.13 illustrates this. The last term encourages the satisfaction of the equality constraints $h_j(\mathbf{x}) = 0$ since the smallest amount is added when all the constraints are zero; otherwise, a substantial penalty is imposed. This means that this approach encourages the maintenance of the feasibility of the solution assuming we start with an initial solution that is within the feasible region of the inequality constraints. These methods are sometimes called interior point methods.

A sequence of problems are generated by starting with an arbitrarily large value for r_0 and then using $r_{k+1} = r_k/c$ where $c > 1$ and solving the resulting sequence of unconstrained optimization problems. The unconstrained minimization steps may of course present formidable difficulties for some problems. A simple stopping criteria is to examine

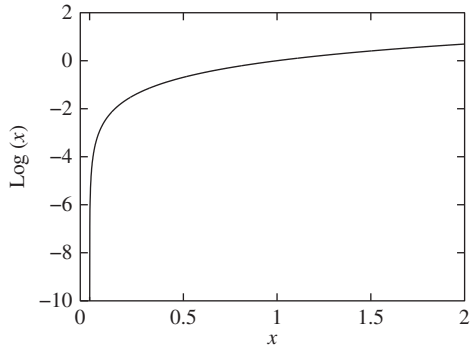

FIGURE 8.13 Graph of $\log_e(x)$.

the difference between the value of $f(\mathbf{x})$ between successive unconstrained optimizations. If the difference is below a specified tolerance then stop the procedure.

There are various alternatives to this algorithm. For example, a reciprocal barrier function can be used instead of the preceding logarithmic function. The barrier term can be replaced by a penalty function term of the form

$$\sum_{i=1}^{p} \max(0, g_i(\mathbf{x}))^2$$

This term will add a substantial penalty if $g_i(\mathbf{x}) < 0$; otherwise, no penalty is applied. This method has the advantage that feasibility is not required and is called an exterior point method. However, the resulting unconstrained problems may present additional problems for the unconstrained minimization procedure. For more details on these methods see Lesdon et al. (1996).

Although purpose-built software is available for this method, a very simple illustration of its operation is given. The program shows some steps of the method for solving a constrained minimization problem; notice that care must be taken in the choice of the initial r value and its reduction factor. To avoid the need for the derivative, the MATLAB function `fminsearch` is used to solve the following unconstrained problem first using the interior point method.

$$\text{Minimize } x_1^2 + 100x_2^2$$

subject to

$$4x_1 + x_2 \geq 6 \tag{8.23}$$

$$x_1 + x_2 = 3 \tag{8.24}$$

$$x_1, x_2 \geq 0 \tag{8.25}$$

```
% e3s810.m
r = 10; x0 = [5 5]
while r>0.01
    fm = @(x) x(1).^2+100*x(2).^2-r*log(-6+4*x(1)+x(2)) ...
        +1/r*(x(1)+x(2)-3).^2-r*log(x(1))-r*log(x(2));
    x1 = fminsearch(fm,x0);
    r = r/5;
    x0 = x1
end
optval = x1(1).^2+100*x1(2).^2

x0 =
     5     5

x0 =
    3.6097    0.2261

x0 =
    2.1946    0.1035

x0 =
    2.2084    0.0553

x0 =
    2.7463    0.0377

x0 =
    2.9217    0.0317

optval =
    8.6366
```

This shows convergence to the optimum solution (3,0), which satisfies the constraints. Using the exterior point method to solve the same problem we have:

```
% e3s811.m
r = 10; x0 = [5 5]
while r>0.01
    fm = @(x) x(1).^2+100*x(2).^2+1/r*min(0,(6+4*x(1)+x(2))).^2 ...
     +1/r*(x(1)+x(2)-3).^2+1/r*min(0,x(1)).^2+1/r*min(0,x(2)).^2;
    x1 = fminsearch(fm,x0);
    r = r/5;
    x0 = x1
end
optval = x1(1).^2+100*x1(2).^2
```

This produces the results:

```
x0 =
     5     5

x0 =
    0.2725    0.0027

x0 =
    0.9967    0.0100

x0 =
    2.1276    0.0213

x0 =
    2.7523    0.0275

x0 =
    2.9240    0.0292

optval =
    8.6352
```

Clearly the results are very similar.

8.12 Summary

In this chapter we have introduced a number of more advanced areas of numerical analysis. The genetic algorithm and simulated annealing are still topics for active research and development, and are mainly used to tackle difficult optimization problems. The conguate gradient method is well established and widely used for problems that present a range of difficulties. MATLAB functions have been provided to allow the reader to experiment and explore the problems more deeply. However, it must be remembered that no optimization technique is guaranteed to solve all optimization problems. The structure of the algorithms is well reflected by the structure of the MATLAB functions. The MathWorks Optimization Toolbox provides a useful selection of optimization functions, which may be used in both education and research.

Problems

8.1. Use the function `barnes` to minimize $z = 5x_1 + 7x_2 + 10x_3$ subject to $x_1 + x_2 + x_3 \geq 4$, $x_1 + 2x_2 + 4x_3 \geq 5$, and $x_1, x_2, x_3 \geq 0$.

8.2. Maximize $p = 4y_1 + 5y_2$ subject to $y_1 + y_2 \leq 5$, $y_1 + 2y_2 \leq 7$, $y_1 + 4y_2 \leq 10$, and $y_1, y_2 \geq 0$.

By introducing slack variables and subtracting one from each equality, write the constraints as equalities. Then apply the function `barnes` to solve this problem. Notice that the optimum value of p for this problem is equal to the optimum value of z in Problem 8.1. Problem 8.2 is called the dual of Problem 8.1. This is an example of an important theorem that the optima of the objective function of a problem and its dual are equal.

8.3. Maximize $z = 2u_1 - 4u_2 + 4u_3$ subject to $u_1 + 2u_2 + u_3 \leq 30$, $u_1 + u_2 = 10$, $u_1 + u_2 + u_3 \geq 8$, and $u_1, u_2, u_3 \geq 0$.
Hint: Remember to use slack variables to ensure that the main constraints are equalities.

8.4. Use the function `mincg`, with tolerance 0.005, to minimize Rosenbrock's function

$$f(x,y) = 100\left(x^2 - y\right)^2 + (1-x)^2$$

starting with the initial approximation $x = 0.5$, $y = 0.5$ and using a line-search accuracy 10 times the machine precision in the MATLAB function `fminsearch`. To obtain an impression of how this function varies, plot it in the range $0 \leq x \leq 2$, $0 \leq y \leq 2$.

8.5. Use the function `mincg`, with tolerance 0.00005, to minimize the five-variable function

$$z = 0.5\left(x_1^4 - 16x_1^2 + 5x_1\right) + 0.5\left(x_2^4 - 16x_2^2 + 5x_2\right) + (x_3 - 1)^2 + (x_4 - 1)^2 + (x_5 - 1)^2$$

Use $x_1 = 1$, $x_2 = 2$, $x_3 = 0$, $x_4 = 2$, and $x_5 = 3$ for the starting values in `mincg`. Experiment further with other starting values.

8.6. Use the function `solvercg` to solve the matrix equation $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Check the accuracy of the solution by finding the value of `norm(b-Ax)`.

8.7. Maximize the function $y = 1/\{(x-1)^2 + 2\}$ in the range $x = 0$ to 2 using the function `optga`. Use different initial population sizes, mutation rates, and numbers of generations. Notice that this is not a simple exercise since for each set of conditions it is necessary to solve the problem several times to take account of the random nature of the process. Given that the optimum value of the function is 0.5, plot the

error in the optimum value of the function for each run under a particular set of parameters. Then change one of the parameters and repeat the process. Differences in the plots may or may not be discernible.

8.8. Plot the function $z = x^2 + y^2$ in the range $0 \le x \le 2$ and $0 \le y \le 2$. The genetic algorithm given in Section 8.6 may be applied to maximize functions in more than one variable. Use the MATLAB function `optga` to determine the maximum value of the preceding function. In order to do this you must modify the `fitness` function so that the first half of the chromosome corresponds to values of x and the second half to values of y and these chromosomes must map to the values of x and y. For example, if an 8-bit chromosome 10010111 is split into two parts, 1001 and 0111, it will convert to $x = 9$ and $y = 7$.

8.9. Use the function `golden`, given in Section 8.3, to minimize the single-variable function $y = e^{-x}\cos(3x)$ for x in the range 0 to 2. Use a tolerance of 0.00001. You should check your result using the MATLAB function `fminsearch` to minimize the same function in the same range. As a further confirmation you might plot the function in the range 0 to 4 using the MATLAB function `fplot`.

8.10. Use the simulated annealing function `asaq` (with the same values of parameters used in the call of the function in Section 8.9) to minimize the two-variable function f where

$$f = (x_1 - 1)^2 + 4(x_2 + 3)^2$$

Compare your result with the exact answer, which is clearly $x_1 = 1$ and $x_2 = -3$. This function has only one minimum in the region considered. As a more demanding test, use the same call as used for the preceding function to minimize the function f where

$$f = 0.5\left(x_1^4 - 16x_1^2 + 5x_1\right) + 0.5\left(x_2^4 - 16x_2^2 + 5x_2\right) - 10\cos\{4(x_1 + 2.9035)\}\cos\{4(x_2 + 2.9035)\}$$

The global optimum for this problem is $x_1 = -2.9035$ and $x_2 = -2.9035$. Try several runs of the function `asaq` for this problem. All runs may not provide the global optimum since this problem has many local optima.

8.11. A method for solving a system of nonlinear equations is to re-express them as an optimization problem. Consider the system of equations

$$2x - \sin((x+y)/2) = 0$$
$$2y - \cos((x-y)/2) = 0$$

These can be rewritten as

$$\text{minimize } z = (2x - \sin((x+y)/2))^2 + (2y - \cos((x-y)/2))^2$$

Use the MATLAB function `minscg` to minimize this function using the starting point $x = 10$ and $y = -10$.

8.12. Write a MATLAB script that provides three-dimensional plots for the function $z = f(x,y)$ defined as follows:

$$z = f(x,y) = (1-x)^2 e^{-p} - pe^{-p} - e^{(-(x+1)^2 - y^2)}$$

where p is defined by

$$p = x^2 + y^2$$

for x and y in the range $x = -4 : 0.1 : 4$ and $y = -4 : 0.1 : 4$. The script should use the MATLAB `surf` and `contour` functions to provide separate three-dimensional and contour plots. Use the MATLAB function `ginput` to select and assign to an appropriate matrix three points that appear to be optimal on the contour plot. Use the function $z = f(x,y)$ defined earlier to find the values of z for these points. Then find the maximum and minimum of these z values using the MATLAB functions `max` and `min`. Finally, approximate the global minimum and maximum of the function.

An alternative method of finding the minimum is to use the MATLAB function `fminsearch` in the form `x = fminsearch(funxy,xv)`, where `funxy` is an anonymous function or user-defined function given by the user and `xv = [-4 4]` is a vector of initial approximations to the location of the minimum. Experiment with different intial approximations to see if your results vary.

8.13. Solve the minimization problem described in Problem 8.12 using the continuous genetic algorithm. Use the MATLAB function `contgaf`.

8.14. Write a MATLAB script to minimize $f(x) = x_1^4 + x_2^2 + x_1$ subject to $4x_1^3 + x_2 > 6$, $x_1 + x_2 = 3$, and $x_1, x_2 > 0$. Use the sequential unconstrained minimization technique with a logarithmic barrier function and an initial approximation vector of $x = [5,\ 5]$. Use an initial value for the parameter r_0 of 10, a reduction parameter $c = 5$, and $r_{k+1} = r_k/c$. Continue iterations while r_k is greater than 0.0001.

8.15. Write a MATLAB script to solve the constrained optimization problem described in Problem 8.14. However, use the penalty function of the form $[\min(0,\ g_i(\mathbf{x}))]^2$ instead of the logarithmic barrier function, where the $g_i(\mathbf{x})$ are greater than or equal to the constraints. Use the same initial starting point and values for r_0 and c. Compare the solution you find with this method to that achieved with Problem 8.14.

8.16. Solve the Rosenbrooks two-variable optimization problem:

$$\text{Minimize } f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

using the MATLAB function `asaq` with initial approximation $[-1.2\ 1]$. With quenching factor 0.9, the upper and lower bounds for the variables given by -10 and 10, respectively, initial temperature value of 100 and the maximum number of main iterations equal to 800. The solution to this problem is $[1\ 1]$.