

Chapter 2

Optimization Algorithms

This chapter will give a survey of some of the most commonly used optimization algorithms within the context of parameter characterization. The idea is not to give any detailed mathematical description of numerical optimization, since, on that topic one can find a decent number of great books (e.g., [1–3]). As the main topic of this book are the inverse analyses in structural engineering context, the goal is to present, to a reasonable extent, mathematical theory behind most commonly used optimization algorithms, so that they can be understood and easily implemented into a practical inverse analysis procedure.

Mathematically speaking, optimization is the minimization or maximization of a function subjected or not to constraints on its variables (parameters). In order to solve any optimization problem numerically, nowadays there is a wide variety of algorithms at our disposal. As we already saw in the previous chapter, these algorithms are starting from some *initial guess* of the parameters, and then they generate sequence of iterates which terminates, when either no more progress can be made, or when it seems that a solution point has been approximated with sufficient accuracy. Within each iteration, certain information on objective function are gathered (i.e. the value of objective function, the values of first derivatives, the values of second derivatives), and based on these computations the new iterate with a lower function value is estimated. It should be mentioned however, that there also exist *non-monotone algorithms* that do not insist on a decrease in the objective function at every step. No matter if the algorithm leads to a monotone decrease of the objective function or not, the main difference between the optimization algorithms is the way on which they pass from one iteration to another. This characteristic of the algorithm determines to a large extent its performance for a given optimization problem.

In the following pages two different strategies for computing next iteration from the previous one will be presented in more details, as they are used most frequently in nowadays available optimization algorithms. The first one is the *line search* strategy in which the algorithm chooses a direction \mathbf{p}_k and then searches along this direction for the lower function value. The second strategy is called the *trust region* in which the information gathered about the objective function is used to construct a model function whose behavior near the current iterate is *trusted* to be similar

enough to the actual function. Then the algorithm searches for the minimizer of the model function inside the trust region.

The last part of this chapter is devoted to a brief description of Genetic Algorithms, as they can be very useful in the problems when the objective function has a lot of local minima.

Before describing above mentioned optimization algorithms let us first say couple of words about least squares problems, or the problems in which the objective function has a form of summation of the squares of differences. As it was shown in Chap. 1, the function to be minimized that emerges from the inverse analyses (Eq. 1.1) is of this type.

2.1 Least Squares Problems

In least squares problems the objective function to be minimized has the following form

$$f(x) = \frac{1}{2} \sum_{i=1}^m r_j^2(x) \quad (2.1)$$

where r_j are called residuals.

The objective function of the least squares type emerges from many problems, and probably represents the most frequent optimization problem. In any engineering or scientific field where a parameterized models are used to fit the actual data, the function of the type (2.1) is used to measure discrepancy of the computed quantities and those that are measured.

If we compare function (2.1) with (1.1) from the previous chapter we can notice that it is of the same type. In this case, each residual represents the difference between computed quantities and their measured counter-part, and therefore represent the function of sought parameters.

To see why this special form of the objective function usually makes the least squares problems easier to solve than the general ones, let us first collect all the individual components into a residual vector \mathbf{R} , namely

$$\mathbf{R}(x) = [r_1(x), r_2(x), \dots, r_m(x)]^T \quad (2.2)$$

Using this notation, the objective function can be written as

$$f(x) = \frac{1}{2} \|\mathbf{R}(x)\|^{L^2} \quad (2.3)$$

The derivatives of the objective function can be expressed in terms of the Jacobian matrix \mathbf{J} , which, in the case when \mathbf{x} is n -dimensional vector will be $m \times n$ matrix, namely

$$\mathbf{J}(\mathbf{x}) = \left[\frac{\partial \mathbf{R}}{\partial \mathbf{x}} \right] = \begin{bmatrix} \frac{\partial r_1}{\partial x_1} & \frac{\partial r_1}{\partial x_2} & \cdots & \frac{\partial r_1}{\partial x_n} \\ \frac{\partial r_2}{\partial x_1} & \frac{\partial r_2}{\partial x_2} & \cdots & \frac{\partial r_2}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial r_m}{\partial x_1} & \frac{\partial r_m}{\partial x_2} & \cdots & \frac{\partial r_m}{\partial x_n} \end{bmatrix} \quad (2.4)$$

The gradient of the objective function can be written in terms of the Jacobian as

$$\nabla f(\mathbf{x}) = \sum_{i=1}^m r_i \frac{\partial r_i}{\partial \mathbf{x}} = \sum_{i=1}^m r_i \begin{bmatrix} \frac{\partial r_i}{\partial x_1} \\ \frac{\partial r_i}{\partial x_2} \\ \cdots \\ \frac{\partial r_i}{\partial x_n} \end{bmatrix} = \mathbf{J}^T \cdot \mathbf{R} \quad (2.5)$$

Hessian matrix of second derivatives of the objective function can be written as

$$\nabla^2 f(\mathbf{x}) = \sum_{i=1}^m \frac{\partial r_i}{\partial \mathbf{x}} \cdot \left(\frac{\partial r_i}{\partial \mathbf{x}} \right)^T + \sum_{i=1}^m r_i \frac{\partial^2 r_i}{\partial \mathbf{x}^2} = \mathbf{J}(\mathbf{x})^T \cdot \mathbf{J}(\mathbf{x}) + \sum_{i=1}^m r_i \frac{\partial^2 r_i}{\partial \mathbf{x}^2} \quad (2.6)$$

What can be observed from the Eq. 2.6 is that the part of the second derivatives can be expressed by the Jacobian matrix. It practically means that, once the first derivatives are computed, we can also compute part of the Hessian matrix for the same computational cost. The possibility to compute “for free” the Hessian matrix once the Jacobian is available represents a distinctive feature of least squares problems. Since near the solution the residuals are close to zero, it also means that the contribution of the second part of Hessian matrix is very small. Therefore, it is reasonable to approximate the Hessian matrix with the first part only.

$$\nabla^2 f(\mathbf{x}) \approx \mathbf{J}(\mathbf{x})^T \cdot \mathbf{J}(\mathbf{x}) \quad (2.7)$$

This approximation is adopted in many applications as it provides an evaluation of the Hessian matrix without computing any second derivatives of the objective function.

2.2 Line Search Method

Within the line search method, in each iteration it is required to find the direction, say \mathbf{p}_k and then to decide how far to go along that direction. Therefore, if the current iteration is denoted by vector \mathbf{x}_k , the new iteration is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (2.8)$$

The new iteration is thus uniquely defined by the direction \mathbf{p}_k and by a positive scalar α_k called the step length. The success of a line search method depends on effective choices of both the direction and the step length. Based in the adopter strategy of solving for the two abovementioned quantities, we can distinguish between different lines search algorithms.

2.2.1 Line Search with Steepest Descend Direction

Most of the line search algorithms are implemented so that they have a monotone decrease of objective function. This means that the direction \mathbf{p}_k needs to be a descending direction. Therefore the most logical direction to move along would be the *steepest descent* direction or the one that defines direction \mathbf{p}_k as

$$\mathbf{p}_k = -\frac{\nabla f_k}{\|\nabla f_k\|} \quad (2.9)$$

The advantage of this choice is that it involves the calculation of only first derivatives.

After the direction is fixed, the algorithm needs to compute the step length. In the step length selection we are facing a tradeoff. Obviously we would like to have as good as possible reduction of the objective function, but at the same time we don't want to spend a lot of time in searching for it. The ideal case would be to find a step length as a global minimizer of the following function

$$\phi(\alpha_k) = f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \quad (2.10)$$

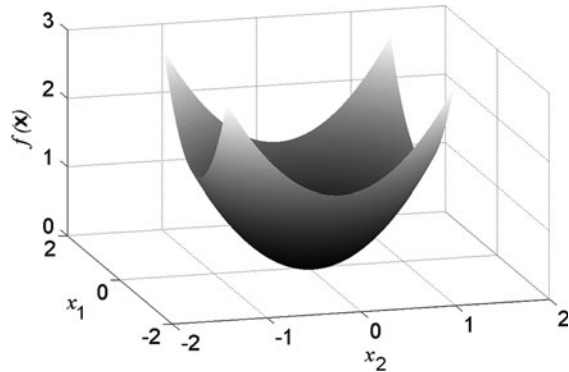
However, it may be computationally too expensive to identify this value. Because of that, most practical strategies that use the line search approach are finding an inexact minimizer of (2.10) for the acceptable computational cost. This minimizer should satisfy some criteria for the function reduction but without a guarantee that it is the global minimizer of the function along that direction. Typical line search algorithms try a sequence of candidate values for the step length and then select the best of them.

In order to see how the step length influences the performance of line search method let us consider a simple example. Let the objective function be given by

$$f(\mathbf{x}) = x_1^2 + x_2^2 \quad (2.11)$$

The function to be minimized is convex and has only one minimum corresponding to coordinates $\mathbf{x} = [0,0]^T$. Figure 2.1 visualizes the function on the domain -2 to 2 .

Fig. 2.1 The objective function given by Eq. 2.11 on the domain $x_1, x_2 \in [-2, 2]$



In the following MATLAB code a simple strategy with three trial step lengths is implemented. Within this strategy the three trial steps are: the reference step length, one-quarter of it, and two times reference length. The algorithm starts in the first iteration with some initial step length, taken as a reference one, and for each of the three trial steps it computes the reduction of the function. It further picks up the step with the largest reduction of the function and in the following iteration adopts it as a reference one. This simple strategy allows for continuous elongation (or shortening) of the step length with respect to the starting value, in order to make better use of the selected direction.

Listing given in the following page includes three MATLAB codes. The first one is the main optimization routine, the second one is used to plot the results and the third one is a MATLAB function that computes the value of the objective function for given parameters.

In the main routine the initial step length, together with other options, is chosen by the user at the beginning of the code. The optimization is terminated when the residual is smaller than a given value. The optimization path is recorded in the matrix *itiner* that has the number of rows equal to the number of iterations. The first derivatives are computed by finite differences, even though in this simple case they can be computed analytically. In more general cases, that will be discussed further in the book, the objective function will not be given in analytical form and therefore the derivatives will always be computed by finite differences.

Second MATLAB code serves for the visualization of the optimization results as mentioned previously. One optimization result is given in Fig. 2.2 that shows the convergence after 8 iterations starting from the point given by $\mathbf{x} = [1.1, 1.5]^T$, for the initial step length equal to 0.1. It may be observed from the figure that, from the first to the fourth iteration the algorithm constantly enlarges the step length after which it starts to shorten it as it approaches the solution. The selection of the initial step is not influencing very much the performance of the optimization algorithm as long as it is selected within a reasonable range for the given problem. Table 2.1 shows the optimization path for this problem for the two different initial step lengths: $\alpha_{IN} = 0.1$ and $\alpha_{IN} = 0.5$. It may be observed that, even though the steps

```

*****
% Optimization algorithm based on line search method.
% Step length identification by three trial steps

% Setting the options for the optimization
guess=rand(2,1)*2-ones(2,1); % Initial vector of parameters
pert=0.001; % Perturbation for the first derivatives
length=0.1; % Initial step length
resMIN=1e-6; % The value of residual at the termination

% Optimization cycle
res=10;
iter=0;
while res>resMIN
    itiner(iter+1,1:2)=guess';
    e=exfun(guess);
    itiner(iter+1,3)=e;
    iter=iter+1;
    for i=1:size(guess,1)
        guesssp=guess;
        guesssp(i)=guesssp(i)+pert;
        e1=examlp2(guesssp);
        grad(i,1)=(e1-e)/pert;
    end
    stpdsc=-grad/norm(grad);
    % Trying different step lengths
    guess1=guess+length*stpdsc;
    guess2=guess+(length/4)*stpdsc;
    guess3=guess+(length*2)*stpdsc;
    eltr=exfun(guess1);
    e2tr=exfun(guess2);
    e3tr=exfun(guess3);
    best=min([eltr,e2tr,e3tr]);
    if eltr==best
        guess=guess1; step=1;
    end
    if e2tr==best
        guess=guess2; step=2; length=length/4;
    end
    if e3tr==best
        guess=guess3; step=3; length=length*2;
    end
    res=guess'*guess; % Computing residual
    itiner(iter+1,1:2)=guess';
    itiner(iter+1,3)=eTR;
end
*****
*****

```

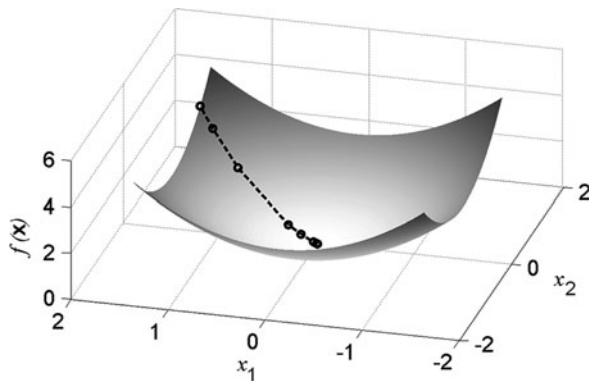
```

% Plotting the results of the optimization
N=0;
for i=-1.2:0.01:1.2
    N=N+1;
    M=0;
    for j=-1.2:0.01:1.2
        M=M+1;
        fun(N,M)=exfun([i;j]);
    end
end
i=-1.2:0.01:1.2; j=-1.2:0.01:1.2;
figure(1)
surf(i,j,fun,'LineStyle','none')
grid on
hold on
plot3(itiner(:,1),itiner(:,2),itiner(:,3),'MarkerSize',10,'Marker','o','LineWidth',3,'LineStyle','--','Color',[0 0 0])
view([-1,1,3])
hold off
*****

*****
% Objective function
function e=exfun(x);
e=x(1)^2+x(2)^2;
*****

```

Fig. 2.2 Result of optimization – objective function and the itinerary of the optimization



were different, both optimizations were terminated practically at the same result, after the same number of iterations. Since it is easy to change this parameter it may be verified that the algorithm will have the same performance (in terms of number of iterations) also for different values of initial step lengths within the range $[0,1]$.

A disadvantage of this approach is that every trial step length requires one computation of the function. In this particular case, for the number of parameters

Table 2.1 Optimization results for two different initial step lengths

	x_1	x_2	$f(\mathbf{x})$	x_1	x_2	$f(\mathbf{x})$
Iteration	$\alpha_{\text{IN}} = 0.1$			$\alpha_{\text{IN}} = 0.5$		
1	1.1000	1.5000	3.4600	1.1000	1.5000	3.4600
2	0.9817	1.3387	2.7560	0.5086	0.6936	0.7398
3	0.7452	1.0162	1.5879	-0.0828	-0.1127	0.0196
4	0.2720	0.3711	0.2117	0.0650	0.0889	0.0121
5	0.1537	0.2098	0.0677	0.0281	0.0385	0.0023
6	0.0355	0.0485	0.0036	-0.0089	-0.0119	0.0002
7	0.0059	0.0082	0.0001	0.0059	0.0082	0.0001
8	-0.0015	-0.0019	0.0001	-0.0015	-0.0019	0.0001

to identify being equal to 2 and with 3 trial steps each iteration included 6 evaluations of function. Considering that the number of iterations was 8, it resulted in total of 48 function evaluations.

Some algorithms that are using inexact line search approach are introducing different criteria that need to be satisfied for the approximated minimization of Eq. 2.10. For example one criterion that can be used is that the step length should give a sufficient reduction of the objective function. This can be measured by the inequality that is found in the literature as *Armijo criterion* ([4, 5]), namely

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \nabla f_k^T \mathbf{p}_k \quad (2.12)$$

where c_1 is some small scalar constant. The right-hand side of the inequality (2.12) is a linear function of step length and it has a negative slope. It lies above the objective function (or at least it is the case in close proximity of the current iterate) which is guaranteed with c_1 being a small positive number.

It is of course evident that for any sufficiently small value of step length, the Armijo criterion is satisfied. Therefore the goal should be to find the largest possible α_k for which it is satisfied.

A possible implementation of this approach is given in the following MATLAB code. Here the algorithm starts from some initial value of step length and then continues doubling it, until it finds the largest values of α_k that satisfies the inequality. This value is accepted as a reference one for the next iteration. In the case when it violates the inequality (2.12) the step length is divided by 2.


```

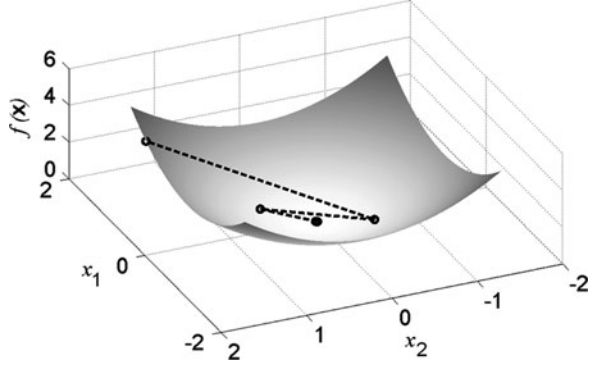
*****
% Optimization algorithm based on line search method.
% Step length identification by Armijo condition

% Setting the options
length=5; % Starting test length for the first iteration
c1=1e-4; % Coefficient for Armijo criterion
pert=0.001; % Perturbation for the first derivatives
guess=rand(2,1)*2-ones(2,1); % Initial vector of parameters
resMIN=1e-6; % The value of residual at the termination

% Optimization cycle
res=10;
iter=0;
while res> resMIN
    itiner(iter+1,1:2)=guess';
    e=exfun(guess);
    itiner(iter+1,3)=e;
    iter=iter+1;
    for i=1:size(guess,1)
        guesssp=guess;
        guesssp(i)=guesssp(i)+pert;
        e1=exfun(guesssp);
        grad(i,1)=(e1-e)/pert;
    end
    stpdsc=-grad/norm(grad);
    % Armijo criterion
    foundlength=0;
    shrt=0; % Total number of eshortening
    while foundlength==0
        guessTR=guess+length*stpdsc;
        eTR=exfun(guessTR);
        if eTR>e+c1*length*grad'*length*stpdsc;
            length=length/2; % Getting back to previous length
            guessTR=guess+length*stpdsc;
            eTR=exfun(guessTR);
            if eTR<e+c1*length*grad'*length*stpdsc;
                foundlength=1;
            else
                length=length/2;
                shrt=shrt+1;
            end
        else
            if shrt>0
                break
            end
            length=length*2;
        end
    end
    % Updating values
    guess=guess+length*stpdsc;
    best=eTR;
    res=guess'*guess;
    itiner(iter+1,1:2)=guess';
    itiner(iter+1,3)=eTR;
end
*****

```

Fig. 2.3 Result of the optimization with Armijo criterion



The remaining two routines are the same as in previous case and can be used also for this optimization. Figure 2.3 visualizes the results of the optimization starting from the same initial point as in previous case, using Armijo criterion for quit large initial step length (equal to 5).

In this case the optimization was terminated after six iterations instead of eight in the previous one visualized in Fig. 2.2. With the adopted strategy however the total number of step lengths that will be tried is not fixed. It means that also the number of function evaluations within the iteration is not limited. Unlike the previous case where in each iteration there were only three candidates, in this one the trials will be repeated until the violation of Armijo criterion. This fact makes the presented implementation more dependent on the initial step length. Repeating the optimization it can be verified that for the initial step length equal to 1, the optimization is terminated after only five iterations, while the starting value of 0.2 involves nine iterations. Some improvements of the approach are of course possible, like for example the one that will put a limitation of the number of increments of the step length in order to avoid waste of computing time in the case when the initial step length is chosen to be quit small.

Satisfying Armijo criterion is not enough by itself to ensure that the algorithm makes reasonable progress since, as previously mentioned, the inequality (2.12) is satisfied for any sufficiently small α_k . This criterion is therefore sometimes combined with the *curvature condition* which requires that the step length satisfies the following condition

$$\nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \cdot \mathbf{p}_k \geq c_2 \nabla f_k^T \cdot \mathbf{p}_k \quad (2.13)$$

where c_2 is some scalar constant with the value within the range $[c_1, 1]$. The curvature criterion practically means that the slope of Eq. 2.10 (a function that we are approximately minimizing) at the α_k is c_2 times larger than the initial slope at \mathbf{x}_k . Since the initial slope is negative it means that the requirement states that the slope at acceptable step length should be less negative. This is a reasonable requirement since strong negative slope would indicate a possible significant reduction of the objective function by moving further along the chosen direction.

The sufficient decrease and curvature condition are known collectively as *Wolfe conditions*. It represents a strategy for selection of stopping criterion which should provide better trials than Armijo condition alone, but since they require additional gradient computation, as the left-hand side of (2.13) is the derivative of (2.10) at α_k , it may involve even larger number of function evaluations in some cases.

The implementation of the two strategies given here should serve to put in evidence that the step length for the steepest descent plays an important role in overall performance of the optimization algorithm. Both of the two discussed strategies showed some oscillatory behavior around the solution which leads to overall increase in the iterations. Figure 2.3 puts in evidence that already third iteration is located quit close to the result. This suggests that the steepest descend algorithm can be combined with some other method once that it approaches the solution.

2.2.2 Line Search with Newton Direction

Another important search direction is the *Newton direction*. This direction is derived from the second-order Taylor series approximation of the objective function. Using this series to approximate the real objective function around current iterate, and truncating it after the second derivative term will give the following model function

$$f(\mathbf{x}_k + \mathbf{p}_k) \approx m_k(\mathbf{p}_k) = f(\mathbf{x}_k) + \mathbf{p}_k^T \nabla f(\mathbf{x}_k) + \frac{1}{2} \mathbf{p}_k^T \nabla^2 f(\mathbf{x}_k) \cdot \mathbf{p}_k \quad (2.14)$$

Assuming that $\nabla^2 f(\mathbf{x}_k)$ is positive definite, Newton direction can be obtained by finding \mathbf{p}_k that minimizes the model function $m_k(\mathbf{p}_k)$. Finding the first derivatives of m_k with respect to direction and simply setting it to zero will give us the minimizer of the model function, namely

$$\mathbf{p}_k = -(\nabla^2 f_k)^{-1} \nabla f_k \quad (2.15)$$

Unlike the steepest descent direction, where as we already saw an additional computational effort needs to be made to further identify the step length, Newton direction has a “natural” step length equal to 1. This can be seen from the way the direction is derived. Since it is determined by setting the first derivative of the model function to zero, it means that the Newton direction computed by (2.15) will be *exact* minimizer of model function. It further implies that the reliability of this direction depends on how much the true objective function differs from the model function. If they are almost the same, Newton direction can provide minimizer in one step only.

Let us now consider the same optimization problem as before where objective function is given by (2.11), and let us use the following MATLAB code to solve this problem using Newton direction.

```

*****
% Optimization algorithm based on line search method.
% That uses Newton direction with the step size equal to 1

% Setting the options
guess=rand(2,1)*2-ones(2,1); % Initial vector of parameters
pert=0.001; % Perturbation for the first derivatives
resMAX=1e-6; % Residual at termination

% Optimization cycle
res=10;
iter=0;

while res>resMAX
    e=exfun(guess);
    itiner(iter+1,1:2)=guess';
    itiner(iter+1,3)=e;
    iter=iter+1;
    % Computing first derivatives
    for i=1:size(guess,1)
        guessp=guess;
        guessp(i)=guessp(i)+pert;
        e1=exfun(guessp);
        grad(i,1)=(e1-e)/pert;
    end
    % Computing Hessian matrix
    HESS=comhess(@exfun,point,pert)
    newton=-inv(HESS)*grad; % Newton direction
    length=1; % Natural length
    guess1=guess+length*newton;
    eltr=exfun(guess1);
    itiner(iter+1,1:2)=guess1';
    itiner(iter+1,3)=eltr;
    guess=guess1;
    res=guess'*guess;
end
*****

*****
function HESS=comhess(FUNNAME,point,pert)
% Computing the Hessian matrix
pointp=point;
e0=FUNNAME(pointp);
for i=1:size(point,1)
    pointp=point;
    pointp(i)=pointp(i)+pert;
    e1=FUNNAME(pointp);
    pointp(i)=pointp(i)+pert;

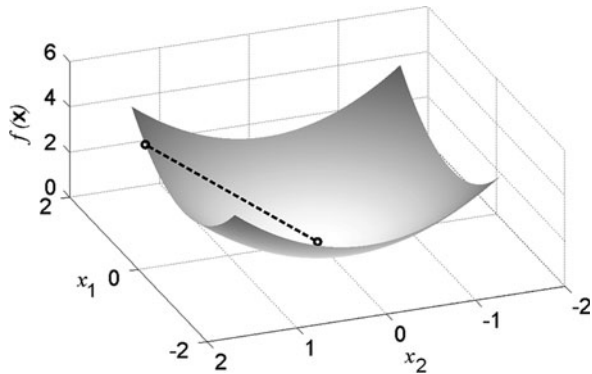
```

```

    e2=FUNNAME(pointp);
    Usnd(i)=(e0-2*e1+e2)/(pert^2);
end
% mixed derivative
pointp=point;
pointp(1)=pointp(1)+pert;
U112=FUNNAME(pointp); % term i+1,j
pointp=point;
pointp(2)=pointp(2)+pert;
U121=FUNNAME(pointp); % term i,j+1
pointp=point;
pointp(1)=pointp(1)+pert;
pointp(2)=pointp(2)+pert;
U1121=FUNNAME(pointp); % term i+1,j+1
mixed=1/pert*((U1121-U112)/pert-(U121-e0)/pert);
HESS(1,1)=Usnd(1);
HESS(1,2)=mixed;
HESS(2,1)=mixed;
HESS(2,2)=Usnd(2);
*****

```

Fig. 2.4 Result of the optimization with Newton direction – convergence after one step only

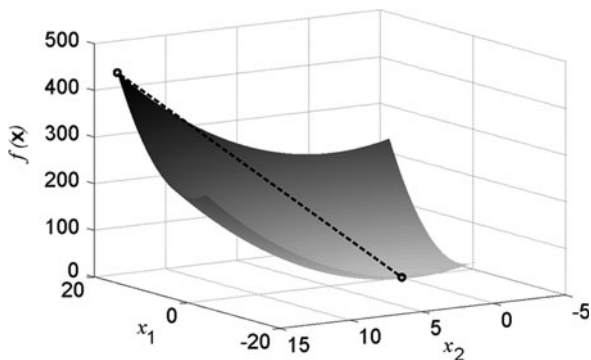


Note that the second routine given in the listing is a MATLAB function used to compute Hessian matrix. This function will be also used later in other optimization algorithms.

The optimization problem was solved starting from the same initial point as in previous cases and the result is visualized in Fig. 2.4.

The real power of Newton direction is evidenced in this example since the algorithm converged in one iteration only. Unlike steepest descend approach, which selects the direction, and then, depending on the strategy of the step length search, the algorithm may take couple of iterations to reach the solution, Newton direction immediately finds the step that exactly minimizes the model function. In this case the objective function was practically the same as the model function, and so the optimization terminated after only one iteration.

Fig. 2.5 Newton direction optimization starting far from the solution



The example considered here have simple, smooth and convex objective function that is suitable to be modeled by second-order Taylor series. It can be shown that it is the case, even farther from the solution. Figure 2.5 shows the result of the optimization when the initialization point was quit far from the solution being equal to $[15, 15]^T$. Still however, the Newton direction provided the solution in one iteration.

In this code, derivatives are computed by finite differences. This represents a disadvantage in terms of computing times. However, if the objective function, like in this case can, with acceptable accuracy, be represented by a second-order model, this drawback is acceptable since, as we saw, the Newton direction, with respect to steepest descent, provides a significant reduction in the number of iterations.

More serious problem of Newton direction is that, when Hessian is not positive definite, the Newton direction may not be defined, or if it is defined it may not be a descending direction. In order to overcome this problem there are different approaches in which the Hessian matrix is modified in order to make it positive definite and thereby yield a descent direction. These problems will arise in the situations where the objective function is more complicated than the one studied in this example. Therefore in the following pages the behavior of both steepest descent and Newton direction method will be analyzed on the least squares type objective function.

2.2.3 Line Search in Least Squares Problems

Let us consider the following analytical function

$$y(t) = t \cdot \sin(t) - \sqrt{t} \quad (2.16)$$

which, at the domain $t \in [0, 3]$ is visualized in Fig. 2.6. Now let us imagine that this expression represents the distribution of a certain physical value (here y) at some

Fig. 2.6 Graph of analytical function (2.16), used as “target” function

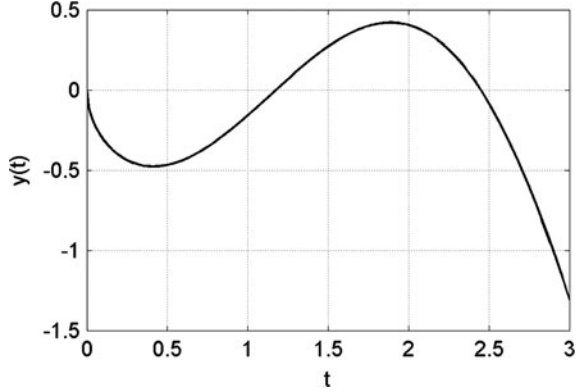
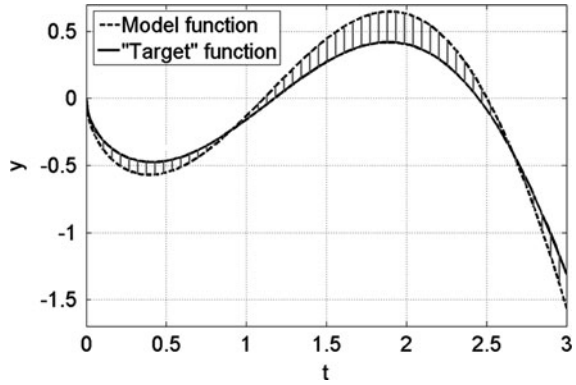


Fig. 2.7 Model and “target” function with the distances between the two curves over some grid of points



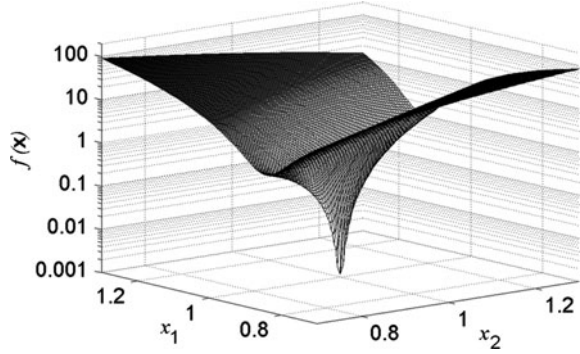
time or spatial coordinate t . Let us further suppose that we have some model function that represents a computed counter-part of this experiment given by

$$y^{COM}(t, \mathbf{x}) = x_1 \cdot t \cdot \sin(t) - \sqrt{x_2} \cdot t \quad (2.17)$$

The model function is function of two parameters collected in vector \mathbf{x} .

Let us assume that we would like to identify the two parameters x_1 and x_2 for which the model function (2.17) will match the “target” one given by (2.16). Obviously, the target function exists within the family of model functions and it is obtained for $\mathbf{x} = [1, 1]^T$. In order to design the inverse analysis procedure that will identify these two parameters the first step is to build the objective function that will quantify the discrepancy between the target and model function. This can be done using the discrepancy function in the least squares form that will represent a summation of squares of differences between the two function values for some grid over t , as shown in Fig. 2.7. The objective function will have the following form

Fig. 2.8 Discrepancy function as summation of squares of differences between the two curves



$$f(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^{N_p} [y(t_i) - y(t_i, \mathbf{x})]^2 \quad (2.18)$$

where N_p is the number of grid points over t used to compute the distance between the two curves. It should be mentioned that this is a typical formulation in any situation where parameterized models are used to predict some phenomenon. In such cases the discrepancy between model prediction and observed behavior is usually measured by the function of the type given by Eq. 2.18. In this case we simplified the situation by considering as observation “clean” analytical data so that the exact solution will exist within our model, in order to study only the behavior of different optimization approaches in the situation in which we know exactly what the solution is.

The objective function defined in this way has more complicated shape than in previously studied case and its form is visualized in Fig. 2.8.

Following MATLAB code can be used to minimize objective function (2.18) using steepest descend approach. The code is practically the same as previously given for optimization of function (2.11) except that here additional lines for different convergence criteria are inserted. Here the optimization is terminated not only in the case when residual is smaller than some prescribed value, but also if the changes of the parameters are smaller than certain value and if the number of iterations is larger than given number NUMIT. This is a common practice in more complicated optimizations since there, a single criterion is not enough and may even lead to the optimization that will never be terminated.


```

*****
% Optimization algorithm based on Line search with steepest
% descend, with Armijo condition
clear

% Setting the options
minchg=1e-5; % Min change in parameters between two iterations
guess=rand(2,1)*2;
minRES=1e-6; % Residual at termination
NUMIT=70; % Max number of iteration allowed
pert=0.001; % Perturbation for the derivatives
length=0.01; % Starting test length for the first iteration
c1=1e-2; % Coefficient for Armijo criterion

% Optimization cycle
res=10;
iter=0;
while res>minRES
    itiner(iter+1,1:2)=guess';
    e=funLSQ(guess);
    itiner(iter+1,3)=e;
    iter=iter+1;
    for i=1:size(guess,1)
        guessp=guess;
        guessp(i)=guessp(i)+pert;
        e1=funLSQ(guessp);
        grad(i,1)=(e1-e)/pert;
    end
    stpdsc=-grad/norm(grad);
    % Armijo criterion
    foundlengt=0;
    while foundlengt==0
        guessTR=guess+length*stpdsc;
        eTR=funLSQ(guessTR);
        if eTR>e+c1*length*grad'*length*stpdsc;
            length=length/2; % Getting back to previous lenght
            guessTR=guess+length*stpdsc;
            eTR=funLSQ(guessTR);
            if eTR<e+c1*length*grad'*length*stpdsc;
                foundlengt=1;
            else
                length=length/2;
            end
        else
            length=length*2;
        end
    end
    % Updating values
    guess=guess+length*stpdsc;

```

```

best=eTR;
res=best;
itiner(iter+1,1:2)=guess';
itiner(iter+1,3)=best;
% Checking convergence options
if abs(itiner(iter+1,1)-itiner(iter,1))<minchg ||
    abs(itiner(iter+1,2)-itiner(iter,2))<minchg;
    res=0;
end
if iter> NUMIT % Terminated after the iteration reach NUMIT
    res=0;
end
end
*****

*****

function e=funLSQ(x);
exper=load('lsqexp.txt');
A=x(1);
B=x(2);
cnt=0;
for i=0:0.01:3
    cnt=cnt+1;
    y(cnt)=A*sin(i)*i-sqrt(B*i);
end
e1=y'-exper(:,2);
e=e1'*e1;
*****

```

The second MATLAB code is used to compute the discrepancy function between model curve and “experimental” one. Here the “experimental” curve is placed in the file `lsqexp.txt` that is actually the graph of Eq. 2.16 (Fig. 2.6).

The result of the optimization is visualized in Fig. 2.9. Setting the total allowed number of iterations to 70 and other convergence criteria quite strict (minimum change in parameters $1\text{E-}5$ and minimum residual at the termination $1\text{E-}6$) the optimization terminates after reaching 70th iteration, at parameter values of $\mathbf{x} = [0.998, 0.995]^T$.

This objective function is a typical example which demonstrates that the steepest descend is not the best direction to go along. As it can be observed from the figure,

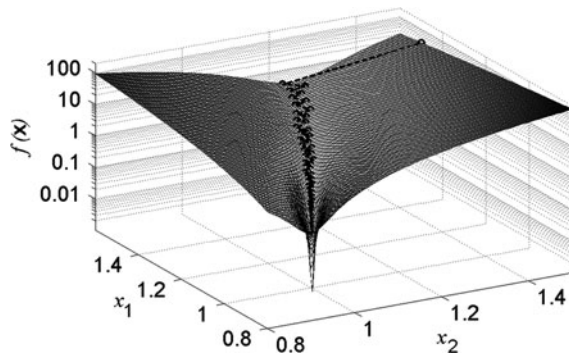


Fig. 2.9 Itinerary of steepest descend optimization of least squares discrepancy function given by Eq. 2.18

practically in all intermediate points within the optimization the steepest descend direction is not matching the direction of global minimum. Therefore, only the small steps can be achieved with which the optimization slowly approaches the minimum, which causes large number of iterations. Since the reason for poor performance is the direction itself, no improvements can be obtained with different strategies for step length selection.

The same problem can be solved using Newton direction. For this purpose a previously given MATLAB code with the same modifications as those for the steepest descend can be used.

The Newton direction approach turns out to be much more effective. Using the same tolerances as for the steepest descend the algorithm converges after only seven iterations but practically already the forth one is in the vicinity of the solution. Figure 2.10 visualizes the itinerary and Table 2.2 lists all the parameter values and corresponding objective function values.

The last example proved the superiority of the Newton direction in the cases when it can be computed, and when it is a descending direction. However, using this direction in the line search approach is not that effective all the time. Apart from being computationally more expensive with respect to steepest descend by involving the computation of second derivatives, there are two main problems connected with the successful implementation of Newton direction.

The first one is that, as already evidenced, the Newton direction minimizes exactly the model function which is a quadratic form of a real objective function around the current iterate. This is a good approximation in the vicinity of the current iterate but by going farther it starts to worsen. It may happen, when the minimizer of the model function is far from the current iterate, that it will be a poor minimizer of the real

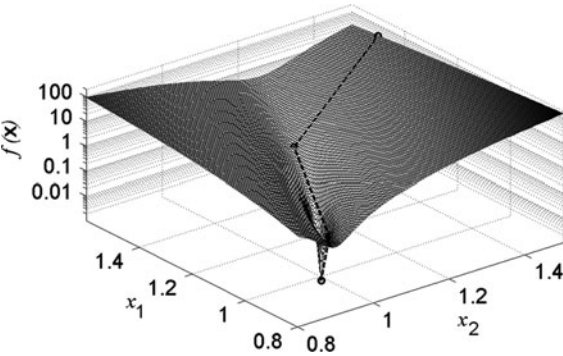


Fig. 2.10 Itinerary of Newton direction optimization of least squares discrepancy function given by Eq. 2.18

Table 2.2 Optimization results for Newton direction and LS function

Iteration	x_1	x_2	$f(\mathbf{x})$
1	1.5000	1.5000	38.3040
2	1.1337	1.2926	0.9928
3	0.9847	0.9450	0.0975
4	0.9966	0.9923	0.0008
5	0.9994	0.9987	0.0000
6	0.9994	0.9988	0.0000
7	0.9994	0.9988	0.0000

function, or it can even increase its value. This can be encountered especially in the cases when iterate is quite far from the global minimum point of the real objective function. An alternative approach that overcomes this problem is *Trust Region* approach and it will be discussed in more details further in this chapter.

The second problem is connected with the fact that Hessian matrix may not be positive definite which means that the Newton direction computed by Eq. 2.15 may not be a descent direction. This problem may be solved in the line search context by modifying the Hessian matrix in order to force it to be positive definite. Unlike the first abovementioned problem, where the difference between the model function and the real one can be verified only after the value of the objective function is evaluated for the next iteration (computed by adding Newton direction to the current iterate) tackling of the second problem is not that time consuming. After the Hessian matrix is computed, if it runs out not to be positive definite, there is no need to use this step to evaluate the objective function as it is not guaranteed that it will reduce the objective function. Instead, an appropriate modification of Hessian matrix can be adopted in order to yield a descending direction. Implementing such modification in the algorithm is not computationally “expensive”, as it is not involving any further evaluation of the objective function.

The Hessian matrix modification is usually obtained by adding either a positive diagonal matrix or full matrix to the true Hessian. However the step computed by modified Hessian matrix loses its “natural” length of 1, as it is not anymore the exact minimizer of the model function. If the modified Hessian is positive definite, the direction computed using it will be a descending one but the step length may have to be adjusted to ensure a reasonable decrease of the objective function like in the line search approach with steepest descent direction. Usually in the line search algorithms that use modified Newton direction, the step length 1 is tried first and then some stopping conditions like those discussed previously are checked. If they are not satisfied, the algorithm should modify the step length, otherwise it may proceed to the next iteration.

Even though the fact that Hessian is not positive definite can be verified a priori avoiding a useless objective function evaluation, still the need to evaluate the step length makes the approach a bit more time consuming with respect to the classical Newton direction line search. However, with this modification, as it is usually not performed at each iteration, the Newton direction line search usually tends to keep good convergence rate, and in general is more effective than the steepest descend.

In order to illustrate the effect of modification of Hessian matrix let us consider the following numerical example. Let us assume that the target distribution of some physical value on the domain $t \in [0, 3]$, that in this context represents “experimental” data, is given by the following equation (Fig. 2.11)

$$y = 2.5 \cdot \sin t + e^{\cos(2.5t)} \quad (2.19)$$

Like in previously discussed case, let us suppose that the computed counter-part of the “experimental” values is given with the following model function

$$y^{COM}(t, \mathbf{x}) = x_1 \cdot \sin t + e^{\cos(x_2 t)} \quad (2.20)$$

Fig. 2.11 Graph of analytical function (2.19), used as “target” function

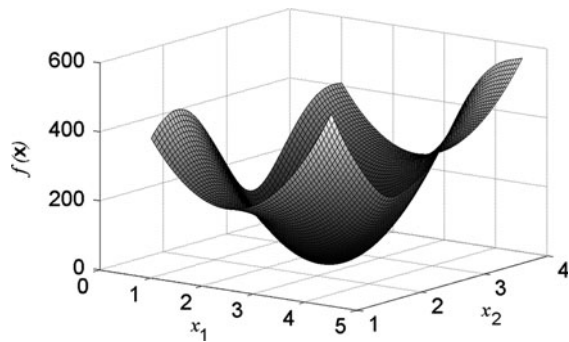
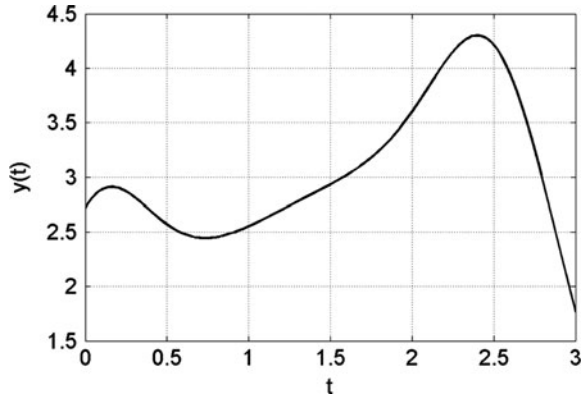


Fig. 2.12 The objective function to be minimized

which depends on two parameters collected in vector \mathbf{x} .

The goal is to identify the two model parameters by constructing a procedure that will minimize the discrepancy function in the least squares form, like the one given by Eq. 2.18. The objective function to be minimized is visualized in Fig. 2.12.

The following MATLAB code can be used to minimize this objective function using both classical Newton direction and modified one. Here, the modification is performed by adding a diagonal matrix that for this particular case turned out to be good enough. In general, a wide variety of types of modifications can be used, and the reader is referred to the reference [2] for more details on this topic. As mentioned earlier, after the Hessian modification, it is also needed to check for the step length, as it cannot be anymore a priori assumed as equal to 1. In the code given here the first tried value was 0.5, and then it is further reduced if necessary. The objective function studied here is characterized with large number of local minima when the parameters are changing in wider ranges. In order to avoid trapping into one of those, as the tackling of local minima is not an issue at this stage, the goal was to avoid large steps along the descending directions. Therefore the maximum allowed step length for modified Newton direction was assumed to be 0.5 unlike the common practice where it is usually 1.

In order to evidence the advantage of least squares method in terms of easy approximation of Hessian matrix by computing only first derivative, the following

code has an option either to compute full Hessian matrix, or to approximate it by using just Jacobian (Eq. 2.7). This option is set by the variable `Hessapp`. By attributing value 1 to it, Hessian approximation will be used, otherwise, also second derivatives will be computed. Both of the functions used for this computation are given at the bottom of the code. Note that with respect to the case presented in Sect. 2.2.2, also the routine for computing Hessian matrix is modified in order to take into account function that gives vector-value (i.e. vector of residuals).

```
*****
% Line search algorithm with Newton direction with possibility
% of Hessian matrix modification and approximation
clear

% Setting the options
minchg=1e-4; % Minimum change in parameters
MAXIT=30; % Maximum allowed number of iterations
guess=[1.5;2.9];
pert=1e-6; % Perturbation for the first derivatives
res=10;
HessMod=0; % Indication for Hessian modification
Hessapp=0; % Approximating Hessian matrix

% Optimization cycle
iter=0;
while res>1e-6
    eV=funLSQ1(guess);
    e=0.5*eV'*eV;
    itiner(iter+1,1:2)=guess';
    itiner(iter+1,3)=e;
    iter=iter+1;
    for i=1:size(guess,1)
        guessp=guess;
        guessp(i)=guessp(i)+pert;
        eV=funLSQ1(guessp);
        e1=0.5*eV'*eV;
        grad(i,1)=(e1-e)/pert;
    end
    guessp=guess;
    eV=funLSQ1(guessp);
    e0=eV'*eV;
    if Hessapp==1
        HESS=comhessapp(@funLSQ1,guess,pert);
    else
        HESS=comhess(@funLSQ1,guess,pert);
    end
    newton=-inv(HESS)*grad;
    length=1;
    lambdas=eigs(HESS);
    if HessMod==0
        lambdas=abs(lambdas); %Trick to avoid Hessian
modification
    end
    % Checking if Hessian is positive definite
    if lambdas(1)>0 && lambdas(2)>0
        hessmod=0;
    else
        coeff=mean(abs(lambdas));
```

```

    posdef=0;
    while posdef<1
        HESSm=HESS+coeff*eye(2); % Adding diagonal matrix
        hessmod=1; % Indication of modified HESSIAN
        lmb=eigs(HESSm);
        if lmb(1)>0 && lmb(2)>0
            posdef=1;
        else
            coeff=coeff*1.5;
        end
    end
end
% Computing next iterate
if hessmod==0
    guess1=guess+length*newton;
    eV=funLSQ1(guess1);
    eltr=0.5*eV'*eV;
else
    foundstep=0;
    length=0.5;
    while foundstep==0
        mnewton=-inv(HESSm)*grad; % Modified Newton
direction
        guess1=guess+length*mnewton;
        PQN=length*mnewton; % Step for quasi newton
direction
        modf=e0+PQN'*grad+PQN'*HESS*PQN;
        if modf<e0
            foundstep=1;
        else
            length=length/1.5;
        end
    end
    eV=funLSQ1(guess1);
    eltr=0.5*eV'*eV;
end
itiner(iter+1,1:2)=guess1';
itiner(iter+1,3)=eltr;
guess=guess1;
res=guess'*guess;
if iter>MAXIT % Terminated after the iteration reaches MAXIT
    res=0;
end
% Checking convergence options
if abs(itiner(iter+1,1)-itiner(iter,1))<minchg ||
abs(itiner(iter+1,2)-itiner(iter,2))<minchg;
    res=0;
end
end
*****

```

```

*****
function e=funLSQ1(x);
exper=load('lsqexpl.txt');
A=x(1);
B=x(2);
cnt=0;
for j=0:0.01:3
    cnt=cnt+1;
    y(cnt)=A*sin(j)+exp(cos(B*j));% *j+B*j^2;
end
e1=y'-exper(:,2);
*****

*****
function HESS=comhess(FUNNAME,point,pert)
% Computing the Hessian matrix
pointp=point;
eV=FUNNAME(pointp);
e0=0.5*eV'*eV;
for i=1:size(point,1)
    pointp=point;
    pointp(i)=pointp(i)+pert;
    eV=FUNNAME(pointp);
    e1=0.5*eV'*eV;
    pointp(i)=pointp(i)+pert;
    eV=FUNNAME(pointp);
    e2=0.5*eV'*eV;
    Usnd(i)=(e0-2*e1+e2)/(pert^2);
end
% mixed derivative
pointp=point;
pointp(1)=pointp(1)+pert;
eV=FUNNAME(pointp); % term i+1,j
U112=0.5*eV'*eV;
pointp=point;
pointp(2)=pointp(2)+pert;
eV=FUNNAME(pointp); % term i,j+1
U121=0.5*eV'*eV;
pointp=point;
pointp(1)=pointp(1)+pert;
pointp(2)=pointp(2)+pert;
eV=FUNNAME(pointp); % term i+1,j+1
U1121=0.5*eV'*eV;
mixed=1/pert*((U1121-U112)/pert-(U121-e0)/pert);
HESS(1,1)=Usnd(1);
HESS(1,2)=mixed;
HESS(2,1)=mixed;
HESS(2,2)=Usnd(2);
*****

```



```

*****
function HESS=comhessapp(FUNNAME,point,pert)
% Computing the Hessian matrix
pointp=point;
e0=FUNNAME(pointp);
% Computing Jacobian
for i=1:size(point,1)
    pointp=point;
    pointp(i)=pointp(i)+pert;
    e1=FUNNAME(pointp);
    J(:,i)=(e1-e0)/pert;
end
% Hessian approximation
HESS=J'*J;
*****

```

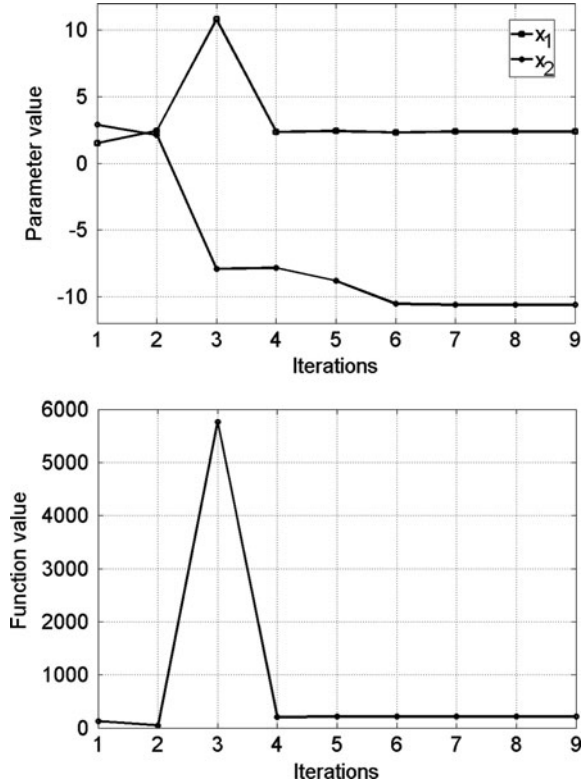
In the code listed above, whether, in each iteration, a true Newton direction is used, or it is modified in order to enforce descending step, is chosen by variable `HessMod`.

The optimization problem was solved two times using as initialization point $\mathbf{x} = [1.5, 2.9]^T$. First time it was solved without Hessian modification. The result is visualized in Fig. 2.13. Upper graph shows how the parameters are changing during the iterations, while the down one shows the changes in the objective function. It may be observed that from second to third iteration there is a large jump in the value of the objective function. It is due to the fact that the Hessian matrix computed at this iteration is not positive-definite. At this point, the eigenvalues of Hessian matrix are equal to: $\lambda_1 = 251.7$ and $\lambda_2 = -8.7$. As a consequence to that, the computed Newton step is not descending even for the model function. Without modifying Hessian matrix, and by accepting this step, third iteration goes far from the solution, and eventually finishes in the local minimum.

The same problem can be solved using given code by setting the variable `HessMod` equal to 1, which will introduce the modification of Hessian matrix in the cases when it is not positive definite. The results are visualized in Fig. 2.14. It may be observed that in this case there is a continuous reduction of the function. In the second iteration the Hessian matrix was modified and the step was descending. After this iteration there was no need to perform any modification and the algorithm finished with true Newton directions showing a very fast rate of convergence and finding the parameters after only five iterations.

Finally, in order to verify that for least squares problems, especially not very far from the solution, also Hessian approximation can be used almost with the same efficiency the problem is once again solved but having set variable `Hessapp` equal to 1. Figure 2.15 visualizes the results, from which it may be confirmed that in this particular case there was no degradation in performance of the optimization

Fig. 2.13 Results of optimization without modifying Hessian matrix



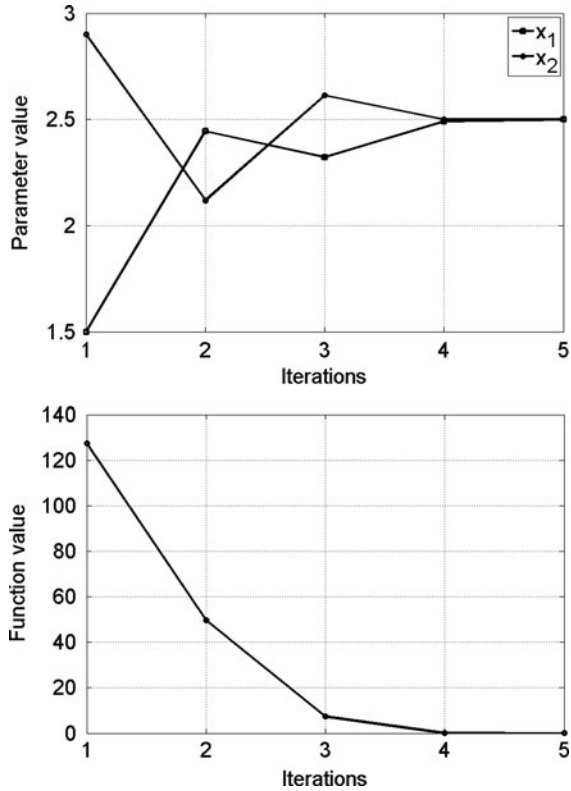
algorithm even when Hessian approximation is used. This particular feature can give a significant savings in computing times when the function evaluation involves possibly time consuming FE analyses.

The same problem can be solved using previously given MATLAB code for steepest descend. If for example the one with three trial steps is employed, for this particular case it turns out to be more effective than in the optimization of objective function given by Eqs. 2.16–2.18, and it converges after 24 iterations. The results are given in Fig. 2.16. It can be noticed that the algorithm with modified Newton direction is overall more effective than steepest descend, as it involves less evaluations of function.

Previous example showed that, even though the use of true Newton direction can lead sometimes to steps with increase of the objective function, this malfunction can be overcome relatively easy by implementing Hessian modification. With this approach the algorithm preserves a fast convergence rate and in most of the cases can be more effective than the steepest descend direction.

Another problem connected with Newton direction is that in some cases, especially far from the solution the model function can be quite different from the real one. It may result that, the Newton direction will provide a step that successfully minimizes the model function but which is not minimizing enough or even not at all

Fig. 2.14 Results of optimization with Hessian modification

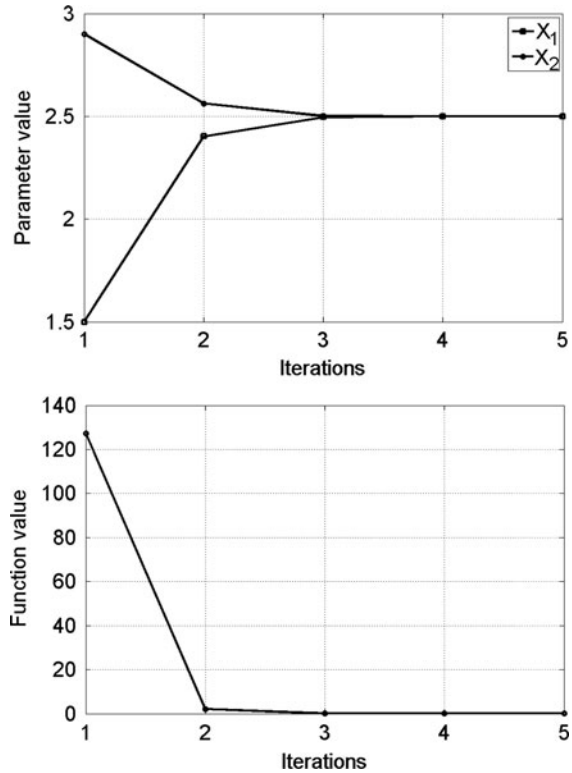


the real function. To tackle this problem a logical approach is to use the model function only in the vicinity of the current iterate. This is approach adopted in the *Trust Region* method.

2.3 Trust Region

Both line search with Newton direction and trust region are using the quadratic model of objective function, but they differ however in the way they make the use of this model. Line search starts by fixing the direction and then identifies an appropriate distance, namely the step length. Trust region on the other hand, first chooses the maximum distance – the trust region radius Δ_k – and then seeks both the direction and the step length that makes the best possible improvement of the function inside the trust region. This approach helps dealing with the situations when the quadratic model is quite different from the actual function as it may occur far from the solution. The model function will be close to the objective function in the vicinity of the current point, so restricting the minimization just to that area is a reasonable strategy.

Fig. 2.15 Results of optimization with Hessian approximation



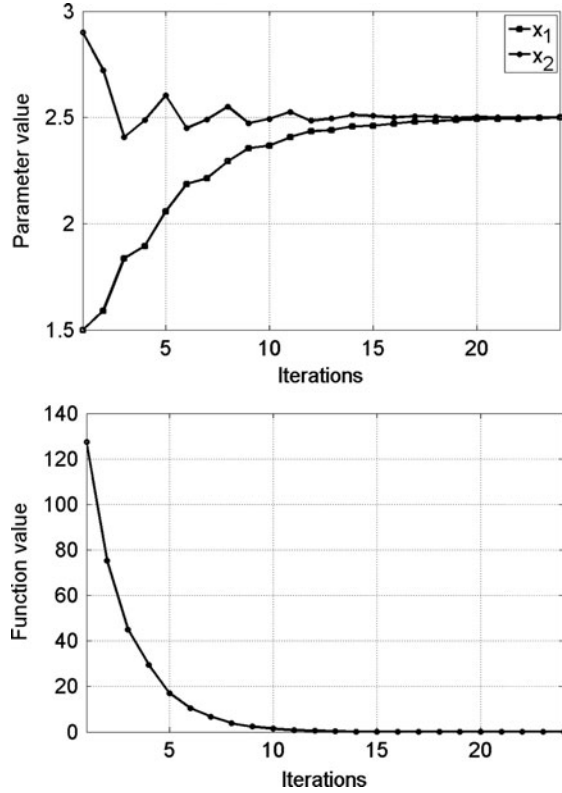
One of the main issues of the trust region approach, that to a large extent determines the success and the performance of this algorithm, is the decision strategy of how large the trusted region should be. Allowing it to be too large can make the algorithm facing the same problem as the classical Newton direction line search, when the minimizer of model function is quite far from the minimizer of the actual objective function. On the other hand using too small region the algorithm misses an opportunity to take a substantial step that could move it much closer to the solution. Some approaches on how to control this important issue will be discussed later.

Each step in the trust region algorithm is obtained by solving the sub-problem defined by

$$\min_{\|\mathbf{p}_k\| \leq \Delta_k} m_k(\mathbf{p}_k) = f(\mathbf{x}_k) + \mathbf{p}_k^T \nabla f(\mathbf{x}_k) + \frac{1}{2} \mathbf{p}_k^T \nabla^2 f(\mathbf{x}_k) \cdot \mathbf{p}_k \quad (2.21)$$

where Δ_k is the trust region radius.

Fig. 2.16 Results of optimization with steepest descend and three trial steps



From previous discussion it may be observed that there are essentially two important parts of any trust region algorithm. The first one, already anticipated, is the way the algorithm controls the radius of trust region. The second one is how efficiently it solves the sub-problem defined by (2.21). In what follows it will be shown how both of these problems are tackled and implemented within different optimization algorithms.

2.3.1 Trust Region Algorithm Based on Cauchy Point

As previously demonstrated on steepest descend line search approach, the algorithm can have global convergence characteristic even when the optimal step length is not used at each iteration. A similar reasoning applies also in trust region methods. Although it is convenient to find optimal solution of the sub-problem (2.21), for global convergence it is enough to find an approximate solution that lays within the trust region and gives a sufficient reduction of the model function.

This approach can be obtained by the use of so-called *Cauchy point*, which is a point that minimizes the model function along the steepest descent direction subjected to the trust region bound. However, the presence of model function makes the minimization of sub-problem a lot easier than in the steepest descent line search procedure. After choosing the search direction, the problem becomes one-dimensional but in the Cauchy point trust region algorithm it can be solved more effectively than in the steepest descent line search since here it is applied to the model function. Therefore, the evaluation of the function is computationally inexpensive.

Considering that the step length anyhow cannot exceed the trust region radius, a simple strategy for finding the Cauchy point can be used. First the value of model function can be computed for the values of steps equal to radius Δ_k , $0.75\Delta_k$ and $0.5\Delta_k$. Based on these values it is easy to see in which zone the minimum is located. In the second iteration algorithm computes the value of model function for the intermediate point between the two boundary points and once again identifies the zone of the solution. While the algorithm propagates the zone of minimum shrinks and already after couple of iterations it is reasonably close to the solution. After the last iteration one of the two boundary points with lower value of model function is taken as a solution. Figure 2.17 shows schematically one case of the Cauchy point identification based on this strategy.

As it can be seen from the figure, in fourth iteration the Cauchy point is identified with an error of about $0.015\Delta_k$. However, considering that these iterations involve only the computation of model function it is computationally inexpensive to repeat this procedure for additional couple of iterations that would practically find the exact minimizer of the model function along steepest descent direction.

Apart from the adopted strategy to solve the sub-problem an important issue of any trust region algorithm is the way it controls the trust region radius Δ_k . In most practical algorithms the choice of size of the region is performed according to performance of the algorithm during previous iteration. In particular the choice is

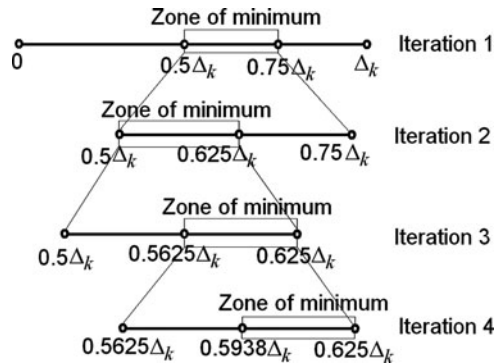


Fig. 2.17 Finding a Cauchy point approximately with four iterations

made based on the agreement between the model function and the objective function. This agreement is quantified by the following ratio

$$\rho_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{p}_k)}{m(0) - m(\mathbf{p}_k)} \quad (2.22)$$

The denominator represents what is called *predicted reduction*, while the numerator is called *actual reduction* obtained with the computed step on the objective function. Therefore, the closer this ratio is to 1, the better agreement between the objective function and the model is. Since predicted reduction will always be a positive number, as this is a condition in solving the sub-problem, if the ratio (2.22) turns out to be negative, it means that there was a significant disagreement between model function and the objective function. In such case the step should be rejected since it increases the objective function and the trust region should be shrunk.

The ratio (2.22) can be computed only after an additional evaluation of the objective function, which therefore represents a wasted computing time if the step is rejected. To avoid this possible inconvenience, this ratio is used as an indicator of how accurately model function describes the real one. Therefore, in most practical algorithms, if the ratio (2.22) is close to zero, the trust region radius should be reduced for the next iteration in order to avoid possible step rejection. On the other hand if it is close to 1 it is a signal that the trust region can be enlarged for the next iteration and therefore allow for possible larger and more ambitious steps. In general however, the trust region radius is not enlarged if the minimizer is found strictly inside the region as in such case it is not interfering with the progress of the algorithm.

Implementation of trust region algorithm that uses Cauchy point approach to solve the sub-problem is given in the following MATLAB code. The listing contains two separate routines – the main one, and an additional function that computes Cauchy point using the strategy schematically presented in Fig. 2.17. The value of initial trust region radius can be selected together with other options. As addition to these also the previously given MATLAB function `funLSQ1` is used. During the optimization if the ratio (2.22) is smaller than 0.2 the algorithm reduces the step by 20%. In the case when this ratio is larger than 0.6 the radius is enlarged by 20%. Otherwise, it remains unaltered.

```

*****
% Trust region algorithm with Cauchy point approach for
% sub-problem
clear

% Setting the options
minchg=1e-4; % Minimum change in parameters
MAXIT=30; % Maximum allowed number of iterations
guess=[1.7;2.9];
pert=1e-6; % Perturbation for the first derivatives
res=10;
TRrad=0.8;

% Optimization cycle
iter=0;
while res>1e-6
    eV=funLSQ1(guess);
    e0=0.5*eV'*eV
    itiner(iter+1,1:2)=guess';
    itiner(iter+1,3)=e0;
    iter=iter+1;
    for i=1:size(guess,1)
        guesssp=guess;
        guesssp(i)=guesssp(i)+pert;
        eV=funLSQ1(guess);
        e1=0.5*eV'*eV;
        grad(i,1)=(e1-e0)/pert;
    end
    % Computing Hessian matrix
    HESS=comhess(@funLSQ1,guess,pert);
    % Computing steepest descent direction
    stpdsc=-grad/norm(grad);
    % Determining TR step
    accepted=0;
    while accepted<1
        % Finding Cauchy point
        pc=cauchypnt(e0,stpdsc,grad,HESS,TRrad);
        predred=-(pc'*grad+0.5*pc'*HESS*pc);
        guess1=guess+pc; % Next iterate
        eV=funLSQ1(guess);
        eltr=0.5*eV'*eV;
        actualred=e0-eltr;
        ratio=actualred/predred;
        if ratio<0
            TRrad=TRrad/1.2;
        else
            accepted=1;
            if ratio<0.2
                TRrad=TRrad/1.2;
            end
        end
    end
end

```



```

        if ratio>0.6
            TRrad=TRrad*1.2;
        end
    end
end
itiner(iter+1,1:2)=guess1';
itiner(iter+1,3)=eltr;
guess=guess1;
res=eltr;
% Checking convergence options
if iter>MAXIT % Terminate if reaching MAXIT
    res=0;
end
if abs(itiner(iter+1,1)-itiner(iter,1))<minchg ||
abs(itiner(iter+1,2)-itiner(iter,2))<minchg;
    res=0;
end
end
end
*****

*****
function pc=cauchypt(e0,stpdsc,grad,HESS,TRrad)
% Function that computes Cauchy point

lenC1=TRrad; % point 1
lenC2=0.75*TRrad; % point 2
lenC3=0.5*TRrad; % point 3
pc1=lenC1*stpdsc;
pc2=lenC2*stpdsc;
pc3=lenC3*stpdsc;
modfun1=e0+pc1'*grad+0.5*pc1'*HESS*pc1;
modfun2=e0+pc2'*grad+0.5*pc2'*HESS*pc2;
modfun3=e0+pc3'*grad+0.5*pc3'*HESS*pc3;
if modfun1<modfun2
    lenC(1)=lenC1; % result between point 1 and 2
    lenC(2)=lenC2;
else
    if modfun3<modfun2
        lenC(1)=lenC2; % result between point 2 and zero
        lenC(2)=0;
    else
        if modfun1<modfun3
            lenC(1)=lenC1; % result between point 1 and 2
            lenC(2)=lenC2;
        else
            lenC(1)=lenC2; % result between point 2 and 3
            lenC(2)=lenC3;
        end
    end
end
end
end

```

```

ic=0;
for NR=1:8
    ic=ic+2;
    pc1=lenC(ic-1)*stpdsc;
    pc2=lenC(ic)*stpdsc;
    modfun1=e0+pc1'*grad+0.5*pc1'*HESS*pc1;
    modfun2=e0+pc2'*grad+0.5*pc2'*HESS*pc2;
    lenC(ic+1)=0.5*(lenC(ic)+lenC(ic-1));
    if modfun1<modfun2
        lenC(ic+2)=lenC(ic-1);
    else
        lenC(ic+2)=lenC(ic);
    end
end
IC=size(lenC,2);
pc1=lenC(IC-1)*stpdsc;
pc2=lenC(IC)*stpdsc;
modfun1=e0+pc1'*grad+0.5*pc1'*HESS*pc1;
modfun2=e0+pc2'*grad+0.5*pc2'*HESS*pc2;
if modfun1<modfun2 % Identifying Cauchy point
    pc=pc1;
else
    pc=pc2;
end
*****

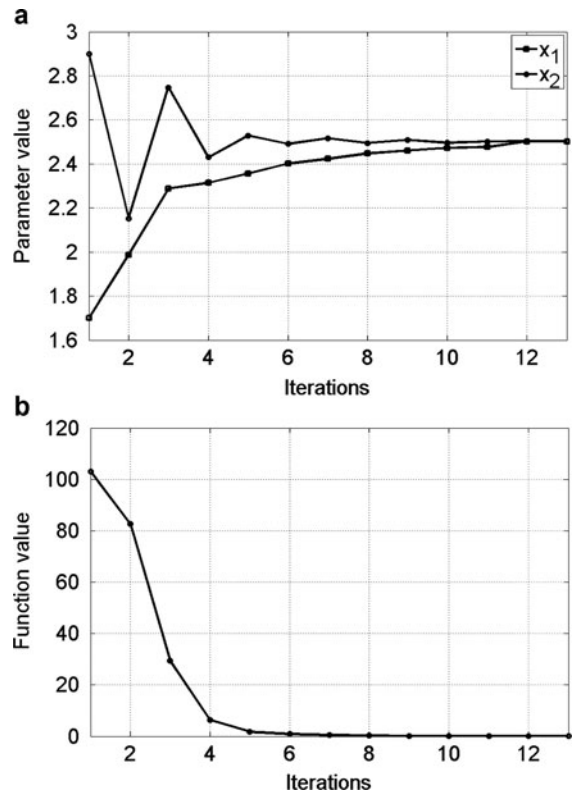
```

Using given routines optimization problem defined by Eqs. 2.19 and 2.20 can be solved. In order to demonstrate the capability of trust region approach to tackle the problems when quadratic model function is quite different from the objective function we can solve the optimization problem starting from the point $[1.7, 2.9]^T$.

The same optimization problem can be solved by using routines given before that use Newton line search approach with Hessian modification. Starting from the same point we can notice that in the second iteration algorithm arrived to the parameter values of $[2.45, 2.05]^T$. Hessian matrix at this point is positive definite so the algorithm proceeds with the exact Newton step. Computed Newton step for this iteration is $[-0.747, 1.552]^T$. Even though this step minimizes the model function, it produces the increment in the actual objective function due to the significant difference between the model and objective function. It represents a classical example where Newton line search method with Hessian modification is ineffective resulting in increase of the objective function, as this increase results not because the Hessian matrix is not positive definite, but because the minimizer of the model function differs significantly from the minimizer of the objective function.

As already anticipated, the trust region approach can keep this problem under control by restricting the search for the minimizer of the model function within the

Fig. 2.18 Results of optimization with Cauchy point trust region



trust region zone. The result of the optimization by trust region is visualized in Fig. 2.18.

It may be observed from Fig. 2.18b that the trust region algorithm provided a monotone decrease of the objective function throughout the whole optimization. The implemented strategy for reduction or increase of trust region radius based on the ratio (2.22) turns out to be effective as there was no step rejection in the optimization. In this case starting value of radius was taken to be equal to 0.8. This value was reduced in second iteration and afterwards, as the algorithm started to approach the solution in the fourth iteration it started to be increased in each subsequent iteration.

Previous example showed that the Cauchy point trust region is effective in solving minimization problems in the situations where line search with Newton direction may fail. Trust region has the global convergence property even when the sub-problem is solved only approximately since the Cauchy point doesn't have to be an absolute minimizer of the model function within the trust region.

Since the Cauchy point provides a sufficient reduction of the model function within trust region, and we showed that this is enough for the global convergence, the logical question that arises is why to use any other strategy to solve the sub-problem? The answer is that, if we are using at every iteration Cauchy point we are practically implementing the steepest descent approach on slightly more effective way since the use of model function allowed us to find the minimizer along steepest descent direction more accurately. But as we saw in previous examples the efficiency of steepest descent usually is not connected to the accuracy of finding the minimizer along this direction (e.g., example given in Fig. 2.9). Cauchy point does not depend on the Hessian matrix and it uses it only to compute the value of model function along the steepest descent direction. Rapid convergence can be expected only if Hessian matrix plays an important role in determining also the direction (like in Newton direction line search). There are many trust region algorithms that compute the Cauchy point and then try to improve it. One of them is a so-called *dog-leg* method originally proposed by Powell [6].

2.3.2 Dog-Leg Trust Region

We already saw that in the cases when model function is a good approximation of the objective function and when Hessian is positive definite Newton direction provides quadratic convergence (e.g. result given in Table 2.2). We also saw that similar convergence can be achieved also by modifying Hessian matrix when it is not positive-definite and using Newton or modified Newton direction, where needed, within line search algorithm (result visualizes in Fig. 2.14).

To incorporate fast quadratic convergence rate offered by full Newton step and global convergence feature of steepest descent into a single trust region algorithm an approach called dog-leg method for solving the sub-problem can be used. This method finds a compromise between steepest descent step and Newton's step based on the size of the trust region.

Let \mathbf{x}_{k+1}^{SD} and \mathbf{x}_{k+1}^N be steepest descend and Newton step respectively. If the Hessian matrix is positive definite then Newton step is actual minimizer of model function. If this point lays inside the trust region than it should be taken as the solution of the sub-problem. Otherwise, the piecewise linear curve defined by the line segments joining \mathbf{x}_k to \mathbf{x}_{k+1}^{SD} and \mathbf{x}_{k+1}^N called *dog-leg trajectory* is taken (Fig. 2.19). The point in which this trajectory intersects trust region, \mathbf{x}_{k+1}^{DL} is taken as the minimizer of the sub-problem.

The dog-leg trajectory can be implemented quit easy. In the first step using the gradient and Hessian matrix a steepest descent direction \mathbf{p}_k^{SD} and Newton direction

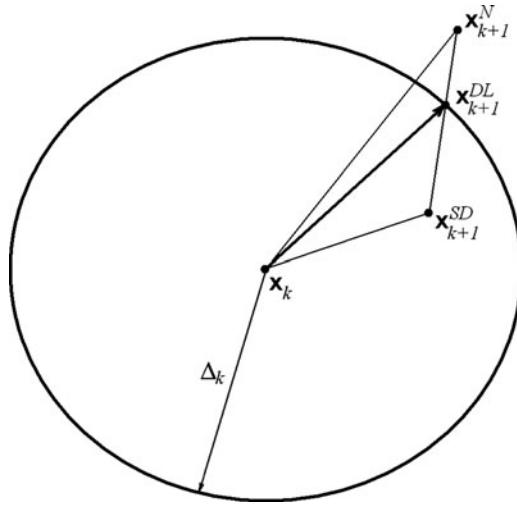


Fig. 2.19 Dog-leg trajectory

\mathbf{p}_k^N are computed. For the case when Newton step lays outside the trust region, an intersection point that corresponds to dog-leg step is found solving the following equation

$$\|\mathbf{p}_k^{SD} + \alpha(\mathbf{p}_k^N - \mathbf{p}_k^{SD})\| = \Delta_k \quad (2.23)$$

This is an effective strategy since both the Cauchy point (guarantying the global convergence) and the full Newton step (ensuring the possibility to have faster rate of convergence) are incorporated into the possible step. It is obvious that the solution will be closer to the Cauchy point for small trust regions while for large enough trust region it will be reduced to Newton's method.

The following listing shows a MATLAB code with the implementation of dog-leg strategy.

```

*****
% This is a Trust region algorithm
% A dog-leg approach for sub-problem
Clear

% Setting the options
minchg=1e-4; % Minimum change in parameters
MAXIT=30; % Maximum allowed number of iterations
guess=[1.7;2.9]; % Initial guess of parameters
pert=1e-6; % Perturbation for the first derivatives
res=10;
TRrad=1; % Initial trust region radius
HessMod=0; % Indication for Hessian modification

% Optimization cycle
iter=0;
while res>1e-6
    eV=funLSQ1(guess);
    e0=0.5*eV'*eV;
    itiner(iter+1,1:2)=guess';
    itiner(iter+1,3)=e0;
    iter=iter+1;
    for i=1:size(guess,1)
        guessp=guess;
        guessp(i)=guessp(i)+pert;
        eV=funLSQ1(guessp);
        e1=0.5*eV'*eV;
        grad(i,1)=(e1-e0)/pert;
    end
    % Computing Hessian matrix
    HESS=comhess(@funLSQ1,guess,pert);
    % Ensuring that Hessian is positive-definite
    if HessMod==1
        lambdas=eigs(HESS);
        if lambdas(1)>0 && lambdas(2)>0
            hessmod=0;
        else
            coeff=mean(abs(lambdas));
            posdef=0;
            while posdef<1
                HESSm=HESS+coeff*eye(2);
                hessmod=1; % Indication of modified HESSIAN
                lmb=eigs(HESSm);
                if lmb(1)>0 && lmb(2)>0
                    posdef=1;
                else
                    coeff=coeff*1.5;
                end
            end
        end
    end
end
end

```

```

else
    hessmod=0;
end
stpdsc=-grad/norm(grad);
if hessmod==0
    newton=-inv(HESS)*grad;
else
    newton=-inv(HESSm)*grad;
end
accepted=0;
while accepted<1
    if norm(newton)<TRrad
        pDL=newton; % Dog Leg step
    else
        % Finding the Cauchy point
        pc=cauchypt(e0,stpdsc,grad,HESS,TRrad);
        % Finding minimizer within trust region (Dog Leg step)
        diff=newton-pc;
        dimV=size(newton,1);
        cf=[0,0,-TRrad^2];
        for ii=1:dimV
            cf(1)=cf(1)+diff(ii)^2;
            cf(2)=cf(2)+2*pc(ii)*diff(ii);
            cf(3)=cf(3)+pc(ii)^2;
        end
        alfa=max(roots(cf)); % Taking the positive root
        pDL=pc+alfa*diff;
    end
    predred=-(pDL'*grad+0.5*pDL'*HESS*pDL);
    guess1=guess+pDL; % Next iterate
    eV=funLSQ1(guess1);
    eltr=0.5*eV'*eV;
    realred=e0-eltr;
    ratio=realred/predred;
    if ratio<0
        TRrad=TRrad/1.2;
    else
        accepted=1;
        if ratio<0.2
            TRrad=TRrad/1.2;
        end
        if ratio>0.6
            TRrad=TRrad*1.2;
        end
    end
end
itiner(iter+1,1:2)=guess1';
itiner(iter+1,3)=eltr;
guess=guess1;
res=eltr;
if iter>MAXIT % Terminated after the iteration reach MAXIT
    res=0;
end

```

```

% Checking convergence options
if abs(itiner(iter+1,1)-itiner(iter,1))<minchg ||
abs(itiner(iter+1,2)-itiner(iter,2))<minchg;
    res=0;
end
end
*****

```

As addition to this MATLAB routine, functions `cauchypnt`, `funLSQ1` and `comhess` with listings given previously need to be used.

The results of the optimization using dog-leg trust region, starting from the same initialization point like for the Cauchy point are visualized in Fig. 2.20. The algorithm proved to be more effective than Cauchy point and it reaches the convergence after seven iterations. It is interesting to note that, dog-leg shows more effective behavior particularly in the zone of the solution as then it uses the full Newton step and therefore is not experiencing the slow approaching to the solution characteristic for steepest descend direction (for example this can be notice in Fig. 2.18 from iteration 8 to iteration 13).

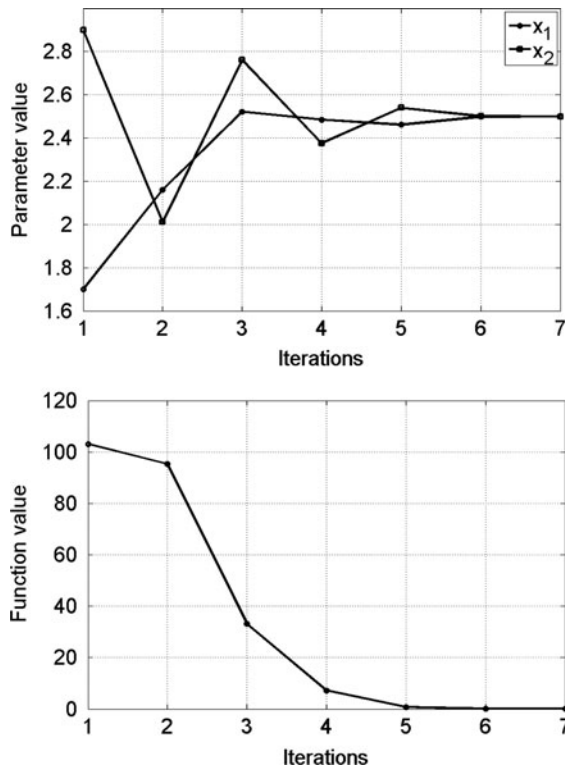


Fig. 2.20 Results of optimization with dog-leg trust region

The implementation given in the above listing takes also into account possibility of modifying Hessian matrix if it is not positive definite on the same way like previously presented for the Newton line search direction. It should be mentioned however that this might not be necessary for the trust region strategy as it introduces its own modification by constraining the minimization of sub-problem. In fact, if we use the same optimization problem starting from parameter set $\mathbf{x} = [1.5, 2.9]^T$, the one which with the Newton direction line search faced the problem of non-positive definite Hessian (Fig. 2.13), with dog-leg trust region it works even without Hessian modification (parameter `HessMod` set to 0) with initial trust region radius set to 1. Since in all steps the dog-leg algorithm makes a combination between steepest descent direction and the Newton direction, this already regularizes the step to be descending in each iteration even without Hessian modification.

The dog-leg method can be made slightly more sophisticated by widening the search for \mathbf{p} to the entire two-dimensional subspace spanned by gradient direction and Newton direction. This approach is described in what follows.

2.3.3 Two-Dimensional Subspace Minimization

By writing the unknown direction as a linear combination of Newton and steepest descend direction, the sub-problem will obtain the following form

$$\begin{aligned} \min m_k(\mathbf{p}_k) = & f(\mathbf{x}_k) + [\alpha_1 \mathbf{p}^{SD} + \alpha_2 \mathbf{p}^N]^T \nabla f(\mathbf{x}_k) \\ & + \frac{1}{2} [\alpha_1 \mathbf{p}^{SD} + \alpha_2 \mathbf{p}^N]^T \cdot \nabla^2 f(\mathbf{x}_k) \cdot [\alpha_1 \mathbf{p}^{SD} + \alpha_2 \mathbf{p}^N] \end{aligned} \quad (2.24)$$

under the constrain

$$\|\alpha_1 \mathbf{p}_k^{SD} + \alpha_2 \mathbf{p}_k^N\| \leq \Delta_k \quad (2.25)$$

The problem now becomes two dimensional and it is solved for the unknown coefficients α_1 and α_2 .

This represents a more general version of dog-leg method, since the solution provided by the dog-leg approach is a part of this 2D sub-space minimization problem.

The reduction of model function achieved by the two-dimensional sub-space minimization strategy is often very close to the reduction achieved by exact solution of (2.21). Since Cauchy point is a feasible solution, the reduction achieved will be at least as the one obtained by Cauchy point, resulting in global convergence. On the other hand, wider search for the minimizer with respect to dog-leg method often provides better reduction.

The solution of two-dimensional sub-problem (2.24) and (2.25) represents a constrained minimization problem. We should recall that within dog-leg approach,

the sub-problem is not solved like constrained minimization problem. Even though the solution satisfies the trust region constrain, the problem is solved in two steps, using geometrical approach in which it is reduced to Eq. 2.23. Here on the other hand, we will solve the sub-problem as a classical minimization problem with inequality constrain.

This minimization problem, defend by (2.24), with one inequality constrain given by (2.25) can be solved with Lagrange multipliers technique. As a first step we should write the minimization function together with its constrain in the following form

$$\ell(\mathbf{p}_k, \lambda_1) = m_k(\mathbf{p}_k) - \lambda_1 \cdot c_1(\mathbf{p}_k) \quad (2.26)$$

Last equation represents a so-called Lagrangian function, where

$$c_1(\mathbf{p}_k) = \Delta_k - \|\alpha_1 \mathbf{p}_k^{SD} + \alpha_2 \mathbf{p}_k^N\| \geq 0 \quad (2.27)$$

is called a constrain function, and scalar λ_1 represents a Lagrange multiplier. The solution of this constrained minimization problem is found as a stationary point of the Lagrangian function. Therefore, if we write the Lagrangian function in the general form for an arbitrary number of constrain functions it will adopt the following form

$$\ell(\mathbf{p}_k, \lambda_i) = m(\mathbf{p}_k) - \sum_{i=1}^N \lambda_i c_i(\mathbf{p}_k) \quad (2.28)$$

A general condition that \mathbf{p}_k^* needs to satisfy in order to represent a local solution of this minimization problem with inequality constrains is defined as follows

$$\nabla_{\mathbf{p}_k} \ell(\mathbf{p}_k^*, \lambda^*) = 0 \quad (2.29a)$$

$$c_i(\mathbf{p}_k^*) \geq 0, \quad i = 1, \dots, N \quad (2.29b)$$

$$\lambda_i^* \geq 0, \quad i = 1, \dots, N \quad (2.29c)$$

$$\lambda_i^* c_i(\mathbf{p}_k^*) = 0 \quad (2.29d)$$

This set of conditions is also known as *Karush-Kuhn-Tucker condition*, or *KKT condition*. This condition practically states, that the solution of the minimization problem should satisfy stationary condition taking the partial derivatives with respect to all the components of vector \mathbf{p}_k and all Lagrange multipliers. Derivatives of Lagrangian with respect to components of vector \mathbf{p}_k results in algebraic equation (2.29a), while partial derivatives of Lagrangian with respect to multipliers

are resulting in constrain inequalities (2.29b). Equation 2.29c state that only non-negative Lagrange multipliers should be taken into account, while (2.29d) are complementarity conditions which imply that, either i^{th} constrain is active or otherwise λ_i^* should be zero. The Lagrange multipliers corresponding to inactive inequalities are zeros, and they can be omitted when writing the general problem with (2.28).

A practical procedure of applying the KKT condition to the present problem defined by (2.26) and (2.27) can proceed as follows. In the first step one should first check if there is a minimizer of the model function m_k within the trust region. It practically means that one should find the unconstrained minimizer, which is found from the condition $\nabla_{\mathbf{p}_k} m_k = 0$, namely finding the first derivatives only of model function with respect to unknown direction. The solution of this problem is obviously Newton step. If Hessian is positive definite, it means that the Newton step is minimizer of model function and if it is inside the TR than it should be taken as a result of minimization problem and the constrain (2.27) is not active, so the conditions (2.29a - 2.29d) are actually reduced to the unconstrained problem solution. It is consistent with the KKT condition since, by having the constrain not active, imposes zero value to Lagrange multiplier which automatically means that the compatibility condition are satisfied.

If Hessian is not positive definite it means that the Newton direction is not pointing to the minimum and it is not the solution of the minimization problem. In this case the minimizer of the constrained problem should be searched on the boundary, meaning that the constrain is active. In this case the problem is solved as minimization with equality constrain, by taking all the partial derivatives with respect to vector \mathbf{p}_k components and Lagrange multipliers and solving the obtained system of algebraic equations for non-negative Lagrange multipliers and vector components \mathbf{p}_k simultaneously.

In the case where Newton direction doesn't exist, the constrain is also active and the problem is solved in the same way as mentioned above.

To illustrate this last case let us assume that the function we would like to minimize has the following form

$$f(\mathbf{x}) = x_1 + x_2 \quad (2.30)$$

Let us assume that we are at the point defined by $\mathbf{x}_k = [1,1]^T$, and that the trust region radius is equal to 1. The minimization sub-problem is therefore defined by

$$\min m(\mathbf{p}_k) = 2 + p_1 + p_2, \quad c_1(\mathbf{p}) = 1 - p_1^2 - p_2^2 \leq 0 \quad (2.31)$$

Obviously, since in this case Hessian matrix is zero, the Newton direction doesn't exist. In fact, the objective function is monotonely decreasing and it doesn't have any minimum in the unconstrained domain. The problem therefore should be solved with the active constrain, and the minimizer should be found on the

boundary, which in this case is a circle. Applying KKT condition, (2.29a) and (2.29b) are resulting in the following algebraic equations

$$\begin{aligned} 1 + 2\lambda_1 p_1 &= 0 \\ 1 + 2\lambda_1 p_2 &= 0 \\ p_1^2 + p_2^2 &= 1 \end{aligned} \tag{2.32}$$

Expressing from the first two equations p_1 and p_2 as functions of λ_1 and substituting to the third one, it results in a quadratic equation that should be solved for the positive values of λ_1 . It is easy to see that the solution of this quadratic equation gives

$$\lambda_1 = \pm \frac{1}{\sqrt{2}} \tag{2.33}$$

Taking just the positive value in order to satisfy (2.29c) the resulting minimizer of the constrained problem is equal to

$$\mathbf{p}_k = \left[-\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2} \right] \tag{2.34}$$

that is the point on the circle with unit radius with the center in $[1,1]^T$ for which the function (2.30) has its minimum value. From the above discussion it can be summarized that the minimizer of the constrained sub-problem that arises within trust region algorithm is taken to be the Newton step in the case when Hessian is positive definite, and when Newton step lays within the trust region. If this is not the case, it means that, either the model function doesn't have the minimum (it monotonely decreases) or the minimizer is found outside of the trust region. In both of these cases the solution should be found on the boundary itself. As demonstrated in previous simple example this will lead us to the system of algebraic equations, like those given in (2.32), where from the first set of equations all the components of unknown direction should be expressed as a function of λ_1 , which after introducing to the last equation will result in a polynomial equation, that is further solved for the positive roots.

Two-dimensional sub-problem defined by (2.24) and (2.25) obviously represent a special case of the presented strategy, where the unknown direction is uniquely defined with two scalars, α_1 and α_2 , that are used as coefficients of linear combination of two known directions (steepest descend and Newton direction). Using the same approach as previously summarized, we should first compute steepest descend and Newton direction using (2.9) and (2.15) respectively. If the Hessian is positive definite, it means that the Newton step is the minimizer of the model function and then, we should check whether it lays inside the trust region. If this is the case, it is

taken as the solution of the sub-problem. In the opposite case, we should proceed with the solution of constrained problem. For general multidimensional case (when the number of parameters is N), Lagrangian will take the following form

$$\begin{aligned} \ell(\mathbf{p}_k, \lambda_i) = & f(\mathbf{x}_k) + [\alpha_1 p_i^{SD} + \alpha_2 p_i^N]^T \cdot [g_i] + \frac{1}{2} [\alpha_1 p_i^{SD} + \alpha_2 p_i^N]^T \cdot [h_{ij}] \\ & \cdot [\alpha_1 p_i^{SD} + \alpha_2 p_i^N]^T + \lambda_1 \left(\Delta^2 - \sum_{i=1}^N (\alpha_1 p_i^{SD} + \alpha_2 p_i^N) \right) \end{aligned} \quad (2.35)$$

whew $[g_i]$ is gradient vector, $[h_{ij}]$ is Hessian matrix and $[p_i^{SD}]$ and $[p_i^N]$ are steepest descend and Newton direction vectors respectively. Writing the first derivatives with respect to α_1 , α_2 and λ_1 will lead us to the following system of algebraic equations

$$C_{11}\alpha_1 + C_{12}\alpha_2 + 2L_{11}\alpha_1\lambda_1 + L_{12}\alpha_2\lambda_1 = F_1 \quad (2.36a)$$

$$C_{21}\alpha_1 + C_{22}\alpha_2 + L_{21}\alpha_1\lambda_1 + 2L_{22}\alpha_2\lambda_1 = F_1 \quad (2.36b)$$

$$L_{11}\alpha_1^2 + L_{12}\alpha_1\alpha_2 + L_{22}\alpha_2^2 = \Delta^2 \quad (2.36c)$$

where the coefficients are given by

$$C_{11} = \sum_{i=1}^N p_i^{SD} \sum_{j=1}^N h_{ij} p_j^{SD} \quad (2.37a)$$

$$C_{12} = C_{21} = \frac{1}{2} \left(\sum_{i=1}^N p_i^{SD} \sum_{j=1}^N h_{ij} p_j^N + \sum_{i=1}^N p_i^N \sum_{j=1}^N h_{ij} p_j^{SD} \right) \quad (2.37b)$$

$$C_{22} = \sum_{i=1}^N p_i^N \sum_{j=1}^N h_{ij} p_j^N \quad (2.37c)$$

$$F_1 = - \sum_{i=1}^N g_i p_i^{SD} \quad (2.37d)$$

$$F_2 = - \sum_{i=1}^N g_i p_i^N \quad (2.37e)$$

$$L_{11} = \sum_{i=1}^N (p_i^{SD})^2 \quad (2.37f)$$

$$L_{12} = L_{21} = 2 \sum_{i=1}^N p_i^{SD} \cdot p_i^N \quad (2.37g)$$

$$L_{22} = \sum_{i=1}^N (p_i^N)^2 \quad (2.37h)$$

Further, the procedure is the same as previously shown. From the first two equations α_1 and α_2 are expressed as functions of λ_1 , which after being introduced to the third one results in polynomial equation that is solved for positive values of λ_1 . Since in general, depending on the values of Hessian matrix and gradient this polynomial can be of higher order, it may have larger number of positive real roots. However, once they are identified it is computationally inexpensive to check which value corresponds to α_1 and α_2 that minimize the objective function.

The reduction of the model function achieved by two-dimensional subspace minimization is often very close to the exact solution of (2.21). The solution of dog-leg method is obviously included in two-dimensional subspace method so at the limit case the performance should match those achieved by the former approach. Usually the advantage of two-dimensional sub-space approach is evidenced in larger reduction of model function in each step. More details of this approach can be found in [7] and [8].

Expressing the polynomial from the system (2.36a - 236c) only in terms of Lagrange multipliers results in rather complicated equation and the implementation is relatively long. In order to shown the abovementioned benefits that are achieved by this approach we can focus on simpler case where the number of parameters to identify is equal to 2. In this case, by considering derivatives directly with respect to components of vector \mathbf{p}_k , the general form of the solution is somewhat simpler, and it leads to the same solution as the one achieved by the two-dimensional sub-problem approach.

Adopting the same notation as in previous case, the model function will take the following form

$$m(\mathbf{p}) = f_k + [p_1 \quad p_2] \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} + \frac{1}{2} [p_1 \quad p_2] \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} \quad (2.38)$$

with the constrain function given by

$$c(\mathbf{p}) = \Delta^2 - p_1^2 - p_2^2 \geq 0 \quad (2.39)$$

After the matrix multiplication in (2.38) is performed, and considering that Hessian is symmetric, Lagrangian function can be written as follows

$$\ell = f_k + g_1 p_1 + g_2 p_2 + \frac{1}{2} h_{11} p_1^2 + h_{12} p_1 p_2 + \frac{1}{2} h_{22} p_2^2 - \lambda (1 - p_1^2 - p_2^2) \quad (2.40)$$

Stationary point of Lagrangian results in the following system of algebraic equations

$$\frac{\partial \ell}{\partial p_1} = g_1 + h_{11}p_1 + h_{12}p_2 + 2\lambda p_1 = 0 \quad (2.41a)$$

$$\frac{\partial \ell}{\partial p_2} = g_2 + h_{12}p_1 + h_{22}p_2 + 2\lambda p_2 = 0 \quad (2.41b)$$

$$\frac{\partial \ell}{\partial \lambda} \Rightarrow p_1^2 + p_2^2 = \Delta^2 \quad (2.41c)$$

From (2.41b) we can express p_2 as a function of p_1 and λ resulting in

$$p_2 = -\frac{h_{12}p_1 + g_2}{2\lambda + h_{22}} \quad (2.42)$$

Combining (2.42) with (2.41a) we can obtain expression of p_1 only as a function of λ

$$p_1 = -\frac{2g_1\lambda + g_1h_{22} - h_{12}g_2}{4\lambda^2 + (2h_{11} + 2h_{22})\lambda + h_{11}h_{22} - h_{12}^2} \quad (2.43)$$

Finally, combining the last two equations also p_2 component can be expressed only as a function of λ resulting in

$$p_2 = \frac{-4g_2\lambda^2 + (2g_1h_{12} - 2h_{11}g_2 - 2h_{22}g_2)\lambda + g_1h_{12}h_{22} - g_2h_{11}h_{22}}{(2\lambda + h_{22})[4\lambda^2 + (2h_{11} + 2h_{22})\lambda + h_{11}h_{22} - h_{12}^2]} \quad (2.44)$$

Equations 2.43 and 2.44 are substituted back in (2.41c) to obtain polynomial of λ .

In a view of easier implementation we can group the coefficients in the following way. Considering that both p_1 and p_2 are polynomial functions of λ we can write the (2.41c) in the following way

$$\frac{(b_1\lambda + a_1)^2}{(c_2\lambda^2 + b_2\lambda + a_2)^2} + \frac{(c_3\lambda^2 + b_3\lambda + a_3)^2}{(b_4\lambda + a_4)^2(c_2\lambda^2 + b_2\lambda + a_2)^2} = \Delta^2 \quad (2.45)$$

where the introduced coefficients are computed using the following equations

$$b_1 = 2g_1 \quad (2.46a)$$

$$a_1 = g_1h_{22} - h_{12}g_2 \quad (2.46b)$$

$$c_2 = 4 \quad (2.46c)$$

$$b_2 = 2h_{11} + 2h_{22} \quad (2.46d)$$

$$a_2 = h_{11}h_{22} - h_{12}^2 \quad (2.46e)$$

$$c_3 = -4g_2 \quad (2.46f)$$

$$b_3 = 2g_1h_{11} - 2h_{11}g_2 - 2h_{22}g_2 \quad (2.46g)$$

$$a_3 = g_1h_{11}h_{22} - g_2h_{11}h_{22} \quad (2.46h)$$

$$b_4 = 2 \quad (2.46i)$$

$$a_4 = h_{22} \quad (2.46j)$$

After multiplication, 2.45 is transformed into a single polynomial equation of sixth order

$$\begin{aligned} & -\Delta b_4^2 c_2^2 \lambda^6 - (2\Delta^2 c_2 b_2 b_4^2 + 2\Delta^2 a_4 b_4 c_2^2) \lambda^5 \\ & + (b_1^2 b_4^2 + c_3^2 - \Delta^2 b_2^2 b_4^2 - 2\Delta^2 a_2 c_2 b_4^2 - 4\Delta^2 c_2 b_2 a_4 b_4 - \Delta^2 a_4^2 c_2^2) \lambda^4 \\ & + (2b_1^2 b_4 a_4 + 2b_1 a_1 b_4^2 + 2c_3 b_3 - 2\Delta^2 a_2 b_2 b_4^2 - 2\Delta^2 a^4 b_4 b_2^2 - 4\Delta^2 a_2 c_2 a_4 b_4 \\ & - 2\Delta^2 a_4^2 c_2 b_2) \lambda^3 + (b_1^2 a_4^2 + 4b_1 a_1 b_4 a_4 + a_1^2 b_4^2 + b_3^2 + 2c_3 a_3 - \Delta^2 a_2^2 b_4^2 - 4\Delta^2 a_2 b_2 a_4 b_4 \\ & - \Delta^2 a_4^2 b_2^2 - 2\Delta^2 a_4^2 a_2 c_2) \lambda^2 + (2b_1 a_1 a_4^2 + 2a_1^2 b_4 a_4 + 2a_3 b_3 - 2\Delta^2 a_4 b_4 a_2^2 \\ & - 2\Delta^2 a_4^2 a_2 b_2) \lambda + a_1^2 a_4^2 + a_3^2 - \Delta^2 a_4^2 a_2^2 = 0 \end{aligned} \quad (2.47)$$

The resulting polynomial is sixth order, but in general not all the roots are real positive numbers. Therefore, in the practical implementation, we will first compute the values of coefficients using Eq. 2.46, and then we will compute roots of polynomial (2.47). Afterwards, only those that are positive real numbers will be considered to compute resulting direction components p_1 and p_2 using (2.43) and (2.44) respectively. In most of the cases there will be more than one pair of components corresponding to real positive Lagrange multipliers. However, once they are computed, by using Eq. 2.38, it is computationally inexpensive to verify which one of the computed directions minimizes the model function.

The implementation of presented procedure is given in the listing below.

```
*****
% This is a Trust region algorithm
% with minimization for components of vector p

clear
% Setting the options
minchg=1e-4; % Minimum change in parameters
MAXIT=30; % Maximum allowed number of iterations
guess=[1.7;2.9];
pert=1e-6; % Perturbation for the first derivatives
res=10;
TRrad=0.8;
HessMod=1; % Indication for Hessian modification

% Optimization cycle
iter=0;
while res>1e-6
    eV=funLSQ1(guess);
    e0=0.5*eV'*eV;
    itiner(iter+1,1:2)=guess';
    itiner(iter+1,3)=e0;
    iter=iter+1;
    for i=1:size(guess,1)
        guessp=guess;
        guessp(i)=guessp(i)+pert;
        eV=funLSQ1(guessp);
        e1=0.5*eV'*eV;
        grad(i,1)=(e1-e0)/pert;
    end
    % Computing Hessian matrix
    HESS=comhess(@funLSQ1,guess,pert);
    % Ensuring that Hessian is positive-definite
    if HessMod==1
        lambdas=eigs(HESS);
        if lambdas(1)>0 && lambdas(2)>0
            hessmod=0;
        else
            coeff=mean(abs(lambdas));
            posdef=0;
            while posdef<1
                HESSm=HESS+coeff*eye(2);
                hessmod=1; % Indication of modified HESSIAN
                lmb=eigs(HESSm);
                if lmb(1)>0 && lmb(2)>0
                    posdef=1;
                else
                    coeff=coeff*1.5;
                end
            end
        end
    else
        hessmod=0;
    end
end
```

```

end
stopdsc=-grad/norm(grad);
if hessmod==0
    newton=-inv(HESS)*grad;
else
    newton=-inv(HESSm)*grad;
end
accepted=0;
while accepted<1
    % Solving for vector p that minimizes the quadratic form
    if norm(newton)<TRrad
        pfnd=newton;
    else
        % Constrain should be active
        % Coefficients for components of vector p
        B1=2*grad(1);
        A1=grad(1)*HESS(2,2)-HESS(1,2)*grad(2);
        C2=4;
        B2=2*HESS(1,1)+2*HESS(2,2);
        A2=HESS(1,1)*HESS(2,2)-HESS(1,2)^2;
        C3=-4*grad(2);
        B3=2*grad(1)*HESS(1,1)-2*HESS(1,1)*grad(2)-
2*HESS(2,2)*grad(2);
        A3=grad(1)*HESS(1,1)*HESS(2,2)-
grad(2)*HESS(1,1)*HESS(2,2);
        B4=2;
        A4=HESS(2,2);
        % Coefficients for polynomial for Lagrange multiplier
        CF(1)=-TRrad^2*B4^2*C2^2;
        CF(2)=-2*TRrad^2*C2*B2*B4^2-2*TRrad^2*A4*B4*C2^2;
        CF(3)=B1^2*B4^2+C3^2-TRrad^2*B2^2*B4^2-
2*TRrad^2*A2*C2*B4^2-4*TRrad^2*C2*B2*A4*B4-TRrad^2*A4^2*C2^2;
        CF(4)=2*B1^2*B4*A4+2*B1*A1*B4^2+2*C3*B3-
2*TRrad^2*A2*B2*B4^2-2*TRrad^2*A4*B4*B2^2-
4*TRrad^2*A2*C2*A4*B4-2*TRrad^2*A4^2*C2*B2;
        CF(5)=B1^2*A4^2+4*B1*A1*B4*A4+A1^2*B4^2+B3^2+2*C3*A3-
TRrad^2*A2^2*B4^2-4*TRrad^2*A2*B2*A4*B4-TRrad^2*A4^2*B2^2-
2*TRrad^2*A4^2*A2*C2;
        CF(6)=2*B1*A1*A4^2+2*A1^2*B4*A4+2*A3*B3-
2*TRrad^2*A4*B4*A2^2-2*TRrad^2*A4^2*A2*B2;
        CF(7)=A1^2*A4^2+A3^2-TRrad^2*A4^2*A2^2;
        LMBDAS=roots(CF);
        rr=0;
        clear LMB
        for i=1:6
            if isreal(LMBDAS(i))==1
                rr=rr+1;
                LMB(rr)=LMBDAS(i);
            end
        end
        % Computing vector p components
        accp=0;
        clear p
    end
end

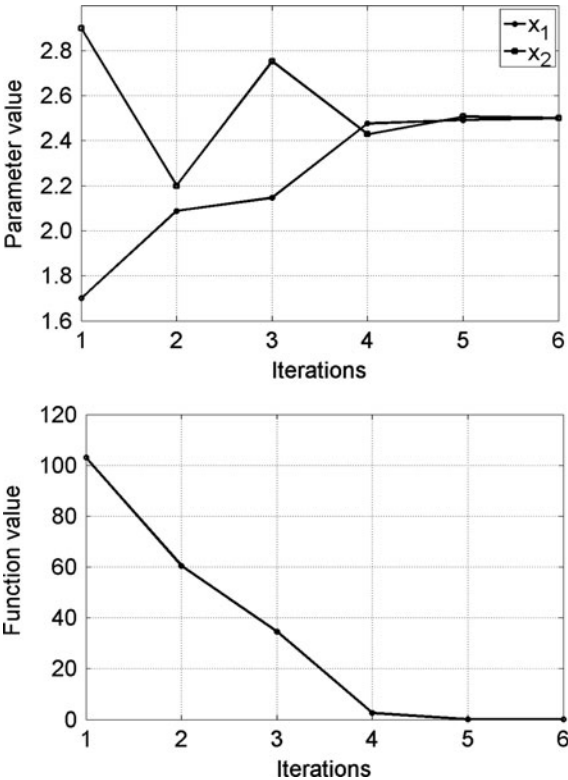
```

```

    for i=1:rr
        accp=accp+1;
        p(1,accp)=- (B1*LMB(i)+A1) / (C2*LMB(i)^2+B2*LMB(i)+A2);
        p(2,accp)=(C3*LMB(i)^2+B3*LMB(i)+A3) /
        ((B4*LMB(i)+A4)*(C2*LMB(i)^2+B2*LMB(i)+A2));
        if norm(p(:,accp)) > (TRrad+0.01)
            accp=accp-1;
        end
    end
    % Checking which solution gives minimum of model function
    minMOD=e0+(p(:,1))*grad+0.5*p(:,1)*HESS*p(:,1));
    pfnd=p(:,1);
    for i=2:accp
        modP=e0+(p(:,i))*grad+0.5*p(:,i)*HESS*p(:,i));
        if modP<minMOD
            pfnd=p(:,i);
            minMOD=modP;
        end
    end
    end
    end
    predred=-(pfnd'*grad+0.5*pfnd'*HESS*pfnd);
    guess1=guess+pfnd; % Next iterate
    eV=funLSQ1(guess1);
    eltr=0.5*eV'*eV;
    realred=e0-eltr;
    ratio=realred/predred;
    if ratio<0
        TRrad=TRrad/1.2;
    else
        accepted=1;
        if ratio<0.2
            TRrad=TRrad/1.2;
        end
        if ratio>0.6
            TRrad=TRrad*1.2;
        end
    end
    end
    end
    itiner(iter+1,1:2)=guess1';
    itiner(iter+1,3)=eltr;
    guess=guess1;
    res=eltr;
    if iter>MAXIT % Terminated after the iteration reaches MAXIT
        res=0;
    end
    % Checking convergence options
    if abs(itiner(iter+1,1)-itiner(iter,1))<minchg ||
        abs(itiner(iter+1,2)-itiner(iter,2))<minchg;
        res=0;
    end
    end
    *****

```

Fig. 2.21 Results of optimization with direct minimization on direction vector components



This algorithm is used to solve the same optimization problem as the one already solved with dog-leg approach (visualized in Fig. 2.20). The results of the optimization are visualized in the same manner as previously in Fig. 2.21. It may be observed that the latter algorithm turned out to be more effective, as expected, and the optimization terminated after six iterations (one less with respect to dog-leg approach).

Comparing the two figures it may be observed that the gain is not significant. In fact, the only thing in which the second approach is more effective is the minimization of the model function under each step. However, also the dog-leg approach is not far from the solution and so the steps are not much different.

This difference can be illustrated if we compare what are the values of model function in the resulting step within each iteration for both algorithms. However, this comparison is fair to make only at the beginning of the optimization if both algorithms are starting from the same point, since, as they will produce different steps in every second iteration the two algorithms will not be at the same point and therefore will not have the same model function to minimize. These differences are given in Table 2.3, referring to the first iteration of three different initialization points.

Table 2.3 Values of model function at the beginning and the end of the first iterations obtained by two different optimization algorithms referring to three different initializations

Initialization	Dog-leg		Minimization with respect to \mathbf{p}	
	m_k at the beginning	m_k at the end of step	m_k at the beginning	m_k at the end of step
[1.7, 2.9]	103.1	-35.3	103.1	-39.9
[1.3, 2.8]	133.4	16.8	133.4	14.4
[1.4, 2.95]	155.8	-33.8	155.8	-33.8

It may be observed from the table that the second approach indeed is obtaining lower values of model function in most of the cases. Since the solution by dog-leg strategy is part of the second approach the latter should perform at least like the former one, which was the case in the third initialization reported in the table, where the reduction of model function achieved by both algorithms was the same. This small advantage of more precise minimization of the model function in some of the cases may result in less iterations. Example of this is the third initialization given in Table 2.3, that converged by dog-leg approach after 8 iterations with respect to 7 obtained by the second approach.

However, in most of the cases the differences are not so large, and also the dog-leg approach manages to minimize the model function almost to the same extent as the minimization with respect to direction vector, so it can be used with almost the same efficiency.

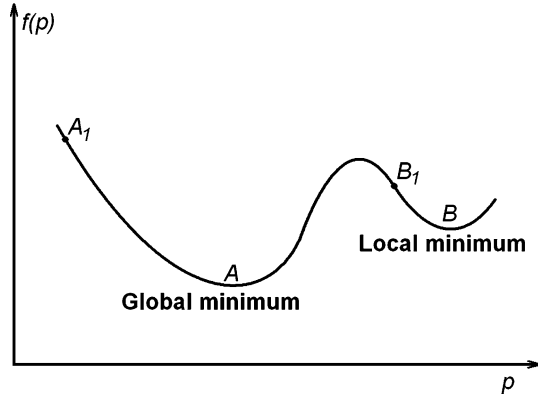
The optimization algorithms presented up to now are some times in the literature found under the name of *derivative based algorithms*, as they rely on the computation of derivatives. This circumstance makes them rather ineffective if the objective function has a large number of local minima, since they are identifying as a solution any mathematical minimum (i.e. first derivative equal to zero and second derivative larger than zero). As a remedy, in these situations Genetic Algorithms can be used as an alternative.

2.4 Genetic Algorithms

The objective functions arising in inverse analyses of structural problems sometimes can be complicated and extremely non-convex. As previously mentioned, in these situations algorithms relying on classical mathematical theory of optimization are behaving rather poor, since they can identify as a solution of the problem also the local minimum.

Using any of derivative based algorithms in order to minimize the function of the type visualized in Fig. 2.22, depending on the initialization point it may occur that the optimization terminates after the point B is reached. Since in point B the first derivative is equal to zero and the second derivative is larger than zero this point represents mathematical minimum, and the algorithm doesn't have information that

Fig. 2.22 Objective function with one local and one global minimum



somewhere within the range of interest there might be a point with lower value of the objective function. Since derivative based algorithms are stopping when the minimum of the objective function within a certain zone is found, the only way to confirm that it also represents a global minimum of the function is to perform the minimization procedure couple of times starting from different initialization points. If for a given problem it turns out that the result of minimization doesn't depend on initialization point then there is a large probability that the identified minimum is an absolute minimum of the function. For example, minimizing the objective function of the type visualized in Fig. 2.22, by the use of any of derivative based algorithms, starting from point B_1 , the algorithm would most likely terminate by identifying point B as a minimum. On the other hand, repeating the same procedure but starting from a different point, say A_1 , the algorithm would probably identify point A as a solution. By comparing the values of the objective functions between the two points it's easy to verify which one of the two is the solution of the problem. In fact, common practice in inverse analyses when derivative based algorithms are used is to perform minimization couple of times and to take as the solution parameter set to which procedure converged three or four times starting from different initialization points.

Using this strategy, problems like the one visualized in Fig. 2.22 can still be relatively effectively solved by derivative based algorithms, as the number of local minima is small. If the objective function is characterized by a presence of large number of local minima, then the use of derivative based algorithms is not effective as the solution becomes extremely dependant on the initialization point, and therefore it is difficult to identify what is the absolute minimum of the function. In these situations it is more appropriate to use Genetic Algorithms to solve the minimization problem.

The Genetic Algorithms (GA) represent a methodology for solving both constrained and unconstrained optimization problems. They belong to a so-called *soft-computing* family as they are not solving in mathematical sense the minimization problems. The approach adopted by GA consists in repeated modification of

“population” of individual solutions, based on the concept of “natural” selection. Over successive generations, the population “evolves” toward an optimal solution.

GA can be used in general optimization problems, but their real advantage comes to the play in optimizing discontinuous, non-differentiable, stochastic, or highly nonlinear objective functions.

In order to understand the way GA works let us describe the main concept and notions of GA within the present context. Each population consists of a certain number of individuals. For the given optimization problem, the number of members is fixed and is usually about 50–100. Each member is represented by a set of parameters, which are called *genes* in the jargon of GA. To each of the individuals it is attributed some *score* which is further used as a selection criterion. This score represents a value of *fitness function* which in the present context is the value of the objective function for the parameters that are defining a certain individual.

Optimization by GA starts by a random selection of individuals that are covering some region of interest. The first step consists of computing the values of objective function for each of the individuals. After these computations are performed, all the individuals are sorted according to their value of the objective function. In the subsequent step, a new generation is created by making a use of the existing one.

There are many different types of GA that differ based on a way how they form the new generation. For a more detailed description on GA readers should refer to [9] and [10]. In what follows, some of more frequently used criteria for building a new generation from the current one will be explained.

All of the GA are using some individuals from the current generation, called *parents*, who contribute their *genes* (i.e. parameter values) to their *children*. Algorithms are usually selecting individuals with better values of the fitness function as parents.

An easy to implement scheme of forming the next generation is described in what follows. This scheme works very well with the problems studied in this book. GA of this type turns out to be capable of solving the problems with multiple minima objective functions, as it will be demonstrated on the examples that will follow.

Within this scheme, every new generation is formed by three groups of children:

- Elite children
- Cross-over children
- Mutation children

Elite children represent the individuals with best fitness function within the present generation. These children are directly passed to the next generation without any modification. Of course, the number of elite children represents an optimization parameter to be adjusted but for a successfully working GA it should not be very large. Usually it is about 2–5% of the population size. The existence of elite children is important for the preservation of the individuals with already good value of fitness function

Cross-over children represent individuals that are formed by combining genes of two parents from the current generation. This group is formed by applying an

adopted cross-over rule on the selected individuals called parents. As parents usually the individuals from best to intermediate values of the fitness function are selected (i.e. the group takes into account both elite members, but also those with somewhat weaker fitness function result). The number of individuals used as parents represents additional optimization parameter, and it is usually about 50–70% of the population size. Two parents are usually combined to reproduce two children so that the total size of the population would be preserved. After the pairs of parents are randomly coupled from previously selected individuals, cross-over rule is applied in order to form children. The idea of crossing over existing individuals is important as it mixes the parameters of existing individuals. Therefore with this operation by combining already assessed parameters in a different way, possibly improved individuals could be formed, as it may occur for certain individuals that the error on some of the parameters is smaller than for the others.

A very simple cross-over rule is the one that uses two random vectors of the same size as the number of parameters, with only zeroes and ones as entries. Applying this rule to the parent pair, the two children are constructed using these two vectors, where entry 1 means taking the corresponding parameter from the first parent, while entry 0 means taking it from the second one.

This scheme of crossing-over is illustratively presented in Fig. 2.23. The figure shows optimization problem with 4 parameters. Parameters of one parent are represented by x_i while those of the other one by y_i . The two children in this case are created by the use of two randomly generated cross-over vectors. It should be mentioned that, in the case of smaller number of optimization parameters (e.g., 3) the scheme can work only with one random extraction and taking the other one as the opposite (e.g., two cross-over vectors can be $[1,1,0]$ for the first child and $[0,0,1]$ for the second one).

Mutation children are created by introduction a random mutation (i.e. changes) to the parameters applied on the selected individuals from the current population. For this operation usually the worst individuals are taken, as they anyhow represent those that should be wasted, so it is reasonable to try on them a fully random modification as it may produce better individuals. The number of individuals that will be subjected to this operation is what remains when previous two groups are excluded. Apart of the total number of mutated individuals this process is also controlled by the parameter that sets the amount of mutation.

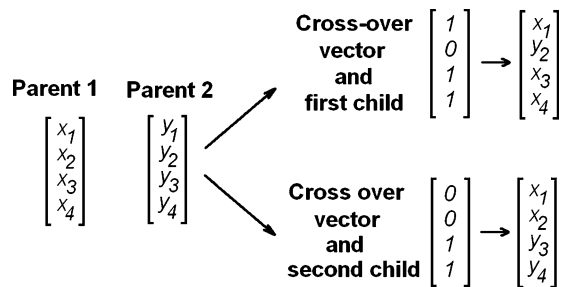
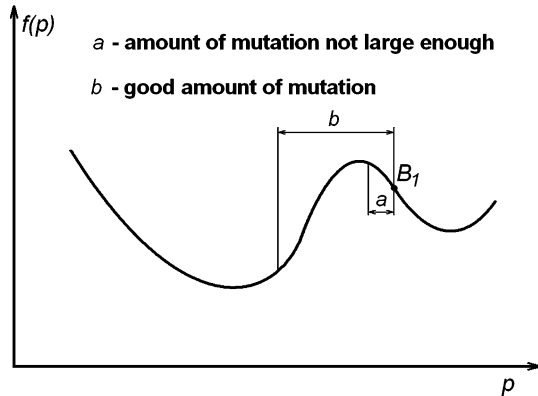


Fig. 2.23 Cross-over scheme with four parameter optimization problem

Fig. 2.24 The influence of amount of mutation to the capability of the algorithm to avoid trapping in local minima



Existence of mutation children is very important for the development of GA with the capability to avoid trapping in the local minima. The success of this feature becomes very much dependant on the proper selection of the mutation amount. Figure 2.24 illustrates the influence of this parameter for the one-dimensional case with a local minimum visualized in Fig. 2.22. Obviously by setting a small amount of mutation, the algorithm will not have capability to “jump” out of the local minimum zones.

Applying previously described scheme, next generation is produced that substitutes the current one. This process iteratively continues until the convergence criteria are met.

Genetic algorithms are randomly driven optimization procedure, and in order to have them working properly they need to involve relatively large number of computations. As previously mentioned, in the structural problems here of interest, a successful GA are usually having population of about 50 members, while the number of generations up to the convergence to the global minimum is usually about 100. GA represent a soft computing technique, and so the convergence criteria are also defined in rather loose manner. In most of the cases it is enough to introduce two stopping criteria for a successful implementation of GA. The first one puts an upper bound to the maximum allowed number of generations, while the second one puts a limit to the so-called “stalling” number of generations that represents the number of consecutive generations in which no improvement is obtained in terms of individuals with the lowest value of the objective function. The latter one usually has the value of about 20–40% of the former.

Within presented scheme, the first two rules (i.e. selection for the elite children and cross-over) are contributing to better exploration of the zone that the present generation is currently occupying. On the other hand, mutation rules are contributing to the exploration of the rest of the region for potentially improved individuals.

In order to explain this mechanism let us consider example in which the objective function is of Rastrigin’s type defined by the following equation

Fig. 2.25 Analytical function with large number of local minima

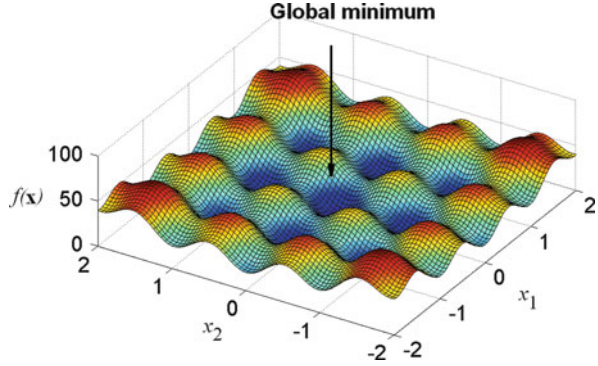
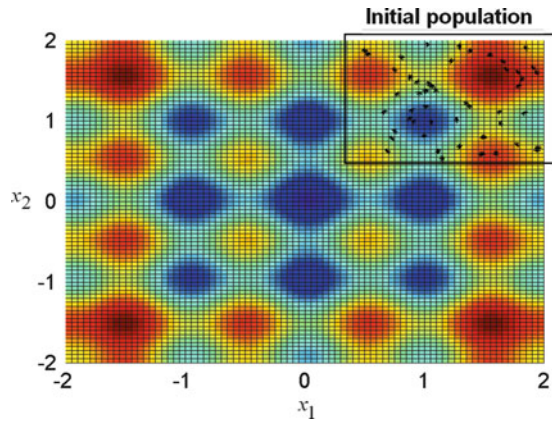


Fig. 2.26 Initial population for GA optimization of analytical function given by Eq. 2.48



$$f(\mathbf{x}) = 20 + 5x_1^2 + 5x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2) \quad (2.48)$$

The graph of this function for the domain $x_1, x_2 \in [-2, 2]$ is visualized in Fig. 2.25. The function is characterized by a large number of local minima, and only one global minimum defined for the coordinate $\mathbf{x} = [0, 0]^T$. This function is frequently used to test GA's capability of optimization of extremely non convex functions.

Let us imagine that the optimization by GA starts with initial population which covers one relatively restricted zone of one of the local minima (see Fig. 2.26). As the optimization starts, driven by first two mechanisms (i.e. selection of the elite children and crossing-over of the existing individuals), the individuals will first start to group in the zone of local minima, as for the present range it offers the lowest value of the objective function. On the other hand at each generation there is a set of individuals to whom a random mutation (i.e. perturbation of the parameter values) is attributed, and therefore within couple of generation there is a strong probability that eventually some of them will "jump" out of this zone of local minima.

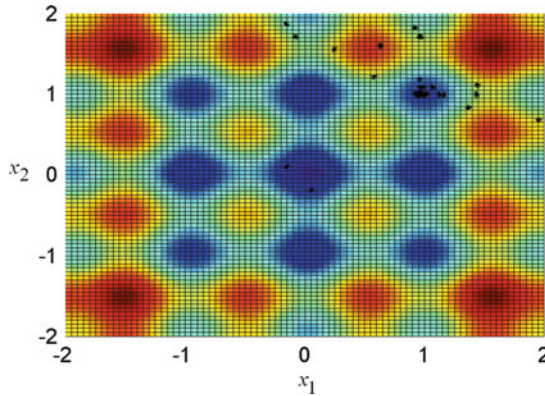


Fig. 2.27 One of the later generations from GA optimization of the analytical function given by Eq. 2.48

Figure 2.27 visualizes some of the later generations of the same optimization. From the figure it may be noticed that even though most of the individuals are grouped in the zone of local minima where the initial population was concentrated, there are two individuals that “jumped” in the zone with smaller values of the objective function. In the following ranking, these will be moved to the elite children and then further by applying cross-over rules they will contribute to the grouping of the individuals in this zone in the following generations.

With this mechanism GA manages to minimize the objective function without mathematically solving the problem. Furthermore, as demonstrated on this example, GA can effectively tackle the problems of multiple local minima as they don’t base computation on derivatives and therefore are not terminating the optimization when mathematical minimum is found.

This nice feature of GA however comes for the price of increased computational cost. The total number of objective function evaluations is equal to the product between number of individuals and the total number of generations reached within the optimization procedure. As previously anticipated in structural problems discussed in this book usually these numbers are about 50 members in each population, while the number of generations is usually between 50 and 100. Clearly, the total number of objective function evaluations is at least one order of magnitude larger than what is required when traditional derivative based algorithms are employed. This contributes to the conclusion that it is numerically justifiable to apply GA only in those situations when the objective function is characterized by a large number of local minima, and so the traditional derivative based algorithms would perform poorly.

The following listing presents a possible implementation of discussed GA.

```
*****
clear
clc
% ~~~~~
% SETTING THE OPTIONS
% ~~~~~
% Basic parameters
% ~~~~~
population=100;
elite=2; % Number of individuals that are considered as elite
cross=0.8; % Cross-over ratio
crossP=0.5; % Ratio of individuals to consider as c-o parents
mutR=0.3; % Ratio of individuals to consider for mutation
mutrange=2.5; % Interval of mutation (0.2 means +/-0.1)
TOTGEN=200; % Total number of generations
TOTSTALL=80; % Total number of stalling generations
% ~~~~~
% Computed parameters
% ~~~~~
% Needed number of the parents to produce next generation
crossN=round(cross*(population-elite));
crsP=round(population*crossP);
mutRN=round(mutR*population);
% Setting the range
npar=2;
range1=[0.5,2];
range2=[0.5,2];
populOld=rand(population,npar);
populOld(:,1)=range1(1)+populOld(:,1)*(range1(2)-range1(1));
populOld(:,2)=range2(1)+populOld(:,2)*(range2(2)-range2(1));
generation=1;
genstl=0;
BEST=1e5;
%~~~~~
% beggining of iterations
while generation<TOTGEN
% Calculating fitness values
for i=1:population
    fitness(i,1)=objrastr([populOld(i,1),populOld(i,2)]);
end
% Scoring the result
result(generation,1)=mean(fitness);
result(generation,2)=min(fitness);
% Sorting the population based on value of fitness function
popsort=[fitness,populOld];
popsort=sortrows(popsort,1);
% Checking if the population stalls
if popsort(1,1)<BEST
    BEST=popsort(1,1);
    genstl=0;
else
    genstl=genstl+1;
end
end
```

```

if genstl>TOTSTALL
    break
end
% Moving the best ones to the elite
eliteKids=popsort(1:elite,2:3);
% Parents for crossover
parX=round(1+(crsP)*rand(crossN,2));
% Individuals for mutation
parMBCK=round(mutRN*rand(population-elite-crossN,1));
parM=population-parMBCK;
% Generating cross-over kids
for i=1:crossN
    first=round(rand(2,1)); % Vector saying which genes will
    be taken from the first parent
    second=[1;1]-first;

    xKids(i,1)=first(1)*populOld(parX(i,1),1)+second(1)*populOld(parX(i,2),1);

    xKids(i,2)=first(2)*populOld(parX(i,1),2)+second(2)*populOld(parX(i,2),2);
end
% Generating mutation kids
for i=1:population-elite-crossN
    mut=mutrange*rand(2,1)-mutrange/2;
    mKids(i,1)=populOld(parM(i),1)+mut(1);
    mKids(i,2)=populOld(parM(i),2)+mut(2);
end
generation=generation+1;
populOld=[eliteKids;xKids;mKids];
end
% end of iterations
%~~~~~
plot(result(:,1))
hold on
plot(result(:,2))
hold off
grid on
*****

*****
function omg=objrastr(x)
omg=20+5*x(1)^2+5*x(2)^2-10*(cos(2*pi*x(1))+cos(2*pi*x(2)));
*****

```

In the GA implementation presented in the above listing, previously discussed operations are implemented in the following way.

The number of elite children is directly given (not as ratio), and those individuals are passed to the successive generation without any modification.

To control cross-over children two variables are used. Variable `cross` represents a ratio between cross-over children and overall population number.

This number needs to be even as it will form number of couples, representing parents, which will cross their genes (parameters) in order to form new children. In order to keep the number of individuals in the population constant during the evolution, from each couple two children are generated. Second variable used to control the crossing-over rule is `crossP` that represents the number of individuals (expressed in ratio of the overall population number) that will be considered as potential parents for crossing over. For example, if this ratio is defined as 0.4, it means that first 40% of the individuals will be considered as potential parents (i.e. the best 40% in terms of fitness function value). From these individuals a number of couples defined by the other variable is randomly selected. Obviously with this implementation repetition of the same individuals in different couples is possible.

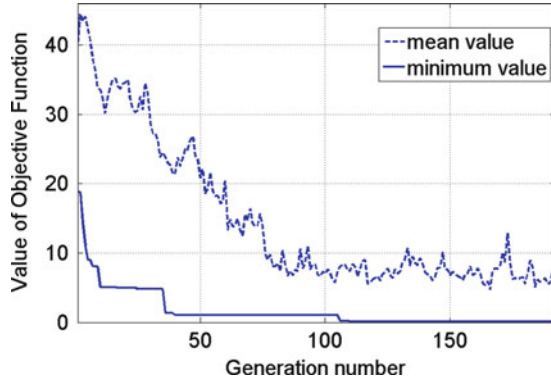
The number of individuals that will mutate is already defined and is equal to what remains from the total number of individuals when previous two groups are subtracted. The rest of the mutation process is controlled by additional two variables: `mutR` and `mutrange`. The former one defines ratio of individuals that will be considered as those to whom a random mutation will be applied, while the latter represents a range of “jump” attributed to the parameters. For example if the `mutR` is set to 0.3 it means that from last 30% of the individuals (i.e. last in terms of fitness function value) a number of individuals will be selected according to previously calculated number of mutation children. This implies that also here the individuals may be repeated as they are selected randomly. On the other hand, `mutrange` refers to the maximum amount of mutation that can be attributed to each of the individuals. As previously illustrated this variable represents an important quantity as it directly influences the performance of the generic algorithm in the presence of local minima.

As for the stopping criteria in the present GA implementation only two of them are implemented, namely the total number of allowed generations is prescribed (variable `TOTGEN`) and the number of stalling generations is given (variable `TOTSTALL`), or number of consecutive generations in which any improvement in the objective function value is not achieved. With this stopping criteria the optimization will be terminated not later than `TOTGEN` generation, or even earlier if during the optimization for more than `TOTSTALL` generations there will be no improvement in the objective function.

During the optimization cycle matrix `result` serves to store the objective function value from the best individual, and the mean value of the objective function for each generation. These results are plotted at the end of the optimization in the graphical representation that is characteristic for the genetic algorithm optimizations.

Implemented GA can be used to solve optimization problem defined by Eq. 2.48 in order to verify the capability of the algorithm to solve difficult cases in terms of the number of local minima. This optimization problem is solved by prescribing the number of individuals in each simulation to be equal to 100, while the stopping criteria are defined by maximum allowed number of generations equal to 200, and stalling generation number equal to 80. By purpose, the initial population is generated in the zone of local minima (i.e. both parameters within the range

Fig. 2.28 Result of the GA optimization of Rastrigin's type function



[0.5, 2]). The amount of mutation prescribed (that also turned out to be sufficient given the performance of the algorithm) was equal to 2.5.

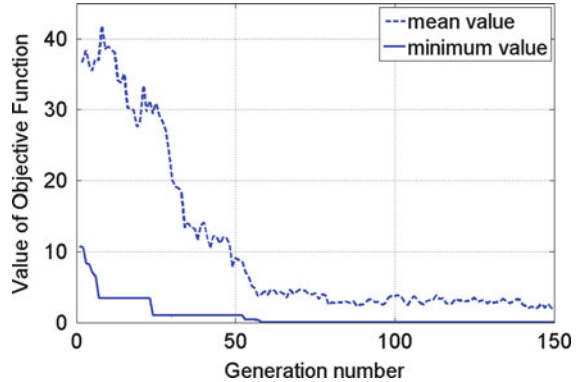
Figure 2.28 visualizes the result of this optimization. From the graph it may be observed how the value of objective function from the best individual in each generation descends monotonely, while keeping its value unchanged for some number of generations. On the other hand, the mean value of the objective function for each population fluctuates, as a result of random nature of the process which not necessarily improves the result of all the individuals. Nevertheless there is also a global descending nature of this graph which indicates the tendency of grouping the individuals in the zone of global minimum. The fact that mean value remains a bit detached from the minimum value shows a certain scattering of the individuals.

Even though our goal is not to bring *all* of the individuals to the global minimum value, since the result of optimization is anyhow represented by the best individual, still the following modification may contribute to overall improvement of the algorithm performance. Instead of having a constant value for the mutation amount, an alternative implementation can start with some larger value attributed to it (e.g. like 2.5 in this case), but at certain point changing it to somewhat smaller number. The reasoning behind this modification consists in the fact that this variable helps the algorithm to “jump” out of possible local minima, but after a certain number of generations it is reasonable to expect that most of the individuals would find themselves in the global minimum zone. Therefore, in order to have all the GA operations (i.e. elite selection, crossing-over and random mutation) working in the direction of improving the objective function within the present zone, a mutation range amount can be decreased.

By implementing a simple modification to previous code that will reduced mutation amount to one fourth of its initial value after the generation reaches number 50, the performance of the algorithm in fact are improved.

Result of the GA optimization with variable mutation amount is visualized in Fig. 2.29. For this particular case the improvement achieved by this modification is not large since the reduction in overall number of generations is not significant. The influence of this modification is evidenced also in the mean value of the objective

Fig. 2.29 Result of the GA optimization with variable mutation amount



function. At the end of the optimization, the mean value of the objective function for the last generation is smaller than in previous case, a circumstance that points out more dense distribution of the individuals in the zone of global minimum. This feature of the algorithm extends the probability of finding the individuals with smaller value of the objective function faster by the applied GA processes resulting therefore in potentially more effective optimization.

2.5 Summary

In this chapter some of the most frequently used optimization algorithms are discussed and their implementation into MATLAB codes is presented. These codes will be used further in this book as a part of inverse analysis procedures designed for the parameter identification.

First part of the chapter showed traditional, derivative based, optimization procedures. In particular two different strategies are discussed: Line search methods, and trust region methods. All of these algorithms have some strong and some weak points, and the text presented in this chapter attempted to point them out. For example, steepest descend line search algorithm has a good feature of being robust as with this approach a global convergence is guaranteed. However, as we could see, in some situations it can perform rather poor involving significantly larger number of function evaluations with respect to other algorithms. Newton direction line search is extremely powerful provided that Hessian matrix is positive definite, and that model function represents a good approximation of real objective function.

Trust region algorithms on the other hand use a different philosophy and at each step they minimize the model function subjected to a trust region constrain. This approach is particularly efficient if the objective function to be minimized is complicated, and therefore cannot be very accurately approximated by a quadratic

form. The minimization is therefore restricted only to a nearby zone where it is trusted that the approximation is good enough.

Both of these groups of algorithms are solving minimization problem mathematically and therefore are sensitive to a presence of local minima (i.e. they cannot distinguish between local and global minimum). If the problem under consideration turns out to have a large number of local minima, all of these algorithms are ineffective. For these situations it is more convenient to use Genetic Algorithms.

A brief description on main principles together with a possible GA implementation is given in the second part of this chapter. In general, GA are involving more evaluations of the objective function. Within the problems of interest presented in this book, the evaluation of the objective function usually involves one simulation of the system response, so sometimes it may be a time consuming task. Therefore, it is computationally unjustifiable to use GA for problems that can be relatively successfully solved by traditional derivative based algorithms.

Optimization algorithm represents one part of the inverse analysis procedure. When it is required to design some parameter characterization procedure, an important issue is its robustness and stability. Examples treated in the chapter should serve to have an idea about the potentialities and limitations of particular optimization algorithms. The behavior of any algorithm depends strongly on the type of the function which should be optimized. Therefore, a general suggestion in selection of the particular algorithm that should be used within the procedure is to keep it as simple and robust as possible for the problem under consideration.

References

1. Giannessi, F.: *Metodi matematici della programmazione. Problemi lineari e non lineari*, Bologna (1982)
2. Nocedal, J., Wright, S.J.: *Numerical Optimization*. Springer, New York (2006)
3. Bonnans, J.F., Gilbert, J.C., Lemarechal, C., Sagastizabal, C.A.: *Numerical Optimization – Theoretical and Practical Aspects*. Springer, New York (2000)
4. Dussault, J.P.: Convergence of Implementable Descent Algorithms for Unconstrained Optimization. *J Optimiz Theory App* **104**(3), 739–745 (2000)
5. De Leone, R., Guadoso, M.: Stopping Criteria For Line Search Methods Without Derivatives. *Math Program* **30**(3), 285–300 (1984)
6. Powell, M.J.D.: *A new algorithm for unconstrained optimization. Non-linear programming – Academic press: 34–65*, New York (1970)
7. Branch, M.A., Coleman, T.F., Li, Y.: A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems. *SIAM J Sci Comput* **21**(1), 1–23 (1999)
8. Byrd, R.H., Schnabel, R.B.: Approximate Solution of the Trust Region Problem by Minimization over Two-Dimensional Subspaces. *Math Program* **40**, 247–263 (1988)
9. Konar, A.: *Artificial Intelligence and Soft Computing – Behavioral and Cognitive Modeling of Human Brain*. CRC Press, New York (2000)
10. Poli, R., Langdon, W.B., McPhee, N.F.: *A field guide to genetic programming*. Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk>, (With contributions by J. R. Koza (2008)

Inverse Analyses with Model Reduction
Proper Orthogonal Decomposition in Structural
Mechanics

Buljak, V.

2012, XIV, 206 p., Hardcover

ISBN: 978-3-642-22702-8