

# CHAPTER 7

# Large-Scale Unconstrained Optimization

Many applications give rise to unconstrained optimization problems with thousands or millions of variables. Problems of this size can be solved efficiently only if the storage and computational costs of the optimization algorithm can be kept at a tolerable level. A diverse collection of large-scale optimization methods has been developed to achieve this goal, each being particularly effective for certain problem types. Some of these methods are straightforward adaptations of the methods described in Chapters 3, 4, and 6. Other approaches are modifications of these basic methods that allow approximate steps to be calculated at lower cost in computation and storage. One set of approaches that we have already discussed—the nonlinear conjugate gradient methods of Section 5.2—can be applied

to large problems without modification, because of its minimal storage demands and its reliance on only first-order derivative information.

The line search and trust-region Newton algorithms of Chapters 3 and 4 require matrix factorizations of the Hessian matrices  $\nabla^2 f_k$ . In the large-scale case, these factorizations can be carried out using sparse elimination techniques. Such algorithms have received much attention, and high quality software implementations are available. If the computational cost and memory requirements of these sparse factorization methods are affordable for a given application, and if the Hessian matrix can be formed explicitly, Newton methods based on sparse factorizations constitute an effective approach for solving such problems.

Often, however, the cost of factoring the Hessian is prohibitive, and it is preferable to compute approximations to the Newton step using iterative linear algebra techniques. Section 7.1 discusses inexact Newton methods that use these techniques, in both line search and trust-region frameworks. The resulting algorithms have attractive global convergence properties and may be superlinearly convergent for suitable choices of parameters. They find effective search directions when the Hessian  $\nabla^2 f_k$  is indefinite, and may even be implemented in a “Hessian-free” manner, without explicit calculation or storage of the Hessian.

The Hessian approximations generated by the quasi-Newton approaches of Chapter 6 are usually dense, even when the true Hessian is sparse, and the cost of storing and working with these approximations can be excessive for large  $n$ . Section 7.2 discusses limited-memory variants of the quasi-Newton approach, which use Hessian approximations that can be stored compactly by using just a few vectors of length  $n$ . These methods are fairly robust, inexpensive, and easy to implement, but they do not converge rapidly. Another approach, discussed briefly in Section 7.3, is to define quasi-Newton approximate Hessians  $B_k$  that preserve sparsity, for example by mimicking the sparsity pattern of the Hessian.

In Section 7.4, we note that objective functions in large problems often possess a structural property known as *partial separability*, which means they can be decomposed into a sum of simpler functions, each of which depends on only a small subspace of  $\mathbf{R}^n$ . Effective Newton and quasi-Newton methods that exploit this property have been developed. Such methods usually converge rapidly and are robust, but they require detailed information about the objective function, which can be difficult to obtain in some applications.

We conclude the chapter with a discussion of software for large-scale unconstrained optimization problems.

## 7.1 INEXACT NEWTON METHODS

Recall from (2.15) that the basic Newton step  $p_k^N$  is obtained by solving the symmetric  $n \times n$  linear system

$$\nabla^2 f_k p_k^N = -\nabla f_k. \quad (7.1)$$

In this section, we describe techniques for obtaining approximations to  $p_k^N$  that are

inexpensive to calculate but are good search directions or steps. These approaches are based on solving (7.1) by using the conjugate gradient (CG) method (see Chapter 5) or the Lanczos method, with modifications to handle negative curvature in the Hessian  $\nabla^2 f_k$ . Both line search and trust-region approaches are described here. We refer to this family of methods by the general name *inexact Newton methods*.

The use of iterative methods for (7.1) spares us from concerns about the expense of a direct factorization of the Hessian  $\nabla^2 f_k$  and the fill-in that may occur during this process. Further, we can customize the solution strategy to ensure that the rapid convergence properties associated with Newton's methods are not lost in the inexact version. In addition, as noted below, we can implement these methods in a Hessian-free manner, so that the Hessian  $\nabla^2 f_k$  need not be calculated or stored explicitly at all.

We examine first how the inexactness in the step calculation determines the local convergence properties of inexact Newton methods. We then consider line search and trust-region approaches based on using CG (possibly with preconditioning) to obtain an approximate solution of (7.1). Finally, we discuss the use of the Lanczos method for solving (7.1) approximately.

### LOCAL CONVERGENCE OF INEXACT NEWTON METHODS

Most rules for terminating the iterative solver for (7.1) are based on the residual

$$r_k = \nabla^2 f_k p_k + \nabla f_k, \quad (7.2)$$

where  $p_k$  is the inexact Newton step. Usually, we terminate the CG iterations when

$$\|r_k\| \leq \eta_k \|\nabla f_k\|, \quad (7.3)$$

where the sequence  $\{\eta_k\}$  (with  $0 < \eta_k < 1$  for all  $k$ ) is called the *forcing sequence*.

We now study how the rate of convergence of inexact Newton methods based on (7.1)–(7.3) is affected by the choice of the forcing sequence. The next two theorems apply not just to Newton–CG procedures but to all inexact Newton methods whose steps satisfy (7.2) and (7.3).

Our first result says that local convergence is obtained simply by ensuring that  $\eta_k$  is bounded away from 1.

#### Theorem 7.1.

*Suppose that  $\nabla^2 f(x)$  exists and is continuous in a neighborhood of a minimizer  $x^*$ , with  $\nabla^2 f(x^*)$  is positive definite. Consider the iteration  $x_{k+1} = x_k + p_k$  where  $p_k$  satisfies (7.3), and assume that  $\eta_k \leq \eta$  for some constant  $\eta \in [0, 1)$ . Then, if the starting point  $x_0$  is sufficiently near  $x^*$ , the sequence  $\{x_k\}$  converges to  $x^*$  and satisfies*

$$\|\nabla^2 f(x^*)(x_{k+1} - x^*)\| \leq \hat{\eta} \|\nabla^2 f(x^*)(x_k - x^*)\|, \quad (7.4)$$

for some constant  $\hat{\eta}$  with  $\eta < \hat{\eta} < 1$ .

Rather than giving a rigorous proof of this theorem, we present an informal derivation that contains the essence of the argument and motivates the next result.

Since the Hessian matrix  $\nabla^2 f$  is positive definite at  $x^*$  and continuous near  $x^*$ , there exists a positive constant  $L$  such that  $\|(\nabla^2 f_k)^{-1}\| \leq L$  for all  $x_k$  sufficiently close to  $x^*$ . We therefore have from (7.2) that the inexact Newton step satisfies

$$\|p_k\| \leq L(\|\nabla f_k\| + \|r_k\|) \leq 2L\|\nabla f_k\|,$$

where the second inequality follows from (7.3) and  $\eta_k < 1$ . Using this expression together with Taylor's theorem and the continuity of  $\nabla^2 f(x)$ , we obtain

$$\begin{aligned} \nabla f_{k+1} &= \nabla f_k + \nabla^2 f_k p_k + \int_0^1 [\nabla f(x_k + tp_k) - \nabla f(x_k)] p_k dt \\ &= \nabla f_k + \nabla^2 f_k p_k + o(\|p_k\|) \\ &= \nabla f_k - (\nabla f_k - r_k) + o(\|\nabla f_k\|) \\ &= r_k + o(\|\nabla f_k\|). \end{aligned} \tag{7.5}$$

Taking norms and recalling (7.3), we have that

$$\|\nabla f_{k+1}\| \leq \eta_k \|\nabla f_k\| + o(\|\nabla f_k\|) \leq (\eta_k + o(1)) \|\nabla f_k\|. \tag{7.6}$$

When  $x_k$  is close enough to  $x^*$  that the  $o(1)$  term in the last estimate is bounded by  $(1 - \eta)/2$ , we have

$$\|\nabla f_{k+1}\| \leq (\eta_k + (1 - \eta)/2) \|\nabla f_k\| \leq \frac{1 + \eta}{2} \|\nabla f_k\|, \tag{7.7}$$

so the gradient norm decreases by a factor of  $(1 + \eta)/2$  at this iteration. By choosing the initial point  $x_0$  sufficiently close to  $x^*$ , we can ensure that this rate of decrease occurs at every iteration.

To prove (7.4), we note that under our smoothness assumptions, we have

$$\nabla f_k = \nabla^2 f(x^*)(x_k - x^*) + o(\|x_k - x^*\|).$$

Hence it can be shown that for  $x_k$  close to  $x^*$ , the gradient  $\nabla f_k$  differs from the scaled error  $\nabla^2 f(x^*)(x_k - x^*)$  by only a relatively small perturbation. A similar estimate holds at  $x_{k+1}$ , so (7.4) follows from (7.7).

From (7.6), we have that

$$\frac{\|\nabla f_{k+1}\|}{\|\nabla f_k\|} \leq \eta_k + o(1). \tag{7.8}$$

If  $\lim_{k \rightarrow \infty} \eta_k = 0$ , we have from this expression that

$$\lim_{k \rightarrow \infty} \frac{\|\nabla f_{k+1}\|}{\|\nabla f_k\|} = 0,$$

indicating Q-superlinear convergence of the gradient norms  $\|\nabla f_k\|$  to zero. Superlinear convergence of the iterates  $\{x_k\}$  to  $x^*$  can be proved as a consequence.

We can obtain quadratic convergence by making the additional assumption that the Hessian  $\nabla^2 f(x)$  is Lipschitz continuous near  $x^*$ . In this case, the estimate (7.5) can be tightened to

$$\nabla f_{k+1} = r_k + O(\|\nabla f_k\|^2).$$

By choosing the forcing sequence so that  $\eta_k = O(\|\nabla f_k\|)$ , we have from this expression that

$$\|\nabla f_{k+1}\| = O(\|\nabla f_k\|^2),$$

indicating Q-quadratic convergence of the gradient norms to zero (and thus also Q-quadratic convergence of the iterates  $x_k$  to  $x^*$ ). The last two observations are summarized in the following theorem.

**Theorem 7.2.**

*Suppose that the conditions of Theorem 7.1 hold, and assume that the iterates  $\{x_k\}$  generated by the inexact Newton method converge to  $x^*$ . Then the rate of convergence is superlinear if  $\eta_k \rightarrow 0$ . If in addition,  $\nabla^2 f(x)$  is Lipschitz continuous for  $x$  near  $x^*$  and if  $\eta_k = O(\|\nabla f_k\|)$ , then the convergence is quadratic.*

To obtain superlinear convergence, we can set, for example,  $\eta_k = \min(0.5, \sqrt{\|\nabla f_k\|})$ ; the choice  $\eta_k = \min(0.5, \|\nabla f_k\|)$  would yield quadratic convergence.

All the results presented in this section, which are proved by Dembo, Eisenstat, and Steihaug [89], are local in nature: They assume that the sequence  $\{x_k\}$  eventually enters the near vicinity of the solution  $x^*$ . They also assume that the unit step length  $\alpha_k = 1$  is taken and hence that globalization strategies do not interfere with rapid convergence. In the following pages we show that inexact Newton strategies can, in fact, be incorporated in practical line search and trust-region implementations of Newton's method, yielding algorithms with good local and global convergence properties. We start with a line search approach.

## LINE SEARCH NEWTON–CG METHOD

In the line search Newton–CG method, also known as the *truncated Newton method*, we compute the search direction by applying the CG method to the Newton equations (7.1) and

attempt to satisfy a termination test of the form (7.3). However, the CG method is designed to solve positive definite systems, and the Hessian  $\nabla^2 f_k$  may have negative eigenvalues when  $x_k$  is not close to a solution. Therefore, we terminate the CG iteration as soon as a direction of negative curvature is generated. This adaptation of the CG method produces a search direction  $p_k$  that is a descent direction. Moreover, the adaptation guarantees that the fast convergence rate of the pure Newton method is preserved, provided that the step length  $\alpha_k = 1$  is used whenever it satisfies the acceptance criteria.

We now describe Algorithm 7.1, a line search algorithm that uses a modification of Algorithm 5.2 as the inner iteration to compute each search direction  $p_k$ . For purposes of this algorithm, we write the linear system (7.1) in the form

$$B_k p = -\nabla f_k, \quad (7.9)$$

where  $B_k$  represents  $\nabla^2 f_k$ . For the inner CG iteration, we denote the search directions by  $d_j$  and the sequence of iterates that it generates by  $z_j$ . When  $B_k$  is positive definite, the inner iteration sequence  $\{z_j\}$  will converge to the Newton step  $p_k^N$  that solves (7.9). At each major iteration, we define a tolerance  $\epsilon_k$  that specifies the required accuracy of the computed solution. For concreteness, we choose the forcing sequence to be  $\eta_k = \min(0.5, \sqrt{\|\nabla f_k\|})$  to obtain a superlinear convergence rate, but other choices are possible.

**Algorithm 7.1** (Line Search Newton–CG).

```

Given initial point  $x_0$ ;
for  $k = 0, 1, 2, \dots$ 
    Define tolerance  $\epsilon_k = \min(0.5, \sqrt{\|\nabla f_k\|}) \|\nabla f_k\|$ ;
    Set  $z_0 = 0, r_0 = \nabla f_k, d_0 = -r_0 = -\nabla f_k$ ;
    for  $j = 0, 1, 2, \dots$ 
        if  $d_j^T B_k d_j \leq 0$ 
            if  $j = 0$ 
                return  $p_k = -\nabla f_k$ ;
            else
                return  $p_k = z_j$ ;
        Set  $\alpha_j = r_j^T r_j / d_j^T B_k d_j$ ;
        Set  $z_{j+1} = z_j + \alpha_j d_j$ ;
        Set  $r_{j+1} = r_j + \alpha_j B_k d_j$ ;
        if  $\|r_{j+1}\| < \epsilon_k$ 
            return  $p_k = z_{j+1}$ ;
        Set  $\beta_{j+1} = r_{j+1}^T r_{j+1} / r_j^T r_j$ ;
        Set  $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$ ;
    end (for)
    Set  $x_{k+1} = x_k + \alpha_k p_k$ , where  $\alpha_k$  satisfies the Wolfe, Goldstein, or
    Armijo backtracking conditions (using  $\alpha_k = 1$  if possible);
end

```

The main differences between the inner loop of Algorithm 7.1 and Algorithm 5.2 are that the specific starting point  $z_0 = 0$  is used; the use of a positive tolerance  $\epsilon_k$  allows the CG iterations to terminate at an inexact solution; and the negative curvature test  $d_j^T B_k d_j \leq 0$  ensures that  $p_k$  is a descent direction for  $f$  at  $x_k$ . If negative curvature is detected on the first inner iteration  $j = 0$ , the returned direction  $p_k = -\nabla f_k$  is both a descent direction and a direction of nonpositive curvature for  $f$  at  $x_k$ .

We can modify the CG iterations in Algorithm 7.1 by introducing preconditioning, in the manner described in Chapter 5.

Algorithm 7.1 is well suited for large problems, but it has a weakness. When the Hessian  $\nabla^2 f_k$  is nearly singular, the line search Newton–CG direction can be long and of poor quality, requiring many function evaluations in the line search and giving only a small reduction in the function. To alleviate this difficulty, we can try to normalize the Newton step, but good rules for doing so are difficult to determine. They run the risk of undermining the rapid convergence of Newton’s method in the case where the pure Newton step is well scaled. It is preferable to introduce a threshold value into the test  $d_j^T B_k d_j \leq 0$ , but good choices of the threshold are difficult to determine. The trust-region Newton–CG method described below deals more effectively with this problematic situation and is therefore preferable, in our opinion.

The line search Newton–CG method does not require explicit knowledge of the Hessian  $B_k = \nabla^2 f_k$ . Rather, it requires only that we can supply Hessian–vector products of the form  $\nabla^2 f_k d$  for any given vector  $d$ . When the user cannot easily supply code to calculate second derivatives, or where the Hessian requires too much storage, the techniques of Chapter 8 (automatic differentiation and finite differencing) can be used to calculate these Hessian–vector products. Methods of this type are known as *Hessian-free* Newton methods.

To illustrate the finite-differencing technique briefly, we use the approximation

$$\nabla^2 f_k d \approx \frac{\nabla f(x_k + hd) - \nabla f(x_k)}{h}, \quad (7.10)$$

for some small differencing interval  $h$ . It is easy to prove that the accuracy of this approximation is  $O(h)$ ; appropriate choices of  $h$  are discussed in Chapter 8. The price we pay for bypassing the computation of the Hessian is one new gradient evaluation per CG iteration.

### TRUST-REGION NEWTON–CG METHOD

In Chapter 4, we discussed approaches for finding an approximate solution of the trust-region subproblem (4.3) that produce improvements on the Cauchy point. Here we define a modified CG algorithm for solving the subproblem with these properties. This algorithm, due to Steihaug [281], is specified below as Algorithm 7.2. A complete algorithm for minimizing  $f$  is obtained by using Algorithm 7.2 to generate the step  $p_k$  required by Algorithm 4.1 of Chapter 4, for some choice of tolerance  $\epsilon_k$  at each iteration.

We use notation similar to (7.9) to define the trust-region subproblem for which Steihaug's method finds an approximate solution:

$$\min_{p \in \mathbb{R}^n} m_k(p) \stackrel{\text{def}}{=} f_k + (\nabla f_k)^T p + \frac{1}{2} p^T B_k p \quad \text{subject to } \|p\| \leq \Delta_k, \quad (7.11)$$

where  $B_k = \nabla^2 f_k$ . As in Algorithm 7.1, we use  $d_j$  to denote the search directions of this modified CG iteration and  $z_j$  to denote the sequence of iterates that it generates.

**Algorithm 7.2** (CG–Steihaug).

Given tolerance  $\epsilon_k > 0$ ;

Set  $z_0 = 0, r_0 = \nabla f_k, d_0 = -r_0 = -\nabla f_k$ ;

**if**  $\|r_0\| < \epsilon_k$

**return**  $p_k = z_0 = 0$ ;

**for**  $j = 0, 1, 2, \dots$

**if**  $d_j^T B_k d_j \leq 0$

        Find  $\tau$  such that  $p_k = z_j + \tau d_j$  minimizes  $m_k(p_k)$  in (4.5)  
        and satisfies  $\|p_k\| = \Delta_k$ ;

**return**  $p_k$ ;

    Set  $\alpha_j = r_j^T r_j / d_j^T B_k d_j$ ;

    Set  $z_{j+1} = z_j + \alpha_j d_j$ ;

**if**  $\|z_{j+1}\| \geq \Delta_k$

        Find  $\tau \geq 0$  such that  $p_k = z_j + \tau d_j$  satisfies  $\|p_k\| = \Delta_k$ ;

**return**  $p_k$ ;

    Set  $r_{j+1} = r_j + \alpha_j B_k d_j$ ;

**if**  $\|r_{j+1}\| < \epsilon_k$

**return**  $p_k = z_{j+1}$ ;

    Set  $\beta_{j+1} = r_{j+1}^T r_{j+1} / r_j^T r_j$ ;

    Set  $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$ ;

**end (for).**

The first **if** statement inside the loop stops the method if its current search direction  $d_j$  is a direction of nonpositive curvature along  $B_k$ , while the second **if** statement inside the loop causes termination if  $z_{j+1}$  violates the trust-region bound. In both cases, the method returns the step  $p_k$  obtained by intersecting the current search direction with the trust-region boundary.

The choice of the tolerance  $\epsilon_k$  at each call to Algorithm 7.2 is important in keeping the overall cost of the trust-region Newton–CG method low. Near a well-behaved solution  $x^*$ , the trust-region bound becomes inactive, and the method reduces to the inexact Newton method analyzed in Theorems 7.1 and 7.2. Rapid convergence can be obtained in these circumstances by choosing  $\epsilon_k$  in a similar fashion to Algorithm 7.1.



The essential differences between Algorithm 5.2 and the inner loop of Algorithm 7.2 are that the latter terminates when it violates the trust-region bound  $\|p\| \leq \Delta$ , when it encounters a direction of negative curvature in  $\nabla^2 f_k$ , or when it satisfies a convergence tolerance defined by a parameter  $\epsilon_k$ . In these respects, Algorithm 7.2 is quite similar to the inner loop of Algorithm 7.1.

The initialization of  $z_0$  to zero in Algorithm 7.2 is a crucial feature of the algorithm. Provided  $\|\nabla f_k\|_2 \geq \epsilon_k$ , Algorithm 7.2 terminates at a point  $p_k$  for which  $m_k(p_k) \leq m_k(p_k^c)$ , that is, when the reduction in model function equals or exceeds that of the Cauchy point. To demonstrate this fact, we consider several cases. First, if  $d_0^T B_k d_0 = (\nabla f_k)^T B_k \nabla f_k \leq 0$ , then the condition in the first **if** statement is satisfied, and the algorithm returns the Cauchy point  $p = -\Delta_k(\nabla f_k)/\|\nabla f_k\|$ . Otherwise, Algorithm 7.2 defines  $z_1$  as follows:

$$z_1 = \alpha_0 d_0 = \frac{r_0^T r_0}{d_0^T B_k d_0} d_0 = -\frac{(\nabla f_k)^T \nabla f_k}{(\nabla f_k)^T B_k \nabla f_k} \nabla f_k.$$

If  $\|z_1\| < \Delta_k$ , then  $z_1$  is exactly the Cauchy point. Subsequent steps of Algorithm 7.2 ensure that the final  $p_k$  satisfies  $m_k(p_k) \leq m_k(z_1)$ . When  $\|z_1\| \geq \Delta_k$ , on the other hand, the second **if** statement is activated, and Algorithm 7.2 terminates at the Cauchy point, proving our claim. This property is important for global convergence: Since each step is at least as good as the Cauchy point in reducing the model  $m_k$ , Algorithm 7.2 is globally convergent.

Another crucial property of the method is that each iterate  $z_j$  is larger in norm than its predecessor. This property is another consequence of the initialization  $z_0 = 0$ . Its main implication is that it is acceptable to stop iterating as soon as the trust-region boundary is reached, because no further iterates giving a lower value of the model function  $m_k$  will lie inside the trust region. We state and prove this property formally in the following theorem, which makes use of the expanding subspace property of the conjugate gradient algorithm, described in Theorem 5.2.

**Theorem 7.3.**

*The sequence of vectors  $\{z_j\}$  generated by Algorithm 7.2 satisfies*

$$0 = \|z_0\|_2 < \cdots < \|z_j\|_2 < \|z_{j+1}\|_2 < \cdots < \|p_k\|_2 \leq \Delta_k.$$

PROOF. We first show that the sequences of vectors generated by Algorithm 7.2 satisfy  $z_j^T r_j = 0$  for  $j \geq 0$  and  $z_j^T d_j > 0$  for  $j \geq 1$ .

Algorithm 7.2 computes  $z_{j+1}$  recursively in terms of  $z_j$ ; but when all the terms of this recursion are written explicitly, we see that

$$z_j = z_0 + \sum_{i=0}^{j-1} \alpha_i d_i = \sum_{i=0}^{j-1} \alpha_i d_i,$$

since  $z_0 = 0$ . Multiplying by  $r_j$  and applying the expanding subspace property of conjugate gradients (see Theorem 5.2), we obtain

$$z_j^T r_j = \sum_{i=0}^{j-1} \alpha_i d_i^T r_j = 0. \quad (7.12)$$

An induction proof establishes the relation  $z_j^T d_j > 0$ . By applying the expanding subspace property again, we obtain

$$z_1^T d_1 = (\alpha_0 d_0)^T (-r_1 + \beta_1 d_0) = \alpha_0 \beta_1 d_0^T d_0 > 0.$$

We now make the inductive hypothesis that  $z_j^T d_j > 0$  and deduce that  $z_{j+1}^T d_{j+1} > 0$ . From (7.12), we have  $z_{j+1}^T r_{j+1} = 0$ , and therefore

$$\begin{aligned} z_{j+1}^T d_{j+1} &= z_{j+1}^T (-r_{j+1} + \beta_{j+1} d_j) \\ &= \beta_{j+1} z_{j+1}^T d_j \\ &= \beta_{j+1} (z_j + \alpha_j d_j)^T d_j \\ &= \beta_{j+1} z_j^T d_j + \alpha_j \beta_{j+1} d_j^T d_j. \end{aligned}$$

Because of the inductive hypothesis and positivity of  $\beta_{j+1}$  and  $\alpha_j$ , the last expression is positive.

We now prove the theorem. If Algorithm 7.2 terminates because  $d_j^T B_k d_j \leq 0$  or  $\|z_{j+1}\|_2 \geq \Delta_k$ , then the final point  $p_k$  is chosen to make  $\|p_k\|_2 = \Delta_k$ , which is the largest possible length. To cover all other possibilities in the algorithm, we must show that  $\|z_j\|_2 < \|z_{j+1}\|_2$  when  $z_{j+1} = z_j + \alpha_j d_j$  and  $j \geq 1$ . Observe that

$$\|z_{j+1}\|_2^2 = (z_j + \alpha_j d_j)^T (z_j + \alpha_j d_j) = \|z_j\|_2^2 + 2\alpha_j z_j^T d_j + \alpha_j^2 \|d_j\|_2^2.$$

It follows from this expression and our intermediate result that  $\|z_j\|_2 < \|z_{j+1}\|_2$ , so our proof is complete.  $\square$

From this theorem we see that Algorithm 7.2 sweeps out points  $z_j$  that move on some interpolating path from  $z_1$  to the final solution  $p_k$ , a path in which every step increases its total distance from the start point. When  $B_k = \nabla^2 f_k$  is positive definite, this path may be compared to the path of the dogleg method: Both methods start by minimizing  $m_k$  along the negative gradient direction  $-\nabla f_k$  and subsequently progress toward  $p_k^N$ , until the trust-region boundary intervenes. One can show that, when  $B_k = \nabla^2 f_k$  is positive definite, Algorithm 7.2 provides a decrease in the model (7.11) that is at least half as good as the optimal decrease [320].

### PRECONDITIONING THE TRUST-REGION NEWTON–CG METHOD

As discussed in Chapter 5, preconditioning can be used to accelerate the CG iteration. Preconditioning techniques are based on finding a nonsingular matrix  $D$  such that the eigenvalues of  $D^{-T} \nabla^2 f_k D^{-1}$  have a more favorable distribution. By generalizing Theorem 7.3, we can show that the iterates  $z_j$  generated by a preconditioned variant of Algorithm 7.2 will grow monotonically in the weighted norm  $\|D \cdot\|$ . To be consistent, we should redefine the trust-region subproblem in terms of the same norm, as follows:

$$\min_{p \in \mathbb{R}^n} m_k(p) \stackrel{\text{def}}{=} f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p \quad \text{subject to } \|Dp\| \leq \Delta_k. \quad (7.13)$$

Making the change of variables  $\hat{p} = Dp$  and defining

$$\hat{g}_k = D^{-T} \nabla f_k, \quad \hat{B}_k = D^{-T} (\nabla^2 f_k) D^{-1},$$

we can write (7.13) as

$$\min_{\hat{p} \in \mathbb{R}^n} f_k + \hat{g}_k^T \hat{p} + \frac{1}{2} \hat{p}^T \hat{B}_k \hat{p} \quad \text{subject to } \|\hat{p}\| \leq \Delta,$$

which has exactly the form of (7.11). We can apply Algorithm 7.2 without any modification to this subproblem, which is equivalent to applying a preconditioned version of Algorithm 7.2 to the problem (7.13).

Many preconditioners can be used within this framework; we discuss some of them in Chapter 5. Of particular interest is *incomplete Cholesky* factorization, which has proved useful in a wide range of optimization problems. The incomplete Cholesky factorization of a positive definite matrix  $B$  finds a lower triangular matrix  $L$  such that

$$B = LL^T - R,$$

where the amount of fill-in in  $L$  is restricted in some way. (For instance, it is constrained to have the same sparsity structure as the lower triangular part of  $B$  or is allowed to have a number of nonzero entries similar to that in  $B$ .) The matrix  $R$  accounts for the inexactness in the approximate factorization. The situation is complicated somewhat by the possible indefiniteness of the Hessian  $\nabla^2 f_k$ ; we must be able to handle this indefiniteness as well as maintain the sparsity. The following algorithm combines incomplete Cholesky and a form of modified Cholesky to define a preconditioner for the trust-region Newton–CG approach.

**Algorithm 7.3** (Inexact Modified Cholesky).

Compute  $T = \text{diag}(\|Be_1\|, \|Be_2\|, \dots, \|Be_n\|)$ , where  $e_i$  is the  $i$ th coordinate vector;  
 Set  $\bar{B} \leftarrow T^{-1/2} B T^{-1/2}$ ; Set  $\beta \leftarrow \|\bar{B}\|$ ;

```

(compute a shift to ensure positive definiteness)
if  $\min_i b_{ii} > 0$ 
     $\alpha_0 \leftarrow 0$ 
else
     $\alpha_0 \leftarrow \beta/2$ ;
for  $k = 0, 1, 2, \dots$ 
    Attempt to apply incomplete Cholesky algorithm to obtain

```

$$LL^T = \bar{B} + \alpha_k I;$$

```

    if the factorization is completed successfully
        stop and return  $L$ ;
    else
         $\alpha_{k+1} \leftarrow \max(2\alpha_k, \beta/2)$ ;
end (for)

```

We can then set the preconditioner to be  $D = L^T$ , where  $L$  is the lower triangular matrix output from Algorithm 7.3. A trust-region Newton–CG method using this preconditioner is implemented in the LANCELOT [72] and TRON [192] codes.

### TRUST-REGION NEWTON-LANCZOS METHOD

A limitation of Algorithm 7.2 is that it accepts *any* direction of negative curvature, even when this direction gives an insignificant reduction in the model. Consider, for example, the case where the subproblem (7.11) is

$$\min_p m(p) = 10^{-3} p_1 - 10^{-4} p_1^2 - p_2^2 \quad \text{subject to } \|p\| \leq 1,$$

where subscripts indicate elements of the vector  $p$ . The steepest descent direction at  $p = 0$  is  $(-10^{-3}, 0)^T$ , which is a direction of negative curvature for the model. Algorithm 7.2 would follow this direction to the boundary of the trust region, yielding a reduction in model function  $m$  of about  $10^{-3}$ . A step along  $e_2$ —also a direction of negative curvature—would yield a much greater reduction of 1.

Several remedies have been proposed. We have seen in Chapter 4 that when the Hessian  $\nabla^2 f_k$  contains negative eigenvalues, the search direction should have a significant component along the eigenvector corresponding to the most negative eigenvalue of  $\nabla^2 f_k$ . This feature would allow the algorithm to move away rapidly from stationary points that are not minimizers. One way to achieve this is to compute a nearly exact solution of the trust-region subproblem (7.11) using the techniques described in Section 4.3. This approach requires the solution of a few linear systems with coefficient matrices of the form

$B_k + \lambda I$ . Although this approach is perhaps too expensive in the large-scale case, it generates productive search directions in all cases.

A more practical alternative is to use the *Lanczos method* (see, for example, [136]) rather than the CG method to solve the linear system  $B_k p = -\nabla f_k$ . The Lanczos method can be seen as a generalization of the CG method that is applicable to indefinite systems, and we can use it to continue the CG process while gathering negative curvature information.

After  $j$  steps, the Lanczos method generates an  $n \times j$  matrix  $Q_j$  with orthogonal columns that span the Krylov subspace (5.15) generated by this method. This matrix has the property that  $Q_j^T B Q_j = T_j$ , where  $T_j$  is a tridiagonal. We can take advantage of this tridiagonal structure and seek to find an approximate solution of the trust-region subproblem in the range of the basis  $Q_j$ . To do so, we solve the problem

$$\min_{w \in \mathbb{R}^j} f_k + e_1^T Q_j (\nabla f_k) e_1^T w + \frac{1}{2} w^T T_j w \quad \text{subject to } \|w\| \leq \Delta_k, \quad (7.14)$$

where  $e_1 = (1, 0, 0, \dots, 0)^T$ , and we define the approximate solution of the trust-region subproblem as  $p_k = Q_j w$ . Since  $T_j$  is tridiagonal, problem (7.14) can be solved by factoring the system  $T_j + \lambda I$  and following the (nearly) exact approach of Section 4.3.

The Lanczos iteration may be terminated, as in the Newton–CG methods, by a test of the form (7.3). Preconditioning can also be incorporated to accelerate the convergence of the Lanczos iteration. The additional robustness in this trust-region algorithm comes at the cost of a more expensive solution of the subproblem than in the Newton–CG approach. A sophisticated implementation of the Newton–Lanczos approach has been implemented in the GLTR package [145].

## **7.2 LIMITED-MEMORY QUASI-NEWTON METHODS**

Limited-memory quasi-Newton methods are useful for solving large problems whose Hessian matrices cannot be computed at a reasonable cost or are not sparse. These methods maintain simple and compact approximations of Hessian matrices: Instead of storing fully dense  $n \times n$  approximations, they save only a few vectors of length  $n$  that represent the approximations implicitly. Despite these modest storage requirements, they often yield an acceptable (albeit linear) rate of convergence. Various limited-memory methods have been proposed; we focus mainly on an algorithm known as L-BFGS, which, as its name suggests, is based on the BFGS updating formula. The main idea of this method is to use curvature information from only the most recent iterations to construct the Hessian approximation. Curvature information from earlier iterations, which is less likely to be relevant to the actual behavior of the Hessian at the current iteration, is discarded in the interest of saving storage.

Following our discussion of L-BFGS and its convergence behavior, we discuss its relationship to the nonlinear conjugate gradient methods of Chapter 5. We then discuss

implementations of limited-memory schemes that make use of a compact representation of approximate Hessian information. These techniques can be applied not only to L-BFGS but also to limited-memory versions of other quasi-Newton procedures such as SR1. Finally, we discuss quasi-Newton updating schemes that impose a particular sparsity pattern on the approximate Hessian.

### LIMITED-MEMORY BFGS

We begin our description of the L-BFGS method by recalling its parent, the BFGS method, which was described in Algorithm 8.1. Each step of the BFGS method has the form

$$x_{k+1} = x_k - \alpha_k H_k \nabla f_k, \quad (7.15)$$

where  $\alpha_k$  is the step length and  $H_k$  is updated at every iteration by means of the formula

$$H_{k+1} = V_k^T H_k V_k + \rho_k s_k s_k^T \quad (7.16)$$

(see (6.17)), where

$$\rho_k = \frac{1}{y_k^T s_k}, \quad V_k = I - \rho_k y_k s_k^T, \quad (7.17)$$

and

$$s_k = x_{k+1} - x_k, \quad y_k = \nabla f_{k+1} - \nabla f_k. \quad (7.18)$$

Since the inverse Hessian approximation  $H_k$  will generally be dense, the cost of storing and manipulating it is prohibitive when the number of variables is large. To circumvent this problem, we store a *modified* version of  $H_k$  *implicitly*, by storing a certain number (say,  $m$ ) of the vector pairs  $\{s_i, y_i\}$  used in the formulas (7.16)–(7.18). The product  $H_k \nabla f_k$  can be obtained by performing a sequence of inner products and vector summations involving  $\nabla f_k$  and the pairs  $\{s_i, y_i\}$ . After the new iterate is computed, the oldest vector pair in the set of pairs  $\{s_i, y_i\}$  is replaced by the new pair  $\{s_k, y_k\}$  obtained from the current step (7.18). In this way, the set of vector pairs includes curvature information from the  $m$  most recent iterations. Practical experience has shown that modest values of  $m$  (between 3 and 20, say) often produce satisfactory results.

We now describe the updating process in a little more detail. At iteration  $k$ , the current iterate is  $x_k$  and the set of vector pairs is given by  $\{s_i, y_i\}$  for  $i = k - m, \dots, k - 1$ . We first choose some initial Hessian approximation  $H_k^0$  (in contrast to the standard BFGS iteration, this initial approximation is allowed to vary from iteration to iteration) and find by repeated application of the formula (7.16) that the L-BFGS approximation  $H_k$  satisfies the following

formula:

$$\begin{aligned}
H_k &= (V_{k-1}^T \cdots V_{k-m}^T) H_k^0 (V_{k-m} \cdots V_{k-1}) \\
&\quad + \rho_{k-m} (V_{k-1}^T \cdots V_{k-m+1}^T) s_{k-m} s_{k-m}^T (V_{k-m+1} \cdots V_{k-1}) \\
&\quad + \rho_{k-m+1} (V_{k-1}^T \cdots V_{k-m+2}^T) s_{k-m+1} s_{k-m+1}^T (V_{k-m+2} \cdots V_{k-1}) \\
&\quad + \cdots \\
&\quad + \rho_{k-1} s_{k-1} s_{k-1}^T.
\end{aligned} \tag{7.19}$$

From this expression we can derive a recursive procedure to compute the product  $H_k \nabla f_k$  efficiently.

**Algorithm 7.4** (L-BFGS two-loop recursion).

```

 $q \leftarrow \nabla f_k;$ 
for  $i = k - 1, k - 2, \dots, k - m$ 
     $\alpha_i \leftarrow \rho_i s_i^T q;$ 
     $q \leftarrow q - \alpha_i y_i;$ 
end (for)
 $r \leftarrow H_k^0 q;$ 
for  $i = k - m, k - m + 1, \dots, k - 1$ 
     $\beta \leftarrow \rho_i y_i^T r;$ 
     $r \leftarrow r + s_i (\alpha_i - \beta)$ 
end (for)
stop with result  $H_k \nabla f_k = r.$ 

```

Without considering the multiplication  $H_k^0 q$ , the two-loop recursion scheme requires  $4mn$  multiplications; if  $H_k^0$  is diagonal, then  $n$  additional multiplications are needed. Apart from being inexpensive, this recursion has the advantage that the multiplication by the initial matrix  $H_k^0$  is isolated from the rest of the computations, allowing this matrix to be chosen freely and to vary between iterations. We may even use an implicit choice of  $H_k^0$  by defining some initial approximation  $B_k^0$  to the Hessian (not its inverse) and obtaining  $r$  by solving the system  $B_k^0 r = q$ .

A method for choosing  $H_k^0$  that has proved effective in practice is to set  $H_k^0 = \gamma_k I$ , where

$$\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}. \tag{7.20}$$

As discussed in Chapter 6,  $\gamma_k$  is the scaling factor that attempts to estimate the size of the true Hessian matrix along the most recent search direction (see (6.21)). This choice helps to ensure that the search direction  $p_k$  is well scaled, and as a result the step length  $\alpha_k = 1$  is accepted in most iterations. As discussed in Chapter 6, it is important that the line search be

based on the Wolfe conditions (3.6) or strong Wolfe conditions (3.7), so that BFGS updating is stable.

The limited-memory BFGS algorithm can be stated formally as follows.

**Algorithm 7.5** (L-BFGS).

Choose starting point  $x_0$ , integer  $m > 0$ ;

$k \leftarrow 0$ ;

**repeat**

    Choose  $H_k^0$  (for example, by using (7.20));

    Compute  $p_k \leftarrow -H_k \nabla f_k$  from Algorithm 7.4;

    Compute  $x_{k+1} \leftarrow x_k + \alpha_k p_k$ , where  $\alpha_k$  is chosen to satisfy the Wolfe conditions;

**if**  $k > m$

        Discard the vector pair  $\{s_{k-m}, y_{k-m}\}$  from storage;

    Compute and save  $s_k \leftarrow x_{k+1} - x_k$ ,  $y_k = \nabla f_{k+1} - \nabla f_k$ ;

$k \leftarrow k + 1$ ;

**until convergence.**

The strategy of keeping the  $m$  most recent correction pairs  $\{s_i, y_i\}$  works well in practice; indeed no other strategy has yet proved to be consistently better. During its first  $m - 1$  iterations, Algorithm 7.5 is equivalent to the BFGS algorithm of Chapter 6 if the initial matrix  $H_0$  is the same in both methods, and if L-BFGS chooses  $H_k^0 = H_0$  at each iteration.

Table 7.1 presents results illustrating the behavior of Algorithm 7.5 for various levels of memory  $m$ . It gives the number of function and gradient evaluations (nfg) and the total CPU time. The test problems are taken from the CUTE collection [35], the number of variables is indicated by  $n$ , and the termination criterion  $\|\nabla f_k\| \leq 10^{-5}$  is used. The table shows that the algorithm tends to be less robust when  $m$  is small. As the amount of storage increases, the number of function evaluations tends to decrease; but since the cost of each iteration increases with the amount of storage, the best CPU time is often obtained for small values of  $m$ . Clearly, the optimal choice of  $m$  is problem dependent.

Because some rival algorithms are inefficient, Algorithm 7.5 is often the approach of choice for large problems in which the true Hessian is not sparse. In particular, a Newton

**Table 7.1** Performance of Algorithm 7.5.

| Problem  | $n$  | L-BFGS<br>$m = 3$ |      | L-BFGS<br>$m = 5$ |      | L-BFGS<br>$m = 17$ |      | L-BFGS<br>$m = 29$ |       |
|----------|------|-------------------|------|-------------------|------|--------------------|------|--------------------|-------|
|          |      | nfg               | time | nfg               | time | nfg                | time | nfg                | time  |
| DIXMAANL | 1500 | 146               | 16.5 | 134               | 17.4 | 120                | 28.2 | 125                | 44.4  |
| EIGENALS | 110  | 821               | 21.5 | 569               | 15.7 | 363                | 16.2 | 168                | 12.5  |
| FREUROTH | 1000 | >999              | —    | >999              | —    | 69                 | 8.1  | 38                 | 6.3   |
| TRIDIA   | 1000 | 876               | 46.6 | 611               | 41.4 | 531                | 84.6 | 462                | 127.1 |



method in which the exact Hessian is computed and factorized is not practical in such circumstances. The L-BFGS approach may also outperform Hessian-free Newton methods such as Newton–CG approaches, in which Hessian–vector products are calculated by finite differences or automatic differentiation. The main weakness of the L-BFGS method is that it converges slowly on ill-conditioned problems—specifically, on problems where the Hessian matrix contains a wide distribution of eigenvalues. On certain applications, the nonlinear conjugate gradient methods discussed in Chapter 5 are competitive with limited-memory quasi-Newton methods.

### RELATIONSHIP WITH CONJUGATE GRADIENT METHODS

Limited-memory methods evolved as an attempt to improve nonlinear conjugate gradient methods, and early implementations resembled conjugate gradient methods more than quasi-Newton methods. The relationship between the two classes is the basis of a *memoryless BFGS iteration*, which we now outline.

We start by considering the Hestenes–Stiefel form of the nonlinear conjugate gradient method (5.46). Recalling that  $s_k = \alpha_k p_k$ , we have that the search direction for this method is given by

$$p_{k+1} = -\nabla f_{k+1} + \frac{\nabla f_{k+1}^T y_k}{y_k^T p_k} p_k = -\left(I - \frac{s_k y_k^T}{y_k^T s_k}\right) \nabla f_{k+1} \equiv -\hat{H}_{k+1} \nabla f_{k+1}. \quad (7.21)$$

This formula resembles a quasi-Newton iteration, but the matrix  $\hat{H}_{k+1}$  is neither symmetric nor positive definite. We could symmetrize it as  $\hat{H}_{k+1}^T \hat{H}_{k+1}$ , but this matrix does not satisfy the secant equation  $\hat{H}_{k+1} y_k = s_k$  and is, in any case, singular. An iteration matrix that is symmetric, positive definite, and satisfies the secant equation is given by

$$H_{k+1} = \left(I - \frac{s_k y_k^T}{y_k^T s_k}\right) \left(I - \frac{y_k s_k^T}{y_k^T s_k}\right) + \frac{s_k s_k^T}{y_k^T s_k}. \quad (7.22)$$

This matrix is exactly the one obtained by applying a single BFGS update (7.16) to the identity matrix. Hence, an algorithm whose search direction is given by  $p_{k+1} = -H_{k+1} \nabla f_{k+1}$ , with  $H_{k+1}$  defined by (7.22), can be thought of as a “memoryless” BFGS method, in which the previous Hessian approximation is always reset to the identity matrix before updating it and where only the most recent correction pair  $(s_k, y_k)$  is kept at every iteration. Alternatively, we can view the method as a variant of Algorithm 7.5 in which  $m = 1$  and  $H_k^0 = I$  at each iteration.

A more direct connection with conjugate gradient methods can be seen if we consider the memoryless BFGS formula (7.22) in conjunction with an exact line search, for which

$\nabla f_{k+1}^T p_k = 0$  for all  $k$ . We then obtain

$$p_{k+1} = -H_{k+1} \nabla f_{k+1} = -\nabla f_{k+1} + \frac{\nabla f_{k+1}^T y_k}{y_k^T p_k} p_k, \quad (7.23)$$

which is none other than the Hestenes–Stiefel conjugate gradient method. Moreover, it is easy to verify that when  $\nabla f_{k+1}^T p_k = 0$ , the Hestenes–Stiefel formula reduces to the Polak–Ribière formula (5.44). Even though the assumption of exact line searches is unrealistic, it is intriguing that the BFGS formula is related in this way to the Polak–Ribière and Hestenes–Stiefel methods.

### GENERAL LIMITED-MEMORY UPDATING

Limited-memory quasi-Newton approximations are useful in a variety of optimization methods. L-BFGS, Algorithm 7.5, is a line search method for unconstrained optimization that (implicitly) updates an approximation  $H_k$  to the inverse of the Hessian matrix. Trust-region methods, on the other hand, require an approximation  $B_k$  to the Hessian matrix, not to its inverse. We would also like to develop limited-memory methods based on the SR1 formula, which is an attractive alternative to BFGS; see Chapter 6. In this section we consider limited-memory updating in a general setting and show that by representing quasi-Newton matrices in a compact (or outer product) form, we can derive efficient implementations of *all* popular quasi-Newton update formulas, and their inverses. These compact representations will also be useful in designing limited-memory methods for constrained optimization, where approximations to the Hessian or reduced Hessian of the Lagrangian are needed; see Chapter 18 and Chapter 19.

We will consider only limited-memory methods (such as L-BFGS) that continuously refresh the correction pairs by removing and adding information at each stage. A different approach saves correction pairs until the available storage is exhausted and then discards all correction pairs (except perhaps one) and starts the process anew. Computational experience suggests that this second approach is less effective in practice.

Throughout this chapter we let  $B_k$  denote an approximation to a Hessian matrix and  $H_k$  the approximation to the inverse. In particular, we always have that  $B_k^{-1} = H_k$ .

### COMPACT REPRESENTATION OF BFGS UPDATING

We now describe an approach to limited-memory updating that is based on representing quasi-Newton matrices in outer-product form. We illustrate it for the case of a BFGS approximation  $B_k$  to the Hessian.

#### Theorem 7.4.

*Let  $B_0$  be symmetric and positive definite, and assume that the  $k$  vector pairs  $\{s_i, y_i\}_{i=0}^{k-1}$  satisfy  $s_i^T y_i > 0$ . Let  $B_k$  be obtained by applying  $k$  BFGS updates with these vector pairs to  $B_0$ ,*

using the formula (6.19). We then have that

$$B_k = B_0 - \begin{bmatrix} B_0 S_k & Y_k \end{bmatrix} \begin{bmatrix} S_k^T B_0 S_k & L_k \\ L_k^T & -D_k \end{bmatrix}^{-1} \begin{bmatrix} S_k^T B_0 \\ Y_k^T \end{bmatrix}, \quad (7.24)$$

where  $S_k$  and  $Y_k$  are the  $n \times k$  matrices defined by

$$S_k = [s_0, \dots, s_{k-1}], \quad Y_k = [y_0, \dots, y_{k-1}], \quad (7.25)$$

while  $L_k$  and  $D_k$  are the  $k \times k$  matrices

$$(L_k)_{i,j} = \begin{cases} s_{i-1}^T y_{j-1} & \text{if } i > j, \\ 0 & \text{otherwise,} \end{cases} \quad (7.26)$$

$$D_k = \text{diag}[s_0^T y_0, \dots, s_{k-1}^T y_{k-1}]. \quad (7.27)$$

This result can be proved by induction. We note that the conditions  $s_i^T y_i > 0$ ,  $i = 0, 1, \dots, k-1$ , ensure that the middle matrix in (7.24) is nonsingular, so that this expression is well defined. The utility of this representation becomes apparent when we consider limited-memory updating.

As in the L-BFGS algorithm, we keep the  $m$  most recent correction pairs  $\{s_i, y_i\}$  and refresh this set at every iteration by removing the oldest pair and adding a newly generated pair. During the first  $m$  iterations, the update procedure described in Theorem 7.4 can be used without modification, except that usually we make the specific choice  $B_k^0 = \delta_k I$  for the basic matrix, where  $\delta_k = 1/\gamma_k$  and  $\gamma_k$  is defined by (7.20).

At subsequent iterations  $k > m$ , the update procedure needs to be modified slightly to reflect the changing nature of the set of vector pairs  $\{s_i, y_i\}$  for  $i = k-m, k-m+1, \dots, k-1$ . Defining the  $n \times m$  matrices  $S_k$  and  $Y_k$  by

$$S_k = [s_{k-m}, \dots, s_{k-1}], \quad Y_k = [y_{k-m}, \dots, y_{k-1}], \quad (7.28)$$

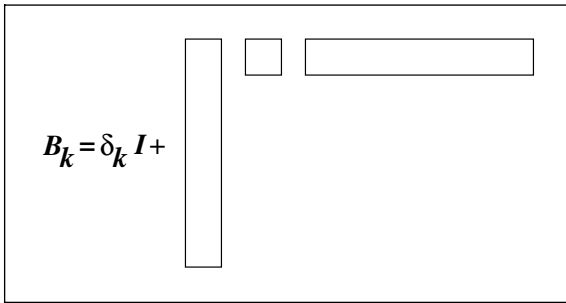
we find that the matrix  $B_k$  resulting from  $m$  updates to the basic matrix  $B_0^{(k)} = \delta_k I$  is given by

$$B_k = \delta_k I - \begin{bmatrix} \delta_k S_k & Y_k \end{bmatrix} \begin{bmatrix} \delta_k S_k^T S_k & L_k \\ L_k^T & -D_k \end{bmatrix}^{-1} \begin{bmatrix} \delta_k S_k^T \\ Y_k^T \end{bmatrix}, \quad (7.29)$$

where  $L_k$  and  $D_k$  are now the  $m \times m$  matrices defined by

$$(L_k)_{i,j} = \begin{cases} (s_{k-m-1+i})^T (y_{k-m-1+j}) & \text{if } i > j, \\ 0 & \text{otherwise,} \end{cases}$$

$$D_k = \text{diag}[s_{k-m}^T y_{k-m}, \dots, s_{k-1}^T y_{k-1}].$$



**Figure 7.1**  
Compact (or outer product) representation of  $B_k$  in (7.29).

After the new iterate  $x_{k+1}$  is generated, we obtain  $S_{k+1}$  by deleting  $s_{k-m}$  from  $S_k$  and adding the new displacement  $s_k$ , and we update  $Y_{k+1}$  in a similar fashion. The new matrices  $L_{k+1}$  and  $D_{k+1}$  are obtained in an analogous way.

Since the middle matrix in (7.29) is small—of dimension  $2m$ —its factorization requires a negligible amount of computation. The key idea behind the compact representation (7.29) is that the corrections to the basic matrix can be expressed as an outer product of two long and narrow matrices— $[\delta_k S_k Y_k]$  and its transpose—with an intervening multiplication by a small  $2m \times 2m$  matrix. See Figure 7.1 for a graphical illustration.

The limited-memory updating procedure of  $B_k$  requires approximately  $2mn + O(m^3)$  operations, and matrix–vector products of the form  $B_k v$  can be performed at a cost of  $(4m + 1)n + O(m^2)$  multiplications. These operation counts indicate that updating and manipulating the direct limited-memory BFGS matrix  $B_k$  is quite economical when  $m$  is small.

This approximation  $B_k$  can be used in a trust-region method for unconstrained optimization or, more significantly, in methods for bound-constrained and general-constrained optimization. The program L-BFGS-B [322] makes extensive use of compact limited-memory approximations to solve large nonlinear optimization problems with bound constraints. In this situation, projections of  $B_k$  into subspaces defined by the constraint gradients must be calculated repeatedly. Several codes for general-constrained optimization, including KNITRO and IPOPT, make use of the compact limited-memory matrix  $B_k$  to approximate the Hessian of the Lagrangians; see Section 19.3

We can derive a formula, similar to (7.24), that provides a compact representation of the inverse BFGS approximation  $H_k$ ; see [52] for details. An implementation of the unconstrained L-BFGS algorithm based on this expression requires a similar amount of computation as the algorithm described in the previous section.

Compact representations can also be derived for matrices generated by the symmetric rank-one (SR1) formula. If  $k$  updates are applied to the symmetric matrix  $B_0$  using the vector pairs  $\{s_i, y_i\}_{i=0}^{k-1}$  and the SR1 formula (6.24), the resulting matrix  $B_k$  can be expressed as

$$B_k = B_0 + (Y_k - B_0 S_k)(D_k + L_k + L_k^T - S_k^T B_0 S_k)^{-1}(Y_k - B_0 S_k)^T, \quad (7.30)$$

where  $S_k$ ,  $Y_k$ ,  $D_k$ , and  $L_k$  are as defined in (7.25), (7.26), and (7.27). Since the SR1 method is self-dual, the inverse formula  $H_k$  can be obtained simply by replacing  $B$ ,  $s$ , and  $y$  by  $H$ ,  $y$ , and  $s$ , respectively. Limited-memory SR1 methods can be derived in the same way as the BFGS method. We replace  $B_0$  with the basic matrix  $B_k^0$  at the  $k$ th iteration, and we redefine  $S_k$  and  $Y_k$  to contain the  $m$  most recent corrections, as in (7.28). We note, however, that limited-memory SR1 updating is sometimes not as effective as L-BFGS updating because it may not produce positive definite approximations near a solution.

### UNROLLING THE UPDATE

The reader may wonder whether limited-memory updating can be implemented in simpler ways. In fact, as we show here, the most obvious implementation of limited-memory BFGS updating is considerably more expensive than the approach based on compact representations discussed in the previous section.

The direct BFGS formula (6.19) can be written as

$$B_{k+1} = B_k - a_k a_k^T + b_k b_k^T, \quad (7.31)$$

where the vectors  $a_k$  and  $b_k$  are defined by

$$a_k = \frac{B_k s_k}{(s_k^T B_k s_k)^{\frac{1}{2}}}, \quad b_k = \frac{y_k}{(y_k^T s_k)^{\frac{1}{2}}}. \quad (7.32)$$

We could continue to save the vector pairs  $\{s_i, y_i\}$  but use the formula (7.31) to compute matrix–vector products. A limited-memory BFGS method that uses this approach would proceed by defining the basic matrix  $B_k^0$  at each iteration and then updating according to the formula

$$B_k = B_k^0 + \sum_{i=k-m}^{k-1} [b_i b_i^T - a_i a_i^T]. \quad (7.33)$$

The vector pairs  $\{a_i, b_i\}$ ,  $i = k - m, k - m + 1, \dots, k - 1$ , would then be recovered from the stored vector pairs  $\{s_i, y_i\}$ ,  $i = k - m, k - m + 1, \dots, k - 1$ , by the following procedure:

**Procedure 7.6** (Unrolling the BFGS formula).

```

for  $i = k - m, k - m + 1, \dots, k - 1$ 
     $b_i \leftarrow y_i / (y_i^T s_i)^{1/2}$ ;
     $a_i \leftarrow B_k^0 s_i + \sum_{j=k-m}^{i-1} [(b_j^T s_i) b_j - (a_j^T s_i) a_j]$ ;
     $a_i \leftarrow a_i / (s_i^T a_i)^{1/2}$ ;
end (for)

```

Note that the vectors  $a_i$  must be recomputed at each iteration because they all depend on the vector pair  $\{s_{k-m}, y_{k-m}\}$ , which is removed at the end of iteration  $k$ . On the other hand, the vectors  $b_i$  and the inner products  $b_j^T s_i$  can be saved from the previous iteration, so only the new values  $b_{k-1}$  and  $b_j^T s_{k-1}$  need to be computed at the current iteration.

By taking all these computations into account, and assuming that  $B_k^0 = I$ , we find that approximately  $\frac{3}{2}m^2n$  operations are needed to determine the limited-memory matrix. The actual computation of the inner product  $B_m v$  (for arbitrary  $v \in \mathbb{R}^n$ ) requires  $4mn$  multiplications. Overall, therefore, this approach is less efficient than the one based on the compact matrix representation described previously. Indeed, while the product  $B_k v$  costs the same in both cases, updating the representation of the limited-memory matrix by using the compact form requires only  $2mn$  multiplications, compared to  $\frac{3}{2}m^2n$  multiplications needed when the BFGS formula is unrolled.

### 7.3 SPARSE QUASI-NEWTON UPDATES

We now discuss a quasi-Newton approach to large-scale problems that has intuitive appeal: We demand that the quasi-Newton approximations  $B_k$  have the same (or similar) sparsity pattern as the true Hessian. This approach would reduce the storage requirements of the algorithm and perhaps give rise to more accurate Hessian approximations.

Suppose that we know which components of the Hessian may be nonzero at some point in the domain of interest. That is, we know the contents of the set  $\Omega$  defined by

$$\Omega \stackrel{\text{def}}{=} \{(i, j) \mid [\nabla^2 f(x)]_{ij} \neq 0 \text{ for some } x \text{ in the domain of } f\}.$$

Suppose also that the current Hessian approximation  $B_k$  mirrors the nonzero structure of the exact Hessian, that is,  $(B_k)_{ij} = 0$  for  $(i, j) \notin \Omega$ . In updating  $B_k$  to  $B_{k+1}$ , then, we could try to find the matrix  $B_{k+1}$  that satisfies the secant condition, has the same sparsity pattern, and is as close as possible to  $B_k$ . Specifically, we define  $B_{k+1}$  to be the solution of the following quadratic program:

$$\min_B \|B - B_k\|_F^2 = \sum_{(i,j) \in \Omega} [B_{ij} - (B_k)_{ij}]^2, \quad (7.34a)$$

$$\text{subject to } Bs_k = y_k, \quad B = B^T, \quad \text{and } B_{ij} = 0 \text{ for } (i, j) \notin \Omega. \quad (7.34b)$$

One can show that the solution  $B_{k+1}$  of this problem can be obtained by solving an  $n \times n$  linear system whose sparsity pattern is  $\Omega$ , the same as the sparsity of the true Hessian. Once  $B_{k+1}$  has been computed, we can use it, within a trust-region method, to obtain the new iterate  $x_{k+1}$ . We note that  $B_{k+1}$  is not guaranteed to be positive definite.

We omit further details of this approach because it has several drawbacks. The updating process does not possess scale invariance under linear transformations of the variables and,

more significantly, its practical performance has been disappointing. The fundamental weakness of this approach is that (7.34a) is an inadequate model and can produce poor Hessian approximations.

An alternative approach is to relax the secant equation, making sure that it is approximately satisfied along the last few steps rather than requiring it to hold strictly on the latest step. To do so, we define  $S_k$  and  $Y_k$  by (7.28) so that they contain the  $m$  most recent difference pairs. We can then define the new Hessian approximation  $B_{k+1}$  to be the solution of

$$\begin{aligned} \min_B \|BS_k - Y_k\|_F^2 \\ \text{subject to } B = B^T \text{ and } B_{ij} = 0 \text{ for } (i, j) \notin \Omega. \end{aligned}$$

This convex optimization problem has a solution, but it is not easy to compute. Moreover, this approach can produce singular or poorly conditioned Hessian approximations. Even though it frequently outperforms methods based on (7.34a), its performance on large problems has not been impressive.

## 7.4 ALGORITHMS FOR PARTIALLY SEPARABLE FUNCTIONS

In a *separable* unconstrained optimization problem, the objective function can be decomposed into a sum of simpler functions that can be optimized independently. For example, if we have

$$f(x) = f_1(x_1, x_3) + f_2(x_2, x_4, x_6) + f_3(x_5),$$

we can find the optimal value of  $x$  by minimizing each function  $f_i$ ,  $i = 1, 2, 3$ , independently, since no variable appears in more than one function. The cost of performing  $m$  lower-dimensional optimizations is much less in general than the cost of optimizing an  $n$ -dimensional function.

In many large problems the objective function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is not separable, but it can still be written as the sum of simpler functions, known as *element functions*. Each element function has the property that it is unaffected when we move along a large number of linearly independent directions. If this property holds, we say that  $f$  is *partially separable*. All functions whose Hessians  $\nabla^2 f$  are sparse are partially separable, but so are many functions whose Hessian is not sparse. Partial separability allows for economical problem representation, efficient automatic differentiation, and effective quasi-Newton updating.

The simplest form of partial separability arises when the objective function can be written as

$$f(x) = \sum_{i=1}^{ne} f_i(x), \tag{7.35}$$

where each of the element functions  $f_i$  depends on only a few components of  $x$ . It follows that the gradients  $\nabla f_i$  and Hessians  $\nabla^2 f_i$  of each element function contain just a few nonzeros. By differentiating (7.35), we obtain

$$\nabla f(x) = \sum_{i=1}^{ne} \nabla f_i(x), \quad \nabla^2 f(x) = \sum_{i=1}^{ne} \nabla^2 f_i(x).$$

A natural question is whether it is more effective to maintain quasi-Newton approximations to each of the element Hessians  $\nabla^2 f_i(x)$  separately, rather than approximating the entire Hessian  $\nabla^2 f(x)$ . We will show that the answer is affirmative, provided that the quasi-Newton approximation fully exploits the structure of each element Hessian.

We introduce the concept by means of a simple example. Consider the objective function

$$\begin{aligned} f(x) &= (x_1 - x_3^2)^2 + (x_2 - x_4^2)^2 + (x_3 - x_2^2)^2 + (x_4 - x_1^2)^2 \\ &\equiv f_1(x) + f_2(x) + f_3(x) + f_4(x). \end{aligned} \quad (7.36)$$

The Hessians of the element functions  $f_i$  are  $4 \times 4$  sparse, singular matrices with 4 nonzero entries.

Let us focus on  $f_1$ ; all other element functions have exactly the same form. Even though  $f_1$  is formally a function of all components of  $x$ , it depends only on  $x_1$  and  $x_3$ , which we call the *element variables* for  $f_1$ . We assemble the element variables into a vector that we call  $x_{[1]}$ , that is,

$$x_{[1]} = \begin{bmatrix} x_1 \\ x_3 \end{bmatrix},$$

and note that

$$x_{[1]} = U_1 x \quad \text{with} \quad U_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

If we define the function  $\phi_1$  by

$$\phi_1(z_1, z_2) = (z_1 - z_2^2)^2,$$

then we can write  $f_1(x) = \phi_1(U_1 x)$ . By applying the chain rule to this representation, we obtain

$$\nabla f_1(x) = U_1^T \nabla \phi_1(U_1 x), \quad \nabla^2 f_1(x) = U_1^T \nabla^2 \phi_1(U_1 x) U_1. \quad (7.37)$$



In our case, we have

$$\nabla^2 \phi_1(U_1 x) = \begin{bmatrix} 2 & -4x_3 \\ -4x_3 & 12x_3^2 - 4x_1 \end{bmatrix}, \quad \nabla^2 f_1(x) = \begin{bmatrix} 2 & 0 & -4x_3 & 0 \\ 0 & 0 & 0 & 0 \\ -4x_3 & 0 & 12x_3^2 - 4x_1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

The matrix  $U_1$ , known as a *compactifying matrix*, allows us to map the derivative information for the low-dimensional function  $\phi_1$  into the derivative information for the element function  $f_1$ .

Now comes the key idea: Instead of maintaining a quasi-Newton approximation to  $\nabla^2 f_1$ , we maintain a  $2 \times 2$  quasi-Newton approximation  $B_{[1]}$  of  $\nabla^2 \phi_1$  and use the relation (7.37) to transform it into a quasi-Newton approximation to  $\nabla^2 f_1$ . To update  $B_{[1]}$  after a typical step from  $x$  to  $x^+$ , we record the information

$$s_{[1]} = x_{[1]}^+ - x_{[1]}, \quad y_{[1]} = \nabla \phi_1(x_{[1]}^+) - \nabla \phi_1(x_{[1]}), \quad (7.38)$$

and use BFGS or SR1 updating to obtain the new approximation  $B_{[1]}^+$ . We therefore update small, dense quasi-Newton approximations with the property

$$B_{[1]} \approx \nabla^2 \phi_1(U_1 x) = \nabla^2 \phi_1(x_{[1]}). \quad (7.39)$$

To obtain an approximation of the element Hessian  $\nabla^2 f_1$ , we use the transformation suggested by the relationship (7.37); that is,

$$\nabla^2 f_1(x) \approx U_1^T B_{[1]} U_1.$$

This operation has the effect of mapping the elements of  $B_{[1]}$  to the correct positions in the full  $n \times n$  Hessian approximation.

The previous discussion concerned only the first element function  $f_1$ , but we can treat all other functions  $f_i$  in the same way. The full objective function can now be written as

$$f(x) = \sum_{i=1}^{ne} \phi_i(U_i x), \quad (7.40)$$

and we maintain a quasi-Newton approximation  $B_{[i]}$  for each of the functions  $\phi_i$ . To obtain a complete approximation to the full Hessian  $\nabla^2 f$ , we simply sum the element Hessian approximations as follows:

$$B = \sum_{i=1}^{ne} U_i^T B_{[i]} U_i. \quad (7.41)$$

We may use this approximate Hessian in a trust-region algorithm, obtaining an approximate solution  $p_k$  of the system

$$B_k p_k = -\nabla f_k. \quad (7.42)$$

We need not assemble  $B_k$  explicitly but rather use the conjugate gradient approach to solve (7.42), computing matrix–vector products of the form  $B_k v$  by performing operations with the matrices  $U_i$  and  $B_{[i]}$ .

To illustrate the usefulness of this element-by-element updating technique, let us consider a problem of the form (7.36) but this time involving 1000 variables, not just 4. The functions  $\phi_i$  still depend on only two internal variables, so that each Hessian approximation  $B_{[i]}$  is a  $2 \times 2$  matrix. After just a few iterations, we will have sampled enough directions  $s_{[i]}$  to make each  $B_{[i]}$  an accurate approximation to  $\nabla^2 \phi_i$ . Hence the full quasi-Newton approximation (7.41) will tend to be a very good approximation to  $\nabla^2 f(x)$ . By contrast, a quasi-Newton method that ignores the partially separable structure of the objective function will attempt to estimate the total average curvature—the sum of the individual curvatures of the element functions—by approximating the  $1000 \times 1000$  Hessian matrix. When the number of variables  $n$  is large, many iterations will be required before this quasi-Newton approximation is of good quality. Hence an algorithm of this type (for example, standard BFGS or L-BFGS) will require many more iterations than a method based on the partially separable approximate Hessian.

It is not always possible to use the BFGS formula to update the partial Hessian  $B_{[i]}$ , because there is no guarantee that the curvature condition  $s_{[i]}^T y_{[i]} > 0$  will be satisfied. That is, even though the full Hessian  $\nabla^2 f(x)$  is at least positive semidefinite at the solution  $x^*$ , some of the individual Hessians  $\nabla^2 \phi_i(\cdot)$  may be indefinite. One way to overcome this obstacle is to apply the SR1 update to each of the element Hessians. This approach has proved effective in the LANCELOT package [72], which is designed to take full advantage of partial separability.

The main limitations of this quasi-Newton approach are the cost of the step computation (7.42), which is comparable to the cost of a Newton step, and the difficulty of identifying the partially separable structure of a function. The performance of quasi-Newton methods is satisfactory provided that we find the *finest* partially separable decomposition of the problem; see [72]. Furthermore, even when the partially separable structure is known, it may be more efficient to compute a Newton step. For example, the modeling language AMPL automatically detects the partially separable structure of a function  $f$  and uses it to compute the Hessian  $\nabla^2 f(x)$ .

## **7.5 PERSPECTIVES AND SOFTWARE**

Newton–CG methods have been used successfully to solve large problems in a variety of applications. Many of these implementations are developed by engineers and

scientists and use problem-specific preconditioners. Freely available packages include TN/TNBC [220] and TNPACK [275]. Software for more general problems, such as LANCELOT [72], KNITRO/CG [50], and TRON [192], employ Newton–CG methods when applied to unconstrained problems. Other packages, such as LOQO [294] implement Newton methods with a sparse factorization modified to ensure positive definiteness. GLTR [145] offers a Newton–Lanczos method. There is insufficient experience to date to say whether the Newton–Lanczos method is significantly better in practice than the Steihaug strategy given in Algorithm 7.2.

Software for computing incomplete Cholesky preconditioners includes the ICFS [193] and MA57 [166] packages. A preconditioner for Newton–CG based on limited-memory BFGS approximations is provided in PREQN [209].

Limited-memory BFGS methods are implemented in LBFGS [194] and M1QN3 [122]; see Gill and Leonard [125] for a variant that requires less storage and appears to be quite efficient. The compact limited-memory representations of Section 7.2 are used in LBFGS-B [322], IPOPT [301], and KNITRO.

The LANCELOT package exploits partial separability. It provides SR1 and BFGS quasi-Newton options as well as a Newton methods. The step computation is obtained by a preconditioned conjugate gradient iteration using trust regions. If  $f$  is partially separable, a general affine transformation will not in general preserve the partially separable structure. The quasi-Newton method for partially separable functions described in Section 7.4 is not invariant to affine transformations of the variables, but this is not a drawback because the method is invariant under transformations that preserve separability.

## NOTES AND REFERENCES


A complete study of inexact Newton methods is given in [74]. For a discussion of the Newton–Lanczos method see [145]. Other iterative methods for the solution of a trust-region problem have been proposed by Hager [160], and by Rendl and Wolkowicz [263].

For further discussion on the L-BFGS method see Nocedal [228], Liu and Nocedal [194], and Gilbert and Lemaréchal [122]. The last paper also discusses various ways in which the scaling parameter can be chosen. Algorithm 7.4, the two-loop L-BFGS recursion, constitutes an economical procedure for computing the product  $H_k \nabla f_k$ . It is based, however, on the specific form of the BFGS update formula (7.16), and recursions of this type have not yet been developed (and may not exist) for other members of the Broyden class (for instance, the SR1 and DFP methods). Our discussion of compact representations of limited-memory matrices is based on Byrd, Nocedal, and Schnabel [52].

Sparse quasi-Newton updates have been studied by Toint [288, 289] and Fletcher et al. [102, 104], among others. The concept of partial separability was introduced by Griewank and Toint [156, 155]. For an extensive treatment of the subject see Conn, Gould, and Toint [72].


---


**EXERCISES**


 **7.1** Code Algorithm 7.5, and test it on the extended Rosenbrock function


$$f(x) = \sum_{i=1}^{n/2} [\alpha(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2],$$

where  $\alpha$  is a parameter that you can vary (for example, 1 or 100). The solution is  $x^* = (1, 1, \dots, 1)^T$ ,  $f^* = 0$ . Choose the starting point as  $(-1, -1, \dots, -1)^T$ . Observe the behavior of your program for various values of the memory parameter  $m$ .

 **7.2** Show that the matrix  $\hat{H}_{k+1}$  in (7.21) is singular.


 **7.3** Derive the formula (7.23) under the assumption that line searches are exact.


 **7.4** Consider limited-memory SR1 updating based on (7.30). Explain how the storage can be cut in half if the basic matrix  $B_k^0$  is kept fixed for all  $k$ . (Hint: Consider the matrix  $Q_k = [q_0, \dots, q_{k-1}] = Y_k - B_0 S_k$ .)

 **7.5** Write the function defined by

$$f(x) = x_2 x_3 e^{x_1 + x_3 - x_4} + (x_2 x_3)^2 + (x_3 - x_4)$$

in the form (7.40). In particular, give the definition of each of the compactifying transformations  $U_i$ .

 **7.6** Does the approximation  $B$  obtained by the partially separable quasi-Newton updating (7.38), (7.41) satisfy the secant equation  $Bs = y$ ?

 **7.7** The minimum surface problem is a classical application of the calculus of variations and can be found in many textbooks. We wish to find the surface of minimum area, defined on the unit square, that interpolates a prescribed continuous function on the boundary of the square. In the standard discretization of this problem, the unknowns are the values of the sought-after function  $z(x, y)$  on a  $q \times q$  rectangular mesh of points over the unit square.

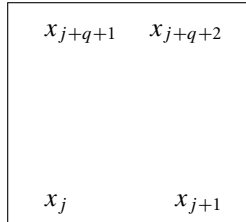
More specifically, we divide each edge of the square into  $q$  intervals of equal length, yielding  $(q + 1)^2$  grid points. We label the grid points as

$$x_{(i-1)(q+1)+1}, \dots, x_{i(q+1)} \quad \text{for } i = 1, 2, \dots, q + 1,$$

so that each value of  $i$  generates a line. With each point we associate a variable  $z_i$  that represents the height of the surface at this point. For the  $4q$  grid points on the boundary of the unit square, the values of these variables are determined by the given function. The

optimization problem is to determine the other  $(q + 1)^2 - 4q$  variables  $z_i$  so that the total surface area is minimized.

A typical subsquare in this partition looks as follows:




We denote this square by  $A_j$  and note that its area is  $q^2$ . The desired function is  $z(x, y)$ , and we wish to compute its surface over  $A_j$ . Calculus books show that the area of the surface is given by

$$f_j(x) \equiv \int \int_{(x,y) \in A_j} \sqrt{1 + \left(\frac{\partial z}{\partial x}\right)^2 + \left(\frac{\partial z}{\partial y}\right)^2} dx dy.$$

Approximate the derivatives by finite differences, and show that  $f_j$  has the form

$$f_j(x) = \frac{1}{q^2} \left[ 1 + \frac{q^2}{2} [(x_j - x_{j+q+1})^2 + (x_{j+1} - x_{j+q})^2] \right]^{\frac{1}{2}}. \quad (7.43)$$

 **7.8** Compute the gradient of the element function (7.43) with respect to the full vector  $x$ . Show that it contains at most four nonzeros, and that two of these four nonzero components are negatives of the other two. Compute the Hessian of  $f_j$ , and show that, among the 16 nonzeros, only three different magnitudes are represented. Also show that this Hessian is singular.