$B_k + \lambda I$. 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 an 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

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

where $e_1 = (1, 0, 0, \ldots, 0)^T$, 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

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

where α_k is the step length and H_k is updated at every iteration by means of the formula

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

(see (6.17)), where

$$\rho_k = \frac{1}{y_k^T s_k}, \qquad V_k = I - \rho_k y_k s_k^T, \tag{7.17}$$

and

$$s_k = x_{k+1} - x_k, \qquad y_k = \nabla f_{k+1} - \nabla f_k. \tag{7.18}$$

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 ∇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 $i = k - m, \ldots, k - 1$. 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:

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

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 $i = k-1, k-2, \ldots, k-m$
  $\alpha_i \leftarrow \rho_i s_i^T q$;
  $q \leftarrow q - \alpha_i y_i$;
end (for)
$r \leftarrow H_k^0 q$;
for $i = k-m, k-m+1, \ldots, k-1$
  $\beta \leftarrow \rho_i y_i^T r$;
  $r \leftarrow r + s_i(\alpha_i - \beta)$
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, γ_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 α_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 $m = 3$		L-BFGS $m = 5$		L-BFGS $m = 17$		L-BFGS $m = 29$	
		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

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

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

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

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

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

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

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

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

$$S_k = [s_0, \ldots, s_{k-1}], \qquad Y_k = [y_0, \ldots, y_{k-1}], \tag{7.25}$$

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

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

$$D_k = \operatorname{diag}\left[s_0^T y_0, \ldots, s_{k-1}^T y_{k-1}\right]. \tag{7.27}$$

This result can be proved by induction. We note that the conditions $s_i^T y_i > 0$, $i = 0, 1, \ldots, k-1$, 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 γ_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 $i = k-m, k-m+1, \ldots, k-1$. Defining the $n \times m$ matrices S_k and Y_k by

$$S_k = [s_{k-m}, \ldots, s_{k-1}], \qquad Y_k = [y_{k-m}, \ldots, y_{k-1}], \tag{7.28}$$

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

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

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

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

$$D_k = \operatorname{diag}\left[s_{k-m}^T y_{k-m}, \ldots, s_{k-1}^T y_{k-1}\right].$$

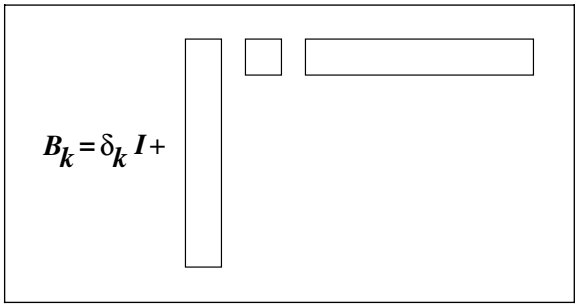

Figure 7.1
Compact (or outer product) representation of B_k in (7.29).

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

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

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 kth 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

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

where the vectors a_k and b_k are defined by

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

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

$$B_k = B_k^0 + \sum_{i=k-m}^{k-1} \left[b_i b_i^T - a_i a_i^T \right]. \tag{7.33}$$

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

Procedure 7.6 (Unrolling the BFGS formula).
for $i = k-m, k-m+1, \ldots, k-1$
  $b_i \leftarrow y_i/(y_i^T s_i)^{1/2}$;
  $a_i \leftarrow B_k^0 s_i + \sum_{j=k-m}^{i-1} \left[(b_j^T s_i) b_j - (a_j^T s_i) a_j \right]$;
  $a_i \leftarrow a_i/(s_i^T a_i)^{1/2}$;
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 Ω 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:

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

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

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 Ω, 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,

Jorge Nocedal Stephen J. Wright

Numerical Optimization

Second Edition

Springer

Jorge Nocedal
EECS Department
Northwestern University
Evanston, IL 60208-3118
USA
nocedal@eecs.northwestern.edu

Stephen J. Wright
Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706–1613
USA
swright@cs.wisc.edu

Mathematics Subject Classification (2000): 90B30, 90C11, 90-01, 90-02

Library of Congress Control Number: 2006923897

ISBN-10: 0-387-30303-0 ISBN-13: 978-0387-30303-1

Printed on acid-free paper.

Printed in the United States of America. (TB/HAM)

9 8 7 6 5 4 3 2 1

springer.com