The general, relative error test is stated as follows. Two scalar quantities, a and b, are said to satisfy a relative error test with respect to a tolerence T if

$$\frac{|a-b|}{|a|} \leq T. \qquad (8.1.4.15)$$

Roughly speaking, if T is of the form 10^{-q} then a and b agree to q digits. Obviously, there is a problem with this test if $a = 0$ and there will be numerical difficulties if a is close to 0. Thus, in practice, (8.1.4.15) is replaced by

$$|a-b| \leq (|a|+1)T, \qquad (8.1.4.16)$$

which reduces to an absolute error test as $a \to 0$. A careful examination may be required to set this tolerance correctly, but, typically, if one of the fast, stable algorithms is used, only a few more iterations are necessary to get six or eight digits if one or two are already known. Note also that the actual value depends on ε, the relative machine precision. It is fruitless to seek more digits of accuracy than are expressed in the machine representation.

A test based on condition (1) is often implemented by using the linear approximation to $\boldsymbol{M}$ or the quadratic approximation to S. Thus, using the quadratic approximation to S, we can compute the predicted reduction by

$$\Delta_{\rm pred} = S(\mathbf{x}_c) - \mathbf{d}^T\boldsymbol{J}^T\boldsymbol{W}[\mathbf{y} - \boldsymbol{M}(\mathbf{x}_c)]. \qquad (8.1.4.17)$$

Similarly, the actual reduction is

$$\Delta_{\rm act} = S(\mathbf{x}_c) - S(\mathbf{x}_+). \qquad (8.1.4.18)$$

The test then becomes $\Delta_{\rm pred} \leq [1 + S(\mathbf{x}_c)]T$, $\Delta_{\rm act} \leq [1 + S(\mathbf{x}_c)]T$, and $\Delta_{\rm act} \leq 2\Delta_{\rm pred}$. That is, we want both the predicted and actual reductions to be small and the actual reduction to agree reasonably well with the predicted reduction. A typical value for T should be 10^{-4}, although the value again depends on ε, and the user is cautioned not to make this tolerance too loose.

For a test on condition (2), we compute the cosine of the angle between the vector of residuals and the linear subspace spanned by the columns of $\mathbf{J}$,

$$\cos\zeta = \{\mathbf{d}^T\boldsymbol{J}^T\boldsymbol{W}[\mathbf{y} - \boldsymbol{M}(\mathbf{x}_+)]\}/[(\mathbf{d}^T\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J}\mathbf{d})S(\mathbf{x}_+)]^{1/2}. \qquad (8.1.4.19)$$

The test is $\cos\zeta \leq T$, where, again, T should be 10^{-4} or smaller.

Test 3 above is usually only present to prevent the process from continuing when almost nothing is happening. Clearly, we do not know $\widehat{\mathbf{x}}$, thus the test is typically that corresponding elements of $\mathbf{x}_+$ and $\mathbf{x}_c$ satisfy (8.1.4.16), where T is chosen to be 10^{-q}. A recommended value of q is half the number of digits carried in the computation, *e.g.* $q = 8$ for standard 64-bit (double-precision or 16 digit) calculations. Sometimes, the relative error test is of the form $|(\mathbf{x}_+)_j - (\mathbf{x}_c)_j|/\sigma_j \leq T$, where σ_j is the standard uncertainty computed from the inverse Hessian in the last iteration. Although this test has some statistical validity, it is quite expensive and usually not worth the work involved to compute.

8.1.4.5. *Recommendations*

One situation in which the Gauss–Newton algorithm behaves particularly poorly is in the vicinity of a saddle point in parameter space, where the true Hessian matrix is not positive definite. This occurs in structure refinement where a symmetric model is refined to convergence and then is replaced by a less symmetric model. The hypersurface of S will have negative curvature in a finite sized region of the parameter space for the less-symmetric model, and it is *essential* to use a safeguarded algorithm, one that incorporates a line search or a trust region, in order to get out of that region.

On the basis of this discussion, we can draw the following conclusions:

(1) In cases where the fit is poor, owing to an incomplete model or in the initial stages of refinement, methods based on the quadratic approximation to S (quasi-Newton methods) often perform better. This is particularly important when the model is close to a more symmetric configuration. These methods are more expensive per iteration and generally require more storage, but their greater stability in such problems usually justifies the cost.

(2) With small residual problems, where the model is complete and close to the solution, a safeguarded Gauss–Newton method is preferred. The trust-region implementation (Levenberg–Marquardt algorithm) has been very successful in practice.

(3) The best advice is to pick a good implementation of either method and stay with it.

8.1.5. Numerical methods for large-scale problems

Because the least-squares problems arising in crystallography are often very large, the methods we have discussed above are not always the most efficient. Some large problems have special structure that can be exploited to produce quite efficient algorithms. A particular special structure is sparsity, that is, the problems have Jacobian matrices that have a large fraction of their entries zero. Of course, not all large problems are sparse, so we shall also discuss approaches applicable to more general problems.

8.1.5.1. *Methods for sparse matrices*

We shall first discuss large, sparse, linear least-squares problems (Heath, 1984), since these techniques form the basis for nonlinear extensions. As we proceed, we shall indicate how the procedures should be modified in order to handle nonlinear problems. Recall that the problem is to find the minimum of the quadratic form $[\mathbf{y} - \boldsymbol{A}\mathbf{x}]^T\boldsymbol{W}[\mathbf{y} - \boldsymbol{A}\mathbf{x}]$, where $\mathbf{y}$ is a vector of observations, $\boldsymbol{A}\mathbf{x}$ represents a vector of the values of a set of linear model functions that predict the values of $\mathbf{y}$, and $\boldsymbol{W}$ is a positive-definite weight matrix. Again, for convenience, we make the transformation $\mathbf{y}' = \boldsymbol{U}\mathbf{y}$, where $\boldsymbol{U}$ is the upper triangular Cholesky factor of $\boldsymbol{W}$, and $\boldsymbol{Z} = \boldsymbol{U}\boldsymbol{A}$, so that the quadratic form becomes $(\mathbf{y}' - \boldsymbol{Z}\mathbf{x})^T(\mathbf{y}' - \boldsymbol{Z}\mathbf{x})$, and the minimum is the solution of the system of normal equations, $\boldsymbol{Z}^T\boldsymbol{Z}\mathbf{x} = \boldsymbol{Z}^T\mathbf{y}'$. Even if $\boldsymbol{Z}$ is sparse, it is easy to see that $\boldsymbol{H} = \boldsymbol{Z}^T\boldsymbol{Z}$ need not be sparse, because if even one row of $\boldsymbol{Z}$ has all of its elements nonzero, all elements of $\boldsymbol{H}$ will be nonzero. Therefore, the direct use of the normal equations may preclude the efficient exploitation of sparsity. But suppose $\boldsymbol{H}$ *is* sparse. The next step in solving the normal equations is to compute the Cholesky decomposition of $\boldsymbol{H}$, and it may turn out that the Cholesky factor is not sparse. For example, if $\boldsymbol{H}$ has the form

$$\boldsymbol{H} = \begin{pmatrix} x & x & x & x \\ x & x & 0 & 0 \\ x & 0 & x & 0 \\ x & 0 & 0 & x \end{pmatrix},$$

where x represents a nonzero element, then the Cholesky factor, $\boldsymbol{R}$, will not be sparse, but if

$$\boldsymbol{H} = \begin{pmatrix} x & 0 & 0 & x \\ 0 & x & 0 & x \\ 0 & 0 & x & x \\ x & x & x & x \end{pmatrix},$$

then $\boldsymbol{R}$ has the form

$$\boldsymbol{R} = \begin{pmatrix} x & 0 & 0 & x \\ 0 & x & 0 & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{pmatrix}.$$

These examples show that, although the sparsity of $\boldsymbol{R}$ is independent of the row ordering of $\boldsymbol{Z}$, the column order can have a profound effect. Procedures exist that analyse $\boldsymbol{Z}$ and select a permutation of the columns that reduces the 'fill' in $\boldsymbol{R}$. An algorithm for using the normal equations is then as follows:

(1) determine a permutation, $\boldsymbol{P}$ (an orthogonal matrix with one and only one 1 in each row and column, and all other elements zero), that tends to ensure a sparse Cholesky factor;
(2) store the elements of $\boldsymbol{R}$ in a sparse format;
(3) compute $\boldsymbol{Z}^T\boldsymbol{Z}$ and $\boldsymbol{Z}^T\mathbf{y}'$;
(4) factor $\boldsymbol{P}^T(\boldsymbol{Z}^T\boldsymbol{Z})\boldsymbol{P}$ to get $\boldsymbol{R}$;
(5) solve $\boldsymbol{R}^T\mathbf{z} = \boldsymbol{P}^T\boldsymbol{Z}^T\mathbf{y}'$;
(6) solve $\boldsymbol{R}\tilde{\mathbf{x}} = \mathbf{z}$;
(7) set $\hat{\mathbf{x}} = \boldsymbol{P}^T\tilde{\mathbf{x}}$.

This algorithm is fast, and it will produce acceptable accuracy if the condition number of $\boldsymbol{Z}$ is not too large. If extension to the nonlinear case is considered, it should be kept in mind that the first two steps need only be done once, since the sparsity pattern of the Jacobian does not, in general, change from iteration to iteration.

The QR decomposition of matrices that may be kept in memory is most often performed by the use of Householder transformations (see Subsection 8.1.1.1). For sparse matrices, or for matrices that are too large to be held in memory, this technique has several drawbacks. First, the method works by inserting zeros in the columns of $\boldsymbol{Z}$, working from left to right, but at each step it tends to fill in the columns to the right of the one currently being worked on, so that columns that are initially sparse cease to be so. Second, each Householder transformation needs to be applied to all of the remaining columns, so that the entire matrix must be retained in memory to make efficient use of this procedure.

The alternative procedure for obtaining the QR decomposition by the use of Givens rotations overcomes these problems if the entire upper triangular matrix, $\boldsymbol{R}$, can be stored in memory. Since this only requires about $p^2/2$ locations, it is usually possible. Also, it may happen that $\boldsymbol{R}$ has a sparse representation, so that even fewer locations will be needed. The algorithm based on Givens rotations is as follows:

(1) bring in the first p rows;
(2) find the QR decomposition of this $p \times p$ matrix;
(3) for $i = p + 1$ to n, do
  (*a*) bring in row i;
  (*b*) eliminate row i using $\boldsymbol{R}$ and at most p Givens rotations.

In order to specify how to use this algorithm to solve the linear least-squares problem, we must also state how to account for $\boldsymbol{Q}$. We could accumulate $\boldsymbol{Q}$ or save enough information to generate it later, but this usually requires excessive storage. The better alternatives are either to apply the steps of $\boldsymbol{Q}$ to $\mathbf{y}'$ as we proceed or to simply discard the information and solve $\boldsymbol{R}^T\boldsymbol{R}\mathbf{x} = \boldsymbol{Z}^T\mathbf{y}'$. It should be noted that the order of rows can make a significant difference. Suppose

$$\boldsymbol{Z} = \begin{pmatrix} x & 0 & 0 & 0 & 0 \\ 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & x \\ x & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & 0 \\ x & x & x & x & x \end{pmatrix}.$$

The work to complete the QR decomposition is of order p^2 operations, because each element below the main diagonal can be eliminated by one Givens rotation with no fill, whereas for

$$\boldsymbol{Z} = \begin{pmatrix} x & x & x & x & x \\ x & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & 0 \\ 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & x \end{pmatrix}$$

each Givens rotation fills an entire row, and the QR decomposition requires of order np^2 operations.

8.1.5.2. *Conjugate-gradient methods*

A numerical procedure that is applicable to large-scale problems that may not be sparse is called the *conjugate-gradient method*. Conjugate-gradient methods were originally designed to solve the quadratic minimization problem, find the minimum of

$$S(\mathbf{x}) = (1/2)\mathbf{x}^T\boldsymbol{H}\mathbf{x} - \mathbf{b}^T\mathbf{x}, \tag{8.1.5.1}$$

where $\boldsymbol{H}$ is a symmetric, positive-definite matrix. The gradient of S is

$$\mathbf{g}(\mathbf{x}) = \boldsymbol{H}\mathbf{x} - \mathbf{b}, \tag{8.1.5.2}$$

and its Hessian matrix is $\boldsymbol{H}$. Given an initial estimate, $\mathbf{x}_0$, the conjugate-gradient algorithm is

(1) define $\mathbf{d}_0 = -\mathbf{g}(\mathbf{x}_0)$;
(2) for $k = 0, 1, 2, ..., p - 1$, $\mathbf{d}_0$
  (*a*) $\alpha_k = -\mathbf{d}_k^T\mathbf{g}(\mathbf{x}_k)/\mathbf{d}_k^T\boldsymbol{H}\mathbf{d}_k$;
  (*b*) $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{d}_k$;
  (*c*) $\gamma_k = \mathbf{g}(\mathbf{x}_{k+1})^T\mathbf{g}(\mathbf{x}_{k+1})/\mathbf{g}(\mathbf{x}_k)^T\mathbf{g}(\mathbf{x}_k)$;
  (*d*) $\mathbf{d}_{k+1} = -\mathbf{g}(\mathbf{x}_k) + \gamma_k\mathbf{d}_k$.

This algorithm finds the exact solution for the quadratic function in not more than p steps.

This algorithm cannot be used directly for the nonlinear case because it requires $\boldsymbol{H}$ to compute α_k, and the goal is to solve the problem without computing the Hessian. To accomplish this, the exact computation of α is replaced by an actual line search, and the termination after at most p steps is replaced by a convergence test. Thus, we obtain, for a given starting value $\mathbf{x}_0$ and a general, nonquadratic function S:

(1) define $\mathbf{d}_0 = -\mathbf{g}(\mathbf{x}_0)$;
(2) set $k = 0$;
(3) do until convergence
  (*a*) $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{d}_k$, where α is chosen by a line search;
  (*b*) $\gamma_k = \mathbf{g}(\mathbf{x}_{k+1})^T\mathbf{g}(\mathbf{x}_{k+1})/\mathbf{g}(\mathbf{x}_k)^T\mathbf{g}(\mathbf{x}_k)$;
  (*c*) $\mathbf{d}_{k+1} = -\mathbf{g}(\mathbf{x}_{k+1}) + \gamma_k\mathbf{d}_k$;
  (*d*) $k = k + 1$.

Note that, as promised, $\boldsymbol{H}$ is not needed. In practice, it has been observed that the line search need not be exact, but that

periodic restarts in the steepest-descent direction are often helpful. This procedure often requires more iterations and function evaluations than methods that store approximate Hessians, but the cost per iteration is small. Thus, it is often the overall least-expensive method for large problems.

For the least-squares problem, recall that we are finding the minimum of

$$S(\mathbf{x}) = (1/2)[\mathbf{y}' - \boldsymbol{Z}\mathbf{x}]^T[\mathbf{y}' - \boldsymbol{Z}\mathbf{x}], \tag{8.1.5.3}$$

for which

$$\mathbf{g}(\mathbf{x}) = \boldsymbol{Z}^T(\boldsymbol{Z}\mathbf{x} - \mathbf{y}'). \tag{8.1.5.4}$$

By using these definitions in the conjugate-gradient algorithm, it is possible to formulate a specific algorithm for linear least squares that requires only the calculation of $\boldsymbol{Z}$ times a vector and $\boldsymbol{Z}^T$ times a vector, and never requires the calculation or factorization of $\boldsymbol{Z}^T\boldsymbol{Z}$.

In practice, such an algorithm will, due to roundoff error, sometimes require more than p iterations to reach a solution. A detailed examination of the performance of the procedure shows, however, that fewer than p iterations will be required if the eigenvalues of $\boldsymbol{Z}^T\boldsymbol{Z}$ are bunched, that is, if there are sets of multiple eigenvalues. Specifically, if the eigenvalues are bunched into k distinct sets, then the conjugate-gradient method will converge in k iterations. Thus, significant improvements can be made if the problem can be transformed to one with bunched eigenvalues. Such a transformation leads to the so-called *preconditioned conjugate-gradient method*. In order to analyse the situation, let $\boldsymbol{C}$ be a $p \times p$ matrix that transforms the variables, such that

$$\mathbf{x}' = \boldsymbol{C}\mathbf{x}. \tag{8.1.5.5}$$

Then,

$$\mathbf{y}' - \boldsymbol{Z}\mathbf{x} = \mathbf{y}' - \boldsymbol{Z}\boldsymbol{C}^{-1}\mathbf{x}'. \tag{8.1.5.6}$$

Therefore, $\boldsymbol{C}$ should be such that the system $\boldsymbol{C}\mathbf{x} = \mathbf{x}'$ is easy to solve, and $(\boldsymbol{Z}\boldsymbol{C}^{-1})^T\boldsymbol{Z}\boldsymbol{C}^{-1}$ has bunched eigenvalues. The ideal choice would be $\boldsymbol{C} = \boldsymbol{R}$, where $\boldsymbol{R}$ is the upper triangular factor of the QR decomposition, since $\boldsymbol{Z}\boldsymbol{R}^{-1} = \boldsymbol{Q}_{\boldsymbol{Z}}$. $\boldsymbol{Q}_{\boldsymbol{Z}}^T\boldsymbol{Q}_{\boldsymbol{Z}} = \boldsymbol{I}$ has all of its eigenvalues equal to one, and, since $\boldsymbol{R}$ is triangular, the system is easy to solve. If $\boldsymbol{R}$ were known, however, the problem would already be exactly solved, so this is not a useful alternative. Unfortunately, no universal best choice seems to exist, but one approach is to choose a sparse approximation to $\boldsymbol{R}$ by ignoring rows that cause too much fill in or by making $\boldsymbol{R}$ a diagonal matrix whose elements are the Euclidean norms of the columns of $\boldsymbol{Z}$. Bear in mind that, in the nonlinear case, an expensive computation to choose $\boldsymbol{C}$ in the first iteration may work very well in subsequent iterations with no further expense. One should be aware of the trade off between the extra work per iteration of the preconditioned-conjugate gradient method *versus* the reduction in the number of iterations. This is especially important in nonlinear problems.

The solution of large, least-squares problems is currently an active area of research, and we have certainly not given an exhaustive list of methods in this chapter. The choice of method or approach for any particular problem is dependent on many conditions. Some of these are:

(1) The size of the problem. Clearly, as computer memories continue to grow, the boundary between small and large problems also grows. Nevertheless, even if a problem can fit into memory, its sparsity structure may be exploited in order to obtain a more efficient algorithm.

(2) The number of times the problem (or similar ones) will be solved. If it is a one-shot problem (a rare occurrence), then one is usually most strongly influenced by easy-to-use, existing software. Exceptions, of course, exist where even a single solution of the problem requires extreme care.

(3) The expense of evaluating the function. With a complicated, nonlinear function like the structure-factor formula, the computational effort to determine the values of the function and its derivatives usually greatly exceeds that required to solve the linearized problem. Therefore, a full Gauss–Newton, trust-region, or quasi-Newton method may be warranted.

(4) Other structure in the problem. Rarely does a problem have a random sparsity pattern. Non-zero values usually occur in blocks or in some regular pattern for which special decomposition methods can be devised.

(5) The machine on which the problem is to be solved. We have said nothing about the existing vector and parallel processors. Suffice it to say that the most efficient procedure for a serial machine may not be the right algorithm for one of these novel machines. Appropriate numerical methods for such architectures are also being actively investigated.

8.1.6. Orthogonal distance regression

It is often useful to consider the data for a least-squares problem to be in the form (t_i, y_i), $i = 1, \ldots, n$, where the t_i are considered to be the *independent* variables and the y_i the dependent variables. The implicit assumption in ordinary least squares is that the independent variables are known exactly. It sometimes occurs, however, that these independent variables also have errors associated with them that are significant with respect to the errors in the observations y_i. In such cases, referred to as 'errors in variables' or 'measurement error models', the ordinary least-squares methodology is not appropriate and its use may give misleading results (see Fuller, 1987).

Let us define $\hat{M}(t_i, \mathbf{x})$ to be the model functions that predict the y_i. Observe that ordinary least squares minimizes the sum of the squares of the vertical distances from the observed points y_i to the curve $\hat{M}(t, \mathbf{x})$. If t_i has an error δ_i, and these errors are normally distributed, then the maximum-likelihood estimate of the parameters is found by minimizing the sum of the squares of the *weighted orthogonal distances* from the point y_i to the curve $\hat{M}(t, \mathbf{x})$. More precisely, the optimization problem to be solved is given by

$$\min_{\mathbf{x},\delta} \sum_{i=1}^{n} \left\{ \left[y_i - \hat{M}(t_i + \delta_i, \mathbf{x})\right]^T \mathbf{W}_{\mathbf{y}} \left[y_i - \hat{M}(t_i + \delta_i, \mathbf{x})\right] + \delta_i^T \mathbf{W}_{\mathbf{t}} \delta_i \right\}, \tag{8.1.6.1}$$

where $\mathbf{W}_{\mathbf{y}}$ and $\mathbf{W}_{\mathbf{t}}$ are appropriately chosen weights. Problem (8.1.6.1) is called the *orthogonal distance regression* (ODR) problem. Problem (8.1.6.1) can be solved as a least-squares problem in the combined variables $\mathbf{x}, \delta$ by the methods given above. This, however, is quite inefficient, since such a procedure would not exploit the special structure of the ODR problem. Few algorithms that exploit this structure exist; one has been given by Boggs, Byrd & Schnabel (1987), and the software, called *ODRPACK*, is by Boggs, Byrd, Donaldson & Schnabel (1989). The algorithm is based on the trust-region (Levenberg–Marquardt) method described above, but it exploits the special structure of (8.1.6.1) so that the cost of each iteration is no more expensive than the cost of a similar iteration of the corresponding

references