International Tables for Crystallography (2006). Vol. C, Section 8.1.6, pp. 687–688.

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

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

for which

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

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,

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

Therefore, $\boldsymbol{C}$ should be such that the system $\boldsymbol{C}\mathbf{x} = \mathbf{x}'$ is easy to solve, and $(\boldsymbol{ZC}^{-1})^T\boldsymbol{ZC}^{-1}$ 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 $\boldsymbol{ZR}^{-1} = \boldsymbol{Q}_{\boldsymbol{Z}}$. $\boldsymbol{Q}_{\boldsymbol{Z}}^T\boldsymbol{Q}_{\boldsymbol{Z}} = \boldsymbol{I}$ 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

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

where $\mathbf{W_y}$ and $\mathbf{W_t}$ 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 ordinary least-squares problems. For a discussion of some of the statistical properties of the resulting estimates, including a procedure for the computation of the variance–covariance matrix, see Boggs & Rogers (1990).

8.1.7. Software for least-squares calculations

Giving even general recommendations on software is a difficult task for several reasons. Clearly, the selection of methods discussed in earlier sections contains implicitly some recommendations for approaches. Among the reasons for avoiding specifics are the following:

(1) Assessing differences in performance among various codes requires a detailed knowledge of the criteria the developer of a particular code used in creating it. A program written to emphasize speed on a certain class of problems on a certain machine is impossible to compare directly with a program written to be very reliable on a wide class of problems and portable over a wide range of machines. Other measures, including ease of maintenance and modification and ease of use, and other design criteria, such as interactive *versus* batch, stand alone *versus* user-callable, automatic computation of related statistics *versus* no statistics, and so forth, make the selection of software analogous to the selection of a car.

(2) Choosing software requires detailed knowledge of the needs of the user and the resources available to the user. Considerations such as problem size, machine size, machine architecture and financial resources all enter into the decision of which software to obtain.

(3) A software recommendation made on the basis of today's knowledge ignores the fact that algorithms continue to be invented, and old algorithm continue to be rethought in the light of new developments and new machine architectures. For example, when vector processors first appeared, algorithms for sparse-matrix calculations were very poor at exploiting this capability, and it was thought that these new machines were simply not appropriate for such calculations. Now, however, recent methods for sparse matrices have achieved a high degree of vectorization. For another example, early programs for crystallographic, full-matrix, least-squares refinement spent a large fraction of the time building the normal-equations matrix. The matrix was then inverted using a procedure called Gaussian elimination, which does not exploit the fact that the matrix is positive definite. Some programs were later converted to use Cholesky decomposition, which is at least twice as fast, but many were not because the inversion process took a small fraction of the total time. Linear algebra, however, is readily adaptable to vector and parallel machines, and procedures such as QR factorization are extremely fast, while the calculation of structure factors, with its repeated evaluations of trigonometric functions, becomes the time-controlling step.

The general recommendation is to analyse carefully the needs and resources in terms of these considerations, and to seek expert assistance whenever possible. *As much as possible, avoid the temptation to write your own codes*. Despite the fact that the quality of existing software is far from uniformly high, the benefits of utilizing high-quality software generally far outweigh the costs of finding, obtaining, and installing it.

Sources of information on software have improved significantly in the past several years. Nevertheless, the task of identifying software in terms of problems that can be solved; organizing, maintaining and updating such a list; and informing the user community still remains formidable.

A current, problem-oriented system that includes both a problem classification scheme and a network tool for obtaining documentation and source code (for software in the public domain) is the *Guide to Available Mathematical Software* (GAMS). This system is maintained by the National Institute of Standards and Technology (NIST) and is continually being updated as new material is received. It gives references to software in several software repositories; the URL is http://math.nist.gov/gams.

references