```
    for (i=1;i<=n;i++) f[i]=f1[i]-f2[i];
    free_vector(y,1,nvar);
    free_vector(f2,1,nvar);
    free_vector(f1,1,nvar);
}
```

There are boundary value problems where even shooting to a fitting point fails — the integration interval has to be partitioned by several fitting points with the solution being matched at each such point. For more details see [1].

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§7.3.5–7.3.6. [1]

17.3 Relaxation Methods

In *relaxation methods* we replace ODEs by approximate *finite-difference equations* (FDEs) on a grid or mesh of points that spans the domain of interest. As a typical example, we could replace a general first-order differential equation

$$\frac{dy}{dx} = g(x, y) \tag{17.3.1}$$

with an algebraic equation relating function values at two points $k, k-1$:

$$y_k - y_{k-1} - (x_k - x_{k-1})\, g\left[\tfrac{1}{2}(x_k + x_{k-1}), \tfrac{1}{2}(y_k + y_{k-1})\right] = 0 \tag{17.3.2}$$

The form of the FDE in (17.3.2) illustrates the idea, but not uniquely: There are many ways to turn the ODE into an FDE. When the problem involves N coupled first-order ODEs represented by FDEs on a mesh of M points, a solution consists of values for N dependent functions given at each of the M mesh points, or $N \times M$ variables in all. The relaxation method determines the solution by starting with a guess and improving it, iteratively. As the iterations improve the solution, the result is said to *relax* to the true solution.

While several iteration schemes are possible, for most problems our old standby, multidimensional Newton's method, works well. The method produces a matrix equation that must be solved, but the matrix takes a special, "block diagonal" form, that allows it to be inverted far more economically both in time and storage than would be possible for a general matrix of size $(MN) \times (MN)$. Since MN can easily be several thousand, this is crucial for the feasibility of the method.

Our implementation couples at most pairs of points, as in equation (17.3.2). More points can be coupled, but then the method becomes more complex. We will provide enough background so that you can write a more general scheme if you have the patience to do so.

Let us develop a general set of algebraic equations that represent the ODEs by FDEs. The ODE problem is exactly identical to that expressed in equations (17.0.1)–(17.0.3) where we had N coupled first-order equations that satisfy n_1 boundary conditions at x_1 and $n_2 = N - n_1$ boundary conditions at x_2. We first define a mesh or grid by a set of $k = 1, 2, ..., M$ points at which we supply values for the independent variable x_k. In particular, x_1 is the initial boundary, and x_M is the final boundary. We use the notation $\mathbf{y}_k$ to refer to the entire set of dependent variables $y_1, y_2, \dots, y_N$ at point x_k. At an arbitrary point k in the middle of the mesh, we approximate the set of N first-order ODEs by algebraic relations of the form

$$0 = \mathbf{E}_k \equiv \mathbf{y}_k - \mathbf{y}_{k-1} - (x_k - x_{k-1})\mathbf{g}_k(x_k, x_{k-1}, \mathbf{y}_k, \mathbf{y}_{k-1}), \quad k = 2, 3, \dots, M \tag{17.3.3}$$

The notation signifies that $\mathbf{g}_k$ can be evaluated using information from both points $k, k-1$. The FDEs labeled by $\mathbf{E}_k$ provide N equations coupling $2N$ variables at points $k, k-1$. There are $M-1$ points, $k = 2, 3, \dots, M$, at which difference equations of the form (17.3.3) apply. Thus the FDEs provide a total of $(M-1)N$ equations for the MN unknowns. The remaining N equations come from the boundary conditions.

At the first boundary we have

$$0 = \mathbf{E}_1 \equiv \mathbf{B}(x_1, \mathbf{y}_1) \tag{17.3.4}$$

while at the second boundary

$$0 = \mathbf{E}_{M+1} \equiv \mathbf{C}(x_M, \mathbf{y}_M) \tag{17.3.5}$$

The vectors $\mathbf{E}_1$ and $\mathbf{B}$ have only n_1 nonzero components, corresponding to the n_1 boundary conditions at x_1. It will turn out to be useful to take these nonzero components to be the *last* n_1 components. In other words, $E_{j,1} \neq 0$ only for $j = n_2 + 1, n_2 + 2, \dots, N$. At the other boundary, only the first n_2 components of $\mathbf{E}_{M+1}$ and $\mathbf{C}$ are nonzero: $E_{j,M+1} \neq 0$ only for $j = 1, 2, \dots, n_2$.

The "solution" of the FDE problem in (17.3.3)–(17.3.5) consists of a set of variables $y_{j,k}$, the values of the N variables y_j at the M points x_k. The algorithm we describe below requires an initial guess for the $y_{j,k}$. We then determine increments $\Delta y_{j,k}$ such that $y_{j,k} + \Delta y_{j,k}$ is an improved approximation to the solution.

Equations for the increments are developed by expanding the FDEs in first-order Taylor series with respect to small changes $\Delta\mathbf{y}_k$. At an interior point, $k = 2, 3, \dots, M$ this gives:

$$\begin{aligned} \mathbf{E}_k(\mathbf{y}_k + \Delta\mathbf{y}_k, \mathbf{y}_{k-1} + \Delta\mathbf{y}_{k-1}) &\approx \mathbf{E}_k(\mathbf{y}_k, \mathbf{y}_{k-1}) \\ &+ \sum_{n=1}^{N} \frac{\partial \mathbf{E}_k}{\partial y_{n,k-1}} \Delta y_{n,k-1} + \sum_{n=1}^{N} \frac{\partial \mathbf{E}_k}{\partial y_{n,k}} \Delta y_{n,k} \end{aligned} \tag{17.3.6}$$

For a solution we want the updated value $\mathbf{E}(\mathbf{y} + \Delta\mathbf{y})$ to be zero, so the general set of equations at an interior point can be written in matrix form as

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,k-1} + \sum_{n=N+1}^{2N} S_{j,n} \Delta y_{n-N,k} = -E_{j,k}, \quad j = 1, 2, \dots, N \tag{17.3.7}$$

where

$$S_{j,n} = \frac{\partial E_{j,k}}{\partial y_{n,k-1}}, \quad S_{j,n+N} = \frac{\partial E_{j,k}}{\partial y_{n,k}}, \quad n = 1, 2, \dots, N \tag{17.3.8}$$

The quantity $S_{j,n}$ is an $N \times 2N$ matrix at each point k. Each interior point thus supplies a block of N equations coupling $2N$ corrections to the solution variables at the points $k, k-1$.

Similarly, the algebraic relations at the boundaries can be expanded in a first-order Taylor series for increments that improve the solution. Since $\mathbf{E}_1$ depends only on $\mathbf{y}_1$, we find at the first boundary:

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,1} = -E_{j,1}, \quad j = n_2 + 1, n_2 + 2, \dots, N \tag{17.3.9}$$

where

$$S_{j,n} = \frac{\partial E_{j,1}}{\partial y_{n,1}}, \quad n = 1, 2, \dots, N \tag{17.3.10}$$

At the second boundary,

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,M} = -E_{j,M+1}, \quad j = 1, 2, \dots, n_2 \tag{17.3.11}$$

where

$$S_{j,n} = \frac{\partial E_{j,M+1}}{\partial y_{n,M}}, \quad n = 1, 2, \ldots, N \tag{17.3.12}$$

We thus have in equations (17.3.7)–(17.3.12) a set of linear equations to be solved for the corrections $\Delta\mathbf{y}$, iterating until the corrections are sufficiently small. The equations have a special structure, because each $S_{j,n}$ couples only points $k, k-1$. Figure 17.3.1 illustrates the typical structure of the complete matrix equation for the case of 5 variables and 4 mesh points, with 3 boundary conditions at the first boundary and 2 at the second. The 3×5 block of nonzero entries in the top left-hand corner of the matrix comes from the boundary condition $S_{j,n}$ at point $k = 1$. The next three 5×10 blocks are the $S_{j,n}$ at the interior points, coupling variables at mesh points (2,1), (3,2), and (4,3). Finally we have the block corresponding to the second boundary condition.

We can solve equations (17.3.7)–(17.3.12) for the increments $\Delta\mathbf{y}$ using a form of Gaussian elimination that exploits the special structure of the matrix to minimize the total number of operations, and that minimizes storage of matrix coefficients by packing the elements in a special blocked structure. (You might wish to review Chapter 2, especially §2.2, if you are unfamiliar with the steps involved in Gaussian elimination.) Recall that Gaussian elimination consists of manipulating the equations by elementary operations such as dividing rows of coefficients by a common factor to produce unity in diagonal elements, and adding appropriate multiples of other rows to produce zeros below the diagonal. Here we take advantage of the block structure by performing a bit more reduction than in pure Gaussian elimination, so that the storage of coefficients is minimized. Figure 17.3.2 shows the form that we wish to achieve by elimination, just prior to the backsubstitution step. Only a small subset of the reduced $MN \times MN$ matrix elements needs to be stored as the elimination progresses. Once the matrix elements reach the stage in Figure 17.3.2, the solution follows quickly by a backsubstitution procedure.

Furthermore, the entire procedure, except the backsubstitution step, operates only on one block of the matrix at a time. The procedure contains four types of operations: (1) partial reduction to zero of certain elements of a block using results from a previous step, (2) elimination of the square structure of the remaining block elements such that the square section contains unity along the diagonal, and zero in off-diagonal elements, (3) storage of the remaining nonzero coefficients for use in later steps, and (4) backsubstitution. We illustrate the steps schematically by figures.

Consider the block of equations describing corrections available from the initial boundary conditions. We have n_1 equations for N unknown corrections. We wish to transform the first block so that its left-hand $n_1 \times n_1$ square section becomes unity along the diagonal, and zero in off-diagonal elements. Figure 17.3.3 shows the original and final form of the first block of the matrix. In the figure we designate matrix elements that are subject to diagonalization by "D", and elements that will be altered by "A"; in the final block, elements that are stored are labeled by "S". We get from start to finish by selecting in turn n_1 "pivot" elements from among the first n_1 columns, normalizing the pivot row so that the value of the "pivot" element is unity, and adding appropriate multiples of this row to the remaining rows so that they contain zeros in the pivot column. In its final form, the reduced block expresses values for the corrections to the first n_1 variables at mesh point 1 in terms of values for the remaining n_2 unknown corrections at point 1, i.e., we now know what the first n_1 elements are in terms of the remaining n_2 elements. We store only the final set of n_2 nonzero columns from the initial block, plus the column for the altered right-hand side of the matrix equation.

We must emphasize here an important detail of the method. To exploit the reduced storage allowed by operating on blocks, it is essential that the ordering of columns in the `s` matrix of derivatives be such that pivot elements can be found among the first n_1 rows of the matrix. This means that the n_1 boundary conditions at the first point must contain some dependence on the first `j=1,2,...,`n_1 dependent variables, `y[j][1]`. If not, then the original square $n_1 \times n_1$ subsection of the first block will appear to be singular, and the method will fail. Alternatively, we would have to allow the search for pivot elements to involve all N columns of the block, and this would require column swapping and far more bookkeeping. The code provides a simple method of reordering the variables, i.e., the columns of the `s` matrix, so that this can be done easily. End of important detail.

```
X X X X X                                V   B
X X X X X                                V   B
X X X X X                                V   B
X X X X X X X X X X                      V   B
X X X X X X X X X X                      V   B
X X X X X X X X X X                      V   B
X X X X X X X X X X                      V   B
X X X X X X X X X X                      V   B
          X X X X X X X X X X            V   B
          X X X X X X X X X X            V   B
          X X X X X X X X X X            V   B
          X X X X X X X X X X            V   B
          X X X X X X X X X X            V   B
                    X X X X X X X X X X  V   B
                    X X X X X X X X X X  V   B
                    X X X X X X X X X X  V   B
                    X X X X X X X X X X  V   B
                    X X X X X X X X X X  V   B
                              X X X X X  V   B
                              X X X X X  V   B
```

Figure 17.3.1. Matrix structure of a set of linear finite-difference equations (FDEs) with boundary conditions imposed at both endpoints. Here X represents a coefficient of the FDEs, V represents a component of the unknown solution vector, and B is a component of the known right-hand side. Empty spaces represent zeros. The matrix equation is to be solved by a special form of Gaussian elimination. (See text for details.)

```
1     X X                                V   B
  1   X X                                V   B
    1 X X                                V   B
      1         X X                      V   B
        1       X X                      V   B
          1     X X                      V   B
            1   X X                      V   B
              1 X X                      V   B
                1         X X            V   B
                  1       X X            V   B
                    1     X X            V   B
                      1   X X            V   B
                        1 X X            V   B
                          1         X X  V   B
                            1       X X  V   B
                              1     X X  V   B
                                1   X X  V   B
                                  1 X X  V   B
                                    1    V   B
                                      1  V   B
```

Figure 17.3.2. Target structure of the Gaussian elimination. Once the matrix of Figure 17.3.1 has been reduced to this form, the solution follows quickly by backsubstitution.

$$
\begin{array}{lccccccc}
\text{(a)} & D & D & D & A & A & V & A \\
 & D & D & D & A & A & V & A \\
 & D & D & D & A & A & V & A \\
 & & & & & & & \\
\text{(b)} & 1 & 0 & 0 & S & S & V & S \\
 & 0 & 1 & 0 & S & S & V & S \\
 & 0 & 0 & 1 & S & S & V & S
\end{array}
$$

Figure 17.3.3. Reduction process for the first (upper left) block of the matrix in Figure 17.3.1. (a) Original form of the block, (b) final form. (See text for explanation.)

$$
\begin{array}{lcccccccccccc}
\text{(a)} & 1 & 0 & 0 & S & S & & & & & & V & S \\
 & 0 & 1 & 0 & S & S & & & & & & V & S \\
 & 0 & 0 & 1 & S & S & & & & & & V & S \\
 & Z & Z & Z & D & D & D & D & D & A & A & V & A \\
 & Z & Z & Z & D & D & D & D & D & A & A & V & A \\
 & Z & Z & Z & D & D & D & D & D & A & A & V & A \\
 & Z & Z & Z & D & D & D & D & D & A & A & V & A \\
 & Z & Z & Z & D & D & D & D & D & A & A & V & A \\
 & & & & & & & & & & & & \\
\text{(b)} & 1 & 0 & 0 & S & S & & & & & & V & S \\
 & 0 & 1 & 0 & S & S & & & & & & V & S \\
 & 0 & 0 & 1 & S & S & & & & & & V & S \\
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & S & S & V & S
\end{array}
$$

Figure 17.3.4. Reduction process for intermediate blocks of the matrix in Figure 17.3.1. (a) Original form, (b) final form. (See text for explanation.)

$$
\begin{array}{lcccccccccccc}
\text{(a)} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & S & S & V & S \\
 & & & & & & Z & Z & Z & D & D & V & A \\
 & & & & & & Z & Z & Z & D & D & V & A \\
 & & & & & & & & & & & & \\
\text{(b)} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & S & S & V & S \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & S & S & V & S \\
 & & & & & & 0 & 0 & 0 & 1 & 0 & V & S \\
 & & & & & & 0 & 0 & 0 & 0 & 1 & V & S
\end{array}
$$

Figure 17.3.5. Reduction process for the last (lower right) block of the matrix in Figure 17.3.1. (a) Original form, (b) final form. (See text for explanation.)

Next consider the block of N equations representing the FDEs that describe the relation between the $2N$ corrections at points 2 and 1. The elements of that block, together with results from the previous step, are illustrated in Figure 17.3.4. Note that by adding suitable multiples of rows from the first block we can reduce to zero the first n_1 columns of the block (labeled by "Z"), and, to do so, we will need to alter only the columns from $n_1 + 1$ to N and the vector element on the right-hand side. Of the remaining columns we can diagonalize a square subsection of $N \times N$ elements, labeled by "D" in the figure. In the process we alter the final set of $n_2 + 1$ columns, denoted "A" in the figure. The second half of the figure shows the block when we finish operating on it, with the stored $(n_2 + 1) \times N$ elements labeled by "S."

If we operate on the next set of equations corresponding to the FDEs coupling corrections at points 3 and 2, we see that the state of available results and new equations exactly reproduces the situation described in the previous paragraph. Thus, we can carry out those steps again for each block in turn through block M. Finally on block $M + 1$ we encounter the remaining boundary conditions.

Figure 17.3.5 shows the final block of n_2 FDEs relating the N corrections for variables at mesh point M, together with the result of reducing the previous block. Again, we can first use the prior results to zero the first n_1 columns of the block. Now, when we diagonalize the remaining square section, we strike gold: We get values for the final n_2 corrections at mesh point M.

With the final block reduced, the matrix has the desired form shown previously in Figure 17.3.2, and the matrix is ripe for backsubstitution. Starting with the bottom row and working up towards the top, at each stage we can simply determine one unknown correction in terms of known quantities.

The function `solvde` organizes the steps described above. The principal procedures used in the algorithm are performed by functions called internally by `solvde`. The function `red` eliminates leading columns of the `s` matrix using results from prior blocks. `pinvs` diagonalizes the square subsection of `s` and stores unreduced coefficients. `bksub` carries out the backsubstitution step. The user of `solvde` must understand the calling arguments, as described below, and supply a function `difeq`, called by `solvde`, that evaluates the `s` matrix for each block.

Most of the arguments in the call to `solvde` have already been described, but some require discussion. Array `y[j][k]` contains the initial guess for the solution, with `j` labeling the dependent variables at mesh points `k`. The problem involves `ne` FDEs spanning points `k=1,..., m`. `nb` boundary conditions apply at the first point `k=1`. The array `indexv[j]` establishes the correspondence between columns of the `s` matrix, equations (17.3.8), (17.3.10), and (17.3.12), and the dependent variables. As described above it is essential that the `nb` boundary conditions at `k=1` involve the dependent variables referenced by the first `nb` columns of the `s` matrix. Thus, columns `j` of the `s` matrix can be ordered by the user in `difeq` to refer to derivatives with respect to the dependent variable `indexv[j]`.

The function only attempts `itmax` correction cycles before returning, even if the solution has not converged. The parameters `conv, slowc, scalv` relate to convergence. Each inversion of the matrix produces corrections for `ne` variables at `m` mesh points. We want these to become vanishingly small as the iterations proceed, but we must define a measure for the size of corrections. This error "norm" is very problem specific, so the user might wish to rewrite this section of the code as appropriate. In the program below we compute a value for the average correction `err` by summing the absolute value of all corrections, weighted by a scale factor appropriate to each type of variable:

$$\mathtt{err} = \frac{1}{\mathtt{m} \times \mathtt{ne}} \sum_{\mathtt{k=1}}^{\mathtt{m}} \sum_{\mathtt{j=1}}^{\mathtt{ne}} \frac{|\Delta Y\mathtt{[j][k]}|}{\mathtt{scalv[j]}} \tag{17.3.13}$$

When `err`$\leq$`conv`, the method has converged. Note that the user gets to supply an array `scalv` which measures the typical size of each variable.

Obviously, if `err` is large, we are far from a solution, and perhaps it is a bad idea to believe that the corrections generated from a first-order Taylor series are accurate. The number `slowc` modulates application of corrections. After each iteration we apply only a fraction of the corrections found by matrix inversion:

$$Y\texttt{[j][k]} \to Y\texttt{[j][k]} + \frac{\texttt{slowc}}{\max(\texttt{slowc},\texttt{err})}\Delta Y\texttt{[j][k]} \tag{17.3.14}$$

Thus, when err$>$slowc only a fraction of the corrections are used, but when err$\leq$slowc the entire correction gets applied.

The call statement also supplies solvde with the array y[1..nyj][1..nyk] containing the initial trial solution, and workspace arrays c[1..ne][1..ne-nb+1][1..m+1], s[1..ne][1..2*ne+1]. The array c is the blockbuster: It stores the unreduced elements of the matrix built up for the backsubstitution step. If there are m mesh points, then there will be m+1 blocks, each requiring ne rows and ne-nb+1 columns. Although large, this is small compared with (ne$\times$m)^2 elements required for the whole matrix if we did not break it into blocks.

We now describe the workings of the user-supplied function difeq. The synopsis of the function is

```
void difeq(int k, int k1, int k2, int jsf, int is1, int isf,
    int indexv[], int ne, float **s, float **y);
```

The only information passed from difeq to solvde is the matrix of derivatives s[1..ne][1..2*ne+1]; all other arguments are input to difeq and should not be altered. k indicates the current mesh point, or block number. k1,k2 label the first and last point in the mesh. If k=k1 or k$>$k2, the block involves the boundary conditions at the first or final points; otherwise the block acts on FDEs coupling variables at points k-1, k.

The convention on storing information into the array s[i][j] follows that used in equations (17.3.8), (17.3.10), and (17.3.12): Rows i label equations, columns j refer to derivatives with respect to dependent variables in the solution. Recall that each equation will depend on the ne dependent variables at either one or two points. Thus, j runs from 1 to either ne or 2*ne. The column ordering for dependent variables at each point must agree with the list supplied in indexv[j]. Thus, for a block not at a boundary, the first column multiplies ΔY(l=indexv[1],k-1), and the column ne+1 multiplies ΔY(l=indexv[1],k). is1,isf give the numbers of the starting and final *rows* that need to be filled in the s matrix for this block. jsf labels the column in which the difference equations $E_{j,k}$ of equations (17.3.3)–(17.3.5) are stored. Thus, $-$s[i][jsf] is the vector on the right-hand side of the matrix. The reason for the minus sign is that difeq supplies the actual difference equation, $E_{j,k}$, not its negative. Note that solvde supplies a value for jsf such that the difference equation is put in the column *just after* all derivatives in the s matrix. Thus, difeq expects to find values entered into s[i][j] for rows is1 $\leq$ i $\leq$ isf and 1 $\leq$ j $\leq$ jsf.

Finally, s[1..nsi][1..nsj] and y[1..nyj][1..nyk] supply difeq with storage for s and the solution variables y for this iteration. An example of how to use this routine is given in the next section.

Many ideas in the following code are due to Eggleton [1].

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"

void solvde(int itmax, float conv, float slowc, float scalv[], int indexv[],
    int ne, int nb, int m, float **y, float ***c, float **s)
```

Driver routine for solution of two point boundary value problems by relaxation. itmax is the maximum number of iterations. conv is the convergence criterion (see text). slowc controls the fraction of corrections actually used after each iteration. scalv[1..ne] contains typical sizes for each dependent variable, used to weight errors. indexv[1..ne] lists the column ordering of variables used to construct the matrix s[1..ne][1..2*ne+1] of derivatives. (The nb boundary conditions at the first mesh point must contain some dependence on the first nb variables listed in indexv.) The problem involves ne equations for ne adjustable dependent variables at each point. At the first mesh point there are nb boundary conditions. There are a total of m mesh points. y[1..ne][1..m] is the two-dimensional array that contains the initial guess for all the dependent variables at each mesh point. On each iteration, it is updated by the calculated correction. The arrays c[1..ne][1..ne-nb+1][1..m+1] and s supply dummy storage used by the relaxation code.

```
{
    void bksub(int ne, int nb, int jf, int k1, int k2, float ***c);
    void difeq(int k, int k1, int k2, int jsf, int is1, int isf,
        int indexv[], int ne, float **s, float **y);
    void pinvs(int ie1, int ie2, int je1, int jsf, int jc1, int k,
        float ***c, float **s);
    void red(int iz1, int iz2, int jz1, int jz2, int jm1, int jm2, int jmf,
        int ic1, int jc1, int jcf, int kc, float ***c, float **s);
    int ic1,ic2,ic3,ic4,it,j,j1,j2,j3,j4,j5,j6,j7,j8,j9;
    int jc1,jcf,jv,k,k1,k2,km,kp,nvars,*kmax;
    float err,errj,fac,vmax,vz,*ermax;

    kmax=ivector(1,ne);
    ermax=vector(1,ne);
    k1=1;                                   Set up row and column markers.
    k2=m;
    nvars=ne*m;
    j1=1;
    j2=nb;
    j3=nb+1;
    j4=ne;
    j5=j4+j1;
    j6=j4+j2;
    j7=j4+j3;
    j8=j4+j4;
    j9=j8+j1;
    ic1=1;
    ic2=ne-nb;
    ic3=ic2+1;
    ic4=ne;
    jc1=1;
    jcf=ic3;
    for (it=1;it<=itmax;it++) {             Primary iteration loop.
        k=k1;                               Boundary conditions at first point.
        difeq(k,k1,k2,j9,ic3,ic4,indexv,ne,s,y);
        pinvs(ic3,ic4,j5,j9,jc1,k1,c,s);
        for (k=k1+1;k<=k2;k++) {            Finite difference equations at all point pairs.
            kp=k-1;
            difeq(k,k1,k2,j9,ic1,ic4,indexv,ne,s,y);
            red(ic1,ic4,j1,j2,j3,j4,j9,ic3,jc1,jcf,kp,c,s);
            pinvs(ic1,ic4,j3,j9,jc1,k,c,s);
        }
        k=k2+1;                             Final boundary conditions.
        difeq(k,k1,k2,j9,ic1,ic2,indexv,ne,s,y);
        red(ic1,ic2,j5,j6,j7,j8,j9,ic3,jc1,jcf,k2,c,s);
        pinvs(ic1,ic2,j7,j9,jcf,k2+1,c,s);
        bksub(ne,nb,jcf,k1,k2,c);           Backsubstitution.
        err=0.0;
        for (j=1;j<=ne;j++) {               Convergence check, accumulate average er-
            jv=indexv[j];                       ror.
            errj=vmax=0.0;
            km=0;
            for (k=k1;k<=k2;k++) {          Find point with largest error, for each de-
                vz=fabs(c[jv][1][k]);           pendent variable.
                if (vz > vmax) {
                    vmax=vz;
                    km=k;
                }
                errj += vz;
            }
            err += errj/scalv[j];           Note weighting for each dependent variable.
            ermax[j]=c[jv][1][km]/scalv[j];
```

```
            kmax[j]=km;
        }
        err /= nvars;
        fac=(err > slowc ? slowc/err : 1.0);
        Reduce correction applied when error is large.
        for (j=1;j<=ne;j++) {                   Apply corrections.
            jv=indexv[j];
            for (k=k1;k<=k2;k++)
                y[j][k] -= fac*c[jv][1][k];
        }
        printf("\n%8s %9s %9s\n","Iter.","Error","FAC");     Summary of corrections
        printf("%6d %12.6f %11.6f\n",it,err,fac);                  for this step.
        if (err < conv) {                       Point with largest error for each variable can
            free_vector(ermax,1,ne);                be monitored by writing out kmax and
            free_ivector(kmax,1,ne);                ermax.
            return;
        }
    }
    nrerror("Too many iterations in solvde");               Convergence failed.
}
```

```
void bksub(int ne, int nb, int jf, int k1, int k2, float ***c)
Backsubstitution, used internally by solvde.
{
    int nbf,im,kp,k,j,i;
    float xx;

    nbf=ne-nb;
    im=1;
    for (k=k2;k>=k1;k--) {              Use recurrence relations to eliminate remaining de-
        if (k == k1) im=nbf+1;              pendences.
        kp=k+1;
        for (j=1;j<=nbf;j++) {
            xx=c[j][jf][kp];
            for (i=im;i<=ne;i++)
                c[i][jf][k] -= c[i][j][k]*xx;
        }
    }
    for (k=k1;k<=k2;k++) {              Reorder corrections to be in column 1.
        kp=k+1;
        for (i=1;i<=nb;i++) c[i][1][k]=c[i+nbf][jf][k];
        for (i=1;i<=nbf;i++) c[i+nb][1][k]=c[i][jf][kp];
    }
}
```

```
#include <math.h>
#include "nrutil.h"

void pinvs(int ie1, int ie2, int je1, int jsf, int jc1, int k, float ***c,
    float **s)
Diagonalize the square subsection of the s matrix, and store the recursion coefficients in c;
used internally by solvde.
{
    int js1,jpiv,jp,je2,jcoff,j,irow,ipiv,id,icoff,i,*indxr;
    float pivinv,piv,dum,big,*pscl;

    indxr=ivector(ie1,ie2);
    pscl=vector(ie1,ie2);
    je2=je1+ie2-ie1;
    js1=je2+1;
```

```
    for (i=ie1;i<=ie2;i++) {            Implicit pivoting, as in §2.1.
        big=0.0;
        for (j=je1;j<=je2;j++)
            if (fabs(s[i][j]) > big) big=fabs(s[i][j]);
        if (big == 0.0) nrerror("Singular matrix - row all 0, in pinvs");
        pscl[i]=1.0/big;
        indxr[i]=0;
    }
    for (id=ie1;id<=ie2;id++) {
        piv=0.0;
        for (i=ie1;i<=ie2;i++) {        Find pivot element.
            if (indxr[i] == 0) {
                big=0.0;
                for (j=je1;j<=je2;j++) {
                    if (fabs(s[i][j]) > big) {
                        jp=j;
                        big=fabs(s[i][j]);
                    }
                }
                if (big*pscl[i] > piv) {
                    ipiv=i;
                    jpiv=jp;
                    piv=big*pscl[i];
                }
            }
        }
        if (s[ipiv][jpiv] == 0.0) nrerror("Singular matrix in routine pinvs");
        indxr[ipiv]=jpiv;               In place reduction. Save column ordering.
        pivinv=1.0/s[ipiv][jpiv];
        for (j=je1;j<=jsf;j++) s[ipiv][j] *= pivinv;      Normalize pivot row.
        s[ipiv][jpiv]=1.0;
        for (i=ie1;i<=ie2;i++) {        Reduce nonpivot elements in column.
            if (indxr[i] != jpiv) {
                if (s[i][jpiv]) {
                    dum=s[i][jpiv];
                    for (j=je1;j<=jsf;j++)
                        s[i][j] -= dum*s[ipiv][j];
                    s[i][jpiv]=0.0;
                }
            }
        }
    }
    jcoff=jc1-js1;                      Sort and store unreduced coefficients.
    icoff=ie1-je1;
    for (i=ie1;i<=ie2;i++) {
        irow=indxr[i]+icoff;
        for (j=js1;j<=jsf;j++) c[irow][j+jcoff][k]=s[i][j];
    }
    free_vector(pscl,ie1,ie2);
    free_ivector(indxr,ie1,ie2);
}
```

```
void red(int iz1, int iz2, int jz1, int jz2, int jm1, int jm2, int jmf,
    int ic1, int jc1, int jcf, int kc, float ***c, float **s)
```

Reduce columns `jz1-jz2` of the `s` matrix, using previous results as stored in the `c` matrix. Only columns `jm1-jm2,jmf` are affected by the prior results. `red` is used internally by `solvde`.

```
{
    int loff,l,j,ic,i;
    float vx;

    loff=jc1-jm1;
    ic=ic1;
```

```
    for (j=jz1;j<=jz2;j++) {              Loop over columns to be zeroed.
        for (l=jm1;l<=jm2;l++) {          Loop over columns altered.
            vx=c[ic][l+loff][kc];
            for (i=iz1;i<=iz2;i++) s[i][l] -= s[i][j]*vx;       Loop over rows.
        }
        vx=c[ic][jcf][kc];
        for (i=iz1;i<=iz2;i++) s[i][jmf] -= s[i][j]*vx;        Plus final element.
        ic += 1;
    }
}
```

"Algebraically Difficult" Sets of Differential Equations

Relaxation methods allow you to take advantage of an additional opportunity that, while not obvious, can speed up some calculations enormously. It is not necessary that the set of variables $y_{j,k}$ correspond exactly with the dependent variables of the original differential equations. They can be related to those variables through algebraic equations. Obviously, it is necessary only that the solution variables allow us to *evaluate* the functions $y, g, \mathbf{B}, \mathbf{C}$ that are used to construct the FDEs from the ODEs. In some problems g depends on functions of y that are known only implicitly, so that iterative solutions are necessary to evaluate functions in the ODEs. Often one can dispense with this "internal" nonlinear problem by defining a new set of variables from which both y, g and the boundary conditions can be obtained directly. A typical example occurs in physical problems where the equations require solution of a complex equation of state that can be expressed in more convenient terms using variables other than the original dependent variables in the ODE. While this approach is analogous to performing an *analytic* change of variables directly on the original ODEs, such an analytic transformation might be prohibitively complicated. The change of variables in the relaxation method is easy and requires no analytic manipulations.

CITED REFERENCES AND FURTHER READING:

Eggleton, P.P. 1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364. [1]

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

Kippenhan, R., Weigert, A., and Hofmeister, E. 1968, in *Methods in Computational Physics*, vol. 7 (New York: Academic Press), pp. 129ff.

17.4 A Worked Example: Spheroidal Harmonics

The best way to understand the algorithms of the previous sections is to see them employed to solve an actual problem. As a sample problem, we have selected the computation of spheroidal harmonics. (The more common name is spheroidal angle functions, but we prefer the explicit reminder of the kinship with spherical harmonics.) We will show how to find spheroidal harmonics, first by the method of relaxation (§17.3), and then by the methods of shooting (§17.1) and shooting to a fitting point (§17.2).

Spheroidal harmonics typically arise when certain partial differential equations are solved by separation of variables in spheroidal coordinates. They satisfy the following differential equation on the interval $-1 \le x \le 1$:

$$\frac{d}{dx}\left[(1-x^2)\frac{dS}{dx}\right] + \left(\lambda - c^2x^2 - \frac{m^2}{1-x^2}\right)S = 0 \tag{17.4.1}$$