Chapter 2. Solution of Linear Algebraic Equations

2.0 Introduction

A set of linear algebraic equations looks like this:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N &= b_3 \\ \cdots \qquad\qquad & \cdots \\ a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N &= b_M \end{aligned} \tag{2.0.1}$$

Here the N unknowns x_j, $j = 1, 2, \ldots, N$ are related by M equations. The coefficients a_{ij} with $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, N$ are known numbers, as are the *right-hand side* quantities b_i, $i = 1, 2, \ldots, M$.

Nonsingular versus Singular Sets of Equations

If $N = M$ then there are as many equations as unknowns, and there is a good chance of solving for a unique solution set of x_j's. Analytically, there can fail to be a unique solution if one or more of the M equations is a linear combination of the others, a condition called *row degeneracy*, or if all equations contain certain variables only in exactly the same linear combination, called *column degeneracy*. (For square matrices, a row degeneracy implies a column degeneracy, and vice versa.) A set of equations that is degenerate is called *singular*. We will consider singular matrices in some detail in §2.6.

Numerically, at least two additional things can go wrong:

- While not exact linear combinations of each other, some of the equations may be so close to linearly dependent that roundoff errors in the machine render them linearly dependent at some stage in the solution process. In this case your numerical procedure will fail, and it can tell you that it has failed.

- Accumulated roundoff errors in the solution process can swamp the true solution. This problem particularly emerges if N is too large. The numerical procedure does not fail algorithmically. However, it returns a set of x's that are wrong, as can be discovered by direct substitution back into the original equations. The closer a set of equations is to being singular, the more likely this is to happen, since increasingly close cancellations will occur during the solution. In fact, the preceding item can be viewed as the special case where the loss of significance is unfortunately total.

Much of the sophistication of complicated "linear equation-solving packages" is devoted to the detection and/or correction of these two pathologies. As you work with large linear sets of equations, you will develop a feeling for when such sophistication is needed. It is difficult to give any firm guidelines, since there is no such thing as a "typical" linear problem. But here is a rough idea: Linear sets with N as large as 20 or 50 can be routinely solved in single precision (32 bit floating representations) without resorting to sophisticated methods, *if* the equations are not close to singular. With double precision (60 or 64 bits), this number can readily be extended to N as large as several hundred, after which point the limiting factor is generally machine time, not accuracy.

Even larger linear sets, N in the thousands or greater, can be solved when the coefficients are sparse (that is, mostly zero), by methods that take advantage of the sparseness. We discuss this further in §2.7.

At the other end of the spectrum, one seems just as often to encounter linear problems which, by their underlying nature, are close to singular. In this case, you *might* need to resort to sophisticated methods even for the case of $N = 10$ (though rarely for $N = 5$). Singular value decomposition (§2.6) is a technique that can sometimes turn singular problems into nonsingular ones, in which case additional sophistication becomes unnecessary.

Matrices

Equation (2.0.1) can be written in matrix form as

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.0.2}$$

Here the raised dot denotes matrix multiplication, $\mathbf{A}$ is the matrix of coefficients, and $\mathbf{b}$ is the right-hand side written as a column vector,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ & \dots & & \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_M \end{bmatrix} \tag{2.0.3}$$

By convention, the first index on an element a_{ij} denotes its row, the second index its column. For most purposes you don't need to know how a matrix is stored in a computer's physical memory; you simply reference matrix elements by their two-dimensional addresses, e.g., a_{34} = `a[3][4]`. We have already seen, in §1.2, that this `C` notation can in fact hide a rather subtle and versatile physical storage scheme, "pointer to array of pointers to rows." You might wish to review that section

at this point. Occasionally it is useful to be able to peer through the veil, for example to pass a whole row `a[i][j], j=`$1, \ldots, N$ by the reference `a[i]`.

Tasks of Computational Linear Algebra

We will consider the following tasks as falling in the general purview of this chapter:

- Solution of the matrix equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for an unknown vector $\mathbf{x}$, where $\mathbf{A}$ is a square matrix of coefficients, raised dot denotes matrix multiplication, and $\mathbf{b}$ is a known right-hand side vector (§2.1–§2.10).
- Solution of more than one matrix equation $\mathbf{A} \cdot \mathbf{x}_j = \mathbf{b}_j$, for a set of vectors $\mathbf{x}_j$, $j = 1, 2, \ldots$, each corresponding to a different, known right-hand side vector $\mathbf{b}_j$. In this task the key simplification is that the matrix $\mathbf{A}$ is held constant, while the right-hand sides, the $\mathbf{b}$'s, are changed (§2.1–§2.10).
- Calculation of the matrix $\mathbf{A}^{-1}$ which is the matrix inverse of a square matrix $\mathbf{A}$, i.e., $\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{1}$, where $\mathbf{1}$ is the identity matrix (all zeros except for ones on the diagonal). This task is equivalent, for an $N \times N$ matrix $\mathbf{A}$, to the previous task with N different $\mathbf{b}_j$'s ($j = 1, 2, \ldots, N$), namely the unit vectors ($\mathbf{b}_j$ = all zero elements except for 1 in the jth component). The corresponding $\mathbf{x}$'s are then the columns of the matrix inverse of $\mathbf{A}$ (§2.1 and §2.3).
- Calculation of the determinant of a square matrix $\mathbf{A}$ (§2.3).

If $M < N$, or if $M = N$ but the equations are degenerate, then there are effectively fewer equations than unknowns. In this case there can be either no solution, or else more than one solution vector $\mathbf{x}$. In the latter event, the solution space consists of a particular solution $\mathbf{x}_p$ added to any linear combination of (typically) $N - M$ vectors (which are said to be in the nullspace of the matrix $\mathbf{A}$). The task of finding the solution space of $\mathbf{A}$ involves

- Singular value decomposition of a matrix $\mathbf{A}$.

This subject is treated in §2.6.

In the opposite case there are more equations than unknowns, $M > N$. When this occurs there is, in general, no solution vector $\mathbf{x}$ to equation (2.0.1), and the set of equations is said to be *overdetermined*. It happens frequently, however, that the best "compromise" solution is sought, the one that comes closest to satisfying all equations simultaneously. If closeness is defined in the least-squares sense, i.e., that the sum of the squares of the differences between the left- and right-hand sides of equation (2.0.1) be minimized, then the overdetermined linear problem reduces to a (usually) solvable linear problem, called the

- Linear least-squares problem.

The reduced set of equations to be solved can be written as the $N \times N$ set of equations

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = (\mathbf{A}^T \cdot \mathbf{b}) \tag{2.0.4}$$

where $\mathbf{A}^T$ denotes the transpose of the matrix $\mathbf{A}$. Equations (2.0.4) are called the *normal equations* of the linear least-squares problem. There is a close connection

between singular value decomposition and the linear least-squares problem, and the latter is also discussed in §2.6. You should be warned that direct solution of the normal equations (2.0.4) is not generally the best way to find least-squares solutions.

Some other topics in this chapter include

- Iterative improvement of a solution (§2.5)
- Various special forms: symmetric positive-definite (§2.9), tridiagonal (§2.4), band diagonal (§2.4), Toeplitz (§2.8), Vandermonde (§2.8), sparse (§2.7)
- Strassen's "fast matrix inversion" (§2.11).

Standard Subroutine Packages

We cannot hope, in this chapter or in this book, to tell you everything there is to know about the tasks that have been defined above. In many cases you will have no alternative but to use sophisticated black-box program packages. Several good ones are available, though not always in C. LINPACK was developed at Argonne National Laboratories and deserves particular mention because it is published, documented, and available for free use. A successor to LINPACK, LAPACK, is now becoming available. Packages available commercially (though not necessarily in C) include those in the IMSL and NAG libraries.

You should keep in mind that the sophisticated packages are designed with very large linear systems in mind. They therefore go to great effort to minimize not only the number of operations, but also the required storage. Routines for the various tasks are usually provided in several versions, corresponding to several possible simplifications in the form of the input coefficient matrix: symmetric, triangular, banded, positive definite, etc. If you have a large matrix in one of these forms, you should certainly take advantage of the increased efficiency provided by these different routines, and not just use the form provided for general matrices.

There is also a great watershed dividing routines that are *direct* (i.e., execute in a predictable number of operations) from routines that are *iterative* (i.e., attempt to converge to the desired answer in however many steps are necessary). Iterative methods become preferable when the battle against loss of significance is in danger of being lost, either due to large N or because the problem is close to singular. We will treat iterative methods only incompletely in this book, in §2.7 and in Chapters 18 and 19. These methods are important, but mostly beyond our scope. We will, however, discuss in detail a technique which is on the borderline between direct and iterative methods, namely the iterative improvement of a solution that has been obtained by direct methods (§2.5).

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press).

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 4.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.).

Coleman, T.F., and Van Loan, C. 1988, *Handbook for Matrix Computations* (Philadelphia: S.I.A.M.).

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag).

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 9.

2.1 Gauss-Jordan Elimination

For inverting a matrix, *Gauss-Jordan elimination* is about as efficient as any other method. For solving sets of linear equations, Gauss-Jordan elimination produces *both* the solution of the equations for one or more right-hand side vectors $\mathbf{b}$, and also the matrix inverse $\mathbf{A}^{-1}$. However, its principal weaknesses are (i) that it requires all the right-hand sides to be stored and manipulated at the same time, and (ii) that when the inverse matrix is *not* desired, Gauss-Jordan is three times slower than the best alternative technique for solving a single linear set (§2.3). The method's principal strength is that it is as stable as any other direct method, perhaps even a bit more stable when full pivoting is used (see below).

If you come along later with an additional right-hand side vector, you can multiply it by the inverse matrix, of course. This does give an answer, but one that is quite susceptible to roundoff error, not nearly as good as if the new vector had been included with the set of right-hand side vectors in the first instance.

For these reasons, Gauss-Jordan elimination should usually not be your method of first choice, either for solving linear equations or for matrix inversion. The decomposition methods in §2.3 are better. Why do we give you Gauss-Jordan at all? Because it is straightforward, understandable, solid as a rock, and an exceptionally good "psychological" backup for those times that something is going wrong and you think it *might* be your linear-equation solver.

Some people believe that the backup is more than psychological, that Gauss-Jordan elimination is an "independent" numerical method. This turns out to be mostly myth. Except for the relatively minor differences in pivoting, described below, the actual sequence of operations performed in Gauss-Jordan elimination is very closely related to that performed by the routines in the next two sections.

For clarity, and to avoid writing endless ellipses ($\cdots$) we will write out equations only for the case of four equations and four unknowns, and with three different right-hand side vectors that are known in advance. You can write bigger matrices and extend the equations to the case of $N \times N$ matrices, with M sets of right-hand side vectors, in completely analogous fashion. The routine implemented below is, of course, general.

Elimination on Column-Augmented Matrices

Consider the linear matrix equation

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \left[\begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{pmatrix} \sqcup \begin{pmatrix} x_{12} \\ x_{22} \\ x_{32} \\ x_{42} \end{pmatrix} \sqcup \begin{pmatrix} x_{13} \\ x_{23} \\ x_{33} \\ x_{43} \end{pmatrix} \sqcup \begin{pmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{pmatrix} \right]$$

$$= \left[\begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{pmatrix} \sqcup \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ b_{42} \end{pmatrix} \sqcup \begin{pmatrix} b_{13} \\ b_{23} \\ b_{33} \\ b_{43} \end{pmatrix} \sqcup \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \tag{2.1.1}$$

Here the raised dot $(\cdot)$ signifies matrix multiplication, while the operator $\sqcup$ just signifies column augmentation, that is, removing the abutting parentheses and making a wider matrix out of the operands of the $\sqcup$ operator.

It should not take you long to write out equation (2.1.1) and to see that it simply states that x_{ij} is the ith component ($i = 1, 2, 3, 4$) of the vector solution of the jth right-hand side ($j = 1, 2, 3$), the one whose coefficients are $b_{ij}, i = 1, 2, 3, 4$; and that the matrix of unknown coefficients y_{ij} is the inverse matrix of a_{ij}. In other words, the matrix solution of

$$[\mathbf{A}] \cdot [\mathbf{x}_1 \sqcup \mathbf{x}_2 \sqcup \mathbf{x}_3 \sqcup \mathbf{Y}] = [\mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \mathbf{b}_3 \sqcup \mathbf{1}] \tag{2.1.2}$$

where $\mathbf{A}$ and $\mathbf{Y}$ are square matrices, the $\mathbf{b}_i$'s and $\mathbf{x}_i$'s are column vectors, and $\mathbf{1}$ is the identity matrix, simultaneously solves the linear sets

$$\mathbf{A} \cdot \mathbf{x}_1 = \mathbf{b}_1 \qquad \mathbf{A} \cdot \mathbf{x}_2 = \mathbf{b}_2 \qquad \mathbf{A} \cdot \mathbf{x}_3 = \mathbf{b}_3 \tag{2.1.3}$$

and

$$\mathbf{A} \cdot \mathbf{Y} = \mathbf{1} \tag{2.1.4}$$

Now it is also elementary to verify the following facts about (2.1.1):

- Interchanging any two *rows* of $\mathbf{A}$ and the corresponding *rows* of the $\mathbf{b}$'s and of $\mathbf{1}$, does not change (or scramble in any way) the solution $\mathbf{x}$'s and $\mathbf{Y}$. Rather, it just corresponds to writing the same set of linear equations in a different order.
- Likewise, the solution set is unchanged and in no way scrambled if we replace any row in $\mathbf{A}$ by a linear combination of itself and any other row, as long as we do the same linear combination of the rows of the $\mathbf{b}$'s and $\mathbf{1}$ (which then is no longer the identity matrix, of course).
- Interchanging any two *columns* of $\mathbf{A}$ gives the same solution set only if we simultaneously interchange corresponding *rows* of the $\mathbf{x}$'s and of $\mathbf{Y}$. In other words, this interchange scrambles the order of the rows in the solution. If we do this, we will need to unscramble the solution by restoring the rows to their original order.

Gauss-Jordan elimination uses one or more of the above operations to reduce the matrix $\mathbf{A}$ to the identity matrix. When this is accomplished, the right-hand side becomes the solution set, as one sees instantly from (2.1.2).

Pivoting

In "Gauss-Jordan elimination with no pivoting," only the second operation in the above list is used. The first row is divided by the element a_{11} (this being a trivial linear combination of the first row with any other row — zero coefficient for the other row). Then the right amount of the first row is subtracted from each other row to make all the remaining a_{i1}'s zero. The first column of $\mathbf{A}$ now agrees with the identity matrix. We move to the second column and divide the second row by a_{22}, then subtract the right amount of the second row from rows 1, 3, and 4, so as to make their entries in the second column zero. The second column is now reduced to the identity form. And so on for the third and fourth columns. As we do these operations to $\mathbf{A}$, we of course also do the corresponding operations to the $\mathbf{b}$'s and to $\mathbf{1}$ (which by now no longer resembles the identity matrix in any way!).

Obviously we will run into trouble if we ever encounter a zero element on the (then current) diagonal when we are going to divide by the diagonal element. (The element that we divide by, incidentally, is called the *pivot element* or *pivot*.) Not so obvious, but true, is the fact that Gauss-Jordan elimination with no pivoting (no use of the first or third procedures in the above list) is numerically unstable in the presence of any roundoff error, even when a zero pivot is not encountered. You must *never* do Gauss-Jordan elimination (or Gaussian elimination, see below) without pivoting!

So what *is* this magic pivoting? Nothing more than interchanging rows (*partial pivoting*) or rows and columns (*full pivoting*), so as to put a particularly desirable element in the diagonal position from which the pivot is about to be selected. Since we don't want to mess up the part of the identity matrix that we have already built up, we can choose among elements that are both (i) on rows below (or on) the one that is about to be normalized, and also (ii) on columns to the right (or on) the column we are about to eliminate. Partial pivoting is easier than full pivoting, because we don't have to keep track of the permutation of the solution vector. Partial pivoting makes available as pivots only the elements already in the correct column. It turns out that partial pivoting is "almost" as good as full pivoting, in a sense that can be made mathematically precise, but which need not concern us here (for discussion and references, see [1]). To show you both variants, we do full pivoting in the routine in this section, partial pivoting in §2.3.

We have to state how to recognize a particularly desirable pivot when we see one. The answer to this is not completely known theoretically. It is known, both theoretically and in practice, that simply picking the largest (in magnitude) available element as the pivot is a very good choice. A curiosity of this procedure, however, is that the choice of pivot will depend on the original scaling of the equations. If we take the third linear equation in our original set and multiply it by a factor of a million, it is almost guaranteed that it will contribute the first pivot; yet the underlying solution of the equations is not changed by this multiplication! One therefore sometimes sees routines which choose as pivot that element which *would* have been largest if the original equations had all been scaled to have their largest coefficient normalized to unity. This is called *implicit pivoting*. There is some extra bookkeeping to keep track of the scale factors by which the rows would have been multiplied. (The routines in §2.3 include implicit pivoting, but the routine in this section does not.)

Finally, let us consider the storage requirements of the method. With a little reflection you will see that at every stage of the algorithm, *either* an element of $\mathbf{A}$ is

predictably a one or zero (if it is already in a part of the matrix that has been reduced to identity form) *or else* the exactly corresponding element of the matrix that started as **1** is predictably a one or zero (if its mate in **A** has not been reduced to the identity form). Therefore the matrix **1** does not have to exist as separate storage: The matrix inverse of **A** is gradually built up in **A** as the original **A** is destroyed. Likewise, the solution vectors **x** can gradually replace the right-hand side vectors **b** and share the same storage, since after each column in **A** is reduced, the corresponding row entry in the **b**'s is never again used.

Here is the routine for Gauss-Jordan elimination with full pivoting:

```
#include <math.h>
#include "nrutil.h"
#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}

void gaussj(float **a, int n, float **b, int m)
```

Linear equation solution by Gauss-Jordan elimination, equation (2.1.1) above. `a[1..n][1..n]` is the input matrix. `b[1..n][1..m]` is input containing the `m` right-hand side vectors. On output, `a` is replaced by its matrix inverse, and `b` is replaced by the corresponding set of solution vectors.

```
{
    int *indxc,*indxr,*ipiv;
    int i,icol,irow,j,k,l,ll;
    float big,dum,pivinv,temp;

    indxc=ivector(1,n);                 The integer arrays ipiv, indxr, and indxc are
    indxr=ivector(1,n);                     used for bookkeeping on the pivoting.
    ipiv=ivector(1,n);
    for (j=1;j<=n;j++) ipiv[j]=0;
    for (i=1;i<=n;i++) {                This is the main loop over the columns to be
        big=0.0;                            reduced.
        for (j=1;j<=n;j++)              This is the outer loop of the search for a pivot
            if (ipiv[j] != 1)               element.
                for (k=1;k<=n;k++) {
                    if (ipiv[k] == 0) {
                        if (fabs(a[j][k]) >= big) {
                            big=fabs(a[j][k]);
                            irow=j;
                            icol=k;
                        }
                    }
                }
        ++(ipiv[icol]);
```

We now have the pivot element, so we interchange rows, if needed, to put the pivot element on the diagonal. The columns are not physically interchanged, only relabeled: `indxc[i]`, the column of the `i`th pivot element, is the `i`th column that is reduced, while `indxr[i]` is the row in which that pivot element was originally located. If `indxr[i]` $\neq$ `indxc[i]` there is an implied column interchange. With this form of bookkeeping, the solution `b`'s will end up in the correct order, and the inverse matrix will be scrambled by columns.

```
        if (irow != icol) {
            for (l=1;l<=n;l++) SWAP(a[irow][l],a[icol][l])
            for (l=1;l<=m;l++) SWAP(b[irow][l],b[icol][l])
        }
        indxr[i]=irow;                  We are now ready to divide the pivot row by the
        indxc[i]=icol;                      pivot element, located at irow and icol.
        if (a[icol][icol] == 0.0) nrerror("gaussj: Singular Matrix");
        pivinv=1.0/a[icol][icol];
        a[icol][icol]=1.0;
        for (l=1;l<=n;l++) a[icol][l] *= pivinv;
        for (l=1;l<=m;l++) b[icol][l] *= pivinv;
```

```
        for (ll=1;ll<=n;ll++)           Next, we reduce the rows...
            if (ll != icol) {           ...except for the pivot one, of course.
                dum=a[ll][icol];
                a[ll][icol]=0.0;
                for (l=1;l<=n;l++) a[ll][l] -= a[icol][l]*dum;
                for (l=1;l<=m;l++) b[ll][l] -= b[icol][l]*dum;
            }
    }
    This is the end of the main loop over columns of the reduction. It only remains to unscram-
    ble the solution in view of the column interchanges. We do this by interchanging pairs of
    columns in the reverse order that the permutation was built up.
    for (l=n;l>=1;l--) {
        if (indxr[l] != indxc[l])
            for (k=1;k<=n;k++)
                SWAP(a[k][indxr[l]],a[k][indxc[l]]);
    }                                   And we are done.
    free_ivector(ipiv,1,n);
    free_ivector(indxr,1,n);
    free_ivector(indxc,1,n);
}
```

Row versus Column Elimination Strategies

The above discussion can be amplified by a modest amount of formalism. Row operations on a matrix $\mathbf{A}$ correspond to pre- (that is, left-) multiplication by some simple matrix $\mathbf{R}$. For example, the matrix $\mathbf{R}$ with components

$$R_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } i \neq 2, 4 \\ 1 & \text{if } i = 2, j = 4 \\ 1 & \text{if } i = 4, j = 2 \\ 0 & \text{otherwise} \end{cases} \tag{2.1.5}$$

effects the interchange of rows 2 and 4. Gauss-Jordan elimination by row operations alone (including the possibility of *partial* pivoting) consists of a series of such left-multiplications, yielding successively

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ (\cdots \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{A}) \cdot \mathbf{x} &= \cdots \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{b} \\ (\mathbf{1}) \cdot \mathbf{x} &= \cdots \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{b} \\ \mathbf{x} &= \cdots \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{b} \end{aligned} \tag{2.1.6}$$

The key point is that since the $\mathbf{R}$'s build from right to left, the right-hand side is simply transformed at each stage from one vector to another.

Column operations, on the other hand, correspond to post-, or right-, multiplications by simple matrices, call them $\mathbf{C}$. The matrix in equation (2.1.5), if right-multiplied onto a matrix $\mathbf{A}$, will interchange $\mathbf{A}$'s second and fourth *columns*. Elimination by column operations involves (conceptually) inserting a column operator, *and also its inverse,* between the matrix $\mathbf{A}$ and the unknown vector $\mathbf{x}$:

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{A} \cdot \mathbf{C}_1 \cdot \mathbf{C}_1^{-1} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{A} \cdot \mathbf{C}_1 \cdot \mathbf{C}_2 \cdot \mathbf{C}_2^{-1} \cdot \mathbf{C}_1^{-1} \cdot \mathbf{x} &= \mathbf{b} \\ (\mathbf{A} \cdot \mathbf{C}_1 \cdot \mathbf{C}_2 \cdot \mathbf{C}_3 \cdots) \cdots \mathbf{C}_3^{-1} \cdot \mathbf{C}_2^{-1} \cdot \mathbf{C}_1^{-1} \cdot \mathbf{x} &= \mathbf{b} \\ (\mathbf{1}) \cdots \mathbf{C}_3^{-1} \cdot \mathbf{C}_2^{-1} \cdot \mathbf{C}_1^{-1} \cdot \mathbf{x} &= \mathbf{b} \end{aligned} \tag{2.1.7}$$

which (peeling of the $\mathbf{C}^{-1}$'s one at a time) implies a solution

$$\mathbf{x} = \mathbf{C}_1 \cdot \mathbf{C}_2 \cdot \mathbf{C}_3 \cdots \mathbf{b} \tag{2.1.8}$$

Notice the essential difference between equation (2.1.8) and equation (2.1.6). In the latter case, the $\mathbf{C}$'s must be applied to $\mathbf{b}$ in the *reverse order* from that in which they become known. That is, they must all be stored along the way. This requirement greatly reduces the usefulness of column operations, generally restricting them to simple permutations, for example in support of full pivoting.

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [1]

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Example 5.2, p. 282.

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Program B-2, p. 298.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.3–1.

2.2 Gaussian Elimination with Backsubstitution

The usefulness of Gaussian elimination with backsubstitution is primarily pedagogical. It stands between full elimination schemes such as Gauss-Jordan, and triangular decomposition schemes such as will be discussed in the next section. Gaussian elimination reduces a matrix not all the way to the identity matrix, but only halfway, to a matrix whose components on the diagonal and above (say) remain nontrivial. Let us now see what advantages accrue.

Suppose that in doing Gauss-Jordan elimination, as described in §2.1, we at each stage subtract away rows only *below* the then-current pivot element. When a_{22} is the pivot element, for example, we divide the second row by its value (as before), but now use the pivot row to zero only a_{32} and a_{42}, not a_{12} (see equation 2.1.1). Suppose, also, that we do only partial pivoting, never interchanging columns, so that the order of the unknowns never needs to be modified.

Then, when we have done this for all the pivots, we will be left with a reduced equation that looks like this (in the case of a single right-hand side vector):

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \tag{2.2.1}$$

Here the primes signify that the a's and b's do not have their original numerical values, but have been modified by all the row operations in the elimination to this point. The procedure up to this point is termed *Gaussian elimination*.

Backsubstitution

But how do we solve for the x's? The last x (x_4 in this example) is already isolated, namely

$$x_4 = b'_4/a'_{44} \tag{2.2.2}$$

With the last x known we can move to the penultimate x,

$$x_3 = \frac{1}{a'_{33}}[b'_3 - x_4 a'_{34}] \tag{2.2.3}$$

and then proceed with the x before that one. The typical step is

$$x_i = \frac{1}{a'_{ii}}\left[b'_i - \sum_{j=i+1}^{N} a'_{ij}x_j\right] \tag{2.2.4}$$

The procedure defined by equation (2.2.4) is called *backsubstitution*. The combination of Gaussian elimination and backsubstitution yields a solution to the set of equations.

The advantage of Gaussian elimination and backsubstitution over Gauss-Jordan elimination is simply that the former is faster in raw operations count: The innermost loops of Gauss-Jordan elimination, each containing one subtraction and one multiplication, are executed N^3 and N^2M times (where there are N equations and M unknowns). The corresponding loops in Gaussian elimination are executed only $\frac{1}{3}N^3$ times (only half the matrix is reduced, and the increasing numbers of predictable zeros reduce the count to one-third), and $\frac{1}{2}N^2M$ times, respectively. Each backsubstitution of a right-hand side is $\frac{1}{2}N^2$ executions of a similar loop (one multiplication plus one subtraction). For $M \ll N$ (only a few right-hand sides) Gaussian elimination thus has about a factor three advantage over Gauss-Jordan. (We could reduce this advantage to a factor 1.5 by *not* computing the inverse matrix as part of the Gauss-Jordan scheme.)

For computing the inverse matrix (which we can view as the case of $M = N$ right-hand sides, namely the N unit vectors which are the columns of the identity matrix), Gaussian elimination and backsubstitution at first glance require $\frac{1}{3}N^3$ (matrix reduction) $+\frac{1}{2}N^3$ (right-hand side manipulations) $+\frac{1}{2}N^3$ (N backsubstitutions) $= \frac{4}{3}N^3$ loop executions, which is more than the N^3 for Gauss-Jordan. However, the unit vectors are quite special in containing all zeros except for one element. If this is taken into account, the right-side manipulations can be reduced to only $\frac{1}{6}N^3$ loop executions, and, for matrix inversion, the two methods have identical efficiencies.

Both Gaussian elimination and Gauss-Jordan elimination share the disadvantage that all right-hand sides must be known in advance. The LU decomposition method in the next section does not share that deficiency, and also has an equally small operations count, both for solution with any number of right-hand sides, and for matrix inversion. For this reason we will not implement the method of Gaussian elimination as a routine.

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.3–1.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §2.1.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.2.1.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

2.3 LU Decomposition and Its Applications

Suppose we are able to write the matrix $\mathbf{A}$ as a product of two matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{2.3.1}$$

where $\mathbf{L}$ is *lower triangular* (has elements only on the diagonal and below) and $\mathbf{U}$ is *upper triangular* (has elements only on the diagonal and above). For the case of a 4×4 matrix $\mathbf{A}$, for example, equation (2.3.1) would look like this:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{2.3.2}$$

We can use a decomposition such as (2.3.1) to solve the linear set

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{2.3.3}$$

by first solving for the vector $\mathbf{y}$ such that

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{2.3.4}$$

and then solving

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{2.3.5}$$

What is the advantage of breaking up one linear set into two successive ones? The advantage is that the solution of a triangular set of equations is quite trivial, as we have already seen in §2.2 (equation 2.2.4). Thus, equation (2.3.4) can be solved by *forward substitution* as follows,

$$\begin{aligned} y_1 &= \frac{b_1}{\alpha_{11}} \\ y_i &= \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \qquad i = 2, 3, \ldots, N \end{aligned} \tag{2.3.6}$$

while (2.3.5) can then be solved by *backsubstitution* exactly as in equations (2.2.2)–(2.2.4),

$$\begin{aligned} x_N &= \frac{y_N}{\beta_{NN}} \\ x_i &= \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right] \qquad i = N-1, N-2, \ldots, 1 \end{aligned} \tag{2.3.7}$$

Equations (2.3.6) and (2.3.7) total (for each right-hand side **b**) N^2 executions of an inner loop containing one multiply and one add. If we have N right-hand sides which are the unit column vectors (which is the case when we are inverting a matrix), then taking into account the leading zeros reduces the total execution count of (2.3.6) from $\frac{1}{2}N^3$ to $\frac{1}{6}N^3$, while (2.3.7) is unchanged at $\frac{1}{2}N^3$.

Notice that, once we have the LU decomposition of **A**, we can solve with as many right-hand sides as we then care to, one at a time. This is a distinct advantage over the methods of §2.1 and §2.2.

Performing the LU Decomposition

How then can we solve for **L** and **U**, given **A**? First, we write out the i,jth component of equation (2.3.1) or (2.3.2). That component always is a sum beginning with

$$\alpha_{i1}\beta_{1j} + \cdots = a_{ij}$$

The number of terms in the sum depends, however, on whether i or j is the smaller number. We have, in fact, the three cases,

$$i < j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{2.3.8}$$

$$i = j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{2.3.9}$$

$$i > j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{2.3.10}$$

Equations (2.3.8)–(2.3.10) total N^2 equations for the $N^2 + N$ unknown α's and β's (the diagonal being represented twice). Since the number of unknowns is greater than the number of equations, we are invited to specify N of the unknowns arbitrarily and then try to solve for the others. In fact, as we shall see, it is always possible to take

$$\alpha_{ii} \equiv 1 \qquad i = 1, \ldots, N \tag{2.3.11}$$

A surprising procedure, now, is *Crout's algorithm*, which quite trivially solves the set of $N^2 + N$ equations (2.3.8)–(2.3.11) for all the α's and β's by just arranging the equations in a certain order! That order is as follows:

- Set $\alpha_{ii} = 1$, $i = 1, \ldots, N$ (equation 2.3.11).
- For each $j = 1, 2, 3, \ldots, N$ do these two procedures: First, for $i = 1, 2, \ldots, j$, use (2.3.8), (2.3.9), and (2.3.11) to solve for β_{ij}, namely

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}. \tag{2.3.12}$$

  (When $i = 1$ in 2.3.12 the summation term is taken to mean zero.) Second, for $i = j+1, j+2, \ldots, N$ use (2.3.10) to solve for α_{ij}, namely

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right). \tag{2.3.13}$$

  Be sure to do both procedures before going on to the next j.

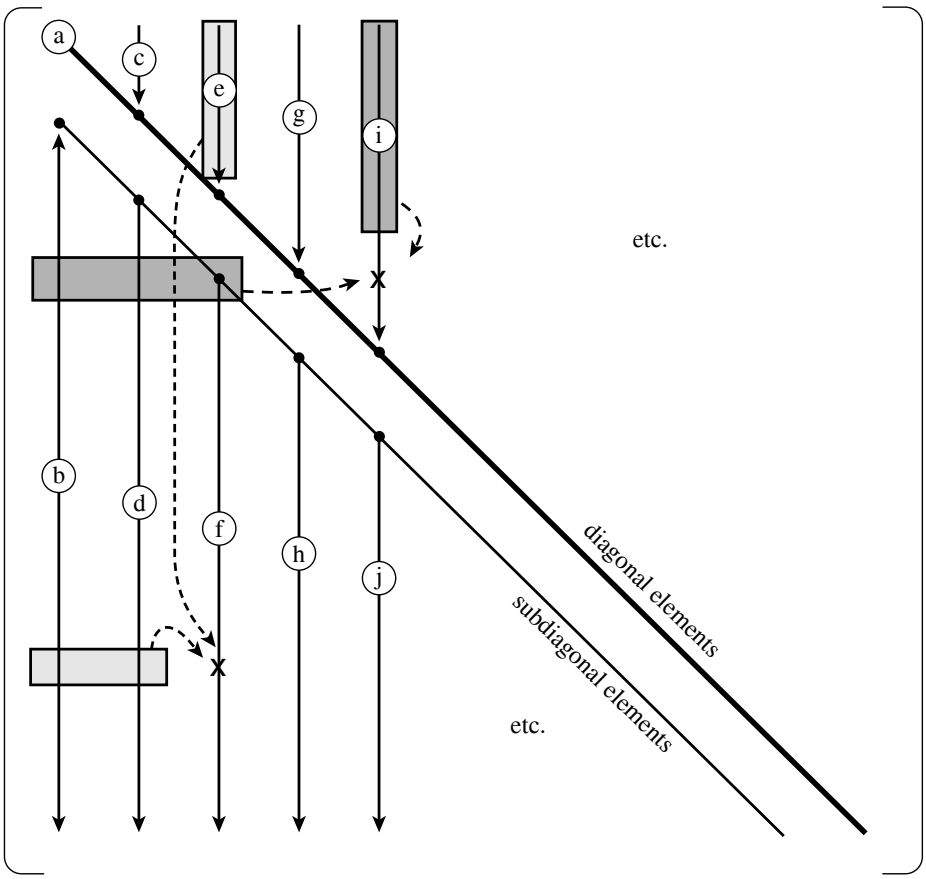

Figure 2.3.1. Crout's algorithm for *LU* decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lower case letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an "x".

If you work through a few iterations of the above procedure, you will see that the α's and β's that occur on the right-hand side of equations (2.3.12) and (2.3.13) are already determined by the time they are needed. You will also see that every a_{ij} is used only once and never again. This means that the corresponding α_{ij} or β_{ij} can be stored in the location that the a used to occupy: the decomposition is "in place." [The diagonal unity elements α_{ii} (equation 2.3.11) are not stored at all.] In brief, Crout's method fills in the combined matrix of α's and β's,

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix} \tag{2.3.14}$$

by columns from left to right, and within each column from top to bottom (see Figure 2.3.1).

What about pivoting? Pivoting (i.e., selection of a salubrious pivot element for the division in equation 2.3.13) is absolutely essential for the stability of Crout's

method. Only partial pivoting (interchange of rows) can be implemented efficiently. However this is enough to make the method stable. This means, incidentally, that we don't actually decompose the matrix **A** into LU form, but rather we decompose a rowwise permutation of **A**. (If we keep track of what that permutation is, this decomposition is just as useful as the original one would have been.)

Pivoting is slightly subtle in Crout's algorithm. The key point to notice is that equation (2.3.12) in the case of $i = j$ (its final application) is *exactly the same* as equation (2.3.13) except for the division in the latter equation; in both cases the upper limit of the sum is $k = j - 1$ $(= i - 1)$. This means that we don't have to commit ourselves as to whether the diagonal element β_{jj} is the one that happens to fall on the diagonal in the first instance, or whether one of the (undivided) α_{ij}'s below it in the column, $i = j+1, \ldots, N$, is to be "promoted" to become the diagonal β. This can be decided after all the candidates in the column are in hand. As you should be able to guess by now, we will choose the largest one as the diagonal β (pivot element), then do all the divisions by that element *en masse*. This is *Crout's method with partial pivoting*. Our implementation has one additional wrinkle: It initially finds the largest element in each row, and subsequently (when it is looking for the maximal pivot element) scales the comparison *as if* we had initially scaled all the equations to make their maximum coefficient equal to unity; this is the *implicit pivoting* mentioned in §2.1.

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-20                    A small number.

void ludcmp(float **a, int n, int *indx, float *d)
```

Given a matrix `a[1..n][1..n]`, this routine replaces it by the LU decomposition of a rowwise permutation of itself. `a` and `n` are input. `a` is output, arranged as in equation (2.3.14) above; `indx[1..n]` is an output vector that records the row permutation effected by the partial pivoting; `d` is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with `lubksb` to solve linear equations or invert a matrix.

```
{
    int i,imax,j,k;
    float big,dum,sum,temp;
    float *vv;                          vv stores the implicit scaling of each row.

    vv=vector(1,n);
    *d=1.0;                             No row interchanges yet.
    for (i=1;i<=n;i++) {                Loop over rows to get the implicit scaling informa-
        big=0.0;                            tion.
        for (j=1;j<=n;j++)
            if ((temp=fabs(a[i][j])) > big) big=temp;
        if (big == 0.0) nrerror("Singular matrix in routine ludcmp");
        No nonzero largest element.
        vv[i]=1.0/big;                  Save the scaling.
    }
    for (j=1;j<=n;j++) {                This is the loop over columns of Crout's method.
        for (i=1;i<j;i++) {             This is equation (2.3.12) except for i = j.
            sum=a[i][j];
            for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
            a[i][j]=sum;
        }
        big=0.0;                        Initialize for the search for largest pivot element.
        for (i=j;i<=n;i++) {            This is i = j of equation (2.3.12) and i = j+1 . . . N
            sum=a[i][j];                    of equation (2.3.13).
            for (k=1;k<j;k++)
```

```
                sum -= a[i][k]*a[k][j];
            a[i][j]=sum;
            if ( (dum=vv[i]*fabs(sum)) >= big) {
            Is the figure of merit for the pivot better than the best so far?
                big=dum;
                imax=i;
            }
        }
        if (j != imax) {                Do we need to interchange rows?
            for (k=1;k<=n;k++) {        Yes, do so...
                dum=a[imax][k];
                a[imax][k]=a[j][k];
                a[j][k]=dum;
            }
            *d = -(*d);                 ...and change the parity of d.
            vv[imax]=vv[j];             Also interchange the scale factor.
        }
        indx[j]=imax;
        if (a[j][j] == 0.0) a[j][j]=TINY;
        If the pivot element is zero the matrix is singular (at least to the precision of the
        algorithm). For some applications on singular matrices, it is desirable to substitute
        TINY for zero.
        if (j != n) {                   Now, finally, divide by the pivot element.
            dum=1.0/(a[j][j]);
            for (i=j+1;i<=n;i++) a[i][j] *= dum;
        }
    }                                   Go back for the next column in the reduction.
    free_vector(vv,1,n);
}
```

Here is the routine for forward substitution and backsubstitution, implementing equations (2.3.6) and (2.3.7).

```
void lubksb(float **a, int n, int *indx, float b[])
```

Solves the set of `n` linear equations $A \cdot X = B$. Here `a[1..n][1..n]` is input, not as the matrix A but rather as its LU decomposition, determined by the routine `ludcmp`. `indx[1..n]` is input as the permutation vector returned by `ludcmp`. `b[1..n]` is input as the right-hand side vector B, and returns with the solution vector X. `a`, `n`, and `indx` are not modified by this routine and can be left in place for successive calls with different right-hand sides `b`. This routine takes into account the possibility that `b` will begin with many zero elements, so it is efficient for use in matrix inversion.

```
{
    int i,ii=0,ip,j;
    float sum;

    for (i=1;i<=n;i++) {            When ii is set to a positive value, it will become the
        ip=indx[i];                     index of the first nonvanishing element of b. We now
        sum=b[ip];                      do the forward substitution, equation (2.3.6). The
        b[ip]=b[i];                     only new wrinkle is to unscramble the permutation
        if (ii)                         as we go.
            for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
        else if (sum) ii=i;         A nonzero element was encountered, so from now on we
        b[i]=sum;                       will have to do the sums in the loop above.
    }
    for (i=n;i>=1;i--) {            Now we do the backsubstitution, equation (2.3.7).
        sum=b[i];
        for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
        b[i]=sum/a[i][i];           Store a component of the solution vector X.
    }                               All done!
}
```

The LU decomposition in `ludcmp` requires about $\frac{1}{3}N^3$ executions of the inner loops (each with one multiply and one add). This is thus the operation count for solving one (or a few) right-hand sides, and is a factor of 3 better than the Gauss-Jordan routine `gaussj` which was given in §2.1, and a factor of 1.5 better than a Gauss-Jordan routine (not given) that does not compute the inverse matrix. For inverting a matrix, the total count (including the forward and backsubstitution as discussed following equation 2.3.7 above) is $(\frac{1}{3} + \frac{1}{6} + \frac{1}{2})N^3 = N^3$, the same as `gaussj`.

To summarize, this is the preferred way to solve the linear set of equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$:

```
float **a,*b,d;
int n,*indx;
...
ludcmp(a,n,indx,&d);
lubksb(a,n,indx,b);
```

The answer **x** will be given back in `b`. Your original matrix **A** will have been destroyed.

If you subsequently want to solve a set of equations with the same **A** but a different right-hand side **b**, you repeat *only*

```
lubksb(a,n,indx,b);
```

not, of course, with the original matrix **A**, but with `a` and `indx` as were already set by `ludcmp`.

Inverse of a Matrix

Using the above LU decomposition and backsubstitution routines, it is completely straightforward to find the inverse of a matrix column by column.

```
#define N ...
float **a,**y,d,*col;
int i,j,*indx;
...
ludcmp(a,N,indx,&d);                    Decompose the matrix just once.
for(j=1;j<=N;j++) {                     Find inverse by columns.
    for(i=1;i<=N;i++) col[i]=0.0;
    col[j]=1.0;
    lubksb(a,N,indx,col);
    for(i=1;i<=N;i++) y[i][j]=col[i];
}
```

The matrix `y` will now contain the inverse of the original matrix `a`, which will have been destroyed. Alternatively, there is nothing wrong with using a Gauss-Jordan routine like `gaussj` (§2.1) to invert a matrix in place, again destroying the original. Both methods have practically the same operations count.

Incidentally, if you ever have the need to compute $\mathbf{A}^{-1} \cdot \mathbf{B}$ from matrices $\mathbf{A}$ and $\mathbf{B}$, you should LU decompose $\mathbf{A}$ and then backsubstitute with the columns of $\mathbf{B}$ instead of with the unit vectors that would give $\mathbf{A}$'s inverse. This saves a whole matrix multiplication, and is also more accurate.

Determinant of a Matrix

The determinant of an LU decomposed matrix is just the product of the diagonal elements,

$$\det = \prod_{j=1}^{N} \beta_{jj} \tag{2.3.15}$$

We don't, recall, compute the decomposition of the original matrix, but rather a decomposition of a rowwise permutation of it. Luckily, we have kept track of whether the number of row interchanges was even or odd, so we just preface the product by the corresponding sign. (You now finally know the purpose of setting `d` in the routine `ludcmp`.)

Calculation of a determinant thus requires one call to `ludcmp`, with *no* subsequent backsubstitutions by `lubksb`.

```
#define N ...
float **a,d;
int j,*indx;
...
ludcmp(a,N,indx,&d);          This returns d as ±1.
for(j=1;j<=N;j++) d *= a[j][j];
```

The variable `d` now contains the determinant of the original matrix `a`, which will have been destroyed.

For a matrix of any substantial size, it is quite likely that the determinant will overflow or underflow your computer's floating-point dynamic range. In this case you can modify the loop of the above fragment and (e.g.) divide by powers of ten, to keep track of the scale separately, or (e.g.) accumulate the sum of logarithms of the absolute values of the factors and the sign separately.

Complex Systems of Equations

If your matrix $\mathbf{A}$ is real, but the right-hand side vector is complex, say $\mathbf{b} + i\mathbf{d}$, then (i) LU decompose $\mathbf{A}$ in the usual way, (ii) backsubstitute $\mathbf{b}$ to get the real part of the solution vector, and (iii) backsubstitute $\mathbf{d}$ to get the imaginary part of the solution vector.

If the matrix itself is complex, so that you want to solve the system

$$(\mathbf{A} + i\mathbf{C}) \cdot (\mathbf{x} + i\mathbf{y}) = (\mathbf{b} + i\mathbf{d}) \tag{2.3.16}$$

then there are two possible ways to proceed. The best way is to rewrite `ludcmp` and `lubksb` as complex routines. Complex modulus substitutes for absolute value in the construction of the scaling vector `vv` and in the search for the largest pivot elements. Everything else goes through in the obvious way, with complex arithmetic used as needed. (See §§1.2 and 5.4 for discussion of complex arithmetic in `C`.)

A quick-and-dirty way to solve complex systems is to take the real and imaginary parts of (2.3.16), giving

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} - \mathbf{C} \cdot \mathbf{y} &= \mathbf{b} \\ \mathbf{C} \cdot \mathbf{x} + \mathbf{A} \cdot \mathbf{y} &= \mathbf{d} \end{aligned} \tag{2.3.17}$$

which can be written as a $2N \times 2N$ set of *real* equations,

$$\begin{pmatrix} \mathbf{A} & -\mathbf{C} \\ \mathbf{C} & \mathbf{A} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{d} \end{pmatrix} \tag{2.3.18}$$

and then solved with `ludcmp` and `lubksb` in their present forms. This scheme is a factor of 2 inefficient in storage, since $\mathbf{A}$ and $\mathbf{C}$ are stored twice. It is also a factor of 2 inefficient in time, since the complex multiplies in a complexified version of the routines would each use 4 real multiplies, while the solution of a $2N \times 2N$ problem involves 8 times the work of an $N \times N$ one. If you can tolerate these factor-of-two inefficiencies, then equation (2.3.18) is an easy way to proceed.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapter 4.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §3.3, and p. 50.

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 9, 16, and 18.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.11.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

2.4 Tridiagonal and Band Diagonal Systems of Equations

The special case of a system of linear equations that is *tridiagonal*, that is, has nonzero elements only on the diagonal plus or minus one column, is one that occurs frequently. Also common are systems that are *band diagonal*, with nonzero elements only along a few diagonal lines adjacent to the main diagonal (above and below).

For tridiagonal sets, the procedures of LU decomposition, forward- and back-substitution each take only $O(N)$ operations, and the whole solution can be encoded very concisely. The resulting routine `tridag` is one that we will use in later chapters.

Naturally, one does not reserve storage for the full $N \times N$ matrix, but only for the nonzero components, stored as three vectors. The set of equations to be solved is

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \cdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.4.1}$$

Notice that a_1 and c_N are undefined and are not referenced by the routine that follows.

```
#include "nrutil.h"

void tridag(float a[], float b[], float c[], float r[], float u[],
    unsigned long n)
Solves for a vector u[1..n] the tridiagonal linear set given by equation (2.4.1). a[1..n],
b[1..n], c[1..n], and r[1..n] are input vectors and are not modified.
{
    unsigned long j;
    float bet,*gam;

    gam=vector(1,n);                    One vector of workspace, gam is needed.
    if (b[1] == 0.0) nrerror("Error 1 in tridag");
    If this happens then you should rewrite your equations as a set of order N − 1, with u2
    trivially eliminated.
    u[1]=r[1]/(bet=b[1]);
    for (j=2;j<=n;j++) {                Decomposition and forward substitution.
        gam[j]=c[j-1]/bet;
        bet=b[j]-a[j]*gam[j];
        if (bet == 0.0)    nrerror("Error 2 in tridag");    Algorithm fails; see be-
        u[j]=(r[j]-a[j]*u[j-1])/bet;                          low.
    }
    for (j=(n-1);j>=1;j--)
        u[j] -= gam[j+1]*u[j+1];        Backsubstitution.
    free_vector(gam,1,n);
}
```

There is no pivoting in `tridag`. It is for this reason that `tridag` can fail even when the underlying matrix is nonsingular: A zero pivot can be encountered even for a nonsingular matrix. In practice, this is not something to lose sleep about. The kinds of problems that lead to tridiagonal linear sets usually have additional properties which guarantee that the algorithm in `tridag` will succeed. For example, if

$$|b_j| > |a_j| + |c_j| \qquad j = 1, \ldots, N \tag{2.4.2}$$

(called *diagonal dominance*) then it can be shown that the algorithm cannot encounter a zero pivot.

It is possible to construct special examples in which the lack of pivoting in the algorithm causes numerical instability. In practice, however, such instability is almost never encountered — unlike the general matrix problem where pivoting is essential.

The tridiagonal algorithm is the rare case of an algorithm that, in practice, is more robust than theory says it should be. Of course, should you ever encounter a problem for which `tridag` fails, you can instead use the more general method for band diagonal systems, now described (routines `bandec` and `banbks`).

Some other matrix forms consisting of tridiagonal with a small number of additional elements (e.g., upper right and lower left corners) also allow rapid solution; see §2.7.

Band Diagonal Systems

Where tridiagonal systems have nonzero elements only on the diagonal plus or minus one, band diagonal systems are slightly more general and have (say) $m_1 \geq 0$ nonzero elements immediately to the left of (below) the diagonal and $m_2 \geq 0$ nonzero elements immediately to its right (above it). Of course, this is only a useful classification if m_1 and m_2 are both $\ll N$.

In that case, the solution of the linear system by LU decomposition can be accomplished much faster, and in much less storage, than for the general $N \times N$ case.

The precise definition of a band diagonal matrix with elements a_{ij} is that

$$a_{ij} = 0 \quad \text{when} \quad j > i + m_2 \quad \text{or} \quad i > j + m_1 \tag{2.4.3}$$

Band diagonal matrices are stored and manipulated in a so-called compact form, which results if the matrix is tilted 45° clockwise, so that its nonzero elements lie in a long, narrow matrix with $m_1 + 1 + m_2$ columns and N rows. This is best illustrated by an example: The band diagonal matrix

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 5 & 0 & 0 & 0 & 0 \\ 9 & 2 & 6 & 5 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 9 & 0 & 0 \\ 0 & 0 & 7 & 9 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 8 & 4 & 6 \\ 0 & 0 & 0 & 0 & 2 & 4 & 4 \end{pmatrix} \tag{2.4.4}$$

which has $N = 7$, $m_1 = 2$, and $m_2 = 1$, is stored compactly as the 7×4 matrix,

$$\begin{pmatrix} x & x & 3 & 1 \\ x & 4 & 1 & 5 \\ 9 & 2 & 6 & 5 \\ 3 & 5 & 8 & 9 \\ 7 & 9 & 3 & 2 \\ 3 & 8 & 4 & 6 \\ 2 & 4 & 4 & x \end{pmatrix} \tag{2.4.5}$$

Here x denotes elements that are wasted space in the compact format; these will not be referenced by any manipulations and can have arbitrary values. Notice that the diagonal of the original matrix appears in column $m_1 + 1$, with subdiagonal elements to its left, superdiagonal elements to its right.

The simplest manipulation of a band diagonal matrix, stored compactly, is to multiply it by a vector to its right. Although this is algorithmically trivial, you might want to study the following routine carefully, as an example of how to pull nonzero elements a_{ij} out of the compact storage format in an orderly fashion.

```
#include "nrutil.h"

void banmul(float **a, unsigned long n, int m1, int m2, float x[], float b[])
```

Matrix multiply $\mathbf{b} = \mathbf{A} \cdot \mathbf{x}$, where $\mathbf{A}$ is band diagonal with `m1` rows below the diagonal and `m2` rows above. The input vector $\mathbf{x}$ and output vector $\mathbf{b}$ are stored as `x[1..n]` and `b[1..n]`, respectively. The array `a[1..n][1..m1+m2+1]` stores $\mathbf{A}$ as follows: The diagonal elements are in `a[1..n][m1+1]`. Subdiagonal elements are in `a[`j`..n][1..m1]` (with $j > 1$ appropriate to the number of elements on each subdiagonal). Superdiagonal elements are in `a[1..`j`][m1+2..m1+m2+1]` with $j <$ `n` appropriate to the number of elements on each superdiagonal.

```
{
    unsigned long i,j,k,tmploop;

    for (i=1;i<=n;i++) {
        k=i-m1-1;
        tmploop=LMIN(m1+m2+1,n-k);
        b[i]=0.0;
        for (j=LMAX(1,1-k);j<=tmploop;j++) b[i] += a[i][j]*x[j+k];
    }
}
```

It is not possible to store the LU decomposition of a band diagonal matrix $\mathbf{A}$ quite as compactly as the compact form of $\mathbf{A}$ itself. The decomposition (essentially by Crout's method, see §2.3) produces additional nonzero "fill-ins." One straightforward storage scheme is to return the upper triangular factor (U) in the same space that $\mathbf{A}$ previously occupied, and to return the lower triangular factor (L) in a separate compact matrix of size $N \times m_1$. The diagonal elements of U (whose product, times $\mathtt{d} = \pm 1$, gives the determinant) are returned in the first column of $\mathbf{A}$'s storage space.

The following routine, `bandec`, is the band-diagonal analog of `ludcmp` in §2.3:

```
#include <math.h>
#define SWAP(a,b) {dum=(a);(a)=(b);(b)=dum;}
#define TINY 1.0e-20

void bandec(float **a, unsigned long n, int m1, int m2, float **al,
    unsigned long indx[], float *d)
```

Given an `n` $\times$ `n` band diagonal matrix $\mathbf{A}$ with `m1` subdiagonal rows and `m2` superdiagonal rows, compactly stored in the array `a[1..n][1..m1+m2+1]` as described in the comment for routine `banmul`, this routine constructs an LU decomposition of a rowwise permutation of $\mathbf{A}$. The upper triangular matrix replaces `a`, while the lower triangular matrix is returned in `al[1..n][1..m1]`. `indx[1..n]` is an output vector which records the row permutation effected by the partial pivoting; `d` is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with `banbks` to solve band-diagonal sets of equations.

```
{
    unsigned long i,j,k,l;
    int mm;
    float dum;

    mm=m1+m2+1;
    l=m1;
    for (i=1;i<=m1;i++) {                Rearrange the storage a bit.
        for (j=m1+2-i;j<=mm;j++) a[i][j-l]=a[i][j];
        l--;
        for (j=mm-l;j<=mm;j++) a[i][j]=0.0;
    }
    *d=1.0;
    l=m1;
    for (k=1;k<=n;k++) {                 For each row...
        dum=a[k][1];
        i=k;
        if (l < n) l++;
        for (j=k+1;j<=l;j++) {           Find the pivot element.
            if (fabs(a[j][1]) > fabs(dum)) {
                dum=a[j][1];
                i=j;
            }
        }
        indx[k]=i;
        if (dum == 0.0) a[k][1]=TINY;
        Matrix is algorithmically singular, but proceed anyway with TINY pivot (desirable in
        some applications).
        if (i != k) {                    Interchange rows.
            *d = -(*d);
            for (j=1;j<=mm;j++) SWAP(a[k][j],a[i][j])
        }
        for (i=k+1;i<=l;i++) {           Do the elimination.
            dum=a[i][1]/a[k][1];
            al[k][i-k]=dum;
            for (j=2;j<=mm;j++) a[i][j-1]=a[i][j]-dum*a[k][j];
            a[i][mm]=0.0;
        }
    }
}
```

Some pivoting is possible within the storage limitations of `bandec`, and the above routine does take advantage of the opportunity. In general, when `TINY` is returned as a diagonal element of U, then the original matrix (perhaps as modified by roundoff error) is in fact singular. In this regard, `bandec` is somewhat more robust than `tridag` above, which can fail algorithmically even for nonsingular matrices; `bandec` is thus also useful (with $m_1 = m_2 = 1$) for some ill-behaved tridiagonal systems.

Once the matrix $\mathbf{A}$ has been decomposed, any number of right-hand sides can be solved in turn by repeated calls to `banbks`, the backsubstitution routine whose analog in §2.3 is `lubksb`.

```
#define SWAP(a,b) {dum=(a);(a)=(b);(b)=dum;}

void banbks(float **a, unsigned long n, int m1, int m2, float **al,
    unsigned long indx[], float b[])
```

Given the arrays `a`, `al`, and `indx` as returned from `bandec`, and given a right-hand side vector `b[1..n]`, solves the band diagonal linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. The solution vector $\mathbf{x}$ overwrites `b[1..n]`. The other input arrays are not modified, and can be left in place for successive calls with different right-hand sides.

```
{
    unsigned long i,k,l;
    int mm;
    float dum;

    mm=m1+m2+1;
    l=m1;
    for (k=1;k<=n;k++) {                Forward substitution, unscrambling the permuted rows
        i=indx[k];                          as we go.
        if (i != k) SWAP(b[k],b[i])
        if (l < n) l++;
        for (i=k+1;i<=l;i++) b[i] -= al[k][i-k]*b[k];
    }
    l=1;
    for (i=n;i>=1;i--) {                Backsubstitution.
        dum=b[i];
        for (k=2;k<=l;k++) dum -= a[i][k]*b[k+i-1];
        b[i]=dum/a[i][1];
        if (l < mm) l++;
    }
}
```

The routines `bandec` and `banbks` are based on the Handbook routines *bandet1* and *bansol1* in [1].

CITED REFERENCES AND FURTHER READING:

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell), p. 74.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Example 5.4.3, p. 166.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.11.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/6. [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.3.

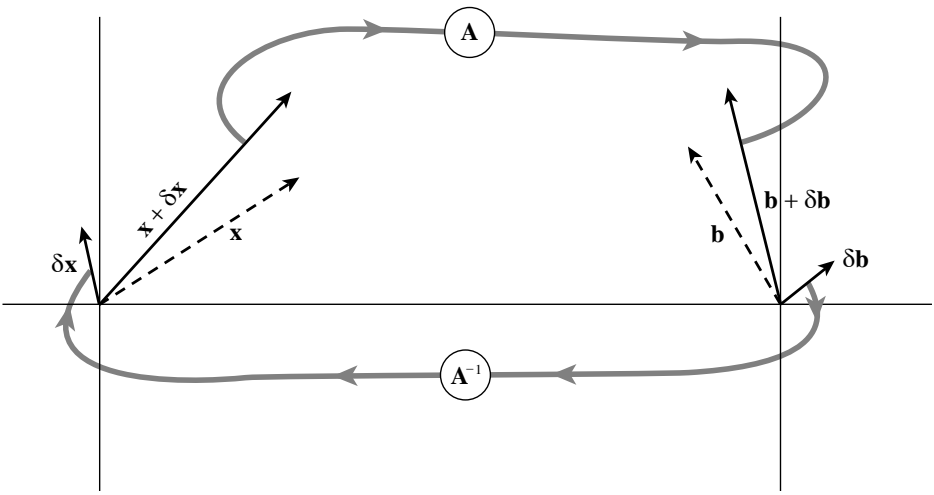

Figure 2.5.1. Iterative improvement of the solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. The first guess $\mathbf{x} + \delta\mathbf{x}$ is multiplied by $\mathbf{A}$ to produce $\mathbf{b} + \delta\mathbf{b}$. The known vector $\mathbf{b}$ is subtracted, giving $\delta\mathbf{b}$. The linear set with this right-hand side is inverted, giving $\delta\mathbf{x}$. This is subtracted from the first guess giving an improved solution $\mathbf{x}$.

2.5 Iterative Improvement of a Solution to Linear Equations

Obviously it is not easy to obtain greater precision for the solution of a linear set than the precision of your computer's floating-point word. Unfortunately, for large sets of linear equations, it is not always easy to obtain precision equal to, or even comparable to, the computer's limit. In direct methods of solution, roundoff errors accumulate, and they are magnified to the extent that your matrix is close to singular. You can easily lose two or three significant figures for matrices which (you thought) were *far* from singular.

If this happens to you, there is a neat trick to restore the full machine precision, called *iterative improvement* of the solution. The theory is very straightforward (see Figure 2.5.1): Suppose that a vector $\mathbf{x}$ is the exact solution of the linear set

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.5.1}$$

You don't, however, know $\mathbf{x}$. You only know some slightly wrong solution $\mathbf{x} + \delta\mathbf{x}$, where $\delta\mathbf{x}$ is the unknown error. When multiplied by the matrix $\mathbf{A}$, your slightly wrong solution gives a product slightly discrepant from the desired right-hand side $\mathbf{b}$, namely

$$\mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \tag{2.5.2}$$

Subtracting (2.5.1) from (2.5.2) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b} \tag{2.5.3}$$

But (2.5.2) can also be solved, trivially, for $\delta\mathbf{b}$. Substituting this into (2.5.3) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) - \mathbf{b} \tag{2.5.4}$$

In this equation, the whole right-hand side is known, since $\mathbf{x} + \delta\mathbf{x}$ is the wrong solution that you want to improve. It is essential to calculate the right-hand side in double precision, since there will be a lot of cancellation in the subtraction of $\mathbf{b}$. Then, we need only solve (2.5.4) for the error $\delta\mathbf{x}$, then subtract this from the wrong solution to get an improved solution.

An important extra benefit occurs if we obtained the original solution by LU decomposition. In this case we already have the LU decomposed form of $\mathbf{A}$, and all we need do to solve (2.5.4) is compute the right-hand side and backsubstitute!

The code to do all this is concise and straightforward:

```
#include "nrutil.h"

void mprove(float **a, float **alud, int n, int indx[], float b[], float x[])
Improves a solution vector x[1..n] of the linear set of equations A · X = B. The matrix
a[1..n][1..n], and the vectors b[1..n] and x[1..n] are input, as is the dimension n.
Also input is alud[1..n][1..n], the LU decomposition of a as returned by ludcmp, and
the vector indx[1..n] also returned by that routine. On output, only x[1..n] is modified,
to an improved set of values.
{
    void lubksb(float **a, int n, int *indx, float b[]);
    int j,i;
    double sdp;
    float *r;

    r=vector(1,n);
    for (i=1;i<=n;i++) {                    Calculate the right-hand side, accumulating
        sdp = -b[i];                            the residual in double precision.
        for (j=1;j<=n;j++) sdp += a[i][j]*x[j];
        r[i]=sdp;
    }
    lubksb(alud,n,indx,r);                  Solve for the error term,
    for (i=1;i<=n;i++) x[i] -= r[i];        and subtract it from the old solution.
    free_vector(r,1,n);
}
```

You should note that the routine `ludcmp` in §2.3 destroys the input matrix as it LU decomposes it. Since iterative improvement requires *both* the original matrix and its LU decomposition, you will need to copy $\mathbf{A}$ before calling `ludcmp`. Likewise `lubksb` destroys $\mathbf{b}$ in obtaining $\mathbf{x}$, so make a copy of $\mathbf{b}$ also. If you don't mind this extra storage, iterative improvement is *highly* recommended: It is a process of order only N^2 operations (multiply vector by matrix, and backsubstitute — see discussion following equation 2.3.7); it never hurts; and it can really give you your money's worth if it saves an otherwise ruined solution on which you have already spent of order N^3 operations.

You can call `mprove` several times in succession if you want. Unless you are starting quite far from the true solution, one call is generally enough; but a second call to verify convergence can be reassuring.

More on Iterative Improvement

It is illuminating (and will be useful later in the book) to give a somewhat more solid analytical foundation for equation (2.5.4), and also to give some additional results. Implicit in the previous discussion was the notion that the solution vector $\mathbf{x} + \delta\mathbf{x}$ has an error term; but we neglected the fact that the LU decomposition of $\mathbf{A}$ is itself not exact.

A different analytical approach starts with some matrix $\mathbf{B}_0$ that is assumed to be an *approximate* inverse of the matrix $\mathbf{A}$, so that $\mathbf{B}_0 \cdot \mathbf{A}$ is approximately the identity matrix $\mathbf{1}$. Define the *residual matrix* $\mathbf{R}$ of $\mathbf{B}_0$ as

$$\mathbf{R} \equiv \mathbf{1} - \mathbf{B}_0 \cdot \mathbf{A} \tag{2.5.5}$$

which is supposed to be "small" (we will be more precise below). Note that therefore

$$\mathbf{B}_0 \cdot \mathbf{A} = \mathbf{1} - \mathbf{R} \tag{2.5.6}$$

Next consider the following formal manipulation:

$$\begin{aligned} \mathbf{A}^{-1} &= \mathbf{A}^{-1} \cdot (\mathbf{B}_0^{-1} \cdot \mathbf{B}_0) = (\mathbf{A}^{-1} \cdot \mathbf{B}_0^{-1}) \cdot \mathbf{B}_0 = (\mathbf{B}_0 \cdot \mathbf{A})^{-1} \cdot \mathbf{B}_0 \\ &= (\mathbf{1} - \mathbf{R})^{-1} \cdot \mathbf{B}_0 = (\mathbf{1} + \mathbf{R} + \mathbf{R}^2 + \mathbf{R}^3 + \cdots) \cdot \mathbf{B}_0 \end{aligned} \tag{2.5.7}$$

We can define the nth partial sum of the last expression by

$$\mathbf{B}_n \equiv (\mathbf{1} + \mathbf{R} + \cdots + \mathbf{R}^n) \cdot \mathbf{B}_0 \tag{2.5.8}$$

so that $\mathbf{B}_\infty \rightarrow \mathbf{A}^{-1}$, if the limit exists.

It now is straightforward to verify that equation (2.5.8) satisfies some interesting recurrence relations. As regards solving $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where $\mathbf{x}$ and $\mathbf{b}$ are vectors, define

$$\mathbf{x}_n \equiv \mathbf{B}_n \cdot \mathbf{b} \tag{2.5.9}$$

Then it is easy to show that

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{B}_0 \cdot (\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_n) \tag{2.5.10}$$

This is immediately recognizable as equation (2.5.4), with $-\delta\mathbf{x} = \mathbf{x}_{n+1} - \mathbf{x}_n$, and with $\mathbf{B}_0$ taking the role of $\mathbf{A}^{-1}$. We see, therefore, that equation (2.5.4) does not require that the LU decomposition of $\mathbf{A}$ be exact, but only that the implied residual $\mathbf{R}$ be small. In rough terms, if the residual is smaller than the square root of your computer's roundoff error, then after one application of equation (2.5.10) (that is, going from $\mathbf{x}_0 \equiv \mathbf{B}_0 \cdot \mathbf{b}$ to $\mathbf{x}_1$) the first neglected term, of order $\mathbf{R}^2$, will be smaller than the roundoff error. Equation (2.5.10), like equation (2.5.4), moreover, can be applied more than once, since it uses only $\mathbf{B}_0$, and not any of the higher $\mathbf{B}$'s.

A much more surprising recurrence which follows from equation (2.5.8) is one that more than *doubles* the order n at each stage:

$$\mathbf{B}_{2n+1} = 2\mathbf{B}_n - \mathbf{B}_n \cdot \mathbf{A} \cdot \mathbf{B}_n \qquad n = 0, 1, 3, 7, \ldots \tag{2.5.11}$$

Repeated application of equation (2.5.11), from a suitable starting matrix $\mathbf{B}_0$, converges *quadratically* to the unknown inverse matrix $\mathbf{A}^{-1}$ (see §9.4 for the definition of "quadratically"). Equation (2.5.11) goes by various names, including *Schultz's Method* and *Hotelling's Method*; see Pan and Reif [1] for references. In fact, equation (2.5.11) is simply the iterative Newton-Raphson method of root-finding (§9.4) applied to matrix inversion.

Before you get too excited about equation (2.5.11), however, you should notice that it involves two full matrix multiplications at each iteration. Each matrix multiplication involves N^3 adds and multiplies. But we already saw in §§2.1–2.3 that direct inversion of $\mathbf{A}$ requires only N^3 adds and N^3 multiplies *in toto*. Equation (2.5.11) is therefore practical only when special circumstances allow it to be evaluated much more rapidly than is the case for general matrices. We will meet such circumstances later, in §13.10.

In the spirit of delayed gratification, let us nevertheless pursue the two related issues: When does the series in equation (2.5.7) converge; and what is a suitable initial guess $\mathbf{B}_0$ (if, for example, an initial LU decomposition is not feasible)?

We can define the norm of a matrix as the largest amplification of length that it is able to induce on a vector,

$$\|\mathbf{R}\| \equiv \max_{\mathbf{v}\neq 0} \frac{|\mathbf{R}\cdot\mathbf{v}|}{|\mathbf{v}|} \tag{2.5.12}$$

If we let equation (2.5.7) act on some arbitrary right-hand side $\mathbf{b}$, as one wants a matrix inverse to do, it is obvious that a sufficient condition for convergence is

$$\|\mathbf{R}\| < 1 \tag{2.5.13}$$

Pan and Reif [1] point out that a suitable initial guess for $\mathbf{B}_0$ is any sufficiently small constant ϵ times the matrix transpose of $\mathbf{A}$, that is,

$$\mathbf{B}_0 = \epsilon\mathbf{A}^T \qquad \text{or} \qquad \mathbf{R} = \mathbf{1} - \epsilon\mathbf{A}^T\cdot\mathbf{A} \tag{2.5.14}$$

To see why this is so involves concepts from Chapter 11; we give here only the briefest sketch: $\mathbf{A}^T\cdot\mathbf{A}$ is a symmetric, positive definite matrix, so it has real, positive eigenvalues. In its diagonal representation, $\mathbf{R}$ takes the form

$$\mathbf{R} = \mathrm{diag}(1-\epsilon\lambda_1, 1-\epsilon\lambda_2, \ldots, 1-\epsilon\lambda_N) \tag{2.5.15}$$

where all the λ_i's are positive. Evidently any ϵ satisfying $0 < \epsilon < 2/(\max_i \lambda_i)$ will give $\|\mathbf{R}\| < 1$. It is not difficult to show that the optimal choice for ϵ, giving the most rapid convergence for equation (2.5.11), is

$$\epsilon = 2/(\max_i \lambda_i + \min_i \lambda_i) \tag{2.5.16}$$

Rarely does one know the eigenvalues of $\mathbf{A}^T\cdot\mathbf{A}$ in equation (2.5.16). Pan and Reif derive several interesting bounds, which are computable directly from $\mathbf{A}$. The following choices guarantee the convergence of $\mathbf{B}_n$ as $n\to\infty$,

$$\epsilon \le 1\Big/\sum_{j,k} a_{jk}^2 \qquad \text{or} \qquad \epsilon \le 1\Big/\left(\max_i \sum_j |a_{ij}| \times \max_j \sum_i |a_{ij}|\right) \tag{2.5.17}$$

The latter expression is truly a remarkable formula, which Pan and Reif derive by noting that the vector norm in equation (2.5.12) need not be the usual L_2 norm, but can instead be either the L_∞ (max) norm, or the L_1 (absolute value) norm. See their work for details.

Another approach, with which we have had some success, is to estimate the largest eigenvalue statistically, by calculating $s_i \equiv |\mathbf{A}\cdot\mathbf{v}_i|^2$ for several unit vector $\mathbf{v}_i$'s with randomly chosen directions in N-space. The largest eigenvalue λ can then be bounded by the maximum of $2\max s_i$ and $2N\mathrm{Var}(s_i)/\mu(s_i)$, where Var and μ denote the sample variance and mean, respectively.

CITED REFERENCES AND FURTHER READING:

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.3.4, p. 55.

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), p. 74.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.5.6, p. 183.

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 13.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.5, p. 437.

Pan, V., and Reif, J. 1985, in Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (New York: Association for Computing Machinery). [1]

2.6 Singular Value Decomposition

There exists a very powerful set of techniques for dealing with sets of equations or matrices that are either singular or else numerically very close to singular. In many cases where Gaussian elimination and LU decomposition fail to give satisfactory results, this set of techniques, known as *singular value decomposition*, or *SVD*, will diagnose for you precisely what the problem is. In some cases, SVD will not only diagnose the problem, it will also solve it, in the sense of giving you a useful numerical answer, although, as we shall see, not necessarily "the" answer that you thought you should get.

SVD is also the method of choice for solving most *linear least-squares* problems. We will outline the relevant theory in this section, but defer detailed discussion of the use of SVD in this application to Chapter 15, whose subject is the parametric modeling of data.

SVD methods are based on the following theorem of linear algebra, whose proof is beyond our scope: Any $M \times N$ matrix $\mathbf{A}$ whose number of rows M is greater than or equal to its number of columns N, can be written as the product of an $M \times N$ column-orthogonal matrix $\mathbf{U}$, an $N \times N$ diagonal matrix $\mathbf{W}$ with positive or zero elements (the *singular values*), and the transpose of an $N \times N$ orthogonal matrix $\mathbf{V}$. The various shapes of these matrices will be made clearer by the following tableau:

$$\begin{pmatrix} & & \\ & \mathbf{A} & \\ & & \\ & & \\ & & \\ & & \end{pmatrix} = \begin{pmatrix} & & \\ & \mathbf{U} & \\ & & \\ & & \\ & & \\ & & \end{pmatrix} \cdot \begin{pmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_N \end{pmatrix} \cdot \begin{pmatrix} & & \\ & \mathbf{V}^T & \\ & & \end{pmatrix} \tag{2.6.1}$$

The matrices $\mathbf{U}$ and $\mathbf{V}$ are each orthogonal in the sense that their columns are orthonormal,

$$\sum_{i=1}^{M} U_{ik} U_{in} = \delta_{kn} \qquad \begin{matrix} 1 \le k \le N \\ 1 \le n \le N \end{matrix} \tag{2.6.2}$$

$$\sum_{j=1}^{N} V_{jk} V_{jn} = \delta_{kn} \qquad \begin{matrix} 1 \le k \le N \\ 1 \le n \le N \end{matrix} \tag{2.6.3}$$

or as a tableau,

$$\begin{pmatrix} & \mathbf{U}^T & \end{pmatrix} \cdot \begin{pmatrix} & \\ & \mathbf{U} \\ & \\ & \end{pmatrix} = \begin{pmatrix} & \mathbf{V}^T & \end{pmatrix} \cdot \begin{pmatrix} & \mathbf{V} & \end{pmatrix}$$

$$= \begin{pmatrix} & \\ & \mathbf{1} \\ & \end{pmatrix} \tag{2.6.4}$$

Since $\mathbf{V}$ is square, it is also row-orthonormal, $\mathbf{V} \cdot \mathbf{V}^T = 1$.

The SVD decomposition can also be carried out when $M < N$. In this case the singular values w_j for $j = M+1, \ldots, N$ are all zero, and the corresponding columns of $\mathbf{U}$ are also zero. Equation (2.6.2) then holds only for $k, n \leq M$.

The decomposition (2.6.1) can always be done, no matter how singular the matrix is, and it is "almost" unique. That is to say, it is unique up to (i) making the same permutation of the columns of $\mathbf{U}$, elements of $\mathbf{W}$, and columns of $\mathbf{V}$ (or rows of $\mathbf{V}^T$), or (ii) forming linear combinations of any columns of $\mathbf{U}$ and $\mathbf{V}$ whose corresponding elements of $\mathbf{W}$ happen to be exactly equal. An important consequence of the permutation freedom is that for the case $M < N$, a numerical algorithm for the decomposition need not return zero w_j's for $j = M+1, \ldots, N$; the $N - M$ zero singular values can be scattered among all positions $j = 1, 2, \ldots, N$.

At the end of this section, we give a routine, `svdcmp`, that performs SVD on an arbitrary matrix $\mathbf{A}$, replacing it by $\mathbf{U}$ (they are the same shape) and giving back $\mathbf{W}$ and $\mathbf{V}$ separately. The routine `svdcmp` is based on a routine by Forsythe et al. [1], which is in turn based on the original routine of Golub and Reinsch, found, in various forms, in [2-4] and elsewhere. These references include extensive discussion of the algorithm used. As much as we dislike the use of black-box routines, we are going to ask you to accept this one, since it would take us too far afield to cover its necessary background material here. Suffice it to say that the algorithm is very stable, and that it is very unusual for it ever to misbehave. Most of the concepts that enter the algorithm (Householder reduction to bidiagonal form, diagonalization by QR procedure with shifts) will be discussed further in Chapter 11.

If you are as suspicious of black boxes as we are, you will want to verify yourself that `svdcmp` does what we say it does. That is very easy to do: Generate an arbitrary matrix $\mathbf{A}$, call the routine, and then verify by matrix multiplication that (2.6.1) and (2.6.4) are satisfied. Since these two equations are the only defining requirements for SVD, this procedure is (for the chosen $\mathbf{A}$) a complete end-to-end check.

Now let us find out what SVD is good for.

SVD of a Square Matrix

If the matrix **A** is square, $N \times N$ say, then **U**, **V**, and **W** are all square matrices of the same size. Their inverses are also trivial to compute: **U** and **V** are orthogonal, so their inverses are equal to their transposes; **W** is diagonal, so its inverse is the diagonal matrix whose elements are the reciprocals of the elements w_j. From (2.6.1) it now follows immediately that the inverse of **A** is

$$\mathbf{A}^{-1} = \mathbf{V} \cdot [\text{diag } (1/w_j)] \cdot \mathbf{U}^T \tag{2.6.5}$$

The only thing that can go wrong with this construction is for one of the w_j's to be zero, or (numerically) for it to be so small that its value is dominated by roundoff error and therefore unknowable. If more than one of the w_j's have this problem, then the matrix is even more singular. So, first of all, SVD gives you a clear diagnosis of the situation.

Formally, the *condition number* of a matrix is defined as the ratio of the largest (in magnitude) of the w_j's to the smallest of the w_j's. A matrix is singular if its condition number is infinite, and it is *ill-conditioned* if its condition number is too large, that is, if its reciprocal approaches the machine's floating-point precision (for example, less than 10^{-6} for single precision or 10^{-12} for double).

For singular matrices, the concepts of *nullspace* and *range* are important. Consider the familiar set of simultaneous equations

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.6.6}$$

where **A** is a square matrix, **b** and **x** are vectors. Equation (2.6.6) defines **A** as a linear mapping from the vector space **x** to the vector space **b**. If **A** is singular, then there is some subspace of **x**, called the nullspace, that is mapped to zero, $\mathbf{A} \cdot \mathbf{x} = 0$. The dimension of the nullspace (the number of linearly independent vectors **x** that can be found in it) is called the *nullity* of **A**.

Now, there is also some subspace of **b** that can be "reached" by **A**, in the sense that there exists some **x** which is mapped there. This subspace of **b** is called the range of **A**. The dimension of the range is called the *rank* of **A**. If **A** is nonsingular, then its range will be all of the vector space **b**, so its rank is N. If **A** is singular, then the rank will be less than N. In fact, the relevant theorem is "rank plus nullity equals N."

What has this to do with SVD? SVD explicitly constructs orthonormal bases for the nullspace and range of a matrix. Specifically, the columns of **U** whose same-numbered elements w_j are *nonzero* are an orthonormal set of basis vectors that span the range; the columns of **V** whose same-numbered elements w_j are *zero* are an orthonormal basis for the nullspace.

Now let's have another look at solving the set of simultaneous linear equations (2.6.6) in the case that **A** is singular. First, the set of *homogeneous* equations, where $\mathbf{b} = 0$, is solved immediately by SVD: Any column of **V** whose corresponding w_j is zero yields a solution.

When the vector **b** on the right-hand side is not zero, the important question is whether it lies in the range of **A** or not. If it does, then the singular set of equations *does* have a solution **x**; in fact it has more than one solution, since any vector in the nullspace (any column of **V** with a corresponding zero w_j) can be added to **x** in any linear combination.

If we want to single out one particular member of this solution-set of vectors as a representative, we might want to pick the one with the smallest length $|\mathbf{x}|^2$. Here is how to find that vector using SVD: Simply *replace* $1/w_j$ *by zero if* $w_j = 0$. (It is not very often that one gets to set $\infty = 0$!) Then compute (working from right to left)

$$\mathbf{x} = \mathbf{V} \cdot [\text{diag } (1/w_j)] \cdot (\mathbf{U}^T \cdot \mathbf{b}) \tag{2.6.7}$$

This will be the solution vector of smallest length; the columns of $\mathbf{V}$ that are in the nullspace complete the specification of the solution set.

Proof: Consider $|\mathbf{x} + \mathbf{x}'|$, where $\mathbf{x}'$ lies in the nullspace. Then, if $\mathbf{W}^{-1}$ denotes the modified inverse of $\mathbf{W}$ with some elements zeroed,

$$\begin{aligned} |\mathbf{x} + \mathbf{x}'| &= \left|\mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{x}'\right| \\ &= \left|\mathbf{V} \cdot (\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}')\right| \\ &= \left|\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}'\right| \end{aligned} \tag{2.6.8}$$

Here the first equality follows from (2.6.7), the second and third from the orthonormality of $\mathbf{V}$. If you now examine the two terms that make up the sum on the right-hand side, you will see that the first one has nonzero j components only where $w_j \neq 0$, while the second one, since $\mathbf{x}'$ is in the nullspace, has nonzero j components only where $w_j = 0$. Therefore the minimum length obtains for $\mathbf{x}' = 0$, q.e.d.

If $\mathbf{b}$ is not in the range of the singular matrix $\mathbf{A}$, then the set of equations (2.6.6) has no solution. But here is some good news: If $\mathbf{b}$ is not in the range of $\mathbf{A}$, then equation (2.6.7) can still be used to construct a "solution" vector $\mathbf{x}$. This vector $\mathbf{x}$ will not exactly solve $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. But, among all possible vectors $\mathbf{x}$, it will do the closest possible job in the least squares sense. In other words (2.6.7) finds

$$\mathbf{x} \quad \text{which minimizes} \quad r \equiv |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}| \tag{2.6.9}$$

The number r is called the *residual* of the solution.

The proof is similar to (2.6.8): Suppose we modify $\mathbf{x}$ by adding some arbitrary $\mathbf{x}'$. Then $\mathbf{A} \cdot \mathbf{x} - \mathbf{b}$ is modified by adding some $\mathbf{b}' \equiv \mathbf{A} \cdot \mathbf{x}'$. Obviously $\mathbf{b}'$ is in the range of $\mathbf{A}$. We then have

$$\begin{aligned} \left|\mathbf{A} \cdot \mathbf{x} - \mathbf{b} + \mathbf{b}'\right| &= \left|(\mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^T) \cdot (\mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b}) - \mathbf{b} + \mathbf{b}'\right| \\ &= \left|(\mathbf{U} \cdot \mathbf{W} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T - 1) \cdot \mathbf{b} + \mathbf{b}'\right| \\ &= \left|\mathbf{U} \cdot \left[(\mathbf{W} \cdot \mathbf{W}^{-1} - 1) \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{U}^T \cdot \mathbf{b}'\right]\right| \\ &= \left|(\mathbf{W} \cdot \mathbf{W}^{-1} - 1) \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{U}^T \cdot \mathbf{b}'\right| \end{aligned} \tag{2.6.10}$$

Now, $(\mathbf{W} \cdot \mathbf{W}^{-1} - 1)$ is a diagonal matrix which has nonzero j components only for $w_j = 0$, while $\mathbf{U}^T\mathbf{b}'$ has nonzero j components only for $w_j \neq 0$, since $\mathbf{b}'$ lies in the range of $\mathbf{A}$. Therefore the minimum obtains for $\mathbf{b}' = 0$, q.e.d.

Figure 2.6.1 summarizes our discussion of SVD thus far.

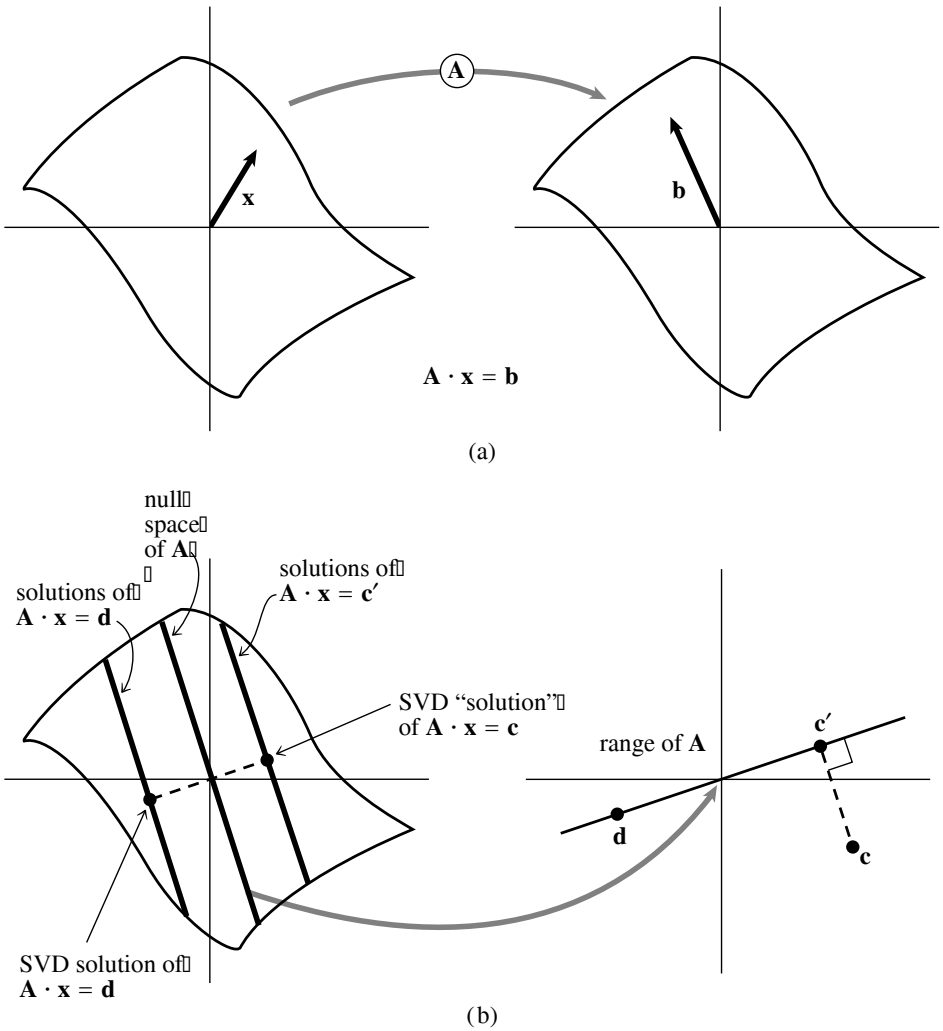

Figure 2.6.1. (a) A nonsingular matrix $\mathbf{A}$ maps a vector space into one of the same dimension. The vector $\mathbf{x}$ is mapped into $\mathbf{b}$, so that $\mathbf{x}$ satisfies the equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. (b) A singular matrix $\mathbf{A}$ maps a vector space into one of lower dimensionality, here a plane into a line, called the "range" of $\mathbf{A}$. The "nullspace" of $\mathbf{A}$ is mapped to zero. The solutions of $\mathbf{A} \cdot \mathbf{x} = \mathbf{d}$ consist of any one particular solution plus any vector in the nullspace, here forming a line parallel to the nullspace. Singular value decomposition (SVD) selects the particular solution closest to zero, as shown. The point $\mathbf{c}$ lies outside of the range of $\mathbf{A}$, so $\mathbf{A} \cdot \mathbf{x} = \mathbf{c}$ has no solution. SVD finds the least-squares best compromise solution, namely a solution of $\mathbf{A} \cdot \mathbf{x} = \mathbf{c}'$, as shown.

In the discussion since equation (2.6.6), we have been pretending that a matrix either is singular or else isn't. That is of course true analytically. Numerically, however, the far more common situation is that some of the w_j's are very small but nonzero, so that the matrix is ill-conditioned. In that case, the direct solution methods of LU decomposition or Gaussian elimination may actually give a formal solution to the set of equations (that is, a zero pivot may not be encountered); but the solution vector may have wildly large components whose algebraic cancellation, when multiplying by the matrix $\mathbf{A}$, may give a very poor approximation to the right-hand vector $\mathbf{b}$. In such cases, the solution vector $\mathbf{x}$ obtained by *zeroing* the

small w_j's and then using equation (2.6.7) is very often better (in the sense of the residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ being smaller) than *both* the direct-method solution *and* the SVD solution where the small w_j's are left nonzero.

It may seem paradoxical that this can be so, since zeroing a singular value corresponds to throwing away one linear combination of the set of equations that we are trying to solve. The resolution of the paradox is that we are throwing away precisely a combination of equations that is so corrupted by roundoff error as to be at best useless; usually it is worse than useless since it "pulls" the solution vector way off towards infinity along some direction that is almost a nullspace vector. In doing this, it compounds the roundoff problem and makes the residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ larger.

SVD cannot be applied blindly, then. You have to exercise some discretion in deciding at what threshold to zero the small w_j's, and/or you have to have some idea what size of computed residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ is acceptable.

As an example, here is a "backsubstitution" routine `svbksb` for evaluating equation (2.6.7) and obtaining a solution vector $\mathbf{x}$ from a right-hand side $\mathbf{b}$, given that the SVD of a matrix $\mathbf{A}$ has already been calculated by a call to `svdcmp`. Note that this routine presumes that *you* have already zeroed the small w_j's. It does not do this for you. If you *haven't* zeroed the small w_j's, then this routine is just as ill-conditioned as any direct method, and you are misusing SVD.

```
#include "nrutil.h"

void svbksb(float **u, float w[], float **v, int m, int n, float b[], float x[])
Solves A·X = B for a vector X, where A is specified by the arrays u[1..m][1..n], w[1..n],
v[1..n][1..n] as returned by svdcmp. m and n are the dimensions of a, and will be equal for
square matrices. b[1..m] is the input right-hand side. x[1..n] is the output solution vector.
No input quantities are destroyed, so the routine may be called sequentially with different b's.
{
    int jj,j,i;
    float s,*tmp;

    tmp=vector(1,n);
    for (j=1;j<=n;j++) {          Calculate U^T B.
        s=0.0;
        if (w[j]) {               Nonzero result only if w_j is nonzero.
            for (i=1;i<=m;i++) s += u[i][j]*b[i];
            s /= w[j];            This is the divide by w_j.
        }
        tmp[j]=s;
    }
    for (j=1;j<=n;j++) {          Matrix multiply by V to get answer.
        s=0.0;
        for (jj=1;jj<=n;jj++) s += v[j][jj]*tmp[jj];
        x[j]=s;
    }
    free_vector(tmp,1,n);
}
```

Note that a typical use of `svdcmp` and `svbksb` superficially resembles the typical use of `ludcmp` and `lubksb`: In both cases, you decompose the left-hand matrix $\mathbf{A}$ just once, and then can use the decomposition either once or many times with different right-hand sides. The crucial difference is the "editing" of the singular values before `svbksb` is called:

```
#define N ...
float wmax,wmin,**a,**u,*w,**v,*b,*x;
int i,j;
...
for(i=1;i<=N;i++)                    Copy a into u if you don't want it to be de-
    for j=1;j<=N;j++)                    stroyed.
        u[i][j]=a[i][j];
svdcmp(u,N,N,w,v);                   SVD the square matrix a.
wmax=0.0;                            Will be the maximum singular value obtained.
for(j=1;j<=N;j++) if (w[j] > wmax) wmax=w[j];
This is where we set the threshold for singular values allowed to be nonzero. The constant
is typical, but not universal. You have to experiment with your own application.
wmin=wmax*1.0e-6;
for(j=1;j<=N;j++) if (w[j] < wmin) w[j]=0.0;
svbksb(u,w,v,N,N,b,x);               Now we can backsubstitute.
```

SVD for Fewer Equations than Unknowns

If you have fewer linear equations M than unknowns N, then you are not expecting a unique solution. Usually there will be an $N - M$ dimensional family of solutions. If you want to find this whole solution space, then SVD can readily do the job.

The SVD decomposition will yield $N - M$ zero or negligible w_j's, since $M < N$. There may be additional zero w_j's from any degeneracies in your M equations. Be sure that you find this many small w_j's, and zero them before calling `svbksb`, which will give you the particular solution vector $\mathbf{x}$. As before, the columns of $\mathbf{V}$ corresponding to zeroed w_j's are the basis vectors whose linear combinations, added to the particular solution, span the solution space.

SVD for More Equations than Unknowns

This situation will occur in Chapter 15, when we wish to find the least-squares solution to an overdetermined set of linear equations. In tableau, the equations to be solved are

$$\begin{pmatrix} & & \\ & \mathbf{A} & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{x} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{b} \\ \\ \end{pmatrix} \tag{2.6.11}$$

The proofs that we gave above for the square case apply without modification to the case of more equations than unknowns. The least-squares solution vector $\mathbf{x}$ is

given by (2.6.7), which, with nonsquare matrices, looks like this,

$$\begin{pmatrix} \\ \mathbf{x} \\ \\ \end{pmatrix} = \begin{pmatrix} & & \\ & \mathbf{V} & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \mathrm{diag}(1/w_j) \\ \\ \end{pmatrix} \cdot \begin{pmatrix} & & \\ & \mathbf{U}^T & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{b} \\ \\ \end{pmatrix} \tag{2.6.12}$$

In general, the matrix $\mathbf{W}$ will not be singular, and no w_j's will need to be set to zero. Occasionally, however, there might be column degeneracies in $\mathbf{A}$. In this case you will need to zero some small w_j values after all. The corresponding column in $\mathbf{V}$ gives the linear combination of $\mathbf{x}$'s that is then ill-determined even by the supposedly overdetermined set.

Sometimes, although you do not need to zero any w_j's for *computational* reasons, you may nevertheless want to take note of any that are unusually small: Their corresponding columns in $\mathbf{V}$ are linear combinations of $\mathbf{x}$'s which are insensitive to your data. In fact, you may then wish to zero these w_j's, to reduce the number of free parameters in the fit. These matters are discussed more fully in Chapter 15.

Constructing an Orthonormal Basis

Suppose that you have N vectors in an M-dimensional vector space, with $N \leq M$. Then the N vectors span some subspace of the full vector space. Often you want to construct an orthonormal set of N vectors that span the same subspace. The textbook way to do this is by Gram-Schmidt orthogonalization, starting with one vector and then expanding the subspace one dimension at a time. Numerically, however, because of the build-up of roundoff errors, naive Gram-Schmidt orthogonalization is *terrible*.

The right way to construct an orthonormal basis for a subspace is by SVD: Form an $M \times N$ matrix $\mathbf{A}$ whose N columns are your vectors. Run the matrix through `svdcmp`. The columns of the matrix $\mathbf{U}$ (which in fact replaces $\mathbf{A}$ on output from `svdcmp`) are your desired orthonormal basis vectors.

You might also want to check the output w_j's for zero values. If any occur, then the spanned subspace was not, in fact, N dimensional; the columns of $\mathbf{U}$ corresponding to zero w_j's should be discarded from the orthonormal basis set.

(QR factorization, discussed in §2.10, also constructs an orthonormal basis, see [5].)

Approximation of Matrices

Note that equation (2.6.1) can be rewritten to express any matrix A_{ij} as a sum of outer products of columns of $\mathbf{U}$ and rows of $\mathbf{V}^T$, with the "weighting factors" being the singular values w_j,

$$A_{ij} = \sum_{k=1}^{N} w_k \, U_{ik} V_{jk} \tag{2.6.13}$$

If you ever encounter a situation where *most* of the singular values w_j of a matrix **A** are very small, then **A** will be well-approximated by only a few terms in the sum (2.6.13). This means that you have to store only a few columns of **U** and **V** (the same k ones) and you will be able to recover, with good accuracy, the whole matrix.

Note also that it is very efficient to multiply such an approximated matrix by a vector **x**: You just dot **x** with each of the stored columns of **V**, multiply the resulting scalar by the corresponding w_k, and accumulate that multiple of the corresponding column of **U**. If your matrix is approximated by a small number K of singular values, then this computation of $\mathbf{A} \cdot \mathbf{x}$ takes only about $K(M+N)$ multiplications, instead of MN for the full matrix.

SVD Algorithm

Here is the algorithm for constructing the singular value decomposition of any matrix. See §11.2–§11.3, and also [4-5], for discussion relating to the underlying method.

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

void svdcmp(float **a, int m, int n, float w[], float **v)
Given a matrix a[1..m][1..n], this routine computes its singular value decomposition, A =
U·W·V^T. The matrix U replaces a on output. The diagonal matrix of singular values W is out-
put as a vector w[1..n]. The matrix V (not the transpose V^T) is output as v[1..n][1..n].
{
    float pythag(float a, float b);
    int flag,i,its,j,jj,k,l,nm;
    float anorm,c,f,g,h,s,scale,x,y,z,*rv1;

    rv1=vector(1,n);
    g=scale=anorm=0.0;                    Householder reduction to bidiagonal form.
    for (i=1;i<=n;i++) {
        l=i+1;
        rv1[i]=scale*g;
        g=s=scale=0.0;
        if (i <= m) {
            for (k=i;k<=m;k++) scale += fabs(a[k][i]);
            if (scale) {
                for (k=i;k<=m;k++) {
                    a[k][i] /= scale;
                    s += a[k][i]*a[k][i];
                }
                f=a[i][i];
                g = -SIGN(sqrt(s),f);
                h=f*g-s;
                a[i][i]=f-g;
                for (j=l;j<=n;j++) {
                    for (s=0.0,k=i;k<=m;k++) s += a[k][i]*a[k][j];
                    f=s/h;
                    for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
                }
                for (k=i;k<=m;k++) a[k][i] *= scale;
            }
        }
        w[i]=scale *g;
        g=s=scale=0.0;
        if (i <= m && i != n) {
            for (k=l;k<=n;k++) scale += fabs(a[i][k]);
            if (scale) {
```

```
            for (k=l;k<=n;k++) {
                a[i][k] /= scale;
                s += a[i][k]*a[i][k];
            }
            f=a[i][l];
            g = -SIGN(sqrt(s),f);
            h=f*g-s;
            a[i][l]=f-g;
            for (k=l;k<=n;k++) rv1[k]=a[i][k]/h;
            for (j=l;j<=m;j++) {
                for (s=0.0,k=l;k<=n;k++) s += a[j][k]*a[i][k];
                for (k=l;k<=n;k++) a[j][k] += s*rv1[k];
            }
            for (k=l;k<=n;k++) a[i][k] *= scale;
        }
    }
    anorm=FMAX(anorm,(fabs(w[i])+fabs(rv1[i])));
}
for (i=n;i>=1;i--) {                Accumulation of right-hand transformations.
    if (i < n) {
        if (g) {
            for (j=l;j<=n;j++)      Double division to avoid possible underflow.
                v[j][i]=(a[i][j]/a[i][l])/g;
            for (j=l;j<=n;j++) {
                for (s=0.0,k=l;k<=n;k++) s += a[i][k]*v[k][j];
                for (k=l;k<=n;k++) v[k][j] += s*v[k][i];
            }
        }
        for (j=l;j<=n;j++) v[i][j]=v[j][i]=0.0;
    }
    v[i][i]=1.0;
    g=rv1[i];
    l=i;
}
for (i=IMIN(m,n);i>=1;i--) {        Accumulation of left-hand transformations.
    l=i+1;
    g=w[i];
    for (j=l;j<=n;j++) a[i][j]=0.0;
    if (g) {
        g=1.0/g;
        for (j=l;j<=n;j++) {
            for (s=0.0,k=l;k<=m;k++) s += a[k][i]*a[k][j];
            f=(s/a[i][i])*g;
            for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
        }
        for (j=i;j<=m;j++) a[j][i] *= g;
    } else for (j=i;j<=m;j++) a[j][i]=0.0;
    ++a[i][i];
}
for (k=n;k>=1;k--) {                Diagonalization of the bidiagonal form: Loop over
    for (its=1;its<=30;its++) {         singular values, and over allowed iterations.
        flag=1;
        for (l=k;l>=1;l--) {        Test for splitting.
            nm=l-1;                 Note that rv1[1] is always zero.
            if ((float)(fabs(rv1[l])+anorm) == anorm) {
                flag=0;
                break;
            }
            if ((float)(fabs(w[nm])+anorm) == anorm) break;
        }
        if (flag) {
            c=0.0;                  Cancellation of rv1[l], if l > 1.
            s=1.0;
            for (i=l;i<=k;i++) {
```

```
            f=s*rv1[i];
            rv1[i]=c*rv1[i];
            if ((float)(fabs(f)+anorm) == anorm) break;
            g=w[i];
            h=pythag(f,g);
            w[i]=h;
            h=1.0/h;
            c=g*h;
            s = -f*h;
            for (j=1;j<=m;j++) {
                y=a[j][nm];
                z=a[j][i];
                a[j][nm]=y*c+z*s;
                a[j][i]=z*c-y*s;
            }
        }
    }
    z=w[k];
    if (l == k) {                Convergence.
        if (z < 0.0) {           Singular value is made nonnegative.
            w[k] = -z;
            for (j=1;j<=n;j++) v[j][k] = -v[j][k];
        }
        break;
    }
    if (its == 30) nrerror("no convergence in 30 svdcmp iterations");
    x=w[l];                      Shift from bottom 2-by-2 minor.
    nm=k-1;
    y=w[nm];
    g=rv1[nm];
    h=rv1[k];
    f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y);
    g=pythag(f,1.0);
    f=((x-z)*(x+z)+h*((y/(f+SIGN(g,f)))-h))/x;
    c=s=1.0;                     Next QR transformation:
    for (j=l;j<=nm;j++) {
        i=j+1;
        g=rv1[i];
        y=w[i];
        h=s*g;
        g=c*g;
        z=pythag(f,h);
        rv1[j]=z;
        c=f/z;
        s=h/z;
        f=x*c+g*s;
        g = g*c-x*s;
        h=y*s;
        y *= c;
        for (jj=1;jj<=n;jj++) {
            x=v[jj][j];
            z=v[jj][i];
            v[jj][j]=x*c+z*s;
            v[jj][i]=z*c-x*s;
        }
        z=pythag(f,h);
        w[j]=z;                  Rotation can be arbitrary if z = 0.
        if (z) {
            z=1.0/z;
            c=f*z;
            s=h*z;
        }
        f=c*g+s*y;
        x=c*y-s*g;
```

```
                for (jj=1;jj<=m;jj++) {
                    y=a[jj][j];
                    z=a[jj][i];
                    a[jj][j]=y*c+z*s;
                    a[jj][i]=z*c-y*s;
                }
            }
            rv1[l]=0.0;
            rv1[k]=f;
            w[k]=x;
        }
    }
    free_vector(rv1,1,n);
}
```

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

float pythag(float a, float b)
Computes (a^2 + b^2)^1/2 without destructive underflow or overflow.
{
    float absa,absb;
    absa=fabs(a);
    absb=fabs(b);
    if (absa > absb) return absa*sqrt(1.0+SQR(absb/absa));
    else return (absb == 0.0 ? 0.0 : absb*sqrt(1.0+SQR(absa/absb)));
}
```

(Double precision versions of `svdcmp`, `svbksb`, and `pythag`, named `dsvdcmp`, `dsvbksb`, and `dpythag`, are used by the routine `ratlsq` in §5.13. You can easily make the conversions, or else get the converted routines from the *Numerical Recipes* diskette.)

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §8.3 and Chapter 12.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 18.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I.10 by G.H. Golub and C. Reinsch. [2]

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.), Chapter 11. [3]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.7. [4]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §5.2.6. [5]

2.7 Sparse Linear Systems

A system of linear equations is called *sparse* if only a relatively small number of its matrix elements a_{ij} are nonzero. It is wasteful to use general methods of linear algebra on such problems, because most of the $O(N^3)$ arithmetic operations devoted to solving the set of equations or inverting the matrix involve zero operands. Furthermore, you might wish to work problems so large as to tax your available memory space, and it is wasteful to reserve storage for unfruitful zero elements. Note that there are two distinct (and not always compatible) goals for any sparse matrix method: saving time and/or saving space.

We have already considered one archetypal sparse form in §2.4, the band diagonal matrix. In the tridiagonal case, e.g., we saw that it was possible to save both time (order N instead of N^3) and space (order N instead of N^2). The method of solution was not different in principle from the general method of LU decomposition; it was just applied cleverly, and with due attention to the bookkeeping of zero elements. Many practical schemes for dealing with sparse problems have this same character. They are fundamentally decomposition schemes, or else elimination schemes akin to Gauss-Jordan, but carefully optimized so as to minimize the number of so-called *fill-ins*, initially zero elements which must become nonzero during the solution process, and for which storage must be reserved.

Direct methods for solving sparse equations, then, depend crucially on the precise pattern of sparsity of the matrix. Patterns that occur frequently, or that are useful as way-stations in the reduction of more general forms, already have special names and special methods of solution. We do not have space here for any detailed review of these. References listed at the end of this section will furnish you with an "in" to the specialized literature, and the following list of buzz words (and Figure 2.7.1) will at least let you hold your own at cocktail parties:

- tridiagonal
- band diagonal (or banded) with bandwidth M
- band triangular
- block diagonal
- block tridiagonal
- block triangular
- cyclic banded
- singly (or doubly) bordered block diagonal
- singly (or doubly) bordered block triangular
- singly (or doubly) bordered band diagonal
- singly (or doubly) bordered band triangular
- other (!)

You should also be aware of some of the special sparse forms that occur in the solution of partial differential equations in two or more dimensions. See Chapter 19.

If your particular pattern of sparsity is not a simple one, then you may wish to try an *analyze/factorize/operate* package, which automates the procedure of figuring out how fill-ins are to be minimized. The *analyze* stage is done once only for each pattern of sparsity. The *factorize* stage is done once for each particular matrix that fits the pattern. The *operate* stage is performed once for each right-hand side to

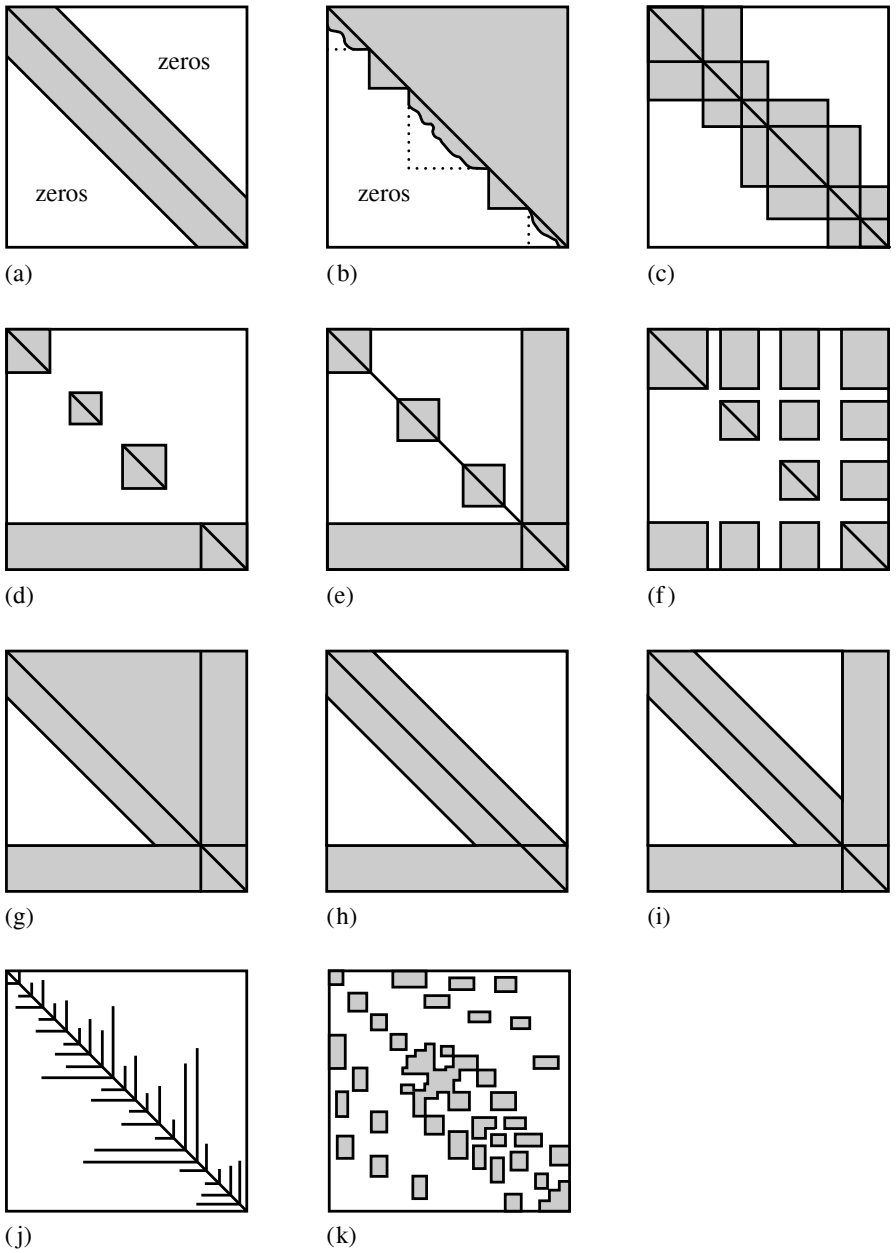

Figure 2.7.1. Some standard forms for sparse matrices. (a) Band diagonal; (b) block triangular; (c) block tridiagonal; (d) singly bordered block diagonal; (e) doubly bordered block diagonal; (f) singly bordered block triangular; (g) bordered band-triangular; (h) and (i) singly and doubly bordered band diagonal; (j) and (k) other! (after Tewarson) [1].

be used with the particular matrix. Consult [2,3] for references on this. The NAG library [4] has an analyze/factorize/operate capability. A substantial collection of routines for sparse matrix calculation is also available from IMSL [5] as the *Yale Sparse Matrix Package* [6].

You should be aware that the special order of interchanges and eliminations,

prescribed by a sparse matrix method so as to minimize fill-ins and arithmetic operations, generally acts to decrease the method's numerical stability as compared to, e.g., regular LU decomposition with pivoting. Scaling your problem so as to make its nonzero matrix elements have comparable magnitudes (if you can do it) will sometimes ameliorate this problem.

In the remainder of this section, we present some concepts which are applicable to some general classes of sparse matrices, and which do not necessarily depend on details of the pattern of sparsity.

Sherman-Morrison Formula

Suppose that you have already obtained, by herculean effort, the inverse matrix $\mathbf{A}^{-1}$ of a square matrix $\mathbf{A}$. Now you want to make a "small" change in $\mathbf{A}$, for example change one element a_{ij}, or a few elements, or one row, or one column. Is there any way of calculating the corresponding change in $\mathbf{A}^{-1}$ without repeating your difficult labors? Yes, if your change is of the form

$$\mathbf{A} \rightarrow (\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \tag{2.7.1}$$

for some vectors $\mathbf{u}$ and $\mathbf{v}$. If $\mathbf{u}$ is a unit vector $\mathbf{e}_i$, then (2.7.1) adds the components of $\mathbf{v}$ to the ith row. (Recall that $\mathbf{u} \otimes \mathbf{v}$ is a matrix whose i, jth element is the product of the ith component of $\mathbf{u}$ and the jth component of $\mathbf{v}$.) If $\mathbf{v}$ is a unit vector $\mathbf{e}_j$, then (2.7.1) adds the components of $\mathbf{u}$ to the jth column. If both $\mathbf{u}$ and $\mathbf{v}$ are proportional to unit vectors $\mathbf{e}_i$ and $\mathbf{e}_j$ respectively, then a term is added only to the element a_{ij}.

The *Sherman-Morrison* formula gives the inverse $(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1}$, and is derived briefly as follows:

$$\begin{aligned}
(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1} &= (\mathbf{1} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v})^{-1} \cdot \mathbf{A}^{-1} \\
&= (\mathbf{1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} - \ldots) \cdot \mathbf{A}^{-1} \\
&= \mathbf{A}^{-1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} \left(1 - \lambda + \lambda^2 - \ldots\right) \\
&= \mathbf{A}^{-1} - \frac{(\mathbf{A}^{-1} \cdot \mathbf{u}) \otimes (\mathbf{v} \cdot \mathbf{A}^{-1})}{1 + \lambda}
\end{aligned} \tag{2.7.2}$$

where

$$\lambda \equiv \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \tag{2.7.3}$$

The second line of (2.7.2) is a formal power series expansion. In the third line, the associativity of outer and inner products is used to factor out the scalars λ.

The use of (2.7.2) is this: Given $\mathbf{A}^{-1}$ and the vectors $\mathbf{u}$ and $\mathbf{v}$, we need only perform two matrix multiplications and a vector dot product,

$$\mathbf{z} \equiv \mathbf{A}^{-1} \cdot \mathbf{u} \qquad \mathbf{w} \equiv (\mathbf{A}^{-1})^T \cdot \mathbf{v} \qquad \lambda = \mathbf{v} \cdot \mathbf{z} \tag{2.7.4}$$

to get the desired change in the inverse

$$\mathbf{A}^{-1} \quad \rightarrow \quad \mathbf{A}^{-1} - \frac{\mathbf{z} \otimes \mathbf{w}}{1 + \lambda} \tag{2.7.5}$$

The whole procedure requires only $3N^2$ multiplies and a like number of adds (an even smaller number if **u** or **v** is a unit vector).

The Sherman-Morrison formula can be directly applied to a class of sparse problems. If you already have a fast way of calculating the inverse of **A** (e.g., a tridiagonal matrix, or some other standard sparse form), then (2.7.4)–(2.7.5) allow you to build up to your related but more complicated form, adding for example a row or column at a time. Notice that you can apply the Sherman-Morrison formula more than once successively, using at each stage the most recent update of $\mathbf{A}^{-1}$ (equation 2.7.5). Of course, if you have to modify *every* row, then you are back to an N^3 method. The constant in front of the N^3 is only a few times worse than the better direct methods, but you have deprived yourself of the stabilizing advantages of pivoting — so be careful.

For some other sparse problems, the Sherman-Morrison formula cannot be directly applied for the simple reason that storage of the whole inverse matrix $\mathbf{A}^{-1}$ is not feasible. If you want to add only a single correction of the form $\mathbf{u} \otimes \mathbf{v}$, and solve the linear system

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \cdot \mathbf{x} = \mathbf{b} \tag{2.7.6}$$

then you proceed as follows. Using the fast method that is presumed available for the matrix **A**, solve the two auxiliary problems

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \qquad \mathbf{A} \cdot \mathbf{z} = \mathbf{u} \tag{2.7.7}$$

for the vectors **y** and **z**. In terms of these,

$$\mathbf{x} = \mathbf{y} - \left[\frac{\mathbf{v} \cdot \mathbf{y}}{1 + (\mathbf{v} \cdot \mathbf{z})} \right] \mathbf{z} \tag{2.7.8}$$

as we see by multiplying (2.7.2) on the right by **b**.

Cyclic Tridiagonal Systems

So-called *cyclic tridiagonal systems* occur quite frequently, and are a good example of how to use the Sherman-Morrison formula in the manner just described. The equations have the form

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \beta \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ \alpha & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.7.9}$$

This is a tridiagonal system, except for the matrix elements α and β in the corners. Forms like this are typically generated by finite-differencing differential equations with periodic boundary conditions (§19.4).

We use the Sherman-Morrison formula, treating the system as tridiagonal plus a correction. In the notation of equation (2.7.6), define vectors **u** and **v** to be

$$\mathbf{u} = \begin{bmatrix} \gamma \\ 0 \\ \vdots \\ 0 \\ \alpha \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ \beta/\gamma \end{bmatrix} \tag{2.7.10}$$

Here γ is arbitrary for the moment. Then the matrix $\mathbf{A}$ is the tridiagonal part of the matrix in (2.7.9), with two terms modified:

$$b_1' = b_1 - \gamma, \qquad b_N' = b_N - \alpha\beta/\gamma \tag{2.7.11}$$

We now solve equations (2.7.7) with the standard tridiagonal algorithm, and then get the solution from equation (2.7.8).

The routine `cyclic` below implements this algorithm. We choose the arbitrary parameter $\gamma = -b_1$ to avoid loss of precision by subtraction in the first of equations (2.7.11). In the unlikely event that this causes loss of precision in the second of these equations, you can make a different choice.

```
#include "nrutil.h"

void cyclic(float a[], float b[], float c[], float alpha, float beta,
    float r[], float x[], unsigned long n)
Solves for a vector x[1..n] the "cyclic" set of linear equations given by equation (2.7.9). a,
b, c, and r are input vectors, all dimensioned as [1..n], while alpha and beta are the corner
entries in the matrix. The input is not modified.
{
    void tridag(float a[], float b[], float c[], float r[], float u[],
        unsigned long n);
    unsigned long i;
    float fact,gamma,*bb,*u,*z;

    if (n <= 2) nrerror("n too small in cyclic");
    bb=vector(1,n);
    u=vector(1,n);
    z=vector(1,n);
    gamma = -b[1];                          Avoid subtraction error in forming bb[1].
    bb[1]=b[1]-gamma;                       Set up the diagonal of the modified tridi-
    bb[n]=b[n]-alpha*beta/gamma;                agonal system.
    for (i=2;i<n;i++) bb[i]=b[i];
    tridag(a,bb,c,r,x,n);                   Solve A · x = r.
    u[1]=gamma;                             Set up the vector u.
    u[n]=alpha;
    for (i=2;i<n;i++) u[i]=0.0;
    tridag(a,bb,c,u,z,n);                   Solve A · z = u.
    fact=(x[1]+beta*x[n]/gamma)/            Form v · x/(1 + v · z).
        (1.0+z[1]+beta*z[n]/gamma);
    for (i=1;i<=n;i++) x[i] -= fact*z[i];   Now get the solution vector x.
    free_vector(z,1,n);
    free_vector(u,1,n);
    free_vector(bb,1,n);
}
```

Woodbury Formula

If you want to add more than a single correction term, then you cannot use (2.7.8) repeatedly, since without storing a new $\mathbf{A}^{-1}$ you will not be able to solve the auxiliary problems (2.7.7) efficiently after the first step. Instead, you need the *Woodbury formula*, which is the block-matrix version of the Sherman-Morrison formula,

$$\begin{aligned}(\mathbf{A} + \mathbf{U}\cdot\mathbf{V}^T)^{-1} \\ = \mathbf{A}^{-1} \; - \; \left[\mathbf{A}^{-1}\cdot\mathbf{U}\cdot(\mathbf{1} + \mathbf{V}^T\cdot\mathbf{A}^{-1}\cdot\mathbf{U})^{-1}\cdot\mathbf{V}^T\cdot\mathbf{A}^{-1}\right]\end{aligned} \tag{2.7.12}$$

Here A is, as usual, an $N \times N$ matrix, while $\mathbf{U}$ and $\mathbf{V}$ are $N \times P$ matrices with $P < N$ and usually $P \ll N$. The inner piece of the correction term may become clearer if written as the tableau,

$$\begin{bmatrix} & & \\ & \mathbf{U} & \\ & & \end{bmatrix} \cdot \left[\mathbf{1} + \mathbf{V}^T \cdot \mathbf{A}^{-1} \cdot \mathbf{U} \right]^{-1} \cdot \begin{bmatrix} & & \\ & \mathbf{V}^T & \\ & & \end{bmatrix} \tag{2.7.13}$$

where you can see that the matrix whose inverse is needed is only $P \times P$ rather than $N \times N$.

The relation between the Woodbury formula and successive applications of the Sherman-Morrison formula is now clarified by noting that, if $\mathbf{U}$ is the matrix formed by columns out of the P vectors $\mathbf{u}_1, \ldots, \mathbf{u}_P$, and $\mathbf{V}$ is the matrix formed by columns out of the P vectors $\mathbf{v}_1, \ldots, \mathbf{v}_P$,

$$\mathbf{U} \equiv \begin{bmatrix} & & \\ \mathbf{u}_1 & \cdots & \mathbf{u}_P \\ & & \end{bmatrix} \qquad \mathbf{V} \equiv \begin{bmatrix} & & \\ \mathbf{v}_1 & \cdots & \mathbf{v}_P \\ & & \end{bmatrix} \tag{2.7.14}$$

then two ways of expressing the same correction to $\mathbf{A}$ are

$$\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{v}_k \right) = (\mathbf{A} + \mathbf{U} \cdot \mathbf{V}^T) \tag{2.7.15}$$

(Note that the subscripts on $\mathbf{u}$ and $\mathbf{v}$ do *not* denote components, but rather distinguish the different column vectors.)

Equation (2.7.15) reveals that, if you have $\mathbf{A}^{-1}$ in storage, then you can either make the P corrections in one fell swoop by using (2.7.12), inverting a $P \times P$ matrix, or else make them by applying (2.7.5) P successive times.

If you don't have storage for $\mathbf{A}^{-1}$, then you *must* use (2.7.12) in the following way: To solve the linear equation

$$\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{v}_k \right) \cdot \mathbf{x} = \mathbf{b} \tag{2.7.16}$$

first solve the P auxiliary problems

$$\begin{aligned} \mathbf{A} \cdot \mathbf{z}_1 &= \mathbf{u}_1 \\ \mathbf{A} \cdot \mathbf{z}_2 &= \mathbf{u}_2 \\ &\cdots \\ \mathbf{A} \cdot \mathbf{z}_P &= \mathbf{u}_P \end{aligned} \tag{2.7.17}$$

and construct the matrix $\mathbf{Z}$ by columns from the $\mathbf{z}$'s obtained,

$$\mathbf{Z} \equiv \begin{bmatrix} & & \\ \mathbf{z}_1 & \cdots & \mathbf{z}_P \\ & & \end{bmatrix} \tag{2.7.18}$$

Next, do the $P \times P$ matrix inversion

$$\mathbf{H} \equiv (\mathbf{1} + \mathbf{V}^T \cdot \mathbf{Z})^{-1} \tag{2.7.19}$$

Finally, solve the one further auxiliary problem

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \tag{2.7.20}$$

In terms of these quantities, the solution is given by

$$\mathbf{x} = \mathbf{y} - \mathbf{Z} \cdot \left[\mathbf{H} \cdot (\mathbf{V}^T \cdot \mathbf{y}) \right] \tag{2.7.21}$$

Inversion by Partitioning

Once in a while, you will encounter a matrix (not even necessarily sparse) that can be inverted efficiently by partitioning. Suppose that the $N \times N$ matrix $\mathbf{A}$ is partitioned into

$$\mathbf{A} = \begin{bmatrix} \mathbf{P} & \mathbf{Q} \\ \mathbf{R} & \mathbf{S} \end{bmatrix} \tag{2.7.22}$$

where $\mathbf{P}$ and $\mathbf{S}$ are square matrices of size $p \times p$ and $s \times s$ respectively ($p + s = N$). The matrices $\mathbf{Q}$ and $\mathbf{R}$ are not necessarily square, and have sizes $p \times s$ and $s \times p$, respectively.

If the inverse of $\mathbf{A}$ is partitioned in the same manner,

$$\mathbf{A}^{-1} = \begin{bmatrix} \widetilde{\mathbf{P}} & \widetilde{\mathbf{Q}} \\ \widetilde{\mathbf{R}} & \widetilde{\mathbf{S}} \end{bmatrix} \tag{2.7.23}$$

then $\widetilde{\mathbf{P}}$, $\widetilde{\mathbf{Q}}$, $\widetilde{\mathbf{R}}$, $\widetilde{\mathbf{S}}$, which have the same sizes as $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{S}$, respectively, can be found by either the formulas

$$\begin{aligned} \widetilde{\mathbf{P}} &= (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \\ \widetilde{\mathbf{Q}} &= -(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \cdot (\mathbf{Q} \cdot \mathbf{S}^{-1}) \\ \widetilde{\mathbf{R}} &= -(\mathbf{S}^{-1} \cdot \mathbf{R}) \cdot (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \\ \widetilde{\mathbf{S}} &= \mathbf{S}^{-1} + (\mathbf{S}^{-1} \cdot \mathbf{R}) \cdot (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \cdot (\mathbf{Q} \cdot \mathbf{S}^{-1}) \end{aligned} \tag{2.7.24}$$

or else by the equivalent formulas

$$\begin{aligned} \widetilde{\mathbf{P}} &= \mathbf{P}^{-1} + (\mathbf{P}^{-1} \cdot \mathbf{Q}) \cdot (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \cdot (\mathbf{R} \cdot \mathbf{P}^{-1}) \\ \widetilde{\mathbf{Q}} &= -(\mathbf{P}^{-1} \cdot \mathbf{Q}) \cdot (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \\ \widetilde{\mathbf{R}} &= -(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \cdot (\mathbf{R} \cdot \mathbf{P}^{-1}) \\ \widetilde{\mathbf{S}} &= (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \end{aligned} \tag{2.7.25}$$

The parentheses in equations (2.7.24) and (2.7.25) highlight repeated factors that you may wish to compute only once. (Of course, by associativity, you can instead do the matrix multiplications in any order you like.) The choice between using equation (2.7.24) and (2.7.25) depends on whether you want $\widetilde{\mathbf{P}}$ or $\widetilde{\mathbf{S}}$ to have the simpler formula; or on whether the repeated expression $(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1}$ is easier

to calculate than the expression $(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1}$; or on the relative sizes of $\mathbf{P}$ and $\mathbf{S}$; or on whether $\mathbf{P}^{-1}$ or $\mathbf{S}^{-1}$ is already known.

Another sometimes useful formula is for the determinant of the partitioned matrix,

$$\det \mathbf{A} = \det \mathbf{P} \det(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q}) = \det \mathbf{S} \det(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R}) \qquad (2.7.26)$$

Indexed Storage of Sparse Matrices

We have already seen (§2.4) that tri- or band-diagonal matrices can be stored in a compact format that allocates storage only to elements which can be nonzero, plus perhaps a few wasted locations to make the bookkeeping easier. What about more general sparse matrices? When a sparse matrix of dimension $N \times N$ contains only a few times N nonzero elements (a typical case), it is surely inefficient — and often physically impossible — to allocate storage for all N^2 elements. Even if one did allocate such storage, it would be inefficient or prohibitive in machine time to loop over all of it in search of nonzero elements.

Obviously some kind of indexed storage scheme is required, one that stores only nonzero matrix elements, along with sufficient auxiliary information to determine where an element logically belongs and how the various elements can be looped over in common matrix operations. Unfortunately, there is no one standard scheme in general use. Knuth [7] describes one method. The Yale Sparse Matrix Package [6] and ITPACK [8] describe several other methods. For most applications, we favor the storage scheme used by PCGPACK [9], which is almost the same as that described by Bentley [10], and also similar to one of the Yale Sparse Matrix Package methods. The advantage of this scheme, which can be called *row-indexed sparse storage mode*, is that it requires storage of only about two times the number of nonzero matrix elements. (Other methods can require as much as three or five times.) For simplicity, we will treat only the case of square matrices, which occurs most frequently in practice.

To represent a matrix $\mathbf{A}$ of dimension $N \times N$, the row-indexed scheme sets up two one-dimensional arrays, call them `sa` and `ija`. The first of these stores matrix element values in single or double precision as desired; the second stores integer values. The storage rules are:

- The first N locations of `sa` store $\mathbf{A}$'s diagonal matrix elements, in order. (Note that diagonal elements are stored even if they are zero; this is at most a slight storage inefficiency, since diagonal elements are nonzero in most realistic applications.)
- Each of the first N locations of `ija` stores the index of the array `sa` that contains the first *off-diagonal* element of the corresponding row of the matrix. (If there are no off-diagonal elements for that row, it is one greater than the index in `sa` of the most recently stored element of a previous row.)
- Location 1 of `ija` is always equal to $N + 2$. (It can be read to determine N.)
- Location $N + 1$ of `ija` is one greater than the index in `sa` of the last off-diagonal element of the last row. (It can be read to determine the number of nonzero elements in the matrix, or the number of elements in the arrays `sa` and `ija`.) Location $N + 1$ of `sa` is not used and can be set arbitrarily.
- Entries in `sa` at locations $\geq N + 2$ contain $\mathbf{A}$'s off-diagonal values, ordered by rows and, within each row, ordered by columns.
- Entries in `ija` at locations $\geq N+2$ contain the column number of the corresponding element in `sa`.

While these rules seem arbitrary at first sight, they result in a rather elegant storage scheme. As an example, consider the matrix

$$\begin{bmatrix} 3. & 0. & 1. & 0. & 0. \\ 0. & 4. & 0. & 0. & 0. \\ 0. & 7. & 5. & 9. & 0. \\ 0. & 0. & 0. & 0. & 2. \\ 0. & 0. & 0. & 6. & 5. \end{bmatrix} \qquad (2.7.27)$$

In row-indexed compact storage, matrix (2.7.27) is represented by the two arrays of length 11, as follows

index k	1	2	3	4	5	6	7	8	9	10	11
ija[k]	7	8	8	10	11	12	3	2	4	5	4
sa[k]	3.	4.	5.	0.	5.	x	1.	7.	9.	2.	6.

(2.7.28)

Here x is an arbitrary value. Notice that, according to the storage rules, the value of N (namely 5) is `ija[1]-2`, and the length of each array is `ija[ija[1]-1]-1`, namely 11. The diagonal element in row `i` is `sa[i]`, and the off-diagonal elements in that row are in `sa[k]` where `k` loops from `ija[i]` to `ija[i+1]-1`, if the upper limit is greater or equal to the lower one (as in C's `for` loops).

Here is a routine, `sprsin`, that converts a matrix from full storage mode into row-indexed sparse storage mode, throwing away any elements that are less than a specified threshold. Of course, the principal use of sparse storage mode is for matrices whose full storage mode won't fit into your machine at all; then you have to generate them directly into sparse format. Nevertheless `sprsin` is useful as a precise algorithmic definition of the storage scheme, for subscale testing of large problems, and for the case where execution time, rather than storage, furnishes the impetus to sparse storage.

```
#include <math.h>

void sprsin(float **a, int n, float thresh, unsigned long nmax, float sa[],
    unsigned long ija[])
Converts a square matrix a[1..n][1..n] into row-indexed sparse storage mode. Only ele-
ments of a with magnitude ≥thresh are retained. Output is in two linear arrays with dimen-
sion nmax (an input parameter): sa[1..] contains array values, indexed by ija[1..]. The
number of elements filled of sa and ija on output are both ija[ija[1]-1]-1 (see text).
{
    void nrerror(char error_text[]);
    int i,j;
    unsigned long k;

    for (j=1;j<=n;j++) sa[j]=a[j][j];        Store diagonal elements.
    ija[1]=n+2;                              Index to 1st row off-diagonal element, if any.
    k=n+1;
    for (i=1;i<=n;i++) {                     Loop over rows.
        for (j=1;j<=n;j++) {                 Loop over columns.
            if (fabs(a[i][j]) >= thresh && i != j) {
                if (++k > nmax) nrerror("sprsin: nmax too small");
                sa[k]=a[i][j];               Store off-diagonal elements and their columns.
                ija[k]=j;
            }
        }
        ija[i+1]=k+1;                        As each row is completed, store index to
    }                                            next.
}
```

The single most important use of a matrix in row-indexed sparse storage mode is to multiply a vector to its right. In fact, the storage mode is optimized for just this purpose. The following routine is thus very simple.

```
void sprsax(float sa[], unsigned long ija[], float x[], float b[],
    unsigned long n)
Multiply a matrix in row-index sparse storage arrays sa and ija by a vector x[1..n], giving
a vector b[1..n].
{
    void nrerror(char error_text[]);
```

```
    unsigned long i,k;

    if (ija[1] != n+2) nrerror("sprsax: mismatched vector and matrix");
    for (i=1;i<=n;i++) {
        b[i]=sa[i]*x[i];                    Start with diagonal term.
        for (k=ija[i];k<=ija[i+1]-1;k++)    Loop over off-diagonal terms.
            b[i] += sa[k]*x[ija[k]];

        }
}
```

It is also simple to multiply the *transpose* of a matrix by a vector to its right. (We will use this operation later in this section.) Note that the transpose matrix is not actually constructed.

```
void sprstx(float sa[], unsigned long ija[], float x[], float b[],
    unsigned long n)
Multiply the transpose of a matrix in row-index sparse storage arrays sa and ija by a vector
x[1..n], giving a vector b[1..n].
{
    void nrerror(char error_text[]);
    unsigned long i,j,k;

    if (ija[1] != n+2) nrerror("mismatched vector and matrix in sprstx");
    for (i=1;i<=n;i++) b[i]=sa[i]*x[i];     Start with diagonal terms.
    for (i=1;i<=n;i++) {                    Loop over off-diagonal terms.
        for (k=ija[i];k<=ija[i+1]-1;k++) {
            j=ija[k];
            b[j] += sa[k]*x[i];
        }
    }
}
```

(Double precision versions of `sprsax` and `sprstx`, named `dsprsax` and `dsprstx`, are used by the routine `atimes` later in this section. You can easily make the conversion, or else get the converted routines from the *Numerical Recipes* diskettes.)

In fact, because the choice of row-indexed storage treats rows and columns quite differently, it is quite an involved operation to construct the transpose of a matrix, given the matrix itself in row-indexed sparse storage mode. When the operation cannot be avoided, it is done as follows: An index of all off-diagonal elements by their columns is constructed (see §8.4). The elements are then written to the output array in column order. As each element is written, its row is determined and stored. Finally, the elements in each column are sorted by row.

```
void sprstp(float sa[], unsigned long ija[], float sb[], unsigned long ijb[])
Construct the transpose of a sparse square matrix, from row-index sparse storage arrays sa and
ija into arrays sb and ijb.
{
    void iindexx(unsigned long n, long arr[], unsigned long indx[]);
    Version of indexx with all float variables changed to long.
    unsigned long j,jl,jm,jp,ju,k,m,n2,noff,inc,iv;
    float v;

    n2=ija[1];                              Linear size of matrix plus 2.
    for (j=1;j<=n2-2;j++) sb[j]=sa[j];      Diagonal elements.
    iindexx(ija[n2-1]-ija[1],(long *)&ija[n2-1],&ijb[n2-1]);
    Index all off-diagonal elements by their columns.
    jp=0;
    for (k=ija[1];k<=ija[n2-1]-1;k++) {     Loop over output off-diagonal elements.
        m=ijb[k]+n2-1;                      Use index table to store by (former) columns.
        sb[k]=sa[m];
        for (j=jp+1;j<=ija[m];j++) ijb[j]=k;    Fill in the index to any omitted rows.
```

```
        jp=ija[m];                                  Use bisection to find which row element
        jl=1;                                           m is in and put that into ijb[k].
        ju=n2-1;
        while (ju-jl > 1) {
            jm=(ju+jl)/2;
            if (ija[jm] > m) ju=jm; else jl=jm;
        }
        ijb[k]=jl;
    }
    for (j=jp+1;j<n2;j++) ijb[j]=ija[n2-1];
    for (j=1;j<=n2-2;j++) {                         Make a final pass to sort each row by
        jl=ijb[j+1]-ijb[j];                             Shell sort algorithm.
        noff=ijb[j]-1;
        inc=1;
        do {
            inc *= 3;
            inc++;
        } while (inc <= jl);
        do {
            inc /= 3;
            for (k=noff+inc+1;k<=noff+jl;k++) {
                iv=ijb[k];
                v=sb[k];
                m=k;
                while (ijb[m-inc] > iv) {
                    ijb[m]=ijb[m-inc];
                    sb[m]=sb[m-inc];
                    m -= inc;
                    if (m-noff <= inc) break;
                }
                ijb[m]=iv;
                sb[m]=v;
            }
        } while (inc > 1);
    }
}
```

The above routine embeds internally a sorting algorithm from §8.1, but calls the external routine `iindexx` to construct the initial column index. This routine is identical to `indexx`, as listed in §8.4, except that the latter's two `float` declarations should be changed to `long`. (The *Numerical Recipes* diskettes include both `indexx` and `iindexx`.) In fact, you can often use `indexx` *without* making these changes, since many computers have the property that numerical values will sort correctly independently of whether they are interpreted as floating or integer values.

As final examples of the manipulation of sparse matrices, we give two routines for the multiplication of two sparse matrices. These are useful for techniques to be described in §13.10.

In general, the product of two sparse matrices is not itself sparse. One therefore wants to limit the size of the product matrix in one of two ways: either compute only those elements of the product that are specified in advance by a known pattern of sparsity, or else compute all nonzero elements, but store only those whose magnitude exceeds some threshold value. The former technique, when it can be used, is quite efficient. The pattern of sparsity is specified by furnishing an index array in row-index sparse storage format (e.g., `ija`). The program then constructs a corresponding value array (e.g., `sa`). The latter technique runs the danger of excessive compute times and unknown output sizes, so it must be used cautiously.

With row-index storage, it is much more natural to multiply a matrix (on the left) by the *transpose* of a matrix (on the right), so that one is crunching rows on rows, rather than rows on columns. Our routines therefore calculate $\mathbf{A} \cdot \mathbf{B}^T$, rather than $\mathbf{A} \cdot \mathbf{B}$. This means that you have to run your right-hand matrix through the transpose routine `sprstp` before sending it to the matrix multiply routine.

The two implementing routines, `sprspm` for "pattern multiply" and `sprstm` for "threshold multiply" are quite similar in structure. Both are complicated by the logic of the various combinations of diagonal or off-diagonal elements for the two input streams and output stream.

```
void sprspm(float sa[], unsigned long ija[], float sb[], unsigned long ijb[],
    float sc[], unsigned long ijc[])
```

Matrix multiply $\mathbf{A} \cdot \mathbf{B}^T$ where $\mathbf{A}$ and $\mathbf{B}$ are two sparse matrices in row-index storage mode, and $\mathbf{B}^T$ is the transpose of $\mathbf{B}$. Here, `sa` and `ija` store the matrix $\mathbf{A}$; `sb` and `ijb` store the matrix $\mathbf{B}$. This routine computes only those components of the matrix product that are *pre-specified* by the input index array `ijc`, which is not modified. On output, the arrays `sc` and `ijc` give the product matrix in row-index storage mode. For sparse matrix multiplication, this routine will often be preceded by a call to `sprstp`, so as to construct the transpose of a known matrix into `sb`, `ijb`.

```
{
    void nrerror(char error_text[]);
    unsigned long i,ijma,ijmb,j,m,ma,mb,mbb,mn;
    float sum;

    if (ija[1] != ijb[1] || ija[1] != ijc[1])
        nrerror("sprspm: sizes do not match");
    for (i=1;i<=ijc[1]-2;i++) {          Loop over rows.
        j=m=i;                           Set up so that first pass through loop does the
        mn=ijc[i];                           diagonal component.
        sum=sa[i]*sb[i];
        for (;;) {                       Main loop over each component to be output.
            mb=ijb[j];
            for (ma=ija[i];ma<=ija[i+1]-1;ma++) {
            Loop through elements in A's row. Convoluted logic, following, accounts for the
            various combinations of diagonal and off-diagonal elements.
                ijma=ija[ma];
                if (ijma == j) sum += sa[ma]*sb[j];
                else {
                    while (mb < ijb[j+1]) {
                        ijmb=ijb[mb];
                        if (ijmb == i) {
                            sum += sa[i]*sb[mb++];
                            continue;
                        } else if (ijmb < ijma) {
                            mb++;
                            continue;
                        } else if (ijmb == ijma) {
                            sum += sa[ma]*sb[mb++];
                            continue;
                        }
                        break;
                    }
                }
            }
            for (mbb=mb;mbb<=ijb[j+1]-1;mbb++) {      Exhaust the remainder of B's row.
                if (ijb[mbb] == i) sum += sa[i]*sb[mbb];
            }
            sc[m]=sum;
            sum=0.0;                     Reset indices for next pass through loop.
            if (mn >= ijc[i+1]) break;
            j=ijc[m=mn++];
        }
    }
}
```

```
#include <math.h>

void sprstm(float sa[], unsigned long ija[], float sb[], unsigned long ijb[],
    float thresh, unsigned long nmax, float sc[], unsigned long ijc[])
```

Matrix multiply $\mathbf{A} \cdot \mathbf{B}^T$ where $\mathbf{A}$ and $\mathbf{B}$ are two sparse matrices in row-index storage mode, and $\mathbf{B}^T$ is the transpose of $\mathbf{B}$. Here, `sa` and `ija` store the matrix $\mathbf{A}$; `sb` and `ijb` store the matrix $\mathbf{B}$. This routine computes all components of the matrix product (which may be non-sparse!), but stores only those whose magnitude exceeds `thresh`. On output, the arrays `sc` and `ijc` (whose maximum size is input as `nmax`) give the product matrix in row-index storage mode. For sparse matrix multiplication, this routine will often be preceded by a call to `sprstp`, so as to construct the transpose of a known matrix into `sb`, `ijb`.

```
{
    void nrerror(char error_text[]);
    unsigned long i,ijma,ijmb,j,k,ma,mb,mbb;
    float sum;

    if (ija[1] != ijb[1]) nrerror("sprstm: sizes do not match");
    ijc[1]=k=ija[1];
    for (i=1;i<=ija[1]-2;i++) {                    Loop over rows of A,
        for (j=1;j<=ijb[1]-2;j++) {                and rows of B.
            if (i == j) sum=sa[i]*sb[j]; else sum=0.0e0;
            mb=ijb[j];
            for (ma=ija[i];ma<=ija[i+1]-1;ma++) {
            Loop through elements in A's row. Convoluted logic, following, accounts for the
            various combinations of diagonal and off-diagonal elements.
                ijma=ija[ma];
                if (ijma == j) sum += sa[ma]*sb[j];
                else {
                    while (mb < ijb[j+1]) {
                        ijmb=ijb[mb];
                        if (ijmb == i) {
                            sum += sa[i]*sb[mb++];
                            continue;
                        } else if (ijmb < ijma) {
                            mb++;
                            continue;
                        } else if (ijmb == ijma) {
                            sum += sa[ma]*sb[mb++];
                            continue;
                        }
                        break;
                    }
                }
            }
            for (mbb=mb;mbb<=ijb[j+1]-1;mbb++) {       Exhaust the remainder of B's row.
                if (ijb[mbb] == i) sum += sa[i]*sb[mbb];
            }
            if (i == j) sc[i]=sum;                     Where to put the answer...
            else if (fabs(sum) > thresh) {
                if (k > nmax) nrerror("sprstm: nmax too small");
                sc[k]=sum;
                ijc[k++]=j;
            }
        }
        ijc[i+1]=k;
    }
}
```

Conjugate Gradient Method for a Sparse System

So-called *conjugate gradient methods* provide a quite general means for solving the $N \times N$ linear system

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.7.29}$$

The attractiveness of these methods for large sparse systems is that they reference $\mathbf{A}$ only through its multiplication of a vector, or the multiplication of its transpose and a vector. As

we have seen, these operations can be very efficient for a properly stored sparse matrix. You, the "owner" of the matrix **A**, can be asked to provide functions that perform these sparse matrix multiplications as efficiently as possible. We, the "grand strategists" supply the general routine, `linbcg` below, that solves the set of linear equations, (2.7.29), using your functions.

The simplest, "ordinary" conjugate gradient algorithm [11-13] solves (2.7.29) only in the case that **A** is symmetric and positive definite. It is based on the idea of minimizing the function

$$f(\mathbf{x}) = \frac{1}{2}\,\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x} - \mathbf{b}\cdot\mathbf{x} \tag{2.7.30}$$

This function is minimized when its gradient

$$\nabla f = \mathbf{A}\cdot\mathbf{x} - \mathbf{b} \tag{2.7.31}$$

is zero, which is equivalent to (2.7.29). The minimization is carried out by generating a succession of search directions $\mathbf{p}_k$ and improved minimizers $\mathbf{x}_k$. At each stage a quantity α_k is found that minimizes $f(\mathbf{x}_k + \alpha_k\mathbf{p}_k)$, and $\mathbf{x}_{k+1}$ is set equal to the new point $\mathbf{x}_k + \alpha_k\mathbf{p}_k$. The $\mathbf{p}_k$ and $\mathbf{x}_k$ are built up in such a way that $\mathbf{x}_{k+1}$ is also the minimizer of f over the whole vector space of directions already taken, $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k\}$. After N iterations you arrive at the minimizer over the entire vector space, i.e., the solution to (2.7.29).

Later, in §10.6, we will generalize this "ordinary" conjugate gradient algorithm to the minimization of arbitrary nonlinear functions. Here, where our interest is in solving linear, but not necessarily positive definite or symmetric, equations, a different generalization is important, the *biconjugate gradient method*. This method does not, in general, have a simple connection with function minimization. It constructs four sequences of vectors, $\mathbf{r}_k$, $\overline{\mathbf{r}}_k$, $\mathbf{p}_k$, $\overline{\mathbf{p}}_k$, $k = 1, 2, \ldots$. You supply the initial vectors $\mathbf{r}_1$ and $\overline{\mathbf{r}}_1$, and set $\mathbf{p}_1 = \mathbf{r}_1$, $\overline{\mathbf{p}}_1 = \overline{\mathbf{r}}_1$. Then you carry out the following recurrence:

$$\begin{aligned}
\alpha_k &= \frac{\overline{\mathbf{r}}_k\cdot\mathbf{r}_k}{\overline{\mathbf{p}}_k\cdot\mathbf{A}\cdot\mathbf{p}_k}\\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k\mathbf{A}\cdot\mathbf{p}_k\\
\overline{\mathbf{r}}_{k+1} &= \overline{\mathbf{r}}_k - \alpha_k\mathbf{A}^T\cdot\overline{\mathbf{p}}_k\\
\beta_k &= \frac{\overline{\mathbf{r}}_{k+1}\cdot\mathbf{r}_{k+1}}{\overline{\mathbf{r}}_k\cdot\mathbf{r}_k}\\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k\\
\overline{\mathbf{p}}_{k+1} &= \overline{\mathbf{r}}_{k+1} + \beta_k\overline{\mathbf{p}}_k
\end{aligned} \tag{2.7.32}$$

This sequence of vectors satisfies the *biorthogonality* condition

$$\overline{\mathbf{r}}_i\cdot\mathbf{r}_j = \mathbf{r}_i\cdot\overline{\mathbf{r}}_j = 0, \qquad j < i \tag{2.7.33}$$

and the *biconjugacy* condition

$$\overline{\mathbf{p}}_i\cdot\mathbf{A}\cdot\mathbf{p}_j = \mathbf{p}_i\cdot\mathbf{A}^T\cdot\overline{\mathbf{p}}_j = 0, \qquad j < i \tag{2.7.34}$$

There is also a mutual orthogonality,

$$\overline{\mathbf{r}}_i\cdot\mathbf{p}_j = \mathbf{r}_i\cdot\overline{\mathbf{p}}_j = 0, \qquad j < i \tag{2.7.35}$$

The proof of these properties proceeds by straightforward induction [14]. As long as the recurrence does not break down earlier because one of the denominators is zero, it must terminate after $m \le N$ steps with $\mathbf{r}_{m+1} = \overline{\mathbf{r}}_{m+1} = 0$. This is basically because after at most N steps you run out of new orthogonal directions to the vectors you've already constructed.

To use the algorithm to solve the system (2.7.29), make an initial guess $\mathbf{x}_1$ for the solution. Choose $\mathbf{r}_1$ to be the *residual*

$$\mathbf{r}_1 = \mathbf{b} - \mathbf{A}\cdot\mathbf{x}_1 \tag{2.7.36}$$

and choose $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. Then form the sequence of improved estimates

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k \tag{2.7.37}$$

while carrying out the recurrence (2.7.32). Equation (2.7.37) guarantees that $\mathbf{r}_{k+1}$ from the recurrence is in fact the residual $\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{k+1}$ corresponding to $\mathbf{x}_{k+1}$. Since $\mathbf{r}_{m+1} = 0$, $\mathbf{x}_{m+1}$ is the solution to equation (2.7.29).

While there is no guarantee that this whole procedure will not break down or become unstable for general $\mathbf{A}$, in practice this is rare. More importantly, the exact termination in at most N iterations occurs only with exact arithmetic. Roundoff error means that you should regard the process as a genuinely iterative procedure, to be halted when some appropriate error criterion is met.

The ordinary conjugate gradient algorithm is the special case of the biconjugate gradient algorithm when $\mathbf{A}$ is symmetric, and we choose $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. Then $\overline{\mathbf{r}}_k = \mathbf{r}_k$ and $\overline{\mathbf{p}}_k = \mathbf{p}_k$ for all k; you can omit computing them and halve the work of the algorithm. This conjugate gradient version has the interpretation of minimizing equation (2.7.30). If $\mathbf{A}$ is positive definite as well as symmetric, the algorithm cannot break down (in theory!). The routine `linbcg` below indeed reduces to the ordinary conjugate gradient method if you input a symmetric $\mathbf{A}$, but it does all the redundant computations.

Another variant of the general algorithm corresponds to a symmetric but non-positive definite $\mathbf{A}$, with the choice $\overline{\mathbf{r}}_1 = \mathbf{A} \cdot \mathbf{r}_1$ instead of $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. In this case $\overline{\mathbf{r}}_k = \mathbf{A} \cdot \mathbf{r}_k$ and $\overline{\mathbf{p}}_k = \mathbf{A} \cdot \mathbf{p}_k$ for all k. This algorithm is thus equivalent to the ordinary conjugate gradient algorithm, but with all dot products $\mathbf{a} \cdot \mathbf{b}$ replaced by $\mathbf{a} \cdot \mathbf{A} \cdot \mathbf{b}$. It is called the *minimum residual* algorithm, because it corresponds to successive minimizations of the function

$$\Phi(\mathbf{x}) = \frac{1}{2}\,\mathbf{r} \cdot \mathbf{r} = \frac{1}{2}\,|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2 \tag{2.7.38}$$

where the successive iterates $\mathbf{x}_k$ minimize Φ over the same set of search directions $\mathbf{p}_k$ generated in the conjugate gradient method. This algorithm has been generalized in various ways for unsymmetric matrices. The *generalized minimum residual* method (GMRES; see [9,15]) is probably the most robust of these methods.

Note that equation (2.7.38) gives

$$\nabla\Phi(\mathbf{x}) = \mathbf{A}^T \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) \tag{2.7.39}$$

For any nonsingular matrix $\mathbf{A}$, $\mathbf{A}^T \cdot \mathbf{A}$ is symmetric and positive definite. You might therefore be tempted to solve equation (2.7.29) by applying the ordinary conjugate gradient algorithm to the problem

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = \mathbf{A}^T \cdot \mathbf{b} \tag{2.7.40}$$

Don't! The condition number of the matrix $\mathbf{A}^T \cdot \mathbf{A}$ is the square of the condition number of $\mathbf{A}$ (see §2.6 for definition of condition number). A large condition number both increases the number of iterations required, and limits the accuracy to which a solution can be obtained. It is almost always better to apply the biconjugate gradient method to the original matrix $\mathbf{A}$.

So far we have said nothing about the *rate* of convergence of these methods. The ordinary conjugate gradient method works well for matrices that are well-conditioned, i.e., "close" to the identity matrix. This suggests applying these methods to the *preconditioned* form of equation (2.7.29),

$$(\widetilde{\mathbf{A}}^{-1} \cdot \mathbf{A}) \cdot \mathbf{x} = \widetilde{\mathbf{A}}^{-1} \cdot \mathbf{b} \tag{2.7.41}$$

The idea is that you might already be able to solve your linear system easily for some $\widetilde{\mathbf{A}}$ close to $\mathbf{A}$, in which case $\widetilde{\mathbf{A}}^{-1} \cdot \mathbf{A} \approx \mathbf{1}$, allowing the algorithm to converge in fewer steps. The matrix $\widetilde{\mathbf{A}}$ is called a *preconditioner* [11], and the overall scheme given here is known as the *preconditioned biconjugate gradient method* or *PBCG*.

For efficient implementation, the PBCG algorithm introduces an additional set of vectors $\mathbf{z}_k$ and $\overline{\mathbf{z}}_k$ defined by

$$\widetilde{\mathbf{A}} \cdot \mathbf{z}_k = \mathbf{r}_k \qquad \text{and} \qquad \widetilde{\mathbf{A}}^T \cdot \overline{\mathbf{z}}_k = \overline{\mathbf{r}}_k \tag{2.7.42}$$

and modifies the definitions of α_k, β_k, $\mathbf{p}_k$, and $\overline{\mathbf{p}}_k$ in equation (2.7.32):

$$\begin{aligned} \alpha_k &= \frac{\overline{\mathbf{r}}_k \cdot \mathbf{z}_k}{\overline{\mathbf{p}}_k \cdot \mathbf{A} \cdot \mathbf{p}_k} \\ \beta_k &= \frac{\overline{\mathbf{r}}_{k+1} \cdot \mathbf{z}_{k+1}}{\overline{\mathbf{r}}_k \cdot \mathbf{z}_k} \\ \mathbf{p}_{k+1} &= \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k \\ \overline{\mathbf{p}}_{k+1} &= \overline{\mathbf{z}}_{k+1} + \beta_k \overline{\mathbf{p}}_k \end{aligned} \tag{2.7.43}$$

For `linbcg`, below, we will ask you to supply routines that solve the auxiliary linear systems (2.7.42). If you have no idea what to use for the preconditioner $\widetilde{\mathbf{A}}$, then use the diagonal part of $\mathbf{A}$, or even the identity matrix, in which case the burden of convergence will be entirely on the biconjugate gradient method itself.

The routine `linbcg`, below, is based on a program originally written by Anne Greenbaum. (See [13] for a different, less sophisticated, implementation.) There are a few wrinkles you should know about.

What constitutes "good" convergence is rather application dependent. The routine `linbcg` therefore provides for four possibilities, selected by setting the flag `itol` on input. If `itol=1`, iteration stops when the quantity $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|/|\mathbf{b}|$ is less than the input quantity `tol`. If `itol=2`, the required criterion is

$$|\widetilde{\mathbf{A}}^{-1} \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b})|/|\widetilde{\mathbf{A}}^{-1} \cdot \mathbf{b}| < \texttt{tol} \tag{2.7.44}$$

If `itol=3`, the routine uses its own estimate of the error in $\mathbf{x}$, and requires its magnitude, divided by the magnitude of $\mathbf{x}$, to be less than `tol`. The setting `itol=4` is the same as `itol=3`, except that the largest (in absolute value) component of the error and largest component of $\mathbf{x}$ are used instead of the vector magnitude (that is, the L_∞ norm instead of the L_2 norm). You may need to experiment to find which of these convergence criteria is best for your problem.

On output, `err` is the tolerance actually achieved. If the returned count `iter` does not indicate that the maximum number of allowed iterations `itmax` was exceeded, then `err` should be less than `tol`. If you want to do further iterations, leave all returned quantities as they are and call the routine again. The routine loses its memory of the spanned conjugate gradient subspace between calls, however, so you should not force it to return more often than about every N iterations.

Finally, note that `linbcg` is furnished in double precision, since it will be usually be used when N is quite large.

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define EPS 1.0e-14

void linbcg(unsigned long n, double b[], double x[], int itol, double tol,
    int itmax, int *iter, double *err)
```

Solves $\mathbf{A} \cdot \mathtt{x} = \mathbf{b}$ for `x[1..n]`, given `b[1..n]`, by the iterative biconjugate gradient method. On input `x[1..n]` should be set to an initial guess of the solution (or all zeros); `itol` is 1,2,3, or 4, specifying which convergence test is applied (see text); `itmax` is the maximum number of allowed iterations; and `tol` is the desired convergence tolerance. On output, `x[1..n]` is reset to the improved solution, `iter` is the number of iterations actually taken, and `err` is the estimated error. The matrix $\mathbf{A}$ is referenced only through the user-supplied routines `atimes`, which computes the product of either $\mathbf{A}$ or its transpose on a vector; and `asolve`, which solves $\widetilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$ or $\widetilde{\mathbf{A}}^T \cdot \mathbf{x} = \mathbf{b}$ for some preconditioner matrix $\widetilde{\mathbf{A}}$ (possibly the trivial diagonal part of $\mathbf{A}$).

```
{
    void asolve(unsigned long n, double b[], double x[], int itrnsp);
    void atimes(unsigned long n, double x[], double r[], int itrnsp);
    double snrm(unsigned long n, double sx[], int itol);
    unsigned long j;
    double ak,akden,bk,bkden,bknum,bnrm,dxnrm,xnrm,zm1nrm,znrm;
    double *p,*pp,*r,*rr,*z,*zz;        Double precision is a good idea in this routine.
```

```
    p=dvector(1,n);
    pp=dvector(1,n);
    r=dvector(1,n);
    rr=dvector(1,n);
    z=dvector(1,n);
    zz=dvector(1,n);

    Calculate initial residual.
    *iter=0;
    atimes(n,x,r,0);                    Input to atimes is x[1..n], output is r[1..n];
    for (j=1;j<=n;j++) {                    the final 0 indicates that the matrix (not its
        r[j]=b[j]-r[j];                     transpose) is to be used.
        rr[j]=r[j];
    }
/*    atimes(n,r,rr,0); */              Uncomment this line to get the "minimum resid-
    if (itol == 1) {                        ual" variant of the algorithm.
        bnrm=snrm(n,b,itol);
        asolve(n,r,z,0);                Input to asolve is r[1..n], output is z[1..n];
    }                                       the final 0 indicates that the matrix Ã (not
    else if (itol == 2) {                   its transpose) is to be used.
        asolve(n,b,z,0);
        bnrm=snrm(n,z,itol);
        asolve(n,r,z,0);
    }
    else if (itol == 3 || itol == 4) {
        asolve(n,b,z,0);
        bnrm=snrm(n,z,itol);
        asolve(n,r,z,0);
        znrm=snrm(n,z,itol);
    } else nrerror("illegal itol in linbcg");
    while (*iter <= itmax) {            Main loop.
        ++(*iter);
        asolve(n,rr,zz,1);              Final 1 indicates use of transpose matrix Ã^T.
        for (bknum=0.0,j=1;j<=n;j++) bknum += z[j]*rr[j];
        Calculate coefficient bk and direction vectors p and pp.
        if (*iter == 1) {
            for (j=1;j<=n;j++) {
                p[j]=z[j];
                pp[j]=zz[j];
            }
        }
        else {
            bk=bknum/bkden;
            for (j=1;j<=n;j++) {
                p[j]=bk*p[j]+z[j];
                pp[j]=bk*pp[j]+zz[j];
            }
        }
        bkden=bknum;                    Calculate coefficient ak, new iterate x, and new
        atimes(n,p,z,0);                    residuals r and rr.
        for (akden=0.0,j=1;j<=n;j++) akden += z[j]*pp[j];
        ak=bknum/akden;
        atimes(n,pp,zz,1);
        for (j=1;j<=n;j++) {
            x[j] += ak*p[j];
            r[j] -= ak*z[j];
            rr[j] -= ak*zz[j];
        }
        asolve(n,r,z,0);                Solve Ã · z = r and check stopping criterion.
        if (itol == 1)
            *err=snrm(n,r,itol)/bnrm;
        else if (itol == 2)
            *err=snrm(n,z,itol)/bnrm;
```

```
        else if (itol == 3 || itol == 4) {
            zm1nrm=znrm;
            znrm=snrm(n,z,itol);
            if (fabs(zm1nrm-znrm) > EPS*znrm) {
                dxnrm=fabs(ak)*snrm(n,p,itol);
                *err=znrm/fabs(zm1nrm-znrm)*dxnrm;
            } else {
                *err=znrm/bnrm;          Error may not be accurate, so loop again.
                continue;
            }
            xnrm=snrm(n,x,itol);
            if (*err <= 0.5*xnrm) *err /= xnrm;
            else {
                *err=znrm/bnrm;          Error may not be accurate, so loop again.
                continue;
            }
        }
        printf("iter=%4d err=%12.6f\n",*iter,*err);
    if (*err <= tol) break;
    }

    free_dvector(p,1,n);
    free_dvector(pp,1,n);
    free_dvector(r,1,n);
    free_dvector(rr,1,n);
    free_dvector(z,1,n);
    free_dvector(zz,1,n);
}
```

The routine `linbcg` uses this short utility for computing vector norms:

```
#include <math.h>

double snrm(unsigned long n, double sx[], int itol)
Compute one of two norms for a vector sx[1..n], as signaled by itol. Used by linbcg.
{
    unsigned long i,isamax;
    double ans;

    if (itol <= 3) {
        ans = 0.0;
        for (i=1;i<=n;i++) ans += sx[i]*sx[i];        Vector magnitude norm.
        return sqrt(ans);
    } else {
        isamax=1;
        for (i=1;i<=n;i++) {                          Largest component norm.
            if (fabs(sx[i]) > fabs(sx[isamax])) isamax=i;
        }
        return fabs(sx[isamax]);
    }
}
```

So that the specifications for the routines `atimes` and `asolve` are clear, we list here simple versions that assume a matrix **A** stored somewhere in row-index sparse format.

```
extern unsigned long ija[];
extern double sa[];              The matrix is stored somewhere.

void atimes(unsigned long n, double x[], double r[], int itrnsp)
{
    void dsprsax(double sa[], unsigned long ija[], double x[], double b[],
        unsigned long n);
```

```
    void dsprstx(double sa[], unsigned long ija[], double x[], double b[],
        unsigned long n);
    These are double versions of sprsax and sprstx.

    if (itrnsp) dsprstx(sa,ija,x,r,n);
    else dsprsax(sa,ija,x,r,n);
}



extern unsigned long ija[];
extern double sa[];                The matrix is stored somewhere.

void asolve(unsigned long n, double b[], double x[], int itrnsp)
{
    unsigned long i;

    for(i=1;i<=n;i++) x[i]=(sa[i] != 0.0 ? b[i]/sa[i] : b[i]);
    The matrix Ã is the diagonal part of A, stored in the first n elements of sa. Since the
    transpose matrix has the same diagonal, the flag itrnsp is not used.
}
```

CITED REFERENCES AND FURTHER READING:

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter I.3 (by J.K. Reid). [2]

George, A., and Liu, J.W.H. 1981, *Computer Solution of Large Sparse Positive Definite Systems* (Englewood Cliffs, NJ: Prentice-Hall). [3]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.). [4]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [5]

Eisenstat, S.C., Gursky, M.C., Schultz, M.H., and Sherman, A.H. 1977, *Yale Sparse Matrix Package*, Technical Reports 112 and 114 (Yale University Department of Computer Science). [6]

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.2.6. [7]

Kincaid, D.R., Respess, J.R., Young, D.M., and Grimes, R.G. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 302–322. [8]

PCGPAK User's Guide (New Haven: Scientific Computing Associates, Inc.). [9]

Bentley, J. 1986, *Programming Pearls* (Reading, MA: Addison-Wesley), §9. [10]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapters 4 and 10, particularly §§10.2–10.3. [11]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 8. [12]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill). [13]

Fletcher, R. 1976, in *Numerical Analysis Dundee 1975*, Lecture Notes in Mathematics, vol. 506, A. Dold and B Eckmann, eds. (Berlin: Springer-Verlag), pp. 73–89. [14]

Saad, Y., and Schulz, M. 1986, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856–869. [15]

Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press).

Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.).

2.8 Vandermonde Matrices and Toeplitz Matrices

In §2.4 the case of a tridiagonal matrix was treated specially, because that particular type of linear system admits a solution in only of order N operations, rather than of order N^3 for the general linear problem. When such particular types exist, it is important to know about them. Your computational savings, should you ever happen to be working on a problem that involves the right kind of particular type, can be enormous.

This section treats two special types of matrices that can be solved in of order N^2 operations, not as good as tridiagonal, but a lot better than the general case. (Other than the operations count, these two types having nothing in common.) Matrices of the first type, termed *Vandermonde matrices*, occur in some problems having to do with the fitting of polynomials, the reconstruction of distributions from their moments, and also other contexts. In this book, for example, a Vandermonde problem crops up in §3.5. Matrices of the second type, termed *Toeplitz matrices*, tend to occur in problems involving deconvolution and signal processing. In this book, a Toeplitz problem is encountered in §13.7.

These are not the *only* special types of matrices worth knowing about. The *Hilbert matrices*, whose components are of the form $a_{ij} = 1/(i+j-1),\ i,j = 1,\dots,N$ can be inverted by an exact integer algorithm, and are very *difficult* to invert in any other way, since they are notoriously ill-conditioned (see [1] for details). The Sherman-Morrison and Woodbury formulas, discussed in §2.7, can sometimes be used to convert new special forms into old ones. Reference [2] gives some other special forms. We have not found these additional forms to arise as frequently as the two that we now discuss.

Vandermonde Matrices

A Vandermonde matrix of size $N \times N$ is completely determined by N arbitrary numbers $x_1, x_2, \dots, x_N$, in terms of which its N^2 components are the integer powers $x_i^{j-1},\ i,j = 1,\dots,N$. Evidently there are two possible such forms, depending on whether we view the i's as rows, j's as columns, or vice versa. In the former case, we get a linear system of equations that looks like this,

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.8.1}$$

Performing the matrix multiplication, you will see that this equation solves for the unknown coefficients c_i which fit a polynomial to the N pairs of abscissas and ordinates (x_j, y_j). Precisely this problem will arise in §3.5, and the routine given there will solve (2.8.1) by the method that we are about to describe.

The alternative identification of rows and columns leads to the set of equations

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ & & \cdots & \\ x_1^{N-1} & x_2^{N-1} & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \cdots \\ w_N \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \cdots \\ q_N \end{bmatrix} \tag{2.8.2}$$

Write this out and you will see that it relates to the *problem of moments*: Given the values of N points x_i, find the unknown weights w_i, assigned so as to match the given values q_j of the first N moments. (For more on this problem, consult [3].) The routine given in this section solves (2.8.2).

The method of solution of both (2.8.1) and (2.8.2) is closely related to Lagrange's polynomial interpolation formula, which we will not formally meet until §3.1 below. Notwithstanding, the following derivation should be comprehensible:

Let $P_j(x)$ be the polynomial of degree $N-1$ defined by

$$P_j(x) = \prod_{\substack{n=1 \\ (n \neq j)}}^{N} \frac{x - x_n}{x_j - x_n} = \sum_{k=1}^{N} A_{jk} x^{k-1} \tag{2.8.3}$$

Here the meaning of the last equality is to define the components of the matrix A_{ij} as the coefficients that arise when the product is multiplied out and like terms collected.

The polynomial $P_j(x)$ is a function of x generally. But you will notice that it is specifically designed so that it takes on a value of zero at all x_i with $i \neq j$, and has a value of unity at $x = x_j$. In other words,

$$P_j(x_i) = \delta_{ij} = \sum_{k=1}^{N} A_{jk} x_i^{k-1} \tag{2.8.4}$$

But (2.8.4) says that A_{jk} is exactly the inverse of the matrix of components x_i^{k-1}, which appears in (2.8.2), with the subscript as the column index. Therefore the solution of (2.8.2) is just that matrix inverse times the right-hand side,

$$w_j = \sum_{k=1}^{N} A_{jk} q_k \tag{2.8.5}$$

As for the transpose problem (2.8.1), we can use the fact that the inverse of the transpose is the transpose of the inverse, so

$$c_j = \sum_{k=1}^{N} A_{kj} y_k \tag{2.8.6}$$

The routine in §3.5 implements this.

It remains to find a good way of multiplying out the monomial terms in (2.8.3), in order to get the components of A_{jk}. This is essentially a bookkeeping problem, and we will let you read the routine itself to see how it can be solved. One trick is to define a master $P(x)$ by

$$P(x) \equiv \prod_{n=1}^{N} (x - x_n) \tag{2.8.7}$$

work out its coefficients, and then obtain the numerators and denominators of the specific P_j's via synthetic division by the one supernumerary term. (See §5.3 for more on synthetic division.) Since each such division is only a process of order N, the total procedure is of order N^2.

You should be warned that Vandermonde systems are notoriously ill-conditioned, by their very nature. (As an aside anticipating §5.8, the reason is the same as that which makes Chebyshev fitting so impressively accurate: there exist high-order polynomials that are very good uniform fits to zero. Hence roundoff error can introduce rather substantial coefficients of the leading terms of these polynomials.) It is a good idea always to compute Vandermonde problems in double precision.

The routine for (2.8.2) which follows is due to G.B. Rybicki.

```
#include "nrutil.h"

void vander(double x[], double w[], double q[], int n)
Solves the Vandermonde linear system sum_{i=1}^{N} x_i^{k-1} w_i = q_k (k = 1,...,N). Input consists of
the vectors x[1..n] and q[1..n]; the vector w[1..n] is output.
{
    int i,j,k;
    double b,s,t,xx;
    double *c;

    c=dvector(1,n);
    if (n == 1) w[1]=q[1];
    else {
        for (i=1;i<=n;i++) c[i]=0.0;          Initialize array.
        c[n] = -x[1];                         Coefficients of the master polynomial are found
        for (i=2;i<=n;i++) {                      by recursion.
            xx = -x[i];
            for (j=(n+1-i);j<=(n-1);j++) c[j] += xx*c[j+1];
            c[n] += xx;
        }
        for (i=1;i<=n;i++) {                  Each subfactor in turn
            xx=x[i];
            t=b=1.0;
            s=q[n];
            for (k=n;k>=2;k--) {              is synthetically divided,
                b=c[k]+xx*b;
                s += q[k-1]*b;                matrix-multiplied by the right-hand side,
                t=xx*t+b;
            }
            w[i]=s/t;                         and supplied with a denominator.
        }
    }
    free_dvector(c,1,n);
}
```

Toeplitz Matrices

An $N \times N$ Toeplitz matrix is specified by giving $2N-1$ numbers R_k, $k = -N+1, \ldots, -1, 0, 1, \ldots, N-1$. Those numbers are then emplaced as matrix elements constant along the (upper-left to lower-right) diagonals of the matrix:

$$\begin{bmatrix} R_0 & R_{-1} & R_{-2} & \cdots & R_{-(N-2)} & R_{-(N-1)} \\ R_1 & R_0 & R_{-1} & \cdots & R_{-(N-3)} & R_{-(N-2)} \\ R_2 & R_1 & R_0 & \cdots & R_{-(N-4)} & R_{-(N-3)} \\ \cdots & & & \cdots & & \\ R_{N-2} & R_{N-3} & R_{N-4} & \cdots & R_0 & R_{-1} \\ R_{N-1} & R_{N-2} & R_{N-3} & \cdots & R_1 & R_0 \end{bmatrix} \tag{2.8.8}$$

The linear Toeplitz problem can thus be written as

$$\sum_{j=1}^{N} R_{i-j} x_j = y_i \qquad (i = 1, \ldots, N) \tag{2.8.9}$$

where the x_j's, $j = 1, \ldots, N$, are the unknowns to be solved for.

The Toeplitz matrix is symmetric if $R_k = R_{-k}$ for all k. Levinson [4] developed an algorithm for fast solution of the symmetric Toeplitz problem, by a *bordering method*, that is,

a recursive procedure that solves the M-dimensional Toeplitz problem

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M)} = y_i \qquad (i = 1, \ldots, M) \tag{2.8.10}$$

in turn for $M = 1, 2, \ldots$ until $M = N$, the desired result, is finally reached. The vector $x_j^{(M)}$ is the result at the Mth stage, and becomes the desired answer only when N is reached.

Levinson's method is well documented in standard texts (e.g., [5]). The useful fact that the method generalizes to the *nonsymmetric* case seems to be less well known. At some risk of excessive detail, we therefore give a derivation here, due to G.B. Rybicki.

In following a recursion from step M to step $M+1$ we find that our developing solution $x^{(M)}$ changes in this way:

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M)} = y_i \qquad i = 1, \ldots, M \tag{2.8.11}$$

becomes

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M+1)} + R_{i-(M+1)} x_{M+1}^{(M+1)} = y_i \qquad i = 1, \ldots, M+1 \tag{2.8.12}$$

By eliminating y_i we find

$$\sum_{j=1}^{M} R_{i-j} \left(\frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \right) = R_{i-(M+1)} \qquad i = 1, \ldots, M \tag{2.8.13}$$

or by letting $i \to M+1-i$ and $j \to M+1-j$,

$$\sum_{j=1}^{M} R_{j-i} G_j^{(M)} = R_{-i} \tag{2.8.14}$$

where

$$G_j^{(M)} \equiv \frac{x_{M+1-j}^{(M)} - x_{M+1-j}^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.15}$$

To put this another way,

$$x_{M+1-j}^{(M+1)} = x_{M+1-j}^{(M)} - x_{M+1}^{(M+1)} G_j^{(M)} \qquad j = 1, \ldots, M \tag{2.8.16}$$

Thus, if we can use recursion to find the order M quantities $x^{(M)}$ and $G^{(M)}$ *and* the single order $M+1$ quantity $x_{M+1}^{(M+1)}$, then all of the other $x_j^{(M+1)}$ will follow. Fortunately, the quantity $x_{M+1}^{(M+1)}$ follows from equation (2.8.12) with $i = M+1$,

$$\sum_{j=1}^{M} R_{M+1-j} x_j^{(M+1)} + R_0 x_{M+1}^{(M+1)} = y_{M+1} \tag{2.8.17}$$

For the unknown order $M+1$ quantities $x_j^{(M+1)}$ we can substitute the previous order quantities in G since

$$G_{M+1-j}^{(M)} = \frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.18}$$

The result of this operation is

$$x_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{M+1-j} x_j^{(M)} - y_{M+1}}{\sum_{j=1}^{M} R_{M+1-j} G_{M+1-j}^{(M)} - R_0} \tag{2.8.19}$$

The only remaining problem is to develop a recursion relation for G. Before we do that, however, we should point out that there are actually two distinct sets of solutions to the original linear problem for a nonsymmetric matrix, namely right-hand solutions (which we have been discussing) and left-hand solutions z_i. The formalism for the left-hand solutions differs only in that we deal with the equations

$$\sum_{j=1}^{M} R_{j-i} z_j^{(M)} = y_i \qquad i = 1, \ldots, M \tag{2.8.20}$$

Then, the same sequence of operations on this set leads to

$$\sum_{j=1}^{M} R_{i-j} H_j^{(M)} = R_i \tag{2.8.21}$$

where

$$H_j^{(M)} \equiv \frac{z_{M+1-j}^{(M)} - z_{M+1-j}^{(M+1)}}{z_{M+1}^{(M+1)}} \tag{2.8.22}$$

(compare with 2.8.14 – 2.8.15). The reason for mentioning the left-hand solutions now is that, by equation (2.8.21), the H_j satisfy exactly the same equation as the x_j except for the substitution $y_i \to R_i$ on the right-hand side. Therefore we can quickly deduce from equation (2.8.19) that

$$H_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{M+1-j} H_j^{(M)} - R_{M+1}}{\sum_{j=1}^{M} R_{M+1-j} G_{M+1-j}^{(M)} - R_0} \tag{2.8.23}$$

By the same token, G satisfies the same equation as z, except for the substitution $y_i \to R_{-i}$. This gives

$$G_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{j-M-1} G_j^{(M)} - R_{-M-1}}{\sum_{j=1}^{M} R_{j-M-1} H_{M+1-j}^{(M)} - R_0} \tag{2.8.24}$$

The same "morphism" also turns equation (2.8.16), and its partner for z, into the final equations

$$\begin{aligned} G_j^{(M+1)} &= G_j^{(M)} - G_{M+1}^{(M+1)} H_{M+1-j}^{(M)} \\ H_j^{(M+1)} &= H_j^{(M)} - H_{M+1}^{(M+1)} G_{M+1-j}^{(M)} \end{aligned} \tag{2.8.25}$$

Now, starting with the initial values

$$x_1^{(1)} = y_1/R_0 \qquad G_1^{(1)} = R_{-1}/R_0 \qquad H_1^{(1)} = R_1/R_0 \tag{2.8.26}$$

we can recurse away. At each stage M we use equations (2.8.23) and (2.8.24) to find $H_{M+1}^{(M+1)}$, $G_{M+1}^{(M+1)}$, and then equation (2.8.25) to find the other components of $H^{(M+1)}$, $G^{(M+1)}$. From there the vectors $x^{(M+1)}$ and/or $z^{(M+1)}$ are easily calculated.

The program below does this. It incorporates the second equation in (2.8.25) in the form

$$H_{M+1-j}^{(M+1)} = H_{M+1-j}^{(M)} - H_{M+1}^{(M+1)} G_j^{(M)} \tag{2.8.27}$$

so that the computation can be done "in place."

Notice that the above algorithm fails if $R_0 = 0$. In fact, because the bordering method does not allow pivoting, the algorithm will fail if any of the diagonal principal minors of the original Toeplitz matrix vanish. (Compare with discussion of the tridiagonal algorithm in §2.4.) If the algorithm fails, your matrix is not necessarily singular — you might just have to solve your problem by a slower and more general algorithm such as LU decomposition with pivoting.

The routine that implements equations (2.8.23)–(2.8.27) is also due to Rybicki. Note that the routine's `r[n+j]` is equal to R_j above, so that subscripts on the `r` array vary from 1 to $2N - 1$.

```
#include "nrutil.h"
#define FREERETURN {free_vector(h,1,n);free_vector(g,1,n);return;}

void toeplz(float r[], float x[], float y[], int n)
```
Solves the Toeplitz system $\sum_{j=1}^{N} R_{(N+i-j)} x_j = y_i$ $(i = 1, \ldots, N)$. The Toeplitz matrix need not be symmetric. `y[1..n]` and `r[1..2*n-1]` are input arrays; `x[1..n]` is the output array.
```
{
    int j,k,m,m1,m2;
    float pp,pt1,pt2,qq,qt1,qt2,sd,sgd,sgn,shn,sxn;
    float *g,*h;

    if (r[n] == 0.0) nrerror("toeplz-1 singular principal minor");
    g=vector(1,n);
    h=vector(1,n);
    x[1]=y[1]/r[n];                 Initialize for the recursion.
    if (n == 1) FREERETURN
    g[1]=r[n-1]/r[n];
    h[1]=r[n+1]/r[n];
    for (m=1;m<=n;m++) {            Main loop over the recursion.
        m1=m+1;
        sxn = -y[m1];               Compute numerator and denominator for x,
        sd = -r[n];
        for (j=1;j<=m;j++) {
            sxn += r[n+m1-j]*x[j];
            sd += r[n+m1-j]*g[m-j+1];
        }
        if (sd == 0.0) nrerror("toeplz-2 singular principal minor");
        x[m1]=sxn/sd;               whence x.
        for (j=1;j<=m;j++) x[j] -= x[m1]*g[m-j+1];
        if (m1 == n) FREERETURN
        sgn = -r[n-m1];             Compute numerator and denominator for G and H,
        shn = -r[n+m1];
        sgd = -r[n];
        for (j=1;j<=m;j++) {
            sgn += r[n+j-m1]*g[j];
            shn += r[n+m1-j]*h[j];
            sgd += r[n+j-m1]*h[m-j+1];
        }
        if (sgd == 0.0) nrerror("toeplz-3 singular principal minor");
        g[m1]=sgn/sgd;              whence G and H.
        h[m1]=shn/sd;
        k=m;
        m2=(m+1) >> 1;
        pp=g[m1];
        qq=h[m1];
        for (j=1;j<=m2;j++) {
            pt1=g[j];
            pt2=g[k];
            qt1=h[j];
            qt2=h[k];
            g[j]=pt1-pp*qt2;
            g[k]=pt2-pp*qt1;
            h[j]=qt1-qq*pt2;
            h[k--]=qt2-qq*pt1;
        }
    }                               Back for another recurrence.
    nrerror("toeplz - should not arrive here!");
}
```

If you are in the business of solving *very* large Toeplitz systems, you should find out about so-called "new, fast" algorithms, which require only on the order of $N(\log N)^2$ operations, compared to N^2 for Levinson's method. These methods are too complicated to include here.

Papers by Bunch [6] and de Hoog [7] will give entry to the literature.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapter 5 [also treats some other special forms].

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), §19. [1]

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley). [2]

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), pp. 394ff. [3]

Levinson, N., Appendix B of N. Wiener, 1949, *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (New York: Wiley). [4]

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall), pp. 163ff. [5]

Bunch, J.R. 1985, *SIAM Journal on Scientific and Statistical Computing*, vol. 6, pp. 349–364. [6]

de Hoog, F. 1987, *Linear Algebra and Its Applications*, vol. 88/89, pp. 123–138. [7]

2.9 Cholesky Decomposition

If a square matrix $\mathbf{A}$ happens to be symmetric and positive definite, then it has a special, more efficient, triangular decomposition. *Symmetric* means that $a_{ij} = a_{ji}$ for $i, j = 1, \ldots, N$, while *positive definite* means that

$$\mathbf{v} \cdot \mathbf{A} \cdot \mathbf{v} > 0 \quad \text{for all vectors } \mathbf{v} \tag{2.9.1}$$

(In Chapter 11 we will see that positive definite has the equivalent interpretation that $\mathbf{A}$ has all positive eigenvalues.) While symmetric, positive definite matrices are rather special, they occur quite frequently in some applications, so their special factorization, called *Cholesky decomposition*, is good to know about. When you can use it, Cholesky decomposition is about a factor of two faster than alternative methods for solving linear equations.

Instead of seeking arbitrary lower and upper triangular factors $\mathbf{L}$ and $\mathbf{U}$, Cholesky decomposition constructs a lower triangular matrix $\mathbf{L}$ whose transpose $\mathbf{L}^T$ can itself serve as the upper triangular part. In other words we replace equation (2.3.1) by

$$\mathbf{L} \cdot \mathbf{L}^T = \mathbf{A} \tag{2.9.2}$$

This factorization is sometimes referred to as "taking the square root" of the matrix $\mathbf{A}$. The components of $\mathbf{L}^T$ are of course related to those of $\mathbf{L}$ by

$$L^T_{ij} = L_{ji} \tag{2.9.3}$$

Writing out equation (2.9.2) in components, one readily obtains the analogs of equations (2.3.12)–(2.3.13),

$$L_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 \right)^{1/2} \tag{2.9.4}$$

and

$$L_{ji} = \frac{1}{L_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right) \qquad j = i+1, i+2, \ldots, N \tag{2.9.5}$$

If you apply equations (2.9.4) and (2.9.5) in the order $i = 1, 2, \ldots, N$, you will see that the L's that occur on the right-hand side are already determined by the time they are needed. Also, only components a_{ij} with $j \geq i$ are referenced. (Since $\mathbf{A}$ is symmetric, these have complete information.) It is convenient, then, to have the factor $\mathbf{L}$ overwrite the subdiagonal (lower triangular but not including the diagonal) part of $\mathbf{A}$, preserving the input upper triangular values of $\mathbf{A}$. Only one extra vector of length N is needed to store the diagonal part of $\mathbf{L}$. The operations count is $N^3/6$ executions of the inner loop (consisting of one multiply and one subtract), with also N square roots. As already mentioned, this is about a factor 2 better than LU decomposition of $\mathbf{A}$ (where its symmetry would be ignored).

A straightforward implementation is

```
#include <math.h>

void choldc(float **a, int n, float p[])
Given a positive-definite symmetric matrix a[1..n][1..n], this routine constructs its Cholesky
decomposition, A = L · L^T. On input, only the upper triangle of a need be given; it is not
modified. The Cholesky factor L is returned in the lower triangle of a, except for its diagonal
elements which are returned in p[1..n].
{
    void nrerror(char error_text[]);
    int i,j,k;
    float sum;

    for (i=1;i<=n;i++) {
        for (j=i;j<=n;j++) {
            for (sum=a[i][j],k=i-1;k>=1;k--) sum -= a[i][k]*a[j][k];
            if (i == j) {
                if (sum <= 0.0)          a, with rounding errors, is not positive definite.
                    nrerror("choldc failed");
                p[i]=sqrt(sum);
            } else a[j][i]=sum/p[i];
        }
    }
}
```

You might at this point wonder about pivoting. The pleasant answer is that Cholesky decomposition is extremely stable numerically, without any pivoting at all. Failure of `choldc` simply indicates that the matrix $\mathbf{A}$ (or, with roundoff error, another very nearby matrix) is not positive definite. In fact, `choldc` is an efficient way to test *whether* a symmetric matrix is positive definite. (In this application, you will want to replace the call to `nrerror` with some less drastic signaling method.)

Once your matrix is decomposed, the triangular factor can be used to solve a linear equation by backsubstitution. The straightforward implementation of this is

```
void cholsl(float **a, int n, float p[], float b[], float x[])
Solves the set of n linear equations A · x = b, where a is a positive-definite symmetric matrix.
a[1..n][1..n] and p[1..n] are input as the output of the routine choldc. Only the lower
subdiagonal portion of a is accessed. b[1..n] is input as the right-hand side vector. The
solution vector is returned in x[1..n]. a, n, and p are not modified and can be left in place
for successive calls with different right-hand sides b. b is not modified unless you identify b and
x in the calling sequence, which is allowed.
{
    int i,k;
    float sum;

    for (i=1;i<=n;i++) {          Solve L · y = b, storing y in x.
        for (sum=b[i],k=i-1;k>=1;k--) sum -= a[i][k]*x[k];
        x[i]=sum/p[i];
    }
    for (i=n;i>=1;i--) {          Solve L^T · x = y.
        for (sum=x[i],k=i+1;k<=n;k++) sum -= a[k][i]*x[k];
```

```
            x[i]=sum/p[i];
        }
    }
```

A typical use of `choldc` and `cholsl` is in the inversion of covariance matrices describing the fit of data to a model; see, e.g., §15.6. In this, and many other applications, one often needs $\mathbf{L}^{-1}$. The lower triangle of this matrix can be efficiently found from the output of `choldc`:

```
    for (i=1;i<=n;i++) {
        a[i][i]=1.0/p[i];
        for (j=i+1;j<=n;j++) {
            sum=0.0;
            for (k=i;k<j;k++) sum -= a[j][k]*a[k][i];
            a[j][i]=sum/p[j];
        }
    }
```

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/1.

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), §4.9.2.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.5.

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.2.

2.10 QR Decomposition

There is another matrix factorization that is sometimes very useful, the so-called QR *decomposition*,

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{2.10.1}$$

Here $\mathbf{R}$ is upper triangular, while $\mathbf{Q}$ is orthogonal, that is,

$$\mathbf{Q}^T \cdot \mathbf{Q} = \mathbf{1} \tag{2.10.2}$$

where $\mathbf{Q}^T$ is the transpose matrix of $\mathbf{Q}$. Although the decomposition exists for a general rectangular matrix, we shall restrict our treatment to the case when all the matrices are square, with dimensions $N \times N$.

Like the other matrix factorizations we have met (LU, SVD, Cholesky), QR decomposition can be used to solve systems of linear equations. To solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.10.3}$$

first form $\mathbf{Q}^T \cdot \mathbf{b}$ and then solve

$$\mathbf{R} \cdot \mathbf{x} = \mathbf{Q}^T \cdot \mathbf{b} \tag{2.10.4}$$

by backsubstitution. Since QR decomposition involves about twice as many operations as LU decomposition, it is not used for typical systems of linear equations. However, we will meet special cases where QR is the method of choice.

The standard algorithm for the QR decomposition involves successive Householder transformations (to be discussed later in §11.2). We write a Householder matrix in the form $\mathbf{1} - \mathbf{u} \otimes \mathbf{u}/c$ where $c = \frac{1}{2}\mathbf{u} \cdot \mathbf{u}$. An appropriate Householder matrix applied to a given matrix can zero all elements in a column of the matrix situated below a chosen element. Thus we arrange for the first Householder matrix $\mathbf{Q}_1$ to zero all elements in the first column of $\mathbf{A}$ below the first element. Similarly $\mathbf{Q}_2$ zeroes all elements in the second column below the second element, and so on up to $\mathbf{Q}_{n-1}$. Thus

$$\mathbf{R} = \mathbf{Q}_{n-1} \cdots \mathbf{Q}_1 \cdot \mathbf{A} \tag{2.10.5}$$

Since the Householder matrices are orthogonal,

$$\mathbf{Q} = (\mathbf{Q}_{n-1} \cdots \mathbf{Q}_1)^{-1} = \mathbf{Q}_1 \cdots \mathbf{Q}_{n-1} \tag{2.10.6}$$

In most applications we don't need to form $\mathbf{Q}$ explicitly; we instead store it in the factored form (2.10.6). Pivoting is not usually necessary unless the matrix $\mathbf{A}$ is very close to singular. A general QR algorithm for rectangular matrices including pivoting is given in [1]. For square matrices, an implementation is the following:

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

void qrdcmp(float **a, int n, float *c, float *d, int *sing)
```

Constructs the QR decomposition of `a[1..n][1..n]`. The upper triangular matrix $\mathbf{R}$ is returned in the upper triangle of `a`, except for the diagonal elements of $\mathbf{R}$ which are returned in `d[1..n]`. The orthogonal matrix $\mathbf{Q}$ is represented as a product of $n-1$ Householder matrices $\mathbf{Q}_1 \cdots \mathbf{Q}_{n-1}$, where $\mathbf{Q}_j = \mathbf{1} - \mathbf{u}_j \otimes \mathbf{u}_j / c_j$. The ith component of $\mathbf{u}_j$ is zero for $i = 1, \ldots, j-1$ while the nonzero components are returned in `a[i][j]` for $i = j, \ldots, n$. `sing` returns as true (1) if singularity is encountered during the decomposition, but the decomposition is still completed in this case; otherwise it returns false (0).

```
{
    int i,j,k;
    float scale,sigma,sum,tau;

    *sing=0;
    for (k=1;k<n;k++) {
        scale=0.0;
        for (i=k;i<=n;i++) scale=FMAX(scale,fabs(a[i][k]));
        if (scale == 0.0) {                    Singular case.
            *sing=1;
            c[k]=d[k]=0.0;
        } else {                               Form Q_k and Q_k · A.
            for (i=k;i<=n;i++) a[i][k] /= scale;
            for (sum=0.0,i=k;i<=n;i++) sum += SQR(a[i][k]);
            sigma=SIGN(sqrt(sum),a[k][k]);
            a[k][k] += sigma;
            c[k]=sigma*a[k][k];
            d[k] = -scale*sigma;
            for (j=k+1;j<=n;j++) {
                for (sum=0.0,i=k;i<=n;i++) sum += a[i][k]*a[i][j];
                tau=sum/c[k];
                for (i=k;i<=n;i++) a[i][j] -= tau*a[i][k];
            }
        }
    }
    d[n]=a[n][n];
    if (d[n] == 0.0) *sing=1;
}
```

The next routine, `qrsolv`, is used to solve linear systems. In many applications only the part (2.10.4) of the algorithm is needed, so we separate it off into its own routine `rsolv`.

```
void qrsolv(float **a, int n, float c[], float d[], float b[])
```
Solves the set of `n` linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. `a[1..n][1..n]`, `c[1..n]`, and `d[1..n]` are input as the output of the routine `qrdcmp` and are not modified. `b[1..n]` is input as the right-hand side vector, and is overwritten with the solution vector on output.

```
{
    void rsolv(float **a, int n, float d[], float b[]);
    int i,j;
    float sum,tau;

    for (j=1;j<n;j++) {          Form Q^T · b.
        for (sum=0.0,i=j;i<=n;i++) sum += a[i][j]*b[i];
        tau=sum/c[j];
        for (i=j;i<=n;i++) b[i] -= tau*a[i][j];
    }
    rsolv(a,n,d,b);              Solve R · x = Q^T · b.
}
```

```
void rsolv(float **a, int n, float d[], float b[])
```
Solves the set of `n` linear equations $\mathbf{R} \cdot \mathbf{x} = \mathbf{b}$, where $\mathbf{R}$ is an upper triangular matrix stored in `a` and `d`. `a[1..n][1..n]` and `d[1..n]` are input as the output of the routine `qrdcmp` and are not modified. `b[1..n]` is input as the right-hand side vector, and is overwritten with the solution vector on output.

```
{
    int i,j;
    float sum;

    b[n] /= d[n];
    for (i=n-1;i>=1;i--) {
        for (sum=0.0,j=i+1;j<=n;j++) sum += a[i][j]*b[j];
        b[i]=(b[i]-sum)/d[i];
    }
}
```

See [2] for details on how to use QR decomposition for constructing orthogonal bases, and for solving least-squares problems. (We prefer to use SVD, §2.6, for these purposes, because of its greater diagnostic capability in pathological cases.)

Updating a QR decomposition

Some numerical algorithms involve solving a succession of linear systems each of which differs only slightly from its predecessor. Instead of doing $O(N^3)$ operations each time to solve the equations from scratch, one can often update a matrix factorization in $O(N^2)$ operations and use the new factorization to solve the next set of linear equations. The LU decomposition is complicated to update because of pivoting. However, QR turns out to be quite simple for a very common kind of update,

$$\mathbf{A} \to \mathbf{A} + \mathbf{s} \otimes \mathbf{t} \tag{2.10.7}$$

(compare equation 2.7.1). In practice it is more convenient to work with the equivalent form

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \quad \to \quad \mathbf{A}' = \mathbf{Q}' \cdot \mathbf{R}' = \mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v}) \tag{2.10.8}$$

One can go back and forth between equations (2.10.7) and (2.10.8) using the fact that $\mathbf{Q}$ is orthogonal, giving

$$\mathbf{t} = \mathbf{v} \quad \text{and either} \quad \mathbf{s} = \mathbf{Q} \cdot \mathbf{u} \quad \text{or} \quad \mathbf{u} = \mathbf{Q}^T \cdot \mathbf{s} \tag{2.10.9}$$

The algorithm [2] has two phases. In the first we apply $N-1$ Jacobi rotations (§11.1) to reduce $\mathbf{R} + \mathbf{u} \otimes \mathbf{v}$ to upper Hessenberg form. Another $N-1$ Jacobi rotations transform this upper Hessenberg matrix to the new upper triangular matrix $\mathbf{R}'$. The matrix $\mathbf{Q}'$ is simply the product of $\mathbf{Q}$ with the $2(N-1)$ Jacobi rotations. In applications we usually want $\mathbf{Q}^T$, and the algorithm can easily be rearranged to work with this matrix instead of with $\mathbf{Q}$.

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

void qrupdt(float **r, float **qt, int n, float u[], float v[])
```
Given the QR decomposition of some n $\times$ n matrix, calculates the QR decomposition of the matrix $\mathbf{Q}\cdot(\mathbf{R}+\mathbf{u}\otimes\mathbf{v})$. The quantities are dimensioned as r[1..n][1..n], qt[1..n][1..n], u[1..n], and v[1..n]. Note that $\mathbf{Q}^T$ is input and returned in qt.
```
{
    void rotate(float **r, float **qt, int n, int i, float a, float b);
    int i,j,k;

    for (k=n;k>=1;k--) {                Find largest k such that u[k] ≠ 0.
        if (u[k]) break;
    }
    if (k < 1) k=1;
    for (i=k-1;i>=1;i--) {              Transform R + u ⊗ v to upper Hessenberg.
        rotate(r,qt,n,i,u[i],-u[i+1]);
        if (u[i] == 0.0) u[i]=fabs(u[i+1]);
        else if (fabs(u[i]) > fabs(u[i+1]))
            u[i]=fabs(u[i])*sqrt(1.0+SQR(u[i+1]/u[i]));
        else u[i]=fabs(u[i+1])*sqrt(1.0+SQR(u[i]/u[i+1]));
    }
    for (j=1;j<=n;j++) r[1][j] += u[1]*v[j];
    for (i=1;i<k;i++)                   Transform upper Hessenberg matrix to upper tri-
        rotate(r,qt,n,i,r[i][i],-r[i+1][i]);      angular.
}
```

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

void rotate(float **r, float **qt, int n, int i, float a, float b)
```
Given matrices r[1..n][1..n] and qt[1..n][1..n], carry out a Jacobi rotation on rows i and i + 1 of each matrix. a and b are the parameters of the rotation: $\cos\theta = a/\sqrt{a^2+b^2}$, $\sin\theta = b/\sqrt{a^2+b^2}$.
```
{
    int j;
    float c,fact,s,w,y;

    if (a == 0.0) {                 Avoid unnecessary overflow or underflow.
        c=0.0;
        s=(b >= 0.0 ? 1.0 : -1.0);
    } else if (fabs(a) > fabs(b)) {
        fact=b/a;
        c=SIGN(1.0/sqrt(1.0+(fact*fact)),a);
        s=fact*c;
    } else {
        fact=a/b;
        s=SIGN(1.0/sqrt(1.0+(fact*fact)),b);
        c=fact*s;
    }
    for (j=i;j<=n;j++) {            Premultiply r by Jacobi rotation.
        y=r[i][j];
        w=r[i+1][j];
        r[i][j]=c*y-s*w;
        r[i+1][j]=s*y+c*w;
    }
    for (j=1;j<=n;j++) {            Premultiply qt by Jacobi rotation.
        y=qt[i][j];
        w=qt[i+1][j];
        qt[i][j]=c*y-s*w;
        qt[i+1][j]=s*y+c*w;
    }
}
```

We will make use of QR decomposition, and its updating, in §9.7.

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/8. [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §§5.2, 5.3, 12.6. [2]

2.11 Is Matrix Inversion an N^3 Process?

We close this chapter with a little entertainment, a bit of algorithmic prestidigitation which probes more deeply into the subject of matrix inversion. We start with a seemingly simple question:

How many individual multiplications does it take to perform the matrix multiplication of two 2×2 matrices,

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \tag{2.11.1}$$

Eight, right? Here they are written explicitly:

$$\begin{aligned} c_{11} &= a_{11} \times b_{11} + a_{12} \times b_{21} \\ c_{12} &= a_{11} \times b_{12} + a_{12} \times b_{22} \\ c_{21} &= a_{21} \times b_{11} + a_{22} \times b_{21} \\ c_{22} &= a_{21} \times b_{12} + a_{22} \times b_{22} \end{aligned} \tag{2.11.2}$$

Do you think that one can write formulas for the c's that involve only *seven* multiplications? (Try it yourself, before reading on.)

Such a set of formulas was, in fact, discovered by Strassen [1]. The formulas are:

$$\begin{aligned} Q_1 &\equiv (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\ Q_2 &\equiv (a_{21} + a_{22}) \times b_{11} \\ Q_3 &\equiv a_{11} \times (b_{12} - b_{22}) \\ Q_4 &\equiv a_{22} \times (-b_{11} + b_{21}) \\ Q_5 &\equiv (a_{11} + a_{12}) \times b_{22} \\ Q_6 &\equiv (-a_{11} + a_{21}) \times (b_{11} + b_{12}) \\ Q_7 &\equiv (a_{12} - a_{22}) \times (b_{21} + b_{22}) \end{aligned} \tag{2.11.3}$$

in terms of which

$$\begin{aligned} c_{11} &= Q_1 + Q_4 - Q_5 + Q_7 \\ c_{21} &= Q_2 + Q_4 \\ c_{12} &= Q_3 + Q_5 \\ c_{22} &= Q_1 + Q_3 - Q_2 + Q_6 \end{aligned} \tag{2.11.4}$$

What's the use of this? There is one fewer multiplication than in equation (2.11.2), but *many more* additions and subtractions. It is not clear that anything has been gained. But notice that in (2.11.3) the a's and b's are never commuted. Therefore (2.11.3) and (2.11.4) are valid when the a's and b's are themselves matrices. The problem of multiplying two very large matrices (of order $N = 2^m$ for some integer *m*) can now be broken down recursively by partitioning the matrices into quarters, sixteenths, etc. And note the key point: The savings is not just a factor "7/8"; it is that factor at *each* hierarchical level of the recursion. In total it reduces the process of matrix multiplication to order $N^{\log_2 7}$ instead of N^3.

What about all the extra additions in (2.11.3)–(2.11.4)? Don't they outweigh the advantage of the fewer multiplications? For large N, it turns out that there are six times as many additions as multiplications implied by (2.11.3)–(2.11.4). But, if N is very large, this constant factor is no match for the change in the *exponent* from N^3 to $N^{\log_2 7}$.

With this "fast" matrix multiplication, Strassen also obtained a surprising result for matrix inversion [1]. Suppose that the matrices

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \tag{2.11.5}$$

are inverses of each other. Then the c's can be obtained from the a's by the following operations (compare equations 2.7.22 and 2.7.25):

$$\begin{aligned} R_1 &= \text{Inverse}(a_{11}) \\ R_2 &= a_{21} \times R_1 \\ R_3 &= R_1 \times a_{12} \\ R_4 &= a_{21} \times R_3 \\ R_5 &= R_4 - a_{22} \\ R_6 &= \text{Inverse}(R_5) \\ c_{12} &= R_3 \times R_6 \\ c_{21} &= R_6 \times R_2 \\ R_7 &= R_3 \times c_{21} \\ c_{11} &= R_1 - R_7 \\ c_{22} &= -R_6 \end{aligned} \tag{2.11.6}$$

In (2.11.6) the "inverse" operator occurs just twice. It is to be interpreted as the reciprocal if the a's and c's are scalars, but as matrix inversion if the a's and c's are themselves submatrices. Imagine doing the inversion of a very large matrix, of order $N = 2^m$, recursively by partitions in half. At each step, halving the order *doubles* the number of inverse operations. But this means that there are only N divisions in all! So divisions don't dominate in the recursive use of (2.11.6). Equation (2.11.6) is dominated, in fact, by its 6 multiplications. Since these can be done by an $N^{\log_2 7}$ algorithm, so can the matrix inversion!

This is fun, but let's look at practicalities: If you estimate how large N has to be before the difference between exponent 3 and exponent $\log_2 7 = 2.807$ is substantial enough to outweigh the bookkeeping overhead, arising from the complicated nature of the recursive Strassen algorithm, you will find that LU decomposition is in no immediate danger of becoming obsolete.

If, on the other hand, you like this kind of fun, then try these: (1) Can you multiply the complex numbers $(a+ib)$ and $(c+id)$ in only *three* real multiplications? [Answer: see §5.4.] (2) Can you evaluate a general fourth-degree polynomial in x for many different values of x with only *three* multiplications per evaluation? [Answer: see §5.3.]

CITED REFERENCES AND FURTHER READING:

Strassen, V. 1969, *Numerische Mathematik*, vol. 13, pp. 354–356. [1]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley).

Winograd, S. 1971, *Linear Algebra and Its Applications*, vol. 4, pp. 381–388.

Pan, V. Ya. 1980, *SIAM Journal on Computing*, vol. 9, pp. 321–342.

Pan, V. 1984, *How to Multiply Matrices Faster*, Lecture Notes in Computer Science, vol. 179 (New York: Springer-Verlag)

Pan, V. 1984, *SIAM Review*, vol. 26, pp. 393–415. [More recent results that show that an exponent of 2.496 can be achieved — theoretically!]