tion at an early stage is successively magnified until it comes to swamp the true answer. An unstable method would be useful on a hypothetical, perfect computer; but in this imperfect world it is necessary for us to require that algorithms be stable – or if unstable that we use them with great caution.

Here is a simple, if somewhat artificial, example of an unstable algorithm: Suppose that it is desired to calculate all integer powers of the so-called "Golden Mean," the number given by

$$\phi \equiv \frac{\sqrt{5}-1}{2} \approx 0.61803398 \tag{1.2.3}$$

It turns out (you can easily verify) that the powers ϕ^n satisfy a simple recursion relation,

$$\phi^{n+1} = \phi^{n-1} - \phi^n \tag{1.2.4}$$

Thus, knowing the first two values $\phi^0 = 1$ and $\phi^1 = 0.61803398$, we can successively apply (1.2.4) performing only a single subtraction, rather than a slower multiplication by ϕ, at each stage.

Unfortunately, the recurrence (1.2.4) also has *another* solution, namely the value $-\frac{1}{2}(\sqrt{5}+1)$. Since the recurrence is linear, and since this undesired solution has magnitude greater than unity, any small admixture of it introduced by roundoff errors will grow exponentially. On a typical machine with 32-bit wordlength, (1.2.4) starts to give completely wrong answers by about $n = 16$, at which point ϕ^n is only down to 10^{-4}. The recurrence (1.2.4) is *unstable*, and cannot be used for the purpose stated.

We will encounter the question of stability in many more sophisticated guises, later in this book.

REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 1.

Johnson, Lee W., and Riess, R. Dean. 1982, *Numerical Analysis*, 2nd ed. (Reading, Mass.: Addison-Wesley), §1.3.

Chapter 2. Solution of Linear Algebraic Equations

2.0 Introduction

A set of linear algebraic equations looks like this:

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

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 §2.9.

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 a few hundred, by which point the limiting factor is almost always machine time, not accuracy.

Even larger linear sets, N in the thousands, can be solved when the coefficients are sparse (that is, mostly zero), by methods which take advantage of the sparseness. We discuss this further in §2.10.

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 (§2.9) is a technique which 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,

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

By convention, the first index on an element a_{ij} denotes its row, the second index its column. A computer will store the matrix $\mathbf{A}$ as a two-dimensional array. However, computer memory is numbered sequentially by its address, and so is intrinsically one-dimensional. Therefore the two-dimensional array A will, at the hardware level, either be ***stored by columns*** in the order

$$a_{11}, a_{21}, \dots, a_{M1}, a_{12}, a_{22}, \dots, a_{M2}, a_{31}, \dots, a_{1N}, a_{2N}, \dots a_{MN}$$

or else ***stored by rows*** in the order

$$a_{11}, a_{12}, \dots, a_{1M}, a_{21}, a_{22}, \dots, a_{2M}, a_{13}, \dots, a_{N1}, a_{N2}, \dots a_{NM}$$

FORTRAN always stores by columns, and user programs are generally allowed to exploit this fact to their advantage. Pascal generally stores by rows, but user programs are discouraged from using this fact, the one exception being that whole rows `A[i,j]`, `j=1,...,`M can be referenced as `A[i]`. Note one confusing point in the terminology, that a matrix which is stored by columns (as in FORTRAN) has its *row* (i.e. first) index changing most rapidly as one goes linearly through memory, the opposite of a car's odometer!

For most purposes you don't *need* to know what the order of storage is, since you reference an element by its two-dimensional address (a_{34} = `A(3,4)`). It is, however, ***essential*** that you understand the difference between an array's ***physical dimensions*** and its ***logical dimensions***. When you pass an array to a subroutine or other procedure, you must, in general, tell the subroutine *both* of these dimensions. The distinction between them is this: It may happen that you have a 4×4 matrix stored in an array dimensioned as 10×10. This occurs most frequently in practice when you have dimensioned to the largest expected value of N, but are at the moment considering a value of N smaller than that largest possible one. In the example posed, the 16 elements of the matrix do not occupy 16 consecutive memory locations. Rather they are spread out among the 100 dimensioned locations of the array as if the whole 10×10 matrix were filled. Figure 2.0.1 shows an additional example.

If you have a subroutine to invert a matrix, its call might typically look like this

```
CALL MATINV(A,AI,N,NP)
```

Here the subroutine has to be told both the logical size of the matrix that you want to invert (here `N=4`), and the physical size of the array in which it is stored (here `NP=10`).

This seems like a trivial point, and we are sorry to belabor it. But it turns out that *most* reported failures of standard linear equation and matrix manipulation packages are due to user errors in passing inappropriate logical or physical dimensions!

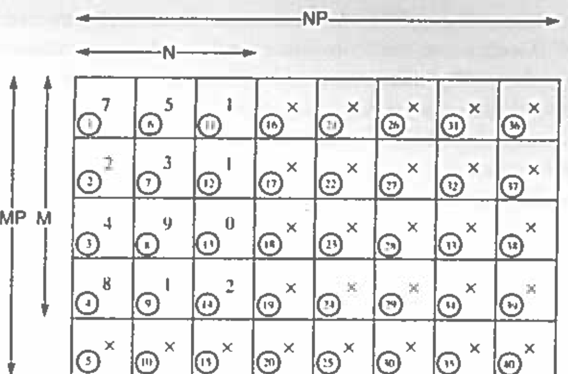

Figure 2.0.1. A matrix of logical dimension M by N is stored in an array of physical dimension NP by NP. Locations marked by "x" contain extraneous information which may be left over from some previous use of the physical array. Circled numbers show the actual ordering of the array in computer memory, not usually relevant to the programmer. Note, however, that the logical array does not occupy consecutive memory locations. To locate an (I,J) element correctly, a subroutine must be told MP and NP, not just I and J.

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–§2.3).
- 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–§2.3).
- 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 §2.4).
- Calculation of the determinant of a square matrix $\mathbf{A}$ (§2.5).

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 either be 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 $\mathbf{A}$ is called

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

This subject is treated in §2.9.

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 which 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 §2.9. 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 (§2.7)
- Various special forms: tridiagonal (§2.6), Toeplitz (§2.8), Vandermonde (§2.8), sparse (§2.10)
- 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. LINPACK was developed at Argonne National Laboratories and deserves particular mention because it is published, documented, and available for free use. Packages available commercially 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 which are *direct* (i.e. execute in a predictable number of operations) from routines which 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 say only a little about iterative methods in this book, in §2.10 and in Chapter 17. These methods are important, but beyond our scope. We will, however, discuss 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 (§2.7).

REFERENCES AND FURTHER READING:

Golub, Gene H., and Van Loan, Charles F. 1983, *Matrix Computations* (Baltimore: Johns Hopkins University Press).

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: Society for Industrial and Applied Mathematics).

Forsythe, George E., and Moler, Cleve B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, N.J.: Prentice-Hall).

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

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

NAG Fortran Library Manual Mark 8, 1980 (NAG Central Office, 7 Banbury Road, Oxford OX26NN, U.K.), chapter F01.

IMSL Library Reference Manual, 1980, ed. 8 (IMSL Inc., 7500 Bellaire Boulevard, Houston TX 77036), chapter L.

Johnson, Lee W., and Riess, R. Dean. 1982, *Numerical Analysis*, 2nd ed. (Reading, Mass.: Addison-Wesley), chapter 2.

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

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

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

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

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

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 **A** and **Y** are square matrices, the $\mathbf{b}_i$'s and $\mathbf{x}_i$'s are column vectors, and **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 **A** and the corresponding *rows* of the **b**'s and of **1**, does not change (or scramble in any way) the solution **x**'s and **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 **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 **b**'s and **1** (which then is no longer the identity matrix, of course).
- Interchanging any two *columns* of **A** gives the same solution set only if we simultaneously interchange corresponding *rows* of the **x**'s and of **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 **A** to the identity matrix. When this is accomplished, the right-hand side becomes the solutions 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 **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 **A**, we of course also do the corresponding operations to the **b**'s and to **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 Wilkinson 1965). 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 **A** is predictably a one or zero (if it is already in a part of the matrix which has been reduced to identity form) *or else* the exactly corresponding element of the matrix which 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:

```
SUBROUTINE GAUSSJ(A,N,NP,B,M,MP)
   Linear equation solution by Gauss-Jordan elimination, equation (2.1.1) above. A is an
   input matrix of N by N elements, stored in an array of physical dimensions NP by NP. B
   is an input matrix of N by M containing the M right-hand side vectors, stored in an array
   of physical dimensions NP by MP. On output, A is replaced by its matrix inverse, and B is
   replaced by the corresponding set of solution vectors.
PARAMETER (NMAX=50)
DIMENSION A(NP,NP),B(NP,MP),IPIV(NMAX),INDXR(NMAX),INDXC(NMAX)
   The integer arrays IPIV, INDXR, and INDXC are used for bookkeeping on the pivoting. NMAX
   should be as large as the largest anticipated value of N.
DO 11 J=1,N
   IPIV(J)=0
11 CONTINUE
DO 22 I=1,N                      This is the main loop over the columns to be reduced.
   BIG=0.
   DO 13 J=1,N                   This is the outer loop of the search for a pivot element.
      IF(IPIV(J).NE.1)THEN
         DO 12 K=1,N
            IF (IPIV(K).EQ.0) THEN
               IF (ABS(A(J,K)).GE.BIG)THEN
                  BIG=ABS(A(J,K))
                  IROW=J
                  ICOL=K
               ENDIF
            ELSE IF (IPIV(K).GT.1) THEN
               PAUSE 'Singular matrix'
            ENDIF
12       CONTINUE
      ENDIF
13 CONTINUE
   IPIV(ICOL)=IPIV(ICOL)+1
   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 Ith pivot element, is the Ith column that is reduced, while INDXR(I) is
   the row in which that pivot element was originally located. If INDXR(I)≠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.NE.ICOL) THEN
      DO 14 L=1,N
         DUM=A(IROW,L)
         A(IROW,L)=A(ICOL,L)
         A(ICOL,L)=DUM
14    CONTINUE
      DO 15 L=1,M
         DUM=B(IROW,L)
         B(IROW,L)=B(ICOL,L)
         B(ICOL,L)=DUM
15    CONTINUE
   ENDIF
   INDXR(I)=IROW                 We are now ready to divide the pivot row by the pivot element, located
   INDXC(I)=ICOL                    at IROW and ICOL.
   IF (A(ICOL,ICOL).EQ.0.) PAUSE 'Singular matrix.'
   PIVINV=1./A(ICOL,ICOL)
```



```
   A(ICOL,ICOL)=1.
   DO 16 L=1,N
      A(ICOL,L)=A(ICOL,L)*PIVINV
16 CONTINUE
   DO 17 L=1,M
      B(ICOL,L)=B(ICOL,L)*PIVINV
17 CONTINUE
   DO 21 LL=1,N                  Next, we reduce the rows...
      IF(LL.NE.ICOL)THEN         ...except for the pivot one, of course.
         DUM=A(LL,ICOL)
         A(LL,ICOL)=0.
         DO 18 L=1,N
            A(LL,L)=A(LL,L)-A(ICOL,L)*DUM
18       CONTINUE
         DO 19 L=1,M
            B(LL,L)=B(LL,L)-B(ICOL,L)*DUM
19       CONTINUE
      ENDIF
21 CONTINUE
22 CONTINUE                      This is the end of the main loop over columns of the reduction.
DO 24 L=N,1,-1                   It only remains to unscramble the solution in view of the column
   IF(INDXR(L).NE.INDXC(L))THEN  interchanges. We do this by interchanging pairs of columns
      DO 23 K=1,N                in the reverse order that the permutation was built up.
         DUM=A(K,INDXR(L))
         A(K,INDXR(L))=A(K,INDXC(L))
         A(K,INDXC(L))=DUM
23    CONTINUE
   ENDIF
24 CONTINUE
RETURN                           And we are done.
END
```

REFERENCES AND FURTHER READING:

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

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

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

Ralston, Anthony, and Rabinowitz, Philip. 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 in 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 advange 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.

REFERENCES AND FURTHER READING:

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

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

Johnson, Lee W., and Riess, R. Dean. 1982, *Numerical Analysis*, 2nd ed. (Reading, Mass.: Addison-Wesley), §2.2.1.

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

2.3 LU Decomposition

Suppose we are able to write the matrix **A** as a product of two matrices,

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

where **L** is *lower triangular* (has elements only on the diagonal and below) and **U** is *upper triangular* (has elements only on the diagonal and above). For the

case of a 4×4 matrix **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 **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,j^{th} 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:

- 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.

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

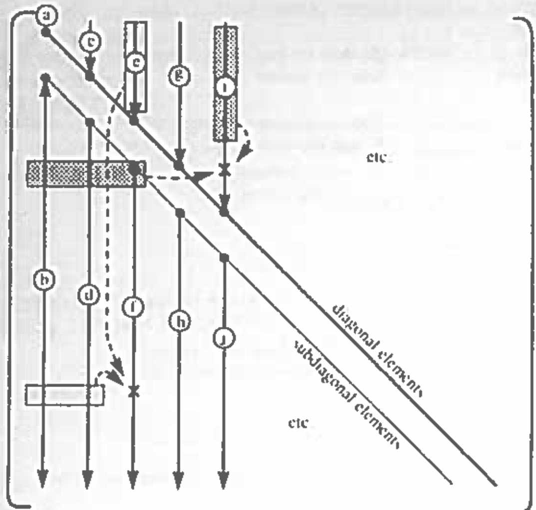

Figure 2.3.1. Crout's algorithm for LU decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lower case letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an "x".

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 which 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 subtlety: 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.

```
SUBROUTINE LUDCMP(A,N,NP,INDX,D)
    Given an N × N matrix A, with physical dimension NP, 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 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
    LUBKSB to solve linear equations or invert a matrix.
PARAMETER (NMAX=100,TINY=1.0E-20)      Largest expected N, and a small number.
DIMENSION A(NP,NP),INDX(N),VV(NMAX)    VV stores the implicit scaling of each row.
D=1.                                   No row interchanges yet.
DO 12 I=1,N                            Loop over rows to get the implicit scaling information.
    AAMAX=0.
    DO 11 J=1,N
```

```
         IF (ABS(A(I,J)).GT.AAMAX) AAMAX=ABS(A(I,J))
11     CONTINUE
       IF (AAMAX.EQ.0.)  PAUSE 'Singular matrix.'   No nonzero largest element.
       VV(I)=1./AAMAX                               Save the scaling.
12   CONTINUE
     DO 19 J=1,N                                    This is the loop over columns of Crout's method.
       DO 14 I=1,J-1                                This is equation 2.3.12 except for i = j.
         SUM=A(I,J)
         DO 13 K=1,I-1
           SUM=SUM-A(I,K)*A(K,J)
13       CONTINUE
         A(I,J)=SUM
14     CONTINUE
       AAMAX=0.                                     Initialize for the search for largest pivot element.
       DO 16 I=J,N                                  This is i = j of equation 2.3.12 and i = j+1...N of equation
         SUM=A(I,J)                                   2.3.13.
         DO 15 K=1,J-1
           SUM=SUM-A(I,K)*A(K,J)
15       CONTINUE
         A(I,J)=SUM
         DUM=VV(I)*ABS(SUM)                         Figure of merit for the pivot.
         IF (DUM.GE.AAMAX) THEN                       Is it better than the best so far?
           IMAX=I
           AAMAX=DUM
         ENDIF
16     CONTINUE
       IF (J.NE.IMAX)THEN                           Do we need to interchange rows?
         DO 17 K=1,N                                Yes, do so...
           DUM=A(IMAX,K)
           A(IMAX,K)=A(J,K)
           A(J,K)=DUM
17       CONTINUE
         D=-D                                       ...and change the parity of D.
         VV(IMAX)=VV(J)                             Also interchange the scale factor.
       ENDIF
       INDX(J)=IMAX
       IF(A(J,J).EQ.0.)A(J,J)=TINY
       IF(J.NE.N)THEN                               Now, finally, divide by the pivot element.
         DUM=1./A(J,J)                              If the pivot element is zero the matrix is singular (at least to the
         DO 18 I=J+1,N                                precision of the algorithm). For some applications on singular
           A(I,J)=A(I,J)*DUM                          matrices, it is desirable to substitute TINY for zero.
18       CONTINUE
       ENDIF
19   CONTINUE                                       Go back for the next column in the reduction.
     RETURN
     END
```

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

```
      SUBROUTINE LUBKSB(A,N,NP,INDX,B)
```

Solves the set of N linear equations $A \cdot X = B$. Here A is input, not as the matrix A but rather as its LU decomposition, determined by the routine LUDCMP. INDX is input as the permutation vector returned by LUDCMP. B is input as the right-hand side vector

B, and returns with the solution vector X. A, N, NP 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.

```
      DIMENSION A(NP,NP),INDX(N),B(N)
      II=0                                When II is set to a positive value, it will become the index of the first
      DO 12 I=1,N                           nonvanishing element of B. We now do the forward substitu-
        LL=INDX(I)                          tion, equation 2.3.6. The only new wrinkle is to unscramble
        SUM=B(LL)                           the permutation as we go.
        B(LL)=B(I)
        IF (II.NE.0)THEN
          DO 11 J=II,I-1
            SUM=SUM-A(I,J)*B(J)
11        CONTINUE
        ELSE IF (SUM.NE.0.)  THEN
          II=I                            A nonzero element was encountered, so from now on we will have to
        ENDIF                               do the sums in the loop above.
        B(I)=SUM
12    CONTINUE
      DO 14 I=N,1,-1                      Now we do the backsubstitution, equation 2.3.7.
        SUM=B(I)
        IF(I.LT.N)THEN
          DO 13 J=I+1,N
            SUM=SUM-A(I,J)*B(J)
13        CONTINUE
        ENDIF
        B(I)=SUM/A(I,I)                   Store a component of the solution vector X.
14    CONTINUE
      RETURN                              All done!
      END
```

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 which 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 $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$:

```
      CALL LUDCMP(A,N,NP,INDX,D)
      CALL LUBKSB(A,N,NP,INDX,B)
```

The answer $\mathbf{x}$ will be returned in B. Your original matrix $\mathbf{A}$ will have been destroyed.

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

```
      CALL LUBKSB(A,N,NP,INDX,B)
```

not, of course, with the original matrix $\mathbf{A}$, but with A and INDX as were already returned from LUDCMP.

REFERENCES AND FURTHER READING:

Golub, Gene H., and Van Loan, Charles F. 1983, *Matrix Computations* (Baltimore: Johns Hopkins University Press), Chapter 4.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: Society for Industrial and Applied Mathematics).

Forsythe, George E., Malcolm, Michael A., and Moler, Cleve B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, N.J.: Prentice-Hall), §3.3.

Forsythe, George E., and Moler, Cleve B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, N.J.: Prentice-Hall), Chapters 9 and 16.

IMSL Library Reference Manual, 1980, ed. 8 (IMSL Inc., 7500 Bellaire Boulevard, Houston TX 77036), Chapter L.

NAG Fortran Library Manual Mark 8, 1980 (NAG Central Office, 7 Banbury Road, Oxford OX26NN, U.K.), Chapters F01, F04.

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

2.4 Inverse of a Matrix

Using the LU decomposition routines of the previous section, it is completely straightforward to find the inverse of a matrix column by column. (There is no better way to do it.)

```
DIMENSION A(NP,NP),Y(NP,NP),INDX(NP)
...
DO 12 I=1,N                                    Set up identity matrix.
   DO 11 J=1,N
      Y(I,J)=0.
11 CONTINUE
   Y(I,I)=1.
12 CONTINUE
CALL LUDCMP(A,N,NP,INDX,D)   Decompose the matrix just once.
DO 13 J=1,N                      Find inverse by columns.
   CALL LUBKSB(A,N,NP,INDX,Y(1,J))     It is necessary to recognize that FORTRAN
13 CONTINUE                            stores two dimensional matrices by column, so that
                                       Y(1,J) is the address of the J^th column of Y. In Pascal you
                                       must create a unit column vector for this call to LUBKSB,
                                       then copy the result into the corresponding column of
                                       the matrix Y on return.
```

The matrix Y will now contain the inverse of the original matrix A, which will have been destroyed.

REFERENCES AND FURTHER READING:

Forsythe, George E., and Moler, Cleve B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 18.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: Society for Industrial and Applied Mathematics).

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

2.5 Determinant of a Matrix

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

$$\det = \prod_{j=1}^{N} \beta_{jj} \tag{2.5.1}$$

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 what was the purpose of returning D in the routine LUDCMP §2.3.)

Calculation of a determinant thus requires one call to LUDCMP, with *no* subsequent backsubstitutions by LUBKSB.

```
DIMENSION A(NP,NP),INDX(NP)
...
CALL LUDCMP(A,N,NP,INDX,D)   This returns D as ±1.
DO 11 J=1,N
   D=D*A(J,J)
11 CONTINUE
```

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 to (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.

REFERENCES AND FURTHER READING:

Forsythe, George E., Malcolm, Michael A., and Moler, Cleve B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, N.J.: Prentice-Hall), p.50.

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

2.6 Tridiagonal 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. For tridiagonal sets, the procedures of LU decomposition, forward- and backsubstitution each take only $O(N)$ operations, and the whole solution can be encoded very concisely. The resulting routine TRIDAG is one which 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

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \cdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.6.1}$$

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

```
SUBROUTINE TRIDAG(A,B,C,R,U,N)
    Solves for a vector U of length N the tridiagonal linear set given by equation (2.6.1). A,
    B, C and R are input vectors and are not modified.
PARAMETER (NMAX=100)                  One vector of workspace, GAM is needed.
DIMENSION GAM(NMAX),A(N),B(N),C(N),R(N),U(N)
IF(B(1).EQ.0.)PAUSE                   If this happens then you should rewrite your equations as a set of
                                          order N - 1, with u2 trivially eliminated.
BET=B(1)
U(1)=R(1)/BET
DO 11 J=2,N                           Decomposition and forward substitution.
    GAM(J)=C(J-1)/BET
    BET=B(J)-A(J)*GAM(J)
    IF(BET.EQ.0.)PAUSE                Algorithm fails; see below.
    U(J)=(R(J)-A(J)*U(J-1))/BET
11  CONTINUE
DO 12 J=N-1,1,-1                      Backsubstitution.
    U(J)=U(J)-GAM(J+1)*U(J+1)
12  CONTINUE
RETURN
END
```

There is no pivoting in TRIDAG. It is for this reason that TRIDAG can fail (PAUSE) 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

$$|b_j| > |a_j| + |c_j| \qquad j = 1, \ldots, N \tag{2.6.2}$$

(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 it appears to be in theory. Of course, should you ever encounter a problem for which TRIDAG fails, you can fall back on elimination with pivoting.

REFERENCES AND FURTHER READING:

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

Dahlquist, Germund, and Bjorck, Ake. 1974, *Numerical Methods* (Englewood Cliffs, N.J.: Prentice-Hall), Example 5.4.3, p.166.

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

2.7 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 *iterated improvement* of the solution. The theory is very straightforward (see Figure 2.7.1): Suppose that a vector $\mathbf{x}$ is the exact solution of the linear set

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.7.1}$$

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

$$\mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \tag{2.7.2}$$

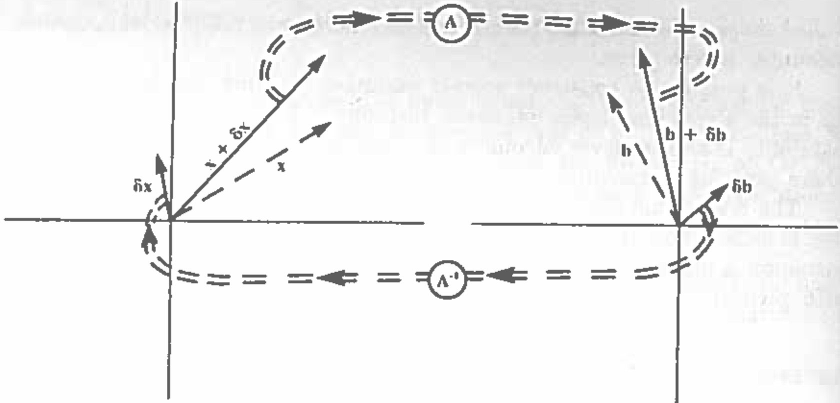

Figure 2.7.1. Iterative improvement of the solution to $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$. The first guess $\mathbf{x}+\delta\mathbf{x}$ is multiplied by $\mathbf{A}$ to produce $\mathbf{b}+\delta\mathbf{b}$. The known vector $\mathbf{b}$ is subtracted, giving $\delta\mathbf{b}$. The linear set with this right hand side is inverted, giving $\delta\mathbf{x}$. This is subtracted from the first guess giving an improved solution $\mathbf{x}$.

Subtracting (2.7.1) from (2.7.2) gives

$$\mathbf{A}\cdot\delta\mathbf{x}=\delta\mathbf{b} \tag{2.7.3}$$

But (2.7.2) can also be solved, trivially, for $\delta\mathbf{b}$. Substituting this into (2.7.3) gives

$$\mathbf{A}\cdot\delta\mathbf{x}=\mathbf{A}\cdot(\mathbf{x}+\delta\mathbf{x})-\mathbf{b} \tag{2.7.4}$$

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 a good idea to calculate the right-hand side in double-precision (if available), since there will be a lot of cancellation in the subtraction of $\mathbf{b}$. Then, we need only solve (2.7.4) 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 (2.7.4) is compute the right-hand side and backsubstitute!

The code to do all this is concise and straightforward:

```
SUBROUTINE MPROVE(A,ALUD,N,NP,INDX,B,X)
   Improves a solution vector X of the linear set of equations A · X = B. The matrix A,
   and the vectors B and X are input, as is the dimension N. Also input is ALUD, the LU
   decomposition of A as returned by LUDCMP, and the vector INDX also returned by that
   routine. On output, only X is modified, to an improved set of values.
PARAMETER (NMAX=100)
DIMENSION A(NP,NP),ALUD(NP,NP),INDX(N),B(N),X(N),R(NMAX)
REAL*8 SDP
DO 12 I=1,N                    Calculate the right-hand side, accumulating the residual in double
  SDP=-B(I)                       precision.
  DO 11 J=1,N
```



```
      SDP=SDP+DBLE(A(I,J))*DBLE(X(J))
11    CONTINUE
    R(I)=SDP
12  CONTINUE
CALL LUBKSB(ALUD,N,NP,INDX,R)       Solve for the error term,
DO 13 I=1,N                         and subtract it from the old solution.
    X(I)=X(I)-R(I)
13  CONTINUE
RETURN
END
```

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, iterated 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.

REFERENCES AND FURTHER READING:

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

Golub, Gene H., and Van Loan, Charles F. 1983, *Matrix Computations* (Baltimore: Johns Hopkins University Press), p.74.

Dahlquist, Germund, and Bjorck, Ake. 1974, *Numerical Methods* (Englewood Cliffs, N.J.: Prentice-Hall), §5.5.6, p.183.

Forsythe, George E., and Moler, Cleve B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 13.

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

2.8 Vandermonde Matrices and Toeplitz Matrices

In §2.6 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 which involves the right kind of particular type, can be considerable.

This section treats two special types of matrices which 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 §12.8.

These are not the *only* special types of matrices worth knowing about. The *Hilbert matrices*, whose components are of the form $a_{ij} = 1/(i+j-1)$, $i,j = 1,\ldots,N$ 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 Forsythe and Moler for details). The Sherman-Morrison and Woodbury formulas, discussed below in §2.10, can sometimes be used to convert new special forms into old ones. Westlake 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 $x_1, x_2, \ldots, x_N$, in terms of which its N^2 components are the integer powers x_i^{j-1}, $i,j = 1,\ldots,N$. 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,

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

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

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

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 von Mises.) 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 which 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

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

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 §5.6, 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. Rybicki.

```
SUBROUTINE VANDER(X,W,Q,N)
    Solves the Vandermonde linear system Σ_{i=1}^{N} x_i^{k-1} w_i = q_k (k = 1,...,N). Input consists
    of the vectors X and Q, each of length N; the vector W is output.
PARAMETER (NMAX=100,ZERO=0.0,ONE=1.0)
    NMAX is the maximum expected value of N. Make constants double precision if you convert
    program to double precision — which is a good idea.
DIMENSION X(N),W(N),Q(N),C(NMAX)
IF(N.EQ.1)THEN
    W(1)=Q(1)
ELSE
    DO 11 I=1,N                      Initialize array.
        C(I)=ZERO
11  CONTINUE
    C(N)=-X(1)                       Coefficients of the master polynomial are found by recursion.
    DO 13 I=2,N
        XX=-X(I)
        DO 12 J=N+1-I,N-1
            C(J)=C(J)+XX*C(J+1)
12      CONTINUE
        C(N)=C(N)+XX
13  CONTINUE
    DO 15 I=1,N                      Each subfactor in turn
        XX=X(I)
        T=ONE
        B=ONE
        S=Q(N)
        K=N
        DO 14 J=2,N                  is synthetically divided,
            K1=K-1
            B=C(K)+XX*B
            S=S+Q(K1)*B              matrix-multiplied by the right-hand side,
            T=XX*T+B
            K=K1
14      CONTINUE
        W(I)=S/T                     and supplied with a denominator.
15  CONTINUE
ENDIF
RETURN
END
```


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:

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

The linear Toeplitz problem can thus be written as

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

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 developed an algorithm for fast solution of the symmetric Toeplitz problem, by a *bordering method*, that is, a recursive procedure which solves the M-dimensional Toeplitz problem

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

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. Robinson and Treitel). 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. 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 §2.6.) 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 (2.8.17)–(2.8.20) 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 $2N-1$.


```
      SUBROUTINE TOEPLZ(R,X,Y,N)
C     Solves the Toeplitz system sum_{j=1}^{N} R_(N+i-j) x_j = y_i (i = 1,...,N). The Toeplitz matrix
C     need not be symmetric. Y and R are input arrays of length N and 2*N-1 respectively. X
C     is the output array, of length N.
      PARAMETER (NMAX=100)
      DIMENSION R(2*N-1),X(N),Y(N),G(NMAX),H(NMAX)
      IF(R(N).EQ.0.) GO TO 99
      X(1)=Y(1)/R(N)                         Initialize for the recursion.
      IF(N.EQ.1)RETURN
      G(1)=R(N-1)/R(N)
      H(1)=R(N+1)/R(N)
      DO 15 M=1,N                            Main loop over the recursion.
        M1=M+1
        SXN=-Y(M1)                           Compute numerator and denominator for x,
        SD=-R(N)
        DO 11 J=1,M
          SXN=SXN+R(N+M1-J)*X(J)
          SD=SD+R(N+M1-J)*G(M-J+1)
11      CONTINUE
        IF(SD.EQ.0.)GO TO 99
        X(M1)=SXN/SD                         whence x.
        DO 12 J=1,M
          X(J)=X(J)-X(M1)*G(M-J+1)
12      CONTINUE
        IF(M1.EQ.N)RETURN
        SGN=-R(N-M1)                         Compute numerator and denominator for G and H.
        SHN=-R(N+M1)
        SGD=-R(N)
        DO 13 J=1,M
          SGN=SGN+R(N+J-M1)*G(J)
          SHN=SHN+R(N+M1-J)*H(J)
          SGD=SGD+R(N+J-M1)*H(M-J+1)
13      CONTINUE
        IF(SD.EQ.0..OR.SGD.EQ.0.)GO TO 99
        G(M1)=SGN/SGD                        whence G and H.
        H(M1)=SHN/SD
        K=M
        M2=(M+1)/2
        PP=G(M1)
        QQ=H(M1)
        DO 14 J=1,M2
          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
          K=K-1
14      CONTINUE
15    CONTINUE                               Back for another recurrence.
      PAUSE 'never get here'
99    PAUSE 'Levinson method fails:  singular principal minor'
      END
```

REFERENCES AND FURTHER READING:

Golub, Gene H., and Van Loan, Charles F. 1983, *Matrix Computations* (Baltimore: Johns Hopkins University Press), Chapter 5 [also treats some other special forms].

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

Forsythe, George E., and Moler, Cleve B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, N.J.: Prentice-Hall), §19.

von Mises, Richard. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), p. 394 ff.

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

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, N.J.: Prentice-Hall), p. 163 ff.

2.9 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 14, 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 **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 **U**, an $N \times N$ diagonal matrix **W** with positive or zero elements, and the transpose of an $N \times N$ orthogonal matrix **V**. The various shapes of these matrices will be made clearer by the following tableau:

$$\begin{pmatrix} \\ \\ \mathbf{A} \\ \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \\ \mathbf{U} \\ \\ \\ \end{pmatrix} \cdot \begin{pmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_N \end{pmatrix} \cdot \begin{pmatrix} & & \\ & \mathbf{V}^T & \\ & & \end{pmatrix} \tag{2.9.1}$$

The matrices **U** and **V** are each orthogonal in the sense that their columns are orthonormal,

$$\sum_{i=1}^{M} U_{ik} U_{in} = \delta_{kn} \qquad \begin{matrix} 1 \le k \le N \\ 1 \le n \le N \end{matrix} \tag{2.9.2}$$

$$\sum_{j=1}^{N} V_{jk} V_{jn} = \delta_{kn} \qquad \begin{matrix} 1 \le k \le N \\ 1 \le n \le N \end{matrix} \tag{2.9.3}$$

or as a tableau,

$$\begin{pmatrix} & & \\ & \mathbf{U}^T & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \\ \mathbf{U} \\ \\ \\ \end{pmatrix} = \begin{pmatrix} & \mathbf{V}^T & \end{pmatrix} \cdot \begin{pmatrix} & \mathbf{V} & \end{pmatrix} = \begin{pmatrix} & \mathbf{1} & \end{pmatrix} \tag{2.9.4}$$

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

The decomposition (2.9.1) 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 **U**, elements of **W**, and columns of **V** (or rows of $\mathbf{V}^T$), or (ii) forming linear combinations of any columns of **U** and **V** whose corresponding elements of **W** happen to be exactly equal.

At the end of this section, we give a routine, `SVDCMP`, that performs SVD on an arbitrary matrix **A**, replacing it by **U** (they are the same shape) and returning **W** and **V** separately. The routine `SVDCMP` is based on a routine by Forsythe et al., which is in turn based on the original routine of Golub and Reinsch, found, in various forms, in Wilkinson and Reinsch, in LINPACK, 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. Along with those already mentioned, another useful reference is Stoer and Bulirsch.

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 **A**, call the routine, and then verify by matrix multiplication that (2.9.1) and (2.9.4) are satisfied. Since these two equations are the only defining requirements for SVD, this procedure is (for the chosen **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 (2.9.1) it now follows immediately that the inverse of **A** is

$$\mathbf{A}^{-1} = \mathbf{V} \cdot [\mathrm{diag}\,(1/w_j)] \cdot \mathbf{U}^T \tag{2.9.5}$$

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 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

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.9.6}$$

where **A** is a square matrix, **b** and **x** are vectors. Equation (2.9.6) 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** which can be found in it) is called the ***nullity*** of **A**.

Now, there is also some subspace of **b** which 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 (2.9.6) in the case that **A** is singular. The important question is whether the vector **b** on the right-hand side 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)

$$\mathbf{x} = \mathbf{V} \cdot [\mathrm{diag}\,(1/w_j)] \cdot (\mathbf{U}^T \cdot \mathbf{b}) \tag{2.9.7}$$

This will be the solution vector of smallest length; the columns of **V** which 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 **W** with some elements zeroed,

$$\begin{aligned} |\mathbf{x} + \mathbf{x}'| &= \left|\mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{x}'\right| \\ &= \left|\mathbf{V} \cdot (\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}')\right| \\ &= \left|\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}'\right| \end{aligned} \tag{2.9.8}$$

Here the first equality follows from (2.9.7), the second and third from the orthonormality of **V**. If you now examine the two terms which 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 **b** is not in the range of the singular matrix **A**, then the set of equations (2.9.6) has no solution. But here is some good news: If **b** is not in the range of **A**, then equation (2.9.7) can still be used to construct a "solution" vector **x**. This vector **x** will not exactly solve $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. But, among all possible vectors **x**, it will do the closest possible job in the least squares sense. In other words (2.9.7) finds

$$\mathbf{x} \quad \text{which minimizes} \quad r \equiv |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}| \tag{2.9.9}$$

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

The proof is similar to (2.9.8): 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

$$\begin{aligned} |\mathbf{A}\cdot\mathbf{x}-\mathbf{b}+\mathbf{b}'| &= \left|(\mathbf{U}\cdot\mathbf{W}\cdot\mathbf{V}^T)\cdot(\mathbf{V}\cdot\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b})-\mathbf{b}+\mathbf{b}'\right| \\ &= \left|(\mathbf{U}\cdot\mathbf{W}\cdot\mathbf{W}^{-1}\cdot\mathbf{U}^T-\mathbf{1})\cdot\mathbf{b}+\mathbf{b}'\right| \\ &= \left|\mathbf{U}\cdot\left[(\mathbf{W}\cdot\mathbf{W}^{-1}-\mathbf{1})\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{U}^T\cdot\mathbf{b}'\right]\right| \\ &= \left|(\mathbf{W}\cdot\mathbf{W}^{-1}-\mathbf{1})\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{U}^T\cdot\mathbf{b}'\right| \end{aligned} \tag{2.9.10}$$

Now, $(\mathbf{W}\cdot\mathbf{W}^{-1}-\mathbf{1})$ 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 2.9.1 summarizes our discussion of SVD thus far.

In the discussion since equation (2.9.6), we have been pretending that a matrix is either 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 (2.9.7) 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 (2.9.7) 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,

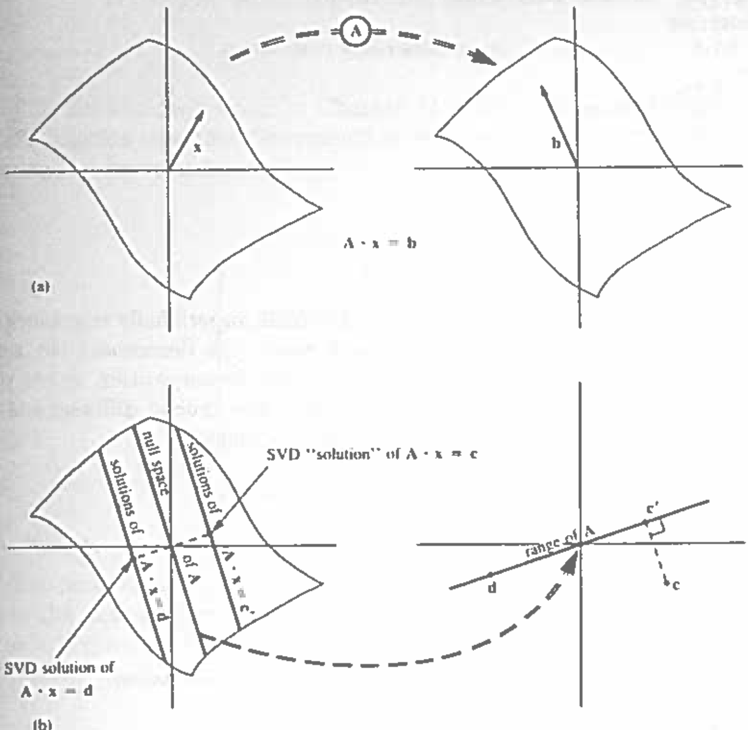

Figure 2.9.1. (a) A nonsingular matrix $\mathbf{A}$ maps a vector space into one of the same dimension. The vector $\mathbf{x}$ is mapped into $\mathbf{b}$, so that $\mathbf{x}$ satisfies the equation $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$. (b) A singular matrix $\mathbf{A}$ maps a vector space into 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.

then this routine is just as ill-conditioned as any direct method, and you are misusing SVD.

```
SUBROUTINE SVBKSB(U,W,V,M,N,MP,NP,B,X)
   Solves A · X = B for a vector X, where A is specified by the arrays U, W, V as returned
   by SVDCMP. M and N are the logical dimensions of A, and will be equal for square matrices.
   MP and NP are the physical dimensions of A. B is the input right-hand side. X is the
   output solution vector. No input quantities are destroyed, so the routine may be called
   sequentially with different B's. M must be greater or equal to N; see SVDCMP.
PARAMETER (NMAX=100)                 Maximum anticipated value of N.
DIMENSION U(MP,NP),W(NP),V(NP,NP),B(MP),X(NP),TMP(NMAX)
DO 12 J=1,N                          Calculate U^T B.
  S=0.
  IF(W(J).NE.0.)THEN                 Nonzero result only if w_j is nonzero.
    DO 11 I=1,M
      S=S+U(I,J)*B(I)
11  CONTINUE
    S=S/W(J)                         This is the divide by w_j.
  ENDIF
```

```
      TMP(J)=S
12    CONTINUE
      DO 14 J=1,N                          Matrix multiply by V to get answer.
        S=0.
        DO 13 JJ=1,N
          S=S+V(J,JJ)*TMP(JJ)
13      CONTINUE
        X(J)=S
14    CONTINUE
      RETURN
      END
```

Note that a typical use of SVDCMP and SVBKSB superficially resembles the typical use of LUDCMP and LUBKSB: In both cases, you decompose the right-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:

```
      DIMENSION A(NP,NP),U(NP,NP),W(NP),V(NP,NP),B(NP),X(NP)
      ...
      DO 12 I=1,N                          Copy A into U if you don't want it to be destroyed.
        DO 11 J=1,N
          U(I,J)=A(I,J)
11      CONTINUE
12    CONTINUE
      CALL SVDCMP(U,N,N,NP,NP,W,V)         SVD the square matrix A.
      WMAX=0.                              Will be the maximum singular value obtained.
      DO 13 J=1,N
        IF(W(J).GT.WMAX)WMAX=W(J)
13    CONTINUE
      WMIN=WMAX*1.0E-6                     This is where we set the threshold for singular values allowed to
      DO 14 J=1,N                            be nonzero. The constant is typical, but not universal.
        IF(W(J).LT.WMIN)W(J)=0.              You have to experiment with your own application.
14    CONTINUE
      CALL SVBKSB(U,W,V,N,N,NP,NP,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.

Augment your left-hand side matrix with rows of zeros underneath its M nonzero rows, until it is filled up to be square, $N \times N$. Similarly augment your right-hand side vector with zeros. You now have a singular set of N equations in N unknowns. Apply SVD as described above. You should expect one zero or negligible w_j for each row of zeros that you added, plus additional ones 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 14, when we wish to find the least-squares solution to an overdetermined set of linear equations. In tableau, the equations to be solved are

$$\begin{pmatrix} \\ \\ \mathbf{A} \\ \\ \\ \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \end{pmatrix} = \begin{pmatrix} \\ \\ \mathbf{b} \\ \\ \\ \end{pmatrix} \tag{2.9.11}$$

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 (2.9.7), which, with non-square matrices, looks like this,

$$\begin{pmatrix} \mathbf{x} \end{pmatrix} = \begin{pmatrix} \mathbf{V} \end{pmatrix} \cdot \begin{pmatrix} \mathrm{diag}(1/w_j) \end{pmatrix} \cdot \begin{pmatrix} \quad \mathbf{U}^T \quad \end{pmatrix} \cdot \begin{pmatrix} \\ \\ \mathbf{b} \\ \\ \\ \end{pmatrix} \tag{2.9.12}$$

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 14.

Approximation of Matrices

Note that equation (2.9.1) can be rewritten to express any matrix A_{ij} as a sum of outer products of columns of **U** and rows of $\mathbf{V}^T$, with the "weighting factors" being the singular values w_j,

$$A_{ij} = \sum_{k=1}^{N} w_k \, U_{ik} V_{jk} \qquad (2.9.13)$$

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 (2.9.13). This means that you only have to store 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

```
SUBROUTINE SVDCMP(A,M,N,MP,NP,W,V)
    Given a matrix A, with logical dimensions M by N and physical dimensions MP by NP, 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 output as a vector W. The matrix
    V (not the transpose V^T) is output as V. M must be greater or equal to N; if it is smaller,
    then A should be filled up to square with zero rows.
PARAMETER (NMAX=100)          Maximum anticipated value of N.
DIMENSION A(MP,NP),W(NP),V(NP,NP),RV1(NMAX)
IF(M.LT.N)PAUSE 'You must augment A with extra zero rows.'
    Householder reduction to bidiagonal form.
G=0.0
SCALE=0.0
ANORM=0.0
DO 25 I=1,N
    L=I+1
    RV1(I)=SCALE*G
    G=0.0
    S=0.0
    SCALE=0.0
    IF (I.LE.M) THEN
        DO 11 K=I,M
            SCALE=SCALE+ABS(A(K,I))
11      CONTINUE
        IF (SCALE.NE.0.0) THEN
            DO 12 K=I,M
                A(K,I)=A(K,I)/SCALE
                S=S+A(K,I)*A(K,I)
12          CONTINUE
            F=A(I,I)
            G=-SIGN(SQRT(S),F)
            H=F*G-S
            A(I,I)=F-G
            IF (I.NE.N) THEN
                DO 15 J=L,N
                    S=0.0
                    DO 13 K=I,M
                        S=S+A(K,I)*A(K,J)
13                  CONTINUE
                    F=S/H
                    DO 14 K=I,M
                        A(K,J)=A(K,J)+F*A(K,I)
14                  CONTINUE
15              CONTINUE
            ENDIF
            DO 16 K= I,M
                A(K,I)=SCALE*A(K,I)
16          CONTINUE
        ENDIF
    ENDIF
    W(I)=SCALE *G
    G=0.0
    S=0.0
    SCALE=0.0
    IF ((I.LE.M).AND.(I.NE.N)) THEN
        DO 17 K=L,N
            SCALE=SCALE+ABS(A(I,K))
17      CONTINUE
        IF (SCALE.NE.0.0) THEN
            DO 18 K=L,N
                A(I,K)=A(I,K)/SCALE
                S=S+A(I,K)*A(I,K)
18          CONTINUE
            F=A(I,L)
            G=-SIGN(SQRT(S),F)
            H=F*G-S
            A(I,L)=F-G
            DO 19 K=L,N
                RV1(K)=A(I,K)/H
19          CONTINUE
            IF (I.NE.M) THEN
                DO 23 J=L,M
                    S=0.0
                    DO 21 K=L,N
                        S=S+A(J,K)*A(I,K)
21                  CONTINUE
                    DO 22 K=L,N
                        A(J,K)=A(J,K)+S*RV1(K)
22                  CONTINUE
23              CONTINUE
            ENDIF
            DO 24 K=L,N
                A(I,K)=SCALE*A(I,K)
24          CONTINUE
        ENDIF
    ENDIF
    ANORM=MAX(ANORM,(ABS(W(I))+ABS(RV1(I))))
25 CONTINUE
    Accumulation of right-hand transformations.
DO 32 I=N,1,-1
    IF (I.LT.N) THEN
        IF (G.NE.0.0) THEN
            DO 26 J=L,N               Double division to avoid possible underflow:
                V(J,I)=(A(I,J)/A(I,L))/G
```

```
          26 CONTINUE
        DO 29 J=L,N
          S=0.0
          DO 27 K=L,N
            S=S+A(I,K)*V(K,J)
27        CONTINUE
          DO 28 K=L,N
            V(K,J)=V(K,J)+S*V(K,I)
28        CONTINUE
29      CONTINUE
      ENDIF
      DO 31 J=L,N
        V(I,J)=0.0
        V(J,I)=0.0
31    CONTINUE
    ENDIF
    V(I,I)=1.0
    G=RV1(I)
    L=I
32  CONTINUE
    Accumulation of left-hand transformations.
  DO 39 I=N,1,-1
    L=I+1
    G=W(I)
    IF (I.LT.N) THEN
      DO 33 J=L,N
        A(I,J)=0.0
33    CONTINUE
    ENDIF
    IF (G.NE.0.0) THEN
      G=1.0/G
      IF (I.NE.N) THEN
        DO 36 J=L,N
          S=0.0
          DO 34 K=L,M
            S=S+A(K,I)*A(K,J)
34        CONTINUE
          F=(S/A(I,I))*G
          DO 35 K=I,M
            A(K,J)=A(K,J)+F*A(K,I)
35        CONTINUE
36      CONTINUE
      ENDIF
      DO 37 J=I,M
        A(J,I)=A(J,I)*G
37    CONTINUE
    ELSE
      DO 38 J= I,M
        A(J,I)=0.0
38    CONTINUE
    ENDIF
    A(I,I)=A(I,I)+1.0
39  CONTINUE
    Diagonalization of the bidiagonal form.
  DO 49 K=N,1,-1                     Loop over singular values.
    DO 48 ITS=1,30                   Loop over allowed iterations.
      DO 41 L=K,1,-1                   Test for splitting:
        NM=L-1                           Note that RV1(1) is always zero.
        IF ((ABS(RV1(L))+ANORM).EQ.ANORM) GO TO 2
        IF ((ABS(W(NM))+ANORM).EQ.ANORM) GO TO 1
41    CONTINUE
1     C=0.0                            Cancellation of RV1(L), if L> 1:
      S=1.0
      DO 43 I=L,K
```



```
        F=S*RV1(I)
        IF ((ABS(F)+ANORM).NE.ANORM) THEN
          G=W(I)
          H=SQRT(F*F+G*G)
          W(I)=H
          H=1.0/H
          C= (G*H)
          S=-(F*H)
          DO 42 J=1,M
            Y=A(J,NM)
            Z=A(J,I)
            A(J,NM)=(Y*C)+(Z*S)
            A(J,I)=-(Y*S)+(Z*C)
42        CONTINUE
        ENDIF
43    CONTINUE
2     Z=W(K)
      IF (L.EQ.K) THEN             Convergence.
        IF (Z.LT.0.0) THEN           Singular value is made nonnegative.
          W(K)=-Z
          DO 44 J=1,N
            V(J,K)=-V(J,K)
44        CONTINUE
        ENDIF
        GO TO 3
      ENDIF
      IF (ITS.EQ.30) PAUSE 'No convergence in 30 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=SQRT(F*F+1.0)
      F=((X-Z)*(X+Z)+H*((Y/(F+SIGN(G,F)))-H))/X
    Next QR transformation:
      C=1.0
      S=1.0
      DO 47 J=L,NM
        I=J+1
        G=RV1(I)
        Y=W(I)
        H=S*G
        G=C*G
        Z=SQRT(F*F+H*H)
        RV1(J)=Z
        C=F/Z
        S=H/Z
        F= (X*C)+(G*S)
        G=-(X*S)+(G*C)
        H=Y*S
        Y=Y*C
        DO 45 JJ=1,N
          X=V(JJ,J)
          Z=V(JJ,I)
          V(JJ,J)= (X*C)+(Z*S)
          V(JJ,I)=-(X*S)+(Z*C)
45      CONTINUE
        Z=SQRT(F*F+H*H)
        W(J)=Z                                   Rotation can be arbitrary if Z=0.
        IF (Z.NE.0.0) THEN
          Z=1.0/Z
          C=F*Z
          S=H*Z
```

```
            ENDIF
            F= (C*G)+(S*Y)
            X=-(S*G)+(C*Y)
            DO 46 JJ=1,M
              Y=A(JJ,J)
              Z=A(JJ,I)
              A(JJ,J)= (Y*C)+(Z*S)
              A(JJ,I)=-(Y*S)+(Z*C)
46          CONTINUE
47        CONTINUE
          RV1(L)=0.0
          RV1(K)=F
          W(K)=X
48      CONTINUE
3     CONTINUE
49    CONTINUE
      RETURN
      END
```

REFERENCES AND FURTHER READING:

Golub, Gene H., and Van Loan, Charles F. 1983, *Matrix Computations* (Baltimore: Johns Hopkins University Press), §8.3 and Chapter 12.

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.

Lawson, Charles L., and Hanson, Richard J. 1974, *Solving Least Squares Problems* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 18.

Forsythe, George E., Malcolm, Michael A., and Moler, Cleve B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 9.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: Society for Industrial and Applied Mathematics), Chapter 11.

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.

2.10 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 §2.6, the tridiagonal matrix. There 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. Most 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 which occur frequently, or which 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 2.10.1) 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 17.

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

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

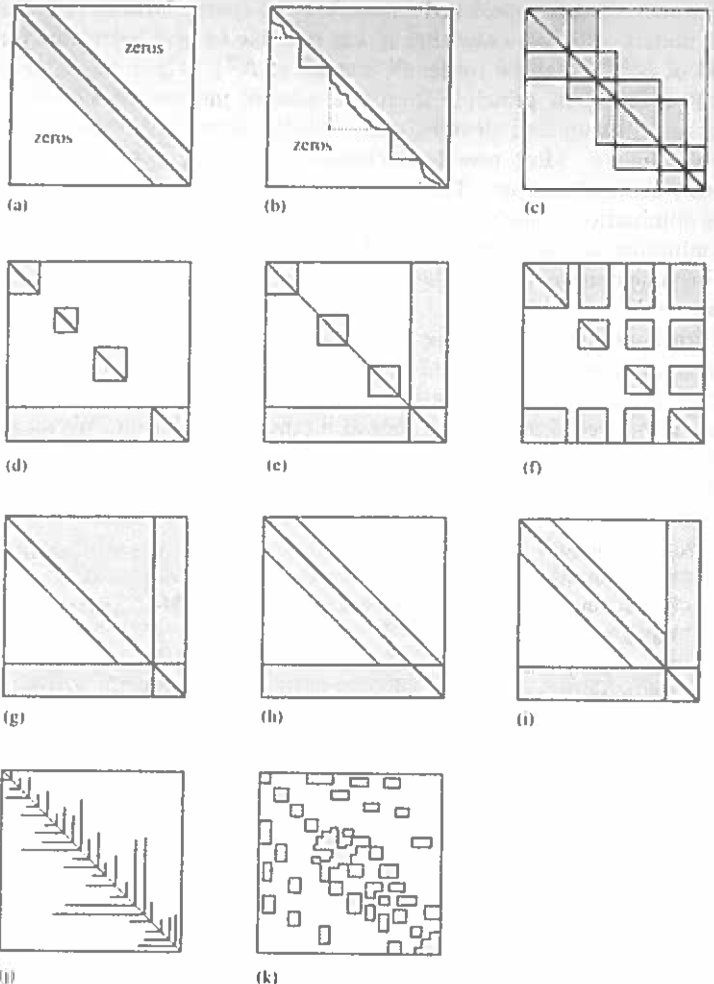

Figure 2.10.1. Some standard forms for sparse matrices. (a) Band diagonal; (b) block triangular; (c) block tridiagonal; (d) singly-bordered block diagonal; (e) doubly-bordered block diagonal; (f) singly-bordered block triangular; (g) bordered band-triangular; (h) and (i) singly- and doubly-bordered band diagonal; (j) and (k) other! (after Tewarson).

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 a couple of ideas which are applicable to some general classes of sparse matrices, and which do not depend on details of the pattern of sparsity.

Sherman-Morrison and Woodbury Formulas

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

$$\mathbf{A} \rightarrow (\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \tag{2.10.1}$$

for some vectors $\mathbf{u}$ and $\mathbf{v}$. If $\mathbf{u}$ is a unit vector $\mathbf{e}_i$, then (2.10.1) adds the components of $\mathbf{v}$ to the ith row. (Recall that $\mathbf{u} \otimes \mathbf{v}$ is a matrix whose i,j^{th} 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 (2.10.1) 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:

$$\begin{aligned}
(\mathbf{A}+\mathbf{u}\otimes\mathbf{v})^{-1} &= (1+\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v})^{-1}\cdot\mathbf{A}^{-1} \\
&= (1-\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v}+\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v}\cdot\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v}-\ldots)\cdot\mathbf{A}^{-1} \\
&= \mathbf{A}^{-1}-\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v}\cdot\mathbf{A}^{-1}(1-\lambda+\lambda^2-\ldots) \\
&= \mathbf{A}^{-1}-\frac{(\mathbf{A}^{-1}\cdot\mathbf{u})\otimes(\mathbf{v}\cdot\mathbf{A}^{-1})}{1+\lambda}
\end{aligned} \tag{2.10.2}$$

where

$$\lambda \equiv \mathbf{v}\cdot\mathbf{A}^{-1}\cdot\mathbf{u} \tag{2.10.3}$$

The second line of (2.10.2) 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 (2.10.2) 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,

$$\mathbf{z} \equiv \mathbf{A}^{-1}\cdot\mathbf{u} \qquad \mathbf{w} \equiv (\mathbf{A}^{-1})^T\cdot\mathbf{v} \qquad \lambda = \mathbf{v}\cdot\mathbf{z} \tag{2.10.4}$$

to get the desired change in the inverse

$$\mathbf{A}^{-1} \quad \rightarrow \quad \mathbf{A}^{-1} - \frac{\mathbf{z}\otimes\mathbf{w}}{1+\lambda} \tag{2.10.5}$$

The whole procedure requires only $3N^2$ multiplies and a like number of adds (an even smaller number if $\mathbf{u}$ or $\mathbf{v}$ 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 $\mathbf{A}$ (e.g.,

a tridiagonal matrix, or some other standard sparse form), then (2.10.4)–(2.10.5) 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 2.10.5). 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

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \cdot \mathbf{x} = \mathbf{b} \tag{2.10.6}$$

then you proceed as follows. Using the fast method that is presumed available for the matrix $\mathbf{A}$, solve the two auxiliary problems

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \qquad \mathbf{A} \cdot \mathbf{z} = \mathbf{u} \tag{2.10.7}$$

for the vectors $\mathbf{y}$ and $\mathbf{z}$. In terms of these,

$$\mathbf{x} = \mathbf{y} - \left[\frac{\mathbf{v} \cdot \mathbf{y}}{1 + (\mathbf{v} \cdot \mathbf{z})}\right] \mathbf{z} \tag{2.10.8}$$

as we see by multiplying (2.10.2) on the right by $\mathbf{b}$.

If you want to add more than a single correction term, then you cannot use (2.10.8) repeatedly, since without storing a new $\mathbf{A}^{-1}$ you will not be able to solve the auxiliary problems (2.10.7) efficiently after the first step. Instead, you need the *Woodbury formula*, which is the block-matrix version of the Sherman-Morrison formula,

$$\begin{aligned}(\mathbf{A} + \mathbf{U} \cdot \mathbf{V}^T)^{-1} \\ = \mathbf{A}^{-1} - \left[\mathbf{A}^{-1} \cdot \mathbf{U} \cdot (\mathbf{1} + \mathbf{V}^T \cdot \mathbf{A}^{-1} \cdot \mathbf{U})^{-1} \cdot \mathbf{V}^T \cdot \mathbf{A}^{-1}\right]\end{aligned} \tag{2.10.9}$$

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,

$$\left[\begin{array}{c} \\ \mathbf{U} \\ \\ \end{array}\right] \cdot \left[\mathbf{1} + \mathbf{V}^T \cdot \mathbf{A}^{-1} \cdot \mathbf{U}\right]^{-1} \cdot \left[\qquad \mathbf{V}^T \qquad\right] \tag{2.10.10}$$

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 $\mathbf{v}_1, \ldots, \mathbf{v}_P$,

$$\mathbf{U} \equiv \left[\mathbf{u}_1\right] \cdots \left[\mathbf{u}_P\right] \qquad \mathbf{V} \equiv \left[\mathbf{v}_1\right] \cdots \left[\mathbf{v}_P\right] \tag{2.10.11}$$

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

$$\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{v}_k\right) = (\mathbf{A} + \mathbf{U} \cdot \mathbf{V}^T) \tag{2.10.12}$$

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

Equation (2.10.12) reveals that, if you have $\mathbf{A}^{-1}$ in storage, then you can either make the P corrections in one fell swoop by using (2.10.9), inverting a $P \times P$ matrix, or else make them by applying (2.10.5) P successive times.

If you don't have storage for $\mathbf{A}^{-1}$, then you *must* use (2.10.9) in the following way: To solve the linear equation

$$\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{v}_k\right) \cdot \mathbf{x} = \mathbf{b} \tag{2.10.13}$$

first solve the P auxiliary problems

$$\begin{aligned} \mathbf{A} \cdot \mathbf{z}_1 &= \mathbf{u}_1 \\ \mathbf{A} \cdot \mathbf{z}_2 &= \mathbf{u}_2 \\ &\cdots \\ \mathbf{A} \cdot \mathbf{z}_P &= \mathbf{u}_P \end{aligned} \tag{2.10.14}$$

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

$$\mathbf{Z} \equiv \begin{bmatrix} \mathbf{z}_1 \end{bmatrix} \cdots \begin{bmatrix} \mathbf{z}_P \end{bmatrix} \tag{2.10.15}$$

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

$$\mathbf{H} \equiv (\mathbf{1} + \mathbf{V}^T \cdot \mathbf{Z})^{-1} \tag{2.10.16}$$

Finally, solve the one further auxiliary problem

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \tag{2.10.17}$$

In terms of these quantities, the solution is given by

$$\mathbf{x} = \mathbf{y} - \mathbf{Z} \cdot \left[\mathbf{H} \cdot (\mathbf{V}^T \cdot \mathbf{y})\right] \tag{2.10.18}$$

Conjugate Gradient Method for a Sparse System

Later in this book, in Chapter 10, we will learn efficient iterative methods that converge to the minimum of a function f of a vector variable $\mathbf{x}$. One method in particular, the ***conjugate gradient method*** demands only that we are able to make two subsidiary calculations, (i) calculate the gradient of the function $\nabla f(\mathbf{x})$ at an arbitrary point, and (ii) minimize f along a specified ray, that is, find the value λ that minimizes the expression $f(\mathbf{x} + \lambda \mathbf{u})$ for specified $\mathbf{x}$ and $\mathbf{u}$. For further explanation and derivation of the conjugate-gradient method, consult §10.6.

Now let us consider the function

$$f(\mathbf{x}) \equiv \frac{1}{2} |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2 \tag{2.10.19}$$

Evidently this function has only a single minimum, at a value $\mathbf{x}$ that satisfies the linear set of equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. A conjugate-gradient minimization will therefore solve that set of equations.

With a short calculation, you will be able to verify that the required two subsidiary calculations can be done as follows:

$$\nabla f(\mathbf{x}) = \mathbf{A}^T \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) \tag{2.10.20}$$

$$\lambda = \frac{-\mathbf{u} \cdot \nabla f}{|\mathbf{A} \cdot \mathbf{u}|^2} \tag{2.10.21}$$

What has this to do with sparse matrices? Equations (2.10.20) and (2.10.21) make only two kinds of references to the matrix $\mathbf{A}$, namely multiplying the matrix by a vector and multiplying its transpose by a vector. If the matrix is sparse, then these multiplications require not the usual N^2 operations, but a smaller number equal to the number of nonzero components. You, the "owner" of the matrix $\mathbf{A}$, can be asked to provide subroutines which perform these sparse matrix multiplications as efficiently as possible. We, the "grand strategists" can supply a general routine which solves the set of linear equations using your subroutines.

This scheme is about the closest thing there is to a "general" sparse matrix routine. In truth, it is not quite as general as it appears, for two related reasons. First, the method is iterative and there are no advance guarantees as to how many iterations it will take to converge (e.g.) to your machine accuracy. It usually takes "on the order of" N iterations, each requiring three of your sparse matrix multiplies; but the constant implicit in the phrase "on the order of" is highly variable. Second, the function f is quadratic in $\mathbf{A}$. This means that the condition-number of the matrix that actually occurs in the function f, $\mathbf{A}^T \cdot \mathbf{A}$, is the square of the condition-number of $\mathbf{A}$ (see §2.9 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.

If you happen to have a sparse matrix $\mathbf{A}$ that is positive definite, then you can eliminate the second of these problems. See the note at the end of §10.6 to find out how to do this. Otherwise, you can give the following routine a try. You may find that it works splendidly – or you may find it to be a bomb – it all depends on your $\mathbf{A}$. (If you want to understand how the routine works, consult §10.6, and compare with the routine FRPRMN.)

```
SUBROUTINE SPARSE(B,N,ASUB,ATSUB,X,RSQ)
```

Solves the linear system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for the vector X of length N, given the right-hand vector B, and given two subroutines, ASUB(XIN,XOUT) and ATSUB(XIN,XOUT), which respectively calculate $\mathbf{A} \cdot \mathbf{x}$ and $\mathbf{A}^T \cdot \mathbf{x}$ for $\mathbf{x}$ given as their first arguments, returning the result in their second arguments. These subroutines should take every advantage of the sparseness of the matrix A. On input, X should be set to a first guess of the desired solution (all zero components is fine). On output, X is the solution vector, and RSQ is the sum of the squares

of the components of the residual vector $A \cdot x - b$. If this is not small, then the matrix is numerically singular and the solution represents a least-squares best approximation.

```
      PARAMETER (NMAX=600,EPS=1.E-6)
C     Maximum anticipated N, and r.m.s. accuracy desired.
      DIMENSION B(N),X(N),G(NMAX),H(NMAX),XI(NMAX),XJ(NMAX)
      EPS2=N*EPS**2                 Criterion for sum-squared residuals.
      IRST=0                        Number of restarts attempted internally.
1     IRST=IRST+1
      CALL ASUB(X,XI)               Evaluate the starting gradient,
      RP=0.
      BSQ=0.
      DO 11 J=1,N
        BSQ=BSQ+B(J)**2             and the magnitude of the right side.
        XI(J)=XI(J)-B(J)
        RP=RP+XI(J)**2
11    CONTINUE
      CALL ATSUB(XI,G)
      DO 12 J=1,N
        G(J)=-G(J)
        H(J)=G(J)
12    CONTINUE
      DO 19 ITER=1,10*N             Main iteration loop.
        CALL ASUB(H,XI)
        ANUM=0.
        ADEN=0.
        DO 13 J=1,N
          ANUM=ANUM+G(J)*H(J)
          ADEN=ADEN+XI(J)**2
13      CONTINUE
        IF(ADEN.EQ.0.)PAUSE 'very singular matrix'
        ANUM=ANUM/ADEN              Equation (2.10.21).
        DO 14 J=1,N
          XI(J)=X(J)
          X(J)=X(J)+ANUM*H(J)
14      CONTINUE
        CALL ASUB(X,XJ)
        RSQ=0.
        DO 15 J=1,N
          XJ(J)=XJ(J)-B(J)
          RSQ=RSQ+XJ(J)**2
15      CONTINUE
        IF(RSQ.EQ.RP.OR.RSQ.LE.BSQ*EPS2)RETURN   Converged. Normal return.
        IF(RSQ.GT.RP)THEN           Not improving. Do a restart.
          DO 16 J=1,N
            X(J)=XI(J)
16        CONTINUE
          IF(IRST.GE.3)RETURN       Return if too many restarts. This is the normal return when we run
                                      into roundoff error before satisfying the return above.
          GO TO 1
        ENDIF
        RP=RSQ
        CALL ATSUB(XJ,XI)           Compute gradient for next iteration.
        GG=0.
        DGG=0.
        DO 17 J=1,N
          GG=GG+G(J)**2
          DGG=DGG+(XI(J)+G(J))*XI(J)
17      CONTINUE
        IF(GG.EQ.0.)RETURN          A rare, but normal, return.
        GAM=DGG/GG
        DO 18 J=1,N
          G(J)=-XI(J)
          H(J)=G(J)+GAM*H(J)
18      CONTINUE
19    CONTINUE
      PAUSE 'too many iterations'
      RETURN
      END
```

So that the specifications for the routines ASUB and ATSUB are clear, we list a couple of dummy versions *which, N.B., do not take any advantage of sparseness!*

```
      SUBROUTINE ASUB(X,V)
      PARAMETER (N=?)
      DIMENSION X(N),V(N)
      COMMON /MAT/ A(N,N)           The matrix is stored somewhere.
      DO 12 I=1,N
        V(I)=0.
        DO 11 J=1,N
          V(I)=V(I)+A(I,J)*X(J)
11      CONTINUE
12    CONTINUE
      RETURN
      END
```

```
      SUBROUTINE ATSUB(X,V)
      PARAMETER (N=?)
      DIMENSION X(N),V(N)
      COMMON /MAT/ A(N,N)
      DO 12 I=1,N
        V(I)=0.
        DO 11 J=1,N
          V(I)=V(I)+A(J,I)*X(J)
11      CONTINUE
12    CONTINUE
      RETURN
      END
```

REFERENCES AND FURTHER READING:

Golub, Gene H., and Van Loan, Charles F. 1983, *Matrix Computations* (Baltimore: Johns Hopkins University Press), Chapters 5 and 10.

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

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

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.).

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

IMSL Library Reference Manual, 1980, ed. 8 (IMSL Inc., 7500 Bellaire Boulevard, Houston TX 77036).

NAG Fortran Library Manual Mark 8, 1980 (NAG Central Office, 7 Banbury Road, Oxford OX26NN, U.K.).

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,

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

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. 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 multiply 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. 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:

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

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:

$$\begin{aligned} ac - bd &= ac - bd \\ ad + bc &= (a + b)(c + d) - ac - bd \end{aligned} \tag{2.11.7}$$

which requires only the three products ac, bd, $(a + b)(c + d)$.] (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.]

REFERENCES AND FURTHER READING:

Strassen, Volker 1969, "Gaussian Elimination is not Optimal," *Numerische Mathematik*, vol. 13, p.354.

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

Pan, V. Ya. 1980, *S.I.A.M. Journal Comput.*, vol. 9, pp. 321-342.

Pan, V. 1984, *S.I.A.M. Review*, vol. 26, pp. 393-415. [More recent results which show that an exponent of 2.496 can be achieved – theoretically!]

Chapter 3. Interpolation and Extrapolation

3.0 Introduction

We sometimes know the value of a function $f(x)$ at a set of points $x_1, x_2, \ldots, x_N$ (say, with $x_1 < \ldots < x_N$), but we don't have an analytic expression for $f(x)$ that lets us calculate its value at an arbitrary point. For example, the $f(x_i)$'s might result from some physical measurement or from long numerical calculation that cannot be cast into a simple functional form. Often the x_i's are equally spaced, but not necessarily.

The task now is to estimate $f(x)$ for arbitrary x by, in some sense, drawing a smooth curve through (and perhaps beyond) the x_i. If the desired x is in between the largest and smallest of the x_i's, the problem is called *interpolation*; if x is outside that range, it is called *extrapolation*, which is considerably more hazardous (as many former stock-market analysts can attest).

Interpolation and extrapolation schemes must model the function, in between or beyond the known points, by some plausible functional form. The form should be sufficiently general so as to be able to approximate large classes of functions which might arise in practice. By far most common among the functional forms used are polynomials (§3.1). Rational functions (quotients of polynomials) also turn out to be extremely useful (§3.2). Trigonometric functions, sines and cosines, give rise to *trigonometric interpolation* and related Fourier methods, which we defer to Chapter 12.

There is an extensive mathematical literature devoted to theorems about what sort of functions can be well approximated by which interpolating functions. These theorems are, alas, almost completely useless in day-to-day work: if we know enough about our function to apply a theorem of any power, we are usually not in the pitiful state of having to interpolate on a table of its values!

Interpolation is related to, but distinct from, *function approximation*. That task consists of finding an approximate (but easily computable) function to use in place of a more complicated one. In the case of interpolation, you are given the function f at points *not of your own choosing*. For the case of function approximation, you are allowed to compute the function f at *any* desired points for the purpose of developing your approximation. We deal with function approximation in Chapter 5.