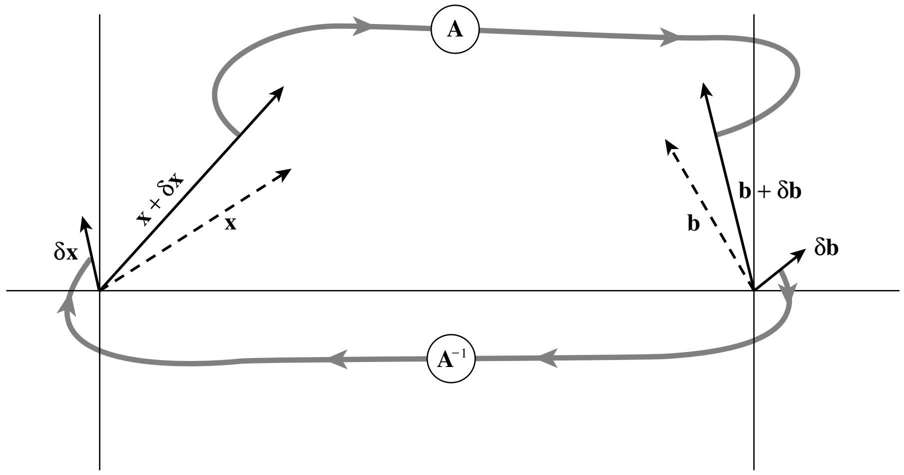

Figure 2.5.1. Iterative improvement of the solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. The first guess $\mathbf{x} + \delta\mathbf{x}$ is multiplied by $\mathbf{A}$ to produce $\mathbf{b} + \delta\mathbf{b}$. The known vector $\mathbf{b}$ is subtracted, giving $\delta\mathbf{b}$. The linear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

2.5 Iterative Improvement of a Solution to Linear Equations

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

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

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

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

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

Subtracting (2.5.1) from (2.5.2) gives

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

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

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

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

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

The code to do all this is concise and straightforward:

```
#include "nrutil.h"

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

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

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

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

More on Iterative Improvement

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

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

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

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

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

Next consider the following formal manipulation:

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

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

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

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

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

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

Then it is easy to show that

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CITED REFERENCES AND FURTHER READING:

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

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

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

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

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

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

2.6 Singular Value Decomposition

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

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

SVD methods are based on the following theorem of linear algebra, whose proof is beyond our scope: Any $M \times N$ matrix **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 (the *singular values*), 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:

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

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

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

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