Linear equations and matrix inversion

We are interested in solving equations of the form

$$\sum_{n=1}^{N} A_{mn} x_n = r_m , \quad m = 1, \ldots , N , \tag{1}$$

—more compactly written $A\,x = r$— with known *coefficient matrix* A_{mn} and known *inhomogeneous term* r_m. We would like to know under what conditions we can find unique values of x_n that simultaneously satisfy the N equations.

The theory of simultaneous linear equations tells us that if not all the r_m's are 0, the necessary and sufficient condition for solvability is that the *determinant*[1] of the matrix A should not be 0. Contrariwise, if $\det(A) = 0$, a solution with $x \neq 0$ can be found only when all the r_m's are 0.

A common way to write such equations explicitly displays the matrix as a square $N \times N$ array and the vectors as columns with N components:

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n1} & \ldots & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \ldots \\ r_n \end{pmatrix} .$$

The formal solution of the linear equation is

$$x = A^{-1} r$$

where A^{-1} is a square matrix such that

$$A^{-1} A = \mathbf{1} \overset{df}{=} \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \ldots & 0 & \ldots & \ldots \\ \ldots & \ldots & \ldots & 1 \end{pmatrix} ,$$

assuming that such a matrix can be constructed. We now investigate how to do this.

1. The determinant of an N'th-order square matix A—denoted by $\det(\mathbf{A})$ or $\|A\|$ — is a number computed from the elements A_{mn} by applying rules familiar from linear algebra. These rules define $\|A\|$ recursively in terms of determinants of square submatrices of A.

1. Cramer's rule

Cramer's rule is a constructive method for solving linear equations by computing determinants. It is completely impractical as a computer algorithm because it requires $O(N!)$ steps to solve N linear equations, whereas pivotal elimination (that we look at below) requires $O(N^3)$ steps, a much smaller number. Nevertheless Cramer's rule is of theoretical interest because it is a closed-form solution.

Consider a square $N\times N$ matrix A. Pretend for the moment we know how to compute the determinant of an $(N-1)\times(N-1)$ matrix. The determinant of A is defined to be

$$\det(A) \overset{df}{=} \sum_{n=0}^{N-1} A_{mn}\, a_{nm} \qquad (2)$$

where the a_{mn}'s are called *co-factors* of the matrix elements A_{mn}, and are in fact determinants of appropriately selected $(N-1)\times(N-1)$ sub-matrices of A.

The sub-matrices are chosen by striking out of A the n'th column and m'th row (leaving an $(N-1)\times(N-1)$ matrix). To illustrate, consider the 3×3 matrix

$$A = \begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix}$$

and produce the co-factor of A_{12} (we attach the factor $(-1)^{m+n}$ to the determinant of the submatrix when we compute a_{nm}):

$$a_{21} = (-1)^{2+1} \times \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & \mathbf{1} & 0 & \mathbf{5} \\ 1 & \mathbf{3} & 2 & \mathbf{4} \\ 2 & 1 & 1 & 6 \end{array}$$

(In the above determinant the struck-out elements are indicated in red while the retained ones are shown in boldface.)

A determinant changes sign when any two rows or any two columns are interchanged. Thus, a determinant with two identical rows or columns vanishes identically. What would happen to Eq. 2 if instead of putting A_{mn} in the sum we put A_{kn} where $k \neq m$? By inspection we realize that this is the same as evaluating a determinant in which two rows are the same, hence we get zero. Thus Eq. 2 can be rewritten more generally

$$\sum_{n=0}^{N-1} A_{kn}\, a_{nm} = \| A \|\, \delta_{km} \ . \qquad (3)$$

This feature of determinants lets us solve the linear equation $A\,x = r$ by construction:

$$x_n = \frac{1}{\det(A)} \sum_{m}^{N-1} a_{nm}\, r_m \,. \qquad (4)$$

We see from Eq. 3 that Eq. 4 solves the equation. Equation 4 also makes clear why the solution cannot be found if $\det(A) = 0$.

A determinant also vanishes when a row is a *linear combination* of any of the other rows. Suppose row 0 can be written

$$a_{0k} \equiv \sum_{m=1}^{N-1} \beta_m\, a_{mk} \;;$$

that is, the 0'th equation can be derived from the other $N-1$ equations, hence it contains no new information. We do not really have N equations for N unknowns, but at most $N-1$ equations. The N unknowns therefore cannot be completely specified, and the determinant tells us this by vanishing.

As an example, we now use Cramer's rule to evaluate the determinant of

$$A = \begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix}.$$

We write

$$\| A \| = A_{00}\, a_{00} + A_{01}\, a_{10} + A_{02}\, a_{20}$$

$$a_{00} = \begin{Vmatrix} 2 & 4 \\ 1 & 6 \end{Vmatrix}$$

$$a_{10} = \begin{Vmatrix} 3 & 4 \\ 1 & 6 \end{Vmatrix} \times (-1)$$

$$a_{20} = \begin{Vmatrix} 3 & 2 \\ 1 & 1 \end{Vmatrix}$$

The determinant of a 1×1 matrix is just the matrix element, hence

$$a_{00} = 2\cdot6 + (-1)\cdot4\cdot1 = 8$$

$$a_{10} = -(18 - 4) = -14$$

$$a_{20} = (3 - 2) = 1$$

and

$$\| A \| = 1\cdot8 + 0\cdot(-14) + 5\cdot1 = 13 \,.$$

How many operations does it take to evaluate a determinant? We see that a determinant of order N requires N determinants of order $N-1$ to be evaluated, as well as N multiplications and $N-1$ additions. If the addition time plus the multiplication time is τ, then

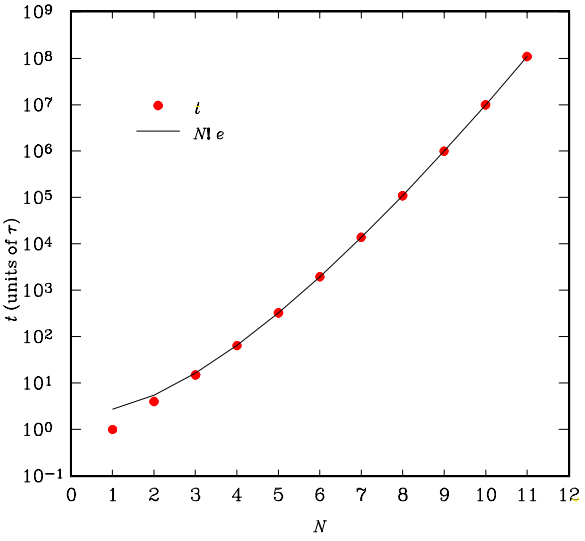
$$T_N \approx N\,(\tau + T_{N-1})\ .$$

The solution to this is[2]

$$T_N = N!\,\tau \sum_{n=0}^{N} \frac{1}{n!} \underset{N\to\infty}{\longrightarrow} N!\,\tau e\ .$$

In other words, the time required to solve N linear equations by Cramer's rule increases so rapidly with N as to render the method thoroughly impractical.

Asymptotic time for Cramer's rule



2. Pivotal elimination

The algorithms we shall use for solving linear equations involve changing the form of the matrix to one whose solution is simpler. Straightforward elimination (as we were taught in high school algebra)—that is, solving for one variable in terms of the rest, substituting that back in the remaining equations and repeating—is still a useful method.

Forty-odd years' experience of solving linear equations on digital computers has taught us to modify the basic elimination procedure to minimize the buildup of round-off error and consequent loss of precision[3]. The necessary additional step involves pivoting—selecting the largest element in a given column to normalize all the other elements. This will be clearer with a concrete illustration rather than further description: consider the 3×3 system of equations:

2. See R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Company, Reading, MA 1983).
3. A computer stores a number with finite precision—perhaps 6–7 decimal places with 32-bit floating-point numbers. This is enough for many purposes, especially in science and engineering, where the data are rarely measured to better than 1% relative precision. Suppose, however, that two numbers, about 10^{-2} in magnitude, are multiplied. Their product is of order 10^{-4} and is known to six significant figures. Now add it to a third number of order unity. The result will be that third number $\pm 10^{-4}$. Later, a fourth number—also of order unity—is subtracted from this sum. The result will be a number of order 10^{-4}, but now known only to *two* significant figures. Matrix arithmetic is full of multiplications and additions. The lesson is clear—to minimize the (inevitable) loss of precision associated with round-off, we must try to keep the magnitudes of products and sums as close as possible.

$$\begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 2 \end{pmatrix} \tag{5}$$

We check that the determinant is $\neq 0$; in fact $\det(A) = 13$. The first step in solving these equations is to transpose rows in A and in r to bring the largest element in the first column to the A_{00} position. We note that

- The x's are not relabeled by row transposition.
- We choose the row ($n=1$ —second row) with the largest (in absolute value) first element $A_{n\,0}$ because we are eventually going to divide by it, and want to minimize the accumulation of roundoff error in the floating point arithmetic.

Transposition gives

$$\begin{pmatrix} 3 & 2 & 4 \\ 1 & 0 & 5 \\ 1 & 1 & 6 \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \\ 2 \end{pmatrix} \tag{6}$$

Now divide row 0 by the new A_{00} (in this case, 3) to get

$$\begin{pmatrix} 1 & \frac{2}{3} & \frac{4}{3} \\ 1 & 0 & 5 \\ 1 & 1 & 6 \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{4}{3} \\ 0 \\ 2 \end{pmatrix} \tag{7}$$

Subtract row 0 times $A_{n\,0}$ from rows with $n>0$:

$$\begin{pmatrix} 1 & \frac{2}{3} & \frac{4}{3} \\ 0 & -\frac{2}{3} & 1\frac{1}{3} \\ 0 & \frac{1}{3} & 1\frac{4}{3} \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{4}{3} \\ -\frac{4}{3} \\ \frac{2}{3} \end{pmatrix} \tag{8}$$

Since $|A_{11}| > |A_{21}|$ we do not bother to switch rows 1 and 2, but divide row 1 by $A_{11} = -2/3$, getting

$$\begin{pmatrix} 1 & \frac{2}{3} & \frac{4}{3} \\ 0 & 1 & -1\frac{1}{2} \\ 0 & \frac{1}{3} & 1\frac{4}{3} \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{4}{3} \\ 2 \\ \frac{2}{3} \end{pmatrix} \tag{9}$$

We now multiply row 1 by $A_{21} = 1/3$ and subtract it from row 2, and also divide through by A_{22} to get

$$\begin{pmatrix} 1 & \frac{2}{3} & \frac{4}{3} \\ 0 & 1 & -1\frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{4}{3} \\ 2 \\ 0 \end{pmatrix} \tag{10}$$

The resulting transformed system of equations now has 0's to the left and below the principal diagonal, and 1's along the diagonal. That is, it has been reduced to upper-triangular form. The solution is almost trivial, as can be seen by actually writing out the equations:

$$1\,x_0 + \tfrac{2}{3}\,x_1 + \tfrac{4}{3}\,x_2 = \tfrac{4}{3} \qquad (11.0)$$

$$0\,x_0 + 1\,x_1 - 11\tfrac{1}{2}\,x_2 = 2 \qquad (11.1)$$

$$0\,x_0 + 0\,x_1 + 1\,x_2 = 0 \qquad (11.2)$$

That is, from Eq. 11.2, x_2=0. We can back-substitute this in 11.1, then solve for x_1 to get

$$x_1 = 2 + 11\tfrac{1}{2} \cdot 0 = 2$$

and similarly, from 21.0 we find

$$x_0 = \tfrac{4}{3} - \tfrac{2}{3}\,2 + \tfrac{4}{3}\,0 = 0\,.$$

We test to see whether this is the correct solution by direct trial:

$$1{\cdot}0 + 0{\cdot}2 + 5{\cdot}0 = 0$$
$$3{\cdot}0 + 2{\cdot}2 + 4{\cdot}0 = 4$$
$$1{\cdot}0 + 2{\cdot}2 + 6{\cdot}0 = 2\,.$$

This works—we have indeed found the solution.

We can express the pivotal elimination algorithm in pseudocode as follows:

```
set n=0
BEGIN
   find pivot element among rows with m>n
   SWAP rows m & n
   divide row n by pivot element:   A(n,m) = A(n,m) / A(n,n)
   subtract:                        A(j,k) = A(j,k) - A(j,n)*A(n,k)
   increment n
UNTIL n=N
   back-substitute
DONE
```

We leave for an exercise the details of implementing and testing this algorithm. However it is worth analyzing its running time. We concentrate on the terms that dominate as $N \to \infty$. The pivot has to be found once for each row; this takes $N-k$ comparisons for the k'th row. Thus we make $\approx N/2$ comparisons of 2 real numbers. (For complex matrices we compare the squared moduli of 2 complex numbers, requiring two multiplications and an addition for each modulus.)

We must divide the k'th (pivot) row by the pivot, at a point in the calculation when the row contains $N-k$ elements that have not been reduced to 0. We do this for $k = 0, 1, \ldots, N-1$, requiring $\approx N^2/2$ divisions.

The back-substitution requires 0 steps for x_{N-1}, 1 multiplication and 1 addition for x_{N-2}, 2 each for x_{N-3}, *etc*. That is, it requires

$$\sum_{k=N-1}^{k=0} (N-1-k) \approx N^2/2$$

multiplications and additions.

The really time-consuming step is multiplying the k'th row by A_{jk}, $j > k$, and subtracting it from row j. Each such step requires $N-k$ multiplications and subtractions, for $j = k+1$ to $N-1$, or $(N-k)\cdot(N-k-1)$ multiplications and subtractions. This has to be repeated for $k = 0$ to $N-2$, giving approximately $N^3/3$ multiplications and subtractions. In other words, the leading contribution to the time is $\tau N^3/3$, which is a lot better than $\tau\, e\, N!$ as with Cramer's rule.

When we optimize for speed, only the innermost loop—requiring $O\left(N^3/3\right)$ operations needs careful tuning; the operations that require time that increases as $O\left(N^2/2\right)$—comparing floating point numbers, dividing by the pivot, and back-substituting—need not be optimized because for large N they are overshadowed by the innermost loop.

Before leaving the subject of elimination methods we should note that it is possible also to interchange columns as well as rows—but this entails permuting the labels on the x_n's. So if elimination with full pivoting is desired, the extra overhead of keeping track of column swaps must be performed.

3. Factorization methods

We now consider methods based on writing the matrix A as a product of two matrices, one that is lower-triangular and one that is upper-triangular,

$$A = L\,U\,.$$

If this can be done without too much trouble, the solution of the linear equations becomes simple: letting

$$y = Ux$$

we solve first the equations

$$Ly = r$$

and then (once y is known)

$$Ux = y\,.$$

Symmetric matrices

If the matrix A is symmetric, $A_{mn} \equiv A_{nm}$, we may write

$$A = S\,\widetilde{S}$$

where S is lower-triangular, and $\widetilde{S}$ is its transpose. Thus we may write

$$\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{pmatrix} = \begin{pmatrix} S_{11} & 0 & \dots & 0 \\ S_{21} & S_{22} & \dots & \dots \\ \dots & \dots & \dots & 0 \\ S_{n1} & S_{n2} & \dots & S_{nn} \end{pmatrix} \begin{pmatrix} S_{11} & S_{21} & \dots & S_{n1} \\ 0 & S_{22} & \dots & S_{n2} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & S_{nn} \end{pmatrix}$$

and find the following equations for the elements of S:

$$\left(S_{11}\right)^2 = A_{11}$$

$$S_{j1}\, S_{11} = A_{j1}\,, \quad j = 2, \dots, n$$

$$\left(S_{22}\right)^2 = A_{22} - \left(S_{21}\right)^2$$

$$S_{j2}\, S_{22} = A_{j2} - S_{j1}\, S_{21}\,, \quad j = 3, \dots, n$$

$$\left(S_{33}\right)^2 = A_{33} - \left(S_{31}\right)^2 - \left(S_{32}\right)^2$$

$$S_{j3}\, S_{33} = A_{j3} - S_{j1}\, S_{31} - S_{j2}\, S_{32}\,, \quad j = 4, \dots, n$$

etc.

The timing is as follows: there are n square roots (they might be imaginary since for a general real-symmetric matrix it is possible to have $\left(S_{jj}\right)^2 < 0$) and about $N^2/2$ subtractions and multiplications to be calculated in getting the diagonal elements of S. But the off-diagonal elements represent the time-consuming part of the algorithm since S_{km} requires $k-1$ subtractions and multiplications (and 1 division) for k running from $m+1$ to n, and of course, m running ftom 1 to n. This results in about $N^3/3$ subtractions and multiplications, as with pivotal elimination. The difference is that the matrix has been factored, hence to solve a second set of linear equations with the same matrix, but a different inhomogeneous term, requires much less time, of order N^2 .

Tridiagonal matrices

Tridiagonal matrices are ones with non-zero elements along the principal diagonal and along the sub- and superdiagonals. That is, they look like

$$A = \begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots \\ a_2 & b_2 & c_2 & 0 & \dots \\ 0 & a_3 & b_3 & c_3 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

One reason they deserve special attention is that triadiagonal matrices arise naturally in many other algorithms—for example solving partial differential equations, or evaluating the coefficients of cubic spline approximations to curves. A second reason is that such matrices require far less storage than general $N \times N$ matrices—$3N$ rather than N^2 memory locations. Finally, to factorize a tridiagonal

matrix into lower- and upper-triangular factors requires time that grows proportional to N rather than N^3.

Factorizing a tridiagonal matrix into a product LU is facilitated by the fact that the diagonal elements of U may be taken to be all 1's, that U may be written as a "bi-diagonal" matrix

$$U = \begin{pmatrix} 1 & u_1 & 0 & \dots \\ 0 & 1 & u_2 & \dots \\ 0 & 0 & 1 & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

and that L can also be taken bi-diagonal, of the form

$$L = \begin{pmatrix} \lambda_1 & 0 & 0 & \dots \\ a_2 & \lambda_2 & 0 & \dots \\ 0 & a_3 & \lambda_3 & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

where the lower sub-diagonal can be taken to be the (already-known) vector a_k , $k = 2, \dots , n$.

The remaining equations to be solved are

$$\begin{aligned} \lambda_1 &= b_1 \\ u_k &= c_k / \lambda_k , \qquad k = 1, \dots , n-1 \\ \lambda_k &= b_k - a_k u_{k-1} , \quad k = 2, \dots , n \end{aligned}$$

The last pair of equations are to be iterated in order, as in the Forth subroutine `}factor` shown below on the following page. The timing of this algorithm is obviously $O(N)$—the time required is proportional to the number of equations to be solved—since there are no nested loops and each loop is traversed only once.

```
\ Linear equations with tridiagonal matrices by LU method
\   reference: Press, et al., "Numerical Recipes" (Cambridge
              U. Press, 1986)
\ -------------------------------------------------
\     (c) Copyright 1999  Julian V. Noble.          \
\       Permission is granted by the author to      \
\       use this software for any application pro-  \
\       vided this copyright notice is preserved.   \
\ -------------------------------------------------
\ This is an ANS Forth program requiring the
\   FLOAT, FLOAT EXT, FILE and TOOLS EXT wordsets.
\ \ Environmental dependences:
\       Assumes independent floating point stack
FALSE [IF] Algorithm:
    Ax = r, A is tridiagonal
    b(k), k=1,...,n are the diagonal elements
    a(k), k=2,...,n are the lower subdiagonal elements
    c(k), k=1,...,n-1 are the upper sudiagonal elements

    Write A = LU where L is lower-triangular and U upper triangular
    If A were a general matrix this would be possible also but the
    decomposition would require O(n^3) steps. For the tridiagonal
    case, however, the decomposition requires only O(n) steps.
    Can assume L and U are bi-diagonal. In the case of L the lower
    subdiagonal is just a(k). In the case of U we can choose the
    diagonal elements to be 1 (unity). Thus we need to determine
    the diagonal elements of L and the upper subdiagonal of U.
    Call them L(k) and U(k) respectively. Then
        L(1) = b(1)
        U(k) = c(k)/L(k), k=1,...,n-1
        L(k) = b(k) - a(k)*U(k-1), k=2,...,n
    Finally let Ux = y and solve first
        Ly = r
    via
        y(1) = r(1)/L(1)
        y(2) = [r(2) - a(2)*y(1)] / L(2) etc.
    then
        x(n) = y(n)
        x(n-1) = y(n-1) - x(n)*U(n-1)  etc.
Usage:
    Say  a{ b{ c{ n }factor  r{ x{ n }backsolve
[THEN]

MARKER -tridiag
BL PARSE undefined DUP PAD C! PAD CHAR+ SWAP CHARS MOVE PAD FIND NIP 0=
[IF]  : undefined    BL WORD  FIND  NIP  0=  ;   [THEN]

include arrays.f
include ftran111.f

20 VALUE Nmax
Nmax long 1 FLOATS 1array a{    \ input array in vector form
Nmax long 1 FLOATS 1array b{
Nmax long 1 FLOATS 1array c{
0 VALUE aa{     0 VALUE bb{     0 VALUE cc{     0 VALUE NN
```

```
Nmax long 1 FLOATS 1array r{    \ inhomogeneous term
Nmax long 1 FLOATS 1array L{    \ diagonal of lower-triangular matrix
Nmax long 1 FLOATS 1array U{    \ subdiagonal of upper-triangular matrix

Nmax long 1 FLOATS 1array x{    \ solution vector

: }factor    ( a{ b{ c{ n --)
    TO NN   TO cc{  TO bb{  TO aa{
    f" bb{ 0 }"   FDUP  F0=  ABORT" Reduce # of equations by 1"
    L{ 0 }  F!
    f" U{ 0 } = cc{ 0 } / L{ 0 }"
    NN 1-  0  DO    f" U{ I } = cc{ I } / L{ I }"
                    f" L{I_1+} = bb{I_1+} - aa{I_1+} * U{I}"
    LOOP ;

: }backsolve    ( r{ x{ n --)
    TO NN   TO aa{  TO bb{
    f" bb{0} = bb{0} / L{0}"
    NN 1 DO     f" bb{ I } = (bb{ I } - a{I}*bb{I_1-}) / L{I}"
    LOOP
    f" aa{ NN_1- } = bb{ NN_1- }"
    0 NN 2 -  DO    f" aa{I} = bb{I} - U{I}*aa{I_1+}"
    LOOP ;
```

LU decomposition of a general matrix

We have seen that it is relatively straightforward to express a tridiagonal matrix as a product of a lower- with an upper-triangular matrix. We now see how this decomposition can be applied to a general matrix[4]. Suppose a given matrix A could be rewritten

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots \\ a_{10} & a_{11} & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix} = L\,U = \begin{pmatrix} \lambda_{00} & 0 & 0 \\ \lambda_{10} & \lambda_{11} & 0 \\ \cdots & \cdots & \cdots \end{pmatrix} \begin{pmatrix} \mu_{00} & \mu_{01} & \cdots \\ 0 & \mu_{11} & \cdots \\ 0 & 0 & \cdots \end{pmatrix};$$

then the solution of

$$L\,U\,x \equiv L\,(U\,x) = r$$

can be found in two steps: as in the tridiagonal case, first solve

$$L\,y = r$$

for $y = U\,x$:

$$\begin{aligned} \lambda_{00}\,y_0 &= r_0 \\ \lambda_{10}\,y_0 + \lambda_{11}\,y_1 &= r_1 \\ \lambda_{20}\,y_0 + \lambda_{21}\,y_1 + \lambda_{22}\,y_2 &= r_2 \\ \cdots \textit{ etc. } \cdots & \end{aligned}$$

—this can be solved successively by forward substitution. Next solve for x by back-substitution:

$$\begin{aligned} \mu_{N-1\,N-1}\,x_{N-1} &= y_{N-1} \\ \mu_{N-2\,N-2}\,x_{N-2} + \mu_{N-2\,N-1}\,x_{N-1} &= y_{N-2} \\ \cdots \textit{ etc. } \cdots & \end{aligned}$$

In solving for y, the n'th term requires n multiplications and n additions. Since we must sum n from 0 to $N-1$, we require $N(N-1)/2 \approx N^2/2$ multiplications and additions. Solving for x similarly requires about $N^2/2$ additions and multiplications. Thus the back-solving process requires about N^2 operations in all, hence the dominant time in solving the equations is the time to LU–decompose the matrix, which turns out to be $O\left(N^3/3\right)$.

The equations to be solved are

$$\sum_{k=0}^{N-1} \lambda_{mk}\,\mu_{kn} = A_{mn}\,,$$

constituting N^2 equations for $N^2 + N$ unknowns. Thus we may arbitrarily impose N extra conditions, which we choose to be

$$\lambda_{kk} = 1\,, \quad k = 0, \ldots, N-1\,.$$

4. See, *e.g.*, W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes* (Cambridge University Press, Cambridge, 1986), p. 31ff.

These equations are easy to solve if we proceed in a sensible order. Clearly,

$$\lambda_{mk} = 0\,,\ m > k$$
$$\mu_{kn} = 0\,,\ k < n$$

so we can divide up the work as follows: for each n, write

$$\mu_{mn} = A_{mn} - \sum_{k=0}^{m-1} \lambda_{mk}\,\mu_{kn}\,,\quad m = 0,\,1,\,\ldots\,,\,n$$

$$\lambda_{mn} = \frac{1}{\mu_{nn}}\left(A_{mn} - \sum_{k=0}^{n-1} \lambda_{mk}\,\mu_{kn}\right),\quad m = n{+}1,\,n{+}2,\,\ldots\,,\,N{-}1$$

It is clear by inspection, that the terms on the right sides of these equations are computed before they are needed. We can store the computed elements λ_{mk} and μ_{kn} in place of the corresponding elements of the original matrix (on the diagonal we store μ_{nn}, since $\lambda_{kk} = 1$ is known).

To limit roundoff error we again pivot, which amounts to permuting so the row with the largest diagonal element is the one we are working on.

4. Eigenvalue problems

Many physical systems can be represented by systems of linear equations. Masses on springs, pendula, electrical circuits, structures[5], and molecules are examples. Such systems often can oscillate sinusoidally. If the amplitude of oscillation remains bounded, such motions are called *stable*. Conversely, sometimes the motions of physical systems are unbounded—the amplitude of any small disturbance will increase exponentially with time. An example is a pencil balanced on its point. Exponentially growing motions are called—for obvious reasons—*unstable*.

Clearly it can be vital to know whether a system is stable or unstable. If stable, we want to know its possible frequencies of free oscillation; whereas for unstable systems we want to know how rapidly disturbances increase in magnitude. Both these problems can be expressed as the question: do linear equations of the form

$$A\,x = \lambda\,\rho\,x$$

have solutions? Here λ is generally a complex number, called the *eigenvalue* (or *characteristic value*) of the preceding equation, and ρ_{mn} is often called the *mass matrix*. Frequently ρ is the unit matrix δ_{mn},

$$\delta_{mn} = \begin{cases} 1, & m = n \\ 0, & m \neq n \end{cases}$$

5. buildings, cars, airplanes, bridges …

but in any case, ρ_{mn} must be positive-definite (we define this below). A non-trivial solution, (that is, with $x \neq 0$), of the equation

$$(A - \lambda \rho)\, x = 0$$

exists if and only if $\det(A - \lambda \rho) = 0$. This fact is useful in solving eigenvalue problems, since the secular equation (or *determinantal equation*)

$$\det(A) = 0$$

is a polynomial of degree N in λ, hence has N roots (either real or complex)[6]. When $\rho_{mn} = \delta_{mn}$, these roots are called the eigenvalues of the matrix A.

Eigenvalue problems arising in physical contexts usually involve a restricted class of matrices, called real-symmetric, or *Hermitian* matrices[7], for which $A^*_{mn} = A_{nm}$. (The superscript * denotes complex conjugation.) All the eigenvalues of Hermitian matrices are real numbers. How do we know? We simply consider the eigenvalaue equation and its complex conjugate:

$$\sum_n A_{mn}\, x_n = \lambda \sum_n \rho_{mn}\, x_n$$

$$\sum_n x^*_n\, A^*_{nm} = \lambda^* \sum_n x^*_n\, \rho^*_{nm}$$

The second of these can be rewritten (using the fact that A and ρ are Hermitian)

$$\sum_m x^*_m\, A_{mn} = \lambda^* \sum_n x^*_m\, \rho_{mn}$$

Multiplying by x^*_m and by x_m, respectively, summing both over m and subtracting gives

$$0 = \left(\lambda^* - \lambda\right) \sum_{n,m} x^*_m\, \rho_{mn}\, x_n \,.$$

However, as noted above, ρ is positive-definite, *i.e.*

$$x^\dagger \cdot \rho \cdot x \equiv \sum_{n,m} x^*_m\, \rho_{mn}\, x_n > 0$$

for any non-zero vector[8] x. Thus, $\lambda^* = \lambda$, that is, λ is real.

---

6. This follows from the **fundamental theorem of algebra**: a polynomial equation, $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_n z^n = 0$, of degree n (in a complex variable z) has exactly n roots.
7. After the French mathematician Charles Hermite (1822-1901).
8. For clarity we now omit the vector "$\rightarrow$" and dyad "$\leftrightarrow$" symbols from vectors and matrices.

In vibration problems, the eigenvalue l usually stands for the square of the (angular) vibration frequency: $\lambda = \omega^2$. Thus, a positive eigenvalue λ corresponds to a (double) real value, $\pm\omega$, of the angular frequency. Real frequencies correspond to sinusoidal vibration with time-dependence $\sin(\omega t)$ or $\cos(\omega t)$.

Conversely, a negative λ corresponds to an imaginary frequency, $\pm i\omega$ and hence to a solution that grows exponentially in time, as

$$\sin(i\,\omega t) = i \sinh(\omega t)$$

$$\cos(i\,\omega t) = \cosh(\omega t)\ .$$

There are many techniques for finding eigenvalues of matrices. If only the largest few are needed, the simplest method is iteration: make an initial guess $x^{(0)}$ and let

$$x^{(n)} = \frac{A\,x^{(n-1)}}{\left(x^{(n-1)},\,\rho\,x^{(n-1)}\right)^{1/2}}\ ;$$

assuming the largest eigenvalue is unique,the sequence of vectors $x^{(n)}$, $n = 1, 2, \ldots$, is guaranteed to converge to the vector corresponding to that eigenvalue, usually after just a few iterations.

If *all* the eigenvalues are wanted, then the only choice is to solve the secular equation for all N roots.

5. Divide and conquer—Strassen's method

Strassen[9] has pointed out that evaluating matrix products by partitioning can substantially speed the most time-consuming matrix operations, multiplication and inversion. For example, it appears as though the product of two partitioned matrices,

$$A\,B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

requires 8 matrix multiplications and 4 matrix additions to evaluate. Strassen has shown that in fact the evaluation can be performed with 7 matrix multiplications:

$$p_1 = \left(A_{11}+A_{22}\right)\left(B_{11}+B_{22}\right)$$

$$p_2 = \left(A_{21}+A_{22}\right)B_{11}$$

$$p_3 = A_{11}\left(B_{12}-B_{22}\right)$$

$$p_4 = \left(-A_{11}+A_{21}\right)\left(B_{11}+B_{12}\right)$$

---

9. V. Strassen, *Numer. Math.* **13** (1969) 184. See also V. Pan, *SIAM Review* **26** (1984) 393.

$$p_5 = (A_{11} + A_{12})B_{22}$$

$$p_6 = A_{22}(-B_{11} + B_{21})$$

$$p_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

and 18 matrix additions:

$$C_{11} = p_1 - p_5 + p_6 + p_7$$

$$C_{12} = p_3 + p_5$$

$$C_{21} = p_2 + p_6$$

$$C_{22} = p_1 - p_2 + p_3 + p_4 .$$

These equations look, at first blush, half as efficient as 8 multiplications and 4 additions. But let us examine the time to multiply two partitioned matrices, first by the straightforward method and then by Strassen's: clearly,

$$M_n = 8\,M_{n/2} + 4\,A_{n/2}$$

where M_n is the multiplication time and A_n the addition time, for square matrices of order n.

Setting $n = 2^k$ we rewrite the above recursion relation as

$$m_k \overset{df}{=} M_{2^k} = 8\,m_k + 4\,a\,4^{k-1} = 8\,m_{k-1} + 4^k\,a$$

where a is the elementary addition time (time to add two numbers). Now we define

$$\sigma_k \overset{df}{=} m_k \times 4^{-k}$$

and see that

$$\sigma_k = 2\,\sigma_{k-1} + a\,,$$

which is a linear difference equation whose solution will be of the form[10]

$$\sigma_k = \mu\,\lambda^k + \beta\,;$$

it is then easy to see that $\lambda = 2$ and $\beta = -a$; thus, recalling that $2^k = n$ we find

$$M_n \approx \mu\,n^3 - a\,n^2$$

where we can identify μ with the time to multiply two numbers.

10. by analogy with linear diferential equations with constant coefficients

Applying the same idea to Strassen's method we obtain the recurrence relation

$$\hat{M}_n = 7\,\hat{M}_{n/2} + 18\,a\,(n/2)^2$$

or, after solving in the same manner,

$$\hat{M}_n \approx \mu\, n^{\lg 7} - 6\,a\,n^2 ,$$

where

$$\lg 7 \overset{df}{=} \log_2 7 = 2.807\ldots$$

That is, partitioning allows a potentially large reduction in the time to multiply dense matrices.

Matrix inversion

By writing a partitioned matrix in the form

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ 0 & Z \end{pmatrix}$$

where

$$Z = A_{22} - A_{21}A_{11}^{-1}A_{12}$$

we may express the *inverse* of A as

$$A^{-1} \equiv \begin{pmatrix} A_{11}^{-1} & -A_{11}^{-1}A_{12}Z^{-1} \\ 0 & Z^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{pmatrix}$$

which leads to the recursion for I_n , the time to invert an n×n matrix:

$$I_n \approx 2\,I_{n/2} + 5\,M_{n/2}$$

whose solution, by the method we used earlier, is

$$I_n = \mu\, n^{\lg 7} + O(n)$$

i.e., the time needed to invert is asymptotically the same as that needed to multiply.

Suppose we merely wish to solve a linear system *without* inverting the matrix: can we gain some speed that way? By partitioning we see that the problem

$$Ax = r$$

can be written

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & Z \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

so that

$$\begin{pmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}.$$

Hence

$$y_1 = r_1$$
$$y_2 = r_2 - A_{21}A_{11}^{-1}r_1$$

and

$$Z x_2 = y_2$$
$$x_1 = A_{11}^{-1}\left(r_1 - A_{12}x_2\right).$$

Thus the solution time satisfies the recurrence relation

$$S_n = S_{n/2} + 3\mu n^{\lg 7} + 3\frac{\mu}{2}n^2$$

whose solution is dominated by

$$S_n = \frac{1}{2}\mu n^{\lg 7}.$$

Recursive solution of linear equations therefore has the same asymptotic running time as matrix multiplication, except that 2× fewer operations are required than for multiplication or inversion. That is, it should be about 2.5× faster to solve a dense system of 1000 linear equations by recursive partitioning than by ordinary Gaussian elimination, even using a scalar processor.

The Appendices list a Forth program for Gaussian elimination with (partial) pivoting, and a FORTRAN program for LU decomposition and back-substitution.

```
\ Linear equation solver using Gaussian elimination with row pivoting

\ This is an ANS Forth program requiring the
\   FLOAT, FLOAT EXT, FILE and TOOLS EXT wordsets.
\
\ Environmental dependences:
\       Assumes independent floating point stack

MARKER -solve

\ conditional compilation

BL PARSE undefined DUP PAD C! PAD CHAR+ SWAP CHARS MOVE PAD FIND NIP 0=
[IF]  : undefined    BL WORD  FIND  NIP  0=  ;   [THEN]

undefined f."       [IF]  INCLUDE ftran111.f        [THEN]
undefined frame|    [IF]  INCLUDE flocals.f         [THEN]
undefined }         [IF]  INCLUDE arrays.f          [THEN]
undefined zdup      [IF]  : zdup  FOVER  FOVER ;    [THEN]
undefined 1/f       [IF]  : 1/f   1.e0  FSWAP  F/ ; [THEN]

\ data structures

0 VALUE Nmax                            \ size of matrix
FVARIABLE  Det                          \ determinant
1.0e-20 FCONSTANT  tiny                 \ condition criterion
1000 long  1 CELLS  1array Iperm{       \ array of permuted row labels

\ locate next pivot row
: }}get_pivot  ( A{{  col# -- Ipiv) ( f: --)
    0                                   \ dummy argument
    LOCALS| Iperm  Col  mat{{ |         \ local names
    Iperm{ Col } @  TO Iperm
    f" ABS( mat{{ Iperm_Col }} ) "      ( f: -- |a[col,col]| )
    Col                                 \ 1st pivot value on stack
    Nmax   Col 1+  ?DO                  \ begin loop
        Iperm{ I } @  TO  Iperm
        f" ABS( mat{{ Iperm_Col }} ) "  ( f: -- |a| |a'| )
        zdup                            ( f: -- |a| |a'| |a| |a'|)
        F<                              \ new.elt > old.elt ?
        IF   FSWAP  DROP  I    THEN
        FDROP
    LOOP                              \ end loop
    FDROP    ;

\ multiply a row by a constant
: }}row*x   ( M{{ row --)  ( f: x -- x)
    0                                   \ dummy argument
    LOCALS| Iperm row#  mat{{ |
    Iperm{ row# }  @  TO Iperm
    Nmax row# ?DO
        FDUP                        ( f: -- x x)
        mat{{ Iperm I }} DUP  F@    ( -- adr[elt])  ( f: -- x x elt)
        F*
        F!
    LOOP  ;   \ Usage:    A{{ 2 }}row*x
```

```
\ subtract a row times a constant from another row
\ this is the innermost loop -- CODE for speed!
: }}r1-r2*x   ( M{{ r1 r2 -- )  ( f: x -- x)  \ initialize assumed
    0 0
    LOCALS| I1 I2 r2  r1  mat{{ |           \ local names
    frame| aa |                             \ local fvariable
    Iperm{ r1 } @  TO  I1
    Iperm{ r2 } @  TO  I2
    Nmax  r2  ?DO                           \ begin loop
        f" mat{{ I1_I }} = mat{{ I1_I }} - mat{{ I2_I }} * aa"
    LOOP                                    \ end loop
    aa F@
    |frame
;

\ v[i] = v[i] - v[j] * x
: }v1-v2*x  ( V{ r1 r2 --  f: x -- )
    LOCALS| r2 r1 v{ |
    FRAME| aa |
    f" v{ Iperm{_r1_}_@ }=v{ Iperm{_r1_}_@ }-v{ Iperm{_r2_}_@ } * aa"
    |FRAME
;

\ permute row labels
: mem_swap  ( adr1 adr2 --)
    LOCALS| a2 a1 |
    a1 @  a2 @   a1 !  a2 !    ;

: }swap  ( I{ m n -- )      \ exchange 2 elts in an integer array
    LOCALS|  N M I{ |
    I{ M }   I{ N }     mem_swap   ;

: initialize    ( A{{  V{ -- A{{  V{ )
    DUP 2@  DROP  TO Nmax               \ Nmax = # of equations
    Nmax 0 DO  I  Iperm{ I } !   LOOP   \ init loop-label array
    f" Det = 1"                         \ init determinant
;

: update_Det    ( -- )  ( f: x -- x )
    FRAME| aa |
    f" - Det * aa"        ( f: -- D' = -x * D)
    FDUP  FABS  tiny  F<
        ABORT" Ill-conditioned matrix"
    Det  F!   aa F@
    |FRAME  ;
```

```
: triangularize ( A{{  V{ -- A{{  V{ )  \ assume INITIALIZEd
    0 0                                 \ dummy arguments
    LOCALS| row Ipiv vec{  mat{{ |      \ local names
    Nmax  0  DO                 \ outer loop - by rows
        mat{{ I }}get_pivot         \ find pivot in col I
        TO  Ipiv                    \ pivot index
        Iperm{ I Ipiv }swap         \ exchange rows
        Iperm{ I } @  TO  row       \ get current row#
        f" mat{{ row_I }}"          \ pivot elt -> fstack
        update_Det                  ( f: x -- x )
        1/f
        mat{{ I }}row*x             \ row[i] = row[i] / pivot
        vec{ row }  DUP  F@  F*  F! \ V[i] = V[i] / pivot
        Nmax   I 1+  ?DO            \ middle loop - by rows
            Iperm{ I } @ TO row
            f" mat{{ row_J }}"        \ multiplier -> fstack
            mat{{ I J }}r1-r2*x       \ row[i] = row[i]-row[j]*x
            vec{  I J  }v1-v2*x       \ same for V{ and drop x
        LOOP                        \ end middle loop
    LOOP                        \ end outer loop
    mat{{  vec{  ;              \ push these addresses

: back_solve ( A{{  V{ -- V{ )  \ assume INITIALIZEd
    0  0e0                      \ dummy arguments
    LOCALS| Jperm vec{ mat{{ |
    FRAME| aa |                 \ aa = sum
    0  Nmax  2 -  DO            \ begin outer loop
        f" aa = 0"              ( f: sum=0)
        Iperm{ I } @  TO Jperm      \ permuted row index
        Nmax I 1+  DO               \ begin inner loop
            f" aa = aa - mat{{ Jperm_I }} * vec{ Iperm{_I_}_@ }"
        LOOP                        \ end inner loop
           f" vec{ Jperm } = vec{ Jperm } + aa"
    -1  +LOOP
    |FRAME
    vec{
;

\ solve Ax = V ; solution vector in V{ , matrix A{{ overwritten
: report    ( V{ --)
    LOCALS| vec{ |
    Nmax 0 DO   CR  ." x(" I . ." )= "
                f" vec{ Iperm{_I_}_@ } "  F.
    LOOP  ;

: }}solve     ( A{{  V{ --)
    initialize  triangularize  back_solve  report ;
\ Usage:  a{{ v{ }}solve
```

```
      SUBROUTINE LUDCMP(A,N,NP,INDX,D)
      PARAMETER (NMAX=100,TINY=1.0E-20)
      DIMENSION A(NP,NP),INDX(N),VV(NMAX)
      D=1.
      DO 12 I=1,N
        AAMAX=0.
        DO 11 J=1,N
          IF (ABS(A(I,J)).GT.AAMAX) THEN
              AAMAX=ABS(A(I,J))
          ENDIF
 11     CONTINUE
        IF (AAMAX.EQ.0.) PAUSE 'Singular ma-
trix.'
        VV(I)=1./AAMAX
 12   CONTINUE
      DO 19 J=1,N
        IF (J.GT.1) THEN
          DO 14 I=1,J-1
            SUM=A(I,J)
            IF (I.GT.1)THEN
              DO 13 K=1,I-1
                SUM=SUM-A(I,K)*A(K,J)
 13           CONTINUE
              A(I,J)=SUM
            ENDIF
 14       CONTINUE
        ENDIF
        AAMAX=0.
        DO 16 I=J,N
          SUM=A(I,J)
          IF (J.GT.1)THEN
            DO 15 K=1,J-1
              SUM=SUM-A(I,K)*A(K,J)
 15          CONTINUE
            A(I,J)=SUM
          ENDIF
          DUM=VV(I)*ABS(SUM)
          IF (DUM.GE.AAMAX) THEN
            IMAX=I
            AAMAX=DUM
          ENDIF
 16     CONTINUE
        IF (J.NE.IMAX)THEN
          DO 17 K=1,N
            DUM=A(IMAX,K)
            A(IMAX,K)=A(J,K)
            A(J,K)=DUM
 17       CONTINUE
          D=-D
          VV(IMAX)=VV(J)
        ENDIF
        INDX(J)=IMAX
        IF(J.NE.N)THEN
          IF(A(J,J).EQ.0.)A(J,J)=TINY
          DUM=1./A(J,J)
          DO 18 I=J+1,N
            A(I,J)=A(I,J)*DUM
 18       CONTINUE
        ENDIF
 19   CONTINUE
      IF(A(N,N).EQ.0.)A(N,N)=TINY
      RETURN
      END
```

```
      SUBROUTINE LUBKSB(A,N,NP,INDX,B)
      DIMENSION A(NP,NP),INDX(N),B(N)
      II=0
      DO 12 I=1,N
        LL=INDX(I)
        SUM=B(LL)
        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
        ENDIF
        B(I)=SUM
12    CONTINUE
      DO 14 I=N,1,-1
        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)
14    CONTINUE
      RETURN
      END
```