Contents

1 Linear Algebra **2**

1 Simultaneous linear equations 2

1 Theory of linear algebraic equations 2

2 Eigenvalue problems 3

2 Solving linear equations 5

1 Cramer's rule 5

2 Pivotal Elimination 7

3 Testing 10

4 Implementing pivotal elimination 12

5 The program 12

6 Timing 21

3 Matrix Inversion 24

1 Linear transformations 24

2 Matrix multiplication 25

3 Matrix Inversion 26

4 Why invert matrices, anyway? 26

5 An Example 27

4 LU decomposition 27

Chapter 1

Linear Algebra

Two common problems in scientific programming are the numerical solution of simultaneous linear algebraic equations and computing the inverse of a given square matrix. This is such an important subject As an exercise in FORTH program development we shall now write programs to solve linear equations and invert matrices.

Simultaneous linear equations

We begin with a dose of mathematics (linear algebra), and then develop programs. Several actual programming session are reproduced *in toto* –mistakes and all– to give the flavor of program development and debugging in FORTH.

Theory of linear algebraic equations

We begin by stating the problem. Given the equations

$$\sum_{n=0}^{N-1} \mathrm{A_{mn} x_n} = \mathrm{b_m} \,. \tag{1.1}$$

– more compactly written $\overleftrightarrow{A} \cdot \vec{x} = \vec{b}$ – with known **coefficient matrix** A_{mn} and known **inhomogeneous term** b_m: under what conditions can we find unique values of x_n that simultaneously satisfy the N equations 1.1?

The theory of simultaneous linear equations tells us that if not all the b_m 's are

0, the necessary and sufficient condition for solvability is that the **determinant**[1] of the matrix $\overleftrightarrow{A}$ should not be 0. Contrariwise, if $\det(\overleftrightarrow{A}) = 0$, a solution with $\vec{x} \neq 0$ *can* be found when $\vec{b} = 0$.

Eigenvalue problems

Many physical systems can be represented by systems of linear equations. Masses on springs, pendula, electrical circuits, structures[2], 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

$$\overleftrightarrow{A} \cdot \vec{x} = \lambda \overleftrightarrow{\rho} \cdot \vec{x} \tag{1.2}$$

have solutions? Here λ is generally a complex number, called the **eigenvalue** (or **characteristic** value) of Eq. 1.2, and $\overleftrightarrow{\rho}$ is often called the **mass matrix**. Frequently $\overleftrightarrow{\rho}$ is the unit matrix **I**,

$$\mathrm{I}_{\mathrm{mn}} = \begin{cases} 1, & m = n \\ 0, & m \neq n, \end{cases} \tag{1.3}$$

but in any case, $\overleftrightarrow{\rho}$ must be **positive-definite** (we define this below). A nontrivial solution, $\vec{x} \neq 0$, of the equation

$$\overleftrightarrow{A} \cdot \vec{x} = 0 \tag{1.4}$$

exists if and only if $\| \overleftrightarrow{A} \| = 0$. This fact is useful in solving **eigenvalue** problems such as Eq. 1.2 above.

The **secular equation** (or determinantal equation)

$$\| \overleftrightarrow{A} - \lambda \overleftrightarrow{\rho} \| = 0 \tag{1.5}$$

[1] The determinant of an N'th-order square matix A – denoted by $\det(A)$ or $\|A\|$ – is a number computed from the elements A_{mn} by apptying rules famitiar from linear algebra. These rules define $\|A\|$ recursively in terms of determinants of matrices of square submatrices of A. See §9§§2.1.

[2] buildings, cars, airplanes, bridges ...

is a polynomial of degree N in λ, hence has N roots (either real or complex)[3]. When $\overleftrightarrow{\rho} = \overleftrightarrow{I}$, these roots are called the **eigenvalues** (or "characteristic values") of the matrix $\overleftrightarrow{A}$.

Eigenvalue problems arising in physical contexts usually involve a restricted class of matrices, called **real-symmetric** or Hermitian (after the French mathematician Hermite) matrices, for which $A^*_{mn} = A_{nm}$. (The superscript * denotes complex conjugation – see Ch. 7.) All the eigenvalues of Hermitian matrices are *real* numbers. How do we know? We simply consider Eq. 1.3 and its **complex conjugate**:

$$\sum_n A_{mn} x_n = \lambda \sum_n \rho_{mn} x_n; \tag{1.6a}$$

$$\sum_n x^*_n A^*_{nm} = \lambda^* \sum_n x^*_n \rho^*_{nm}; \tag{1.6b}$$

Equation 1.6b can be rewritten (using the fact that $\overleftrightarrow{A}$ and $\overleftrightarrow{\rho}$ are Hermitian)

$$\sum_m x^*_m A_{mn} = \lambda^* \sum_n x^*_m \rho_{mn} \tag{1.6c}$$

Multiply 1.6c by x_m and 1.6a by x^*_m, sum both over at and subtract this gives

$$0 = \left(\lambda^* - \lambda\right) \sum_{n,m} x^*_m \rho_{mn} x_n \ . \tag{1.7}$$

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

$$x^\dagger \cdot \rho \cdot x \equiv \sum_{n,m} x^*_m \rho_{mn} x_n > 0 \tag{1.8}$$

for any non-zero vector[4] x. Thus, Eq. 1.7 $\lambda^* \equiv \lambda$, that is, λ is real.

In vibration problems, the eigenvalue λ 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

$$\begin{gathered} \sin(i\,\omega t) = i \sinh(\omega t) \\ \cos(i\,\omega t) = i \cosh(\omega t) \end{gathered} \tag{1.9}$$

[3] 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 solutions.

[4] For clarity we now omit the vector "→" and dyad "↔" symbol from vectors and matrices.

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 1.5 for all N roots.

Solving linear equations

The test-and-development cycle in FORTH is short compared with most languages. It is usually easy to create a working program that subsequently can be tuned for speed, again much more rapidly than with other languages. For me this is the chief benefit of the language.

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

$$\det(A) = \sum_{n=0}^{N-1} A_{mn} a_{nm} \tag{1.10}$$

where the a_{mn}'s are called **co-factors** of the matrix elements A_{mn} and are in fact determinants of specially 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

$$A = \begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix} \tag{1.11}$$

and produce the co-factor of A_{12}:

Column labels

$$a_{21} = (-1)^{2+1} \left\| \begin{array}{cccc} & 0 & 1 & 2 \\ 0 & 1 & \cancel{0} & 5 \\ 1 & \cancel{3} & \cancel{2} & \cancel{4} \\ 2 & 1 & \cancel{1} & 6 \end{array} \right\| \tag{1.12}$$

Row labels

We also attach the factor $(-1)^{m+n}$ to the determinant of the submatrix when we compute a_{nm}.

A determinant changes sign when any two rows or any two columns are interchanged. Thus, a determinant with two identical rows or columns is exactly zeros[5]. What would happen to Eq. 1.11 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. 1.11 can be rewritten more generally

$$\sum_{n=0}^{N-1} A_{kn} a_{nm} = \|A\| \, \delta_{km} \; . \tag{1.13}$$

This feature of determinants lets us solve the linear equation $A \cdot x = b$ by construction: try

$$x_n = \frac{1}{\det(A)} \sum_{m}^{N-1} a_{nm} b_m . \tag{1.14}$$

We see from Eq. 1.13 that Eq. 1.14 solves the equation. Equation 1.14 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

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

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.

[5] The only number equal to its negative is 0.

As an example, we now use Cramer's rule to evaluate the determinant of Eq. 1.11. We will write

$$\begin{aligned}
\|A\| &= A_{00}a_{00} + A_{01}a_{10} + A_{02}a_{20} \\
a_{00} &= \begin{Vmatrix} 2 & 4 \\ 1 & 6 \end{Vmatrix} \\
a_{10} &= \begin{Vmatrix} 3 & 4 \\ 1 & 6 \end{Vmatrix}(-1) \\
a_{20} &= \begin{Vmatrix} 3 & 2 \\ 1 & 1 \end{Vmatrix}
\end{aligned}$$

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

$$\begin{aligned}
[a_{00}]_r^1 &= 2 \cdot 6 + (-1) \cdot 4 \cdot 1 = 8 \qquad \text{Err(1)} \\
a_{10} &= -(18 - 4) = -14 \\
a_{20} &= (3 - 2) = 1 \\
\|A\| &= 1 \cdot 8 + 0 \cdot (-14) + 5 \cdot 1 = 13 \ .
\end{aligned}$$

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

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

It is easy to see[67] the solution to this is

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

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.

Pivotal Elimination

The algorithm we shall use for solving linear equations is elimination, just as we were taught in high school algebra. However we modify it to take into account the experience of 40 years's solving linear equations on digital computers (not

[1] ERRATUM! *typo* (original text was: "a_{10}")

[6] Professors always say this, hee, hee! See R. Sedgewick, Algorithms. (Addison-Wesley Publishing Company, Reading, MA 1983).

[7] $N!$ means $N\times(N-1)\times(N-2)\times\ldots\times2\times1$. The base of the "natural" fogarithms is $e = 2.7182818\ldots$

to mention a previous 20 years's worth on mechanical calculators!), to minimize the buildup of round-off error and consequent loss of precision [8].

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:

$$\begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 2 \end{pmatrix} \tag{1.15}$$

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 $\mathbf{A}$ and in $\mathbf{b}$ to bring the largest element in the first column to the A_{00} position.

Notes:

- The x's are not relabeled by this transposition.
- We choose the row (n=1 –second row) with the largest (in absolute value) element A_{n0} 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

$$\begin{pmatrix} 3 & 2 & 4 \\ 1 & 0 & 5 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \\ 2 \end{pmatrix} \tag{1.16}$$

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

$$\begin{pmatrix} 1 & {}^{2}/_{3} & {}^{4}/_{3} \\ 1 & 0 & 5 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} {}^{4}/_{3} \\ 0 \\ 2 \end{pmatrix} \tag{1.17}$$

Subtract row 0 times A_{n0} from rows with n > 0

$$\begin{pmatrix} 1 & {}^{2}/_{3} & {}^{4}/_{3} \\ 0 & {}^{-2}/_{3} & {}^{11}/_{3} \\ 0 & {}^{1}/_{3} & {}^{14}/_{3} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} {}^{4}/_{3} \\ {}^{-4}/_{3} \\ {}^{2}/_{3} \end{pmatrix} \tag{1.18}$$

[8] A Computer stores a number with finite precision – say 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 dose as possible.

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

$$\begin{pmatrix} 1 & {}^{2}/_{3} & {}^{4}/_{3} \\ 0 & 1 & {}^{-11}/_{2} \\ 0 & {}^{1}/_{3} & {}^{14}/_{3} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} {}^{4}/_{3} \\ 2 \\ {}^{2}/_{3} \end{pmatrix} \tag{1.19}$$

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

$$\begin{pmatrix} 1 & {}^{2}/_{3} & {}^{4}/_{3} \\ 0 & 1 & {}^{-11}/_{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} {}^{4}/_{3} \\ 2 \\ 0 \end{pmatrix} \tag{1.20}$$

The resulting transformed system of equations now has 0's to the left and below the principal diagonal, and 1's along the diagonal. Its solution is almost trivial, as can be seen by actually writing out the equations:

$$1x_0 + 2/3x_1 + 4/3x_2 = 4/3 \tag{1.21a}$$

$$0x_0 + 1x_1 + (-11/2)x_2 = 2 \tag{1.21b}$$

$$0x_0 + 0x_1 + 1x_2 = 0 \tag{1.21c}$$

That is, from Eq. 1.21c, $x_2 = 0$. We can back-substitute this in 1.21b, then solve for x_1, to get

$$x_1 = 2 - (-11/2) \cdot (0) = 2$$

and similarly, from 1.21a we find

$$x_0 = 4/3 - 2/3\,(2) + 4/3\,(0) = 0\ .$$

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

$$\begin{aligned} 1 \cdot 0 + 0 \cdot 2 + 5 \cdot 0 &= 0 \\ 3 \cdot 0 + 2 \cdot 2 + 4 \cdot 0 &= 4 \\ 1 \cdot 0 + 2 \cdot 2 + 6 \cdot 0 &= 2 \end{aligned}$$

This works – we have indeed found the solution.

How much time is needed to perform pivotal elimination? 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 ≈ 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 have to 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 have to do this for k = 0, 1, ..., 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 $O(N^3/3)$ operations needs careful tuning; the $O(N^2/2)$ operations – comparing floating point numbers, dividing by the pivot, and back-substituting – need not be optimized bgecause for large N they are overshadowed by the innermost loop[9].

Testing

Since we have worked out a specific 3×3 set of linear equations, we might as well use it for testing and debugging. Begin with some test words that do the following jobs:

- Create matrix and vector (inhomogeneous term)
- Initialize them to the values in the example
- Display the matrix at any stage of the calculation
- Display inhomogeneous term at any stage

```
\ Display words
\ V{ or M{{ stands for what an array puts on stack
GU: G. F. X. ;
: .M ( M{{ --) FINIT DUP
    D.LEN 0 DO CR DUP
        D.LEN 0 DO DUP ( -- M{{ M{{ )
            J }{ I }} DUP>R G@ G.
        LOOP
    LOOP DROP ;

: .V ( V{ --) DUP D.LEN
```

[9] The exception to this general rule, where more complete optimization would pay, would be an application that requires solving many sets of equations of relatively small order.

```
    0 DO CR DUP I }{ 0 } G@ G. LOOP DROP ;
\ end{verbatim}

We define a word to put ASCII numbers into matrices:
\begin{verbatim}
: GET-F# BL TEXT PAD $->F ;
```

This word takes the next string[10] from the input stream (set off by ASCII blank, **BL**) and uses the HS/FORTH word $\$ \to F$ to convert the string to floating point format and put it on the 87stack.

GET-F# is used in the following:

```
: <DO-VEC>   0 DO GET-F#  DUP  I 0}   G!
             LOOP DROP ;
: TAB->VEC  DUP D.LEN <DO-VEC> ;
: TAB->MAT  DUP D.LEN DUP * <DO-VEC> ;
```

The file EX.3 will be found in the accompanying program diskette. The explanatory notes below refer to EX.3.

Notes:

- The word .(emits everything up to a terminating) .
- HS/FORTH, because of its segmented dictionary, uses a word TASK to define a task-name, and FORGET-TASK to FORGET everything following the task-name. The FORTH word FORGET fails in HS/FORTH when it has to reach too deeply into the dictionary.
- Ex.3 uses the word }row{ defined below.

This is what it looks like when we load EX.3:

```
FLOAD EX.3 Loading EX.3
Read a dimenslon 3 vector and 3x3 matrix from a table, then dispiay them.
Now displaying vector:
V{ .V CR
0.0000000
4.0000000
2.0000000
Now displayhg matrix
A{{ .M CR
1.0000000 0.0000000 5.0000000
3.0000000 2.0000000 4.0000000
1.0000000 1.0000000 6.0000000
say EX.3 FORGET-TASK to graceitfly FORGET these words ok
```

[10] The word **TEXT** inputs a counted string, using the FORTH-79 standard word **WORD** and places it at **PAD**. This is why PAD appears in the definition of **GET-F#**. A definition of **TEXT** might be : `TEXT ( delimiter --) WORD DUP C@ 1 + PAD SWAP CMOVE ;` Note that all words prefaced with C are byte operations by FORTH convention.

```
A{{ B{ }}SOLVE ( solution to Ax=B left in B)
BEGIN
    set n=0
    find pivot (row m > n)
    swap rows m & n

    divide row n by pivot $A'_{nm} = A_{nm}/A_{nn}$

    subtract: $A'_{kl} = A_{kl} - A_{kn}A'_{nl}$
    increment n
    n N =
UNTIL
    Back-substitute: $x_n = b'_n - \sum_{m>n} A_{nm}x_m (12)$
DONE
```

Figure 1.1: Pseudocode for pivotal elimination

Implementing pivotal elimination

Now we have to define the FORTH words that will implement this algorithm. In pseudocode, we can express pivotal elimination as shown in Fig. 1.1 below.

The program

Whenever we write a complex program we are faced with the problem "Where do we begin?" FORTH emphasizes the construction of components, and generally cares little which component we define first.

For example, we must swap two rows of a matrix. The most obvious way moves data:

```
row1 -> temp
row2 -> row1
temp -> row2
```

Although data moves are relatively fast, up to $N^2/2$ swaps may be needed, each row containing N data; row-swapping is an $O(N^3)$ operation which must be avoided. Indirection, setting up a list of row pointers and exchanging those, uses $O(N^2)$ time.

We anticipate computing the address of an element of **A{{** in a **DO** loop via the phrase

```
A{{ J I }} ( J is row, I is col)
```

Suppose we have an array of row pointers and a word `}row{` that looks them up. Then the (swapped) element would be

```
A{{ J }row{ I }}.
```

To implement this syntax we define[11]

```
: swapper ( n --)
    CREATE 0 DO I , LOOP
    DOES> OVER + + @ ;
    \ this is a defining word
```

We would define the array of row indices *via*

```
A{{ D.LEN swapper }row{ \ create array }row{
```

We have initialized the vector `}row{` by filling it with integers from 0 to N-1 while defining it.

Suppose we wanted to re-initialize the currently vectored row-index array: this is accomplished *via*

```
A{{ D.LEN '}row{ refill
```

where

```
: refill (n adr --)
    SWAP 0 DO I 2* DDUP + !
        2 +LOOP DROP;
```

To swap the two rows

```
:ADRS >R 2* OVER + SWAP R> 2* + ;
( a m n -- a + 2m a + 2n)
0 VAR ?SWAP           \to keep track of swaps

: }SWAP ( a m n --) DDUP =
    IF DDROP DROP \ noswap -- clean up
    ELSE ADRS DDUP ( --1 2 1 2)
        @ SWAP @ ( -- 1 2 [2] [1])
        ROT ! SWAP !
        -1 ?SWAP XOR IS ?SWAP \ ˜ ?SWAP
    THEN ;
```

Test this with a 3-dimensional row-index array:

[11] The HS/FORTH words **VAR** and **IS** define a data structure that can be changed, like a FORTH **VARIABLE: 0 VAR X 3 IS X** but has the run-time behavior of a **CONSTANT: X . 3 ok**.

```
3 swapper }row{

: TEST 0 DO I }row{ CR . LOOP ;
3 TEST
0
1
2 ok
```

Now swap rows 1 and 2:

```
'}row{ 1 2 }SWAP 3 TEST
0
2
1 ok
```

and back again:

```
'}row{ 1 2 }SWAP 3 TEST
0
1
2 ok
```

Next , we need a word to find the pivot element in a given column, searching from a given n. The steps in this procedure are shown in Fig. 1.2 on page 15 below.

We anticipate using generic operations Gx defined in Chapter 5 to perform fetch, store and other useful manipulations on the matrix elements, without specifying their type until run-time. It is thus useful to have a place to store the type (other than the second cell of the array data structure). We arrange this *via*

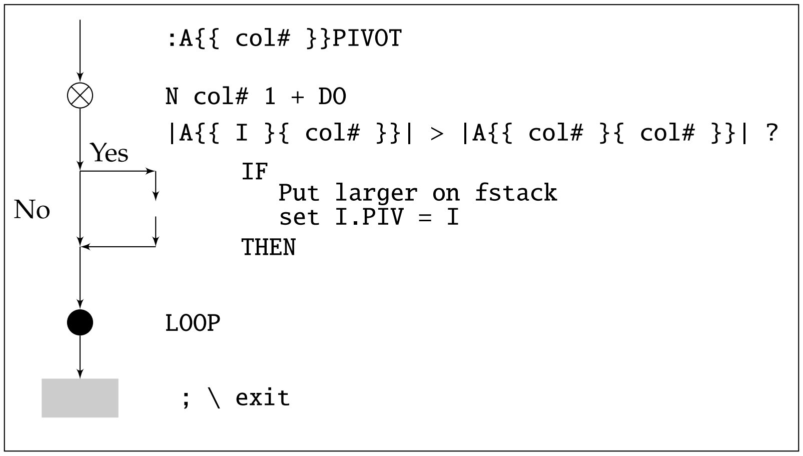
```
0 VAR T             \ a place to store data type
```

The rest of the definition is rendered self-explanatory by the vertical commenting style (a must for long[12] words):

```
0 VAR Length        \ Length of matrix
0 VAR COL           \ current col#
0 VAR I.PIV         \ I.PIV is used to return the result
0 VAR a{{           \ a{{ stores the adress of M{{

: INITIALIZE        ( M{{ -- ::--)
    IS a{{
    a{{ type@ IS T          \ initialize T
    a{{ LEN@ IS Length ;    \ length - > Length

: }}PIVOT           ( M{{ col-- ::--)
    IS COL ZECOL IS I.PIV
```

[12] While Brodie (*TF*, p.180) quotes Charles Moore. FORTH's inventor, as offering 1-line definitions as the goal in FORTH programming, sometimes this is just not possible.

```
    INITIALIZE
    a{{ COL }row{ COL }}    ( --seg.off[M{{ col,col}}]t)
    >FS GABS FS>F           ( 87:--|1st.elt|)
    R> COL1+                \ loop limits:L, COL+1
    DO                      \ begin loop
        a{{ I }row{ COL }}  ( -- seg.off[M{{i,col}}] t)
        >FS GABS FS>F       ( 87: -- |old.elt| |new.elt|)
        XDUP F<             \ test if new.elt > old.elt
        IF   FSSWAP I IS I.PIV   THEN   FDROP
    LOOP                    \ end loop
    FDROP    ;              \ clean up ifstack

\ Usage:    A{{ 2 }}PIVOT
```



```
:A{{ col# }}PIVOT

N col# 1 + DO
|A{{ I }{ col# }}| > |A{{ col# }{ col# }}| ?
     IF
        Put larger on fstack
        set I.PIV = I
     THEN

LOOP

 ; \ exit
```

Figure 1.2: Pseudocode for finding the pivot element

Notes:

- We avoid filling up the parameter stack with addresses that have to be juggled, by putting arguments in named storage locations (**VAR**s). We anticificiate setting all the parameters in the beginning using **INITIALIZE** which can be extended later if necessary.
- We have used a word from the COMPLEX arithmetic lexicon, namely **XDUP(FOVER FOVER)** to double-copy the contents of the fstack, since the test *F* < drops both arguments and leaves a flag.
- There is little to be gained by optimizing (and if we did we should have had to avoid the generic **Gx** words because they cannot be optimized by the HS/FORTH recursive-descent optimizer) because only $N^2/2$ time is used, negligible (for large N) compared with the $N^3/3$ in the innermost loop.

The **Gx** operations are found in the file **GLIB.FTH** .

To test the word }}PIVOT we can add some lines to the test file:

```
CR CR .( flnd the pivot row for columns 0,1,2)
CR
CR .( A{{ 0 }}PIVOT IPIV .) BL EMTT A{{ 0 }}PIVOT IPIV .
CR .( A{{ 1 }}PIVOT IPIV .) BL EMTT A{{ 1 }}PIVOT IPIV .
CR .( A{{ 2 }}PIVOT IPIV .) BL EMTT A{{ 2 }}PIVOT IPIV .
CR CR
```

The result is the additional screen output

```
A{{ 0 }}PIVOT IPIV 1
A{{ 1 }}PIVOT IPIV 1
A{{ 2 }}PIVOT IPIV 2
ok
```

From our pseudocode expression (see Fig. 1.1, p. 12) of the pivotal elimination algorithm we see that the next operation is to multiply a row by a constant. To do this we define

```
0 VAR ISTEP
\ include phrase
\ T #BYTES DROP IS ISTEP
\ in INITIALIZE

: DO(ROW*X)LOOP (seg off I.fin I.beg -- :: x -- x)
    DO DDUP I + DDUP T >FS G*NP T FS>
    ISTEP /LOOP DDROP ;
\ G*NP means "(generic) multiply, no pop"

: }}ROW*X       ( M{{ row -- :: x -- x)
    UNDER }row{ 0 }}            ( -- r seg off t)
    DROP ROT                    ( --seg off r)
    ISTEP * Length ISTEP * SWAP \ loop indices
    DO(ROW*X)LOOP ;             \ loop

\Ex: A{{ 2 }ROW*X
```

Notes:

- While **DO(ROW*X)LOOP** and **}}ROW*X** clear the parameter stack in approved FORTH fashion, they leave the constant multiplier **x** on the fstack. That is, we anticipate next multiplying the corresponding row of the inhomogeneous term **b** (in the matrix equation **A-x = b**) by the same constant **x**.
- We assume **INITIALIZE** (including **INIT.ISTEP**) has been invoked before **}}PIVOT** or **}}ROW*X**.

- Anticipating the (possible) need to optimize, we factor the loop right out of the word. We also com me the addresses fast by putting base addresses on the stack and then incrementing them by the cell-size, in bytes.

Subtracting row I (times a constant) from row J is quite similar to multiplying a row by a constant. The process can be broken down into the pseudocoded actions in Fig.1.3 below.

```
⊗  DO (K = J TO L)
│       X G@
↓
│       M[I,K] G@ G*NP (::--X M[I,K]*X)
↓
│       M[J,K] G@ GR- (::--X M[J,K]-X*M[I,K])
↓
│       M[J,K] G!
●  LOOP
```

Figure 1.3: Steps in }}R1-P2*X

A first attempt might look as follows:

```
\ auxiliary words
: 4DUP DOVER DOVER ;
( a b c d -- a b c d a b c d )
: ++   ( s1.o1 s2.o2 n -- s1.o1+n s2.o2+n)
    DUP>R + ROT R> + -ROT ;

: }}R1-R2*X (r1r2 -- ::x--x)
    >R >R
    a{{ R> }row{ 0 }} DROP \ M{{ r1 0 }}
    a{{ R@ }row{ 0 }} DROP \ M{{ r2 0 }}
    Length ISTEP *  \ L*ISTEP (upper limit)
    R> ISTEP *      \ r2*ISTEP (lower limit)
    DO 4DUP I ++    \ duplicate & inc base adrs
        T >FS G*NP  ( :: -- xx*M[r2,k])
        DDUP T >FS GR-
        T >FS       \ ! result
    ISTEP /LOOP DDROP DDROP ;
\ INITIALIZE is assumed
```

As with **}}ROW*X** we avoid the execution-speed penalty of calculating addresses within the loop by *not* using the phrase

```
a{{ I }row{ J }} .
```

However, we are doing a lot of unnecessary work inside the loop by making 5 choices based on datatype. There are also a lot of unnecessary moves to/from the ifstack. This is an example where speed has been sacrificed to compactness of code: one program solves equations of any (”scientific”) data format - REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16.

Conversely, we can both eliminate the extra work and accelerate execution simply by defining a *separate* inner loop for each data; type, letting the choice take place once, outside the inner loop. This multiplies the needed code fourfold (or more-fold, if we optimize).

The 4 inner loops are

```
: Re.LP ( s1.o1 s2.o2 L*istep r2*istep -- ::x--x)
    FS>F >R
    DO 4DUP I ++ R32@L F*NP
        DDUP R32@L FR- R32!L
    4 /LOOP R> F>FS ;

: DRe.LP ( s1.o1 s2.o2 L*istep r2*istep -- ::x--x)
    FS>F >R
    DO 4DUP l ++ R64@L F*NP
        DDUP R64@L FR- R64!L
    8 /LOOP R> F>FS ;

: X.LP ( s1.o1 s2.o2 L*istep r2*istep -- ::x--x)
    FS>F >R
    DO 4DUP I ++ CP@L X*NP
        DDUP CP@L CPR- CP!L
    8 /LOOP R> F>FS ;

: DX.LP ( s1.o1 s2.o2 L*istep r2*istep -- ::x--x)
    FS>F >R
    DO 4DUP I ++ DCP@L X*NP
        DDUP DCP@L CPR- DCP!L
    16 /LOOP R> F>FS ;
```

Not surprisingly, the loops contain some duplicate code, but this is a small price to pay for the speed increase. A significant further increase can be obtained easily, using the HS/FORTH recursive- descent optimizer to define these inner loops, or by redefining the loops in assembler (for ultimate speed).

Now we can redefine **}}R1-R2*X** using vectored execution, *via* HS/FORTH’s **CASE: ... ;CASE** construct, or the equivalent high-level version given here:

```
: CASE: CREATE ]
    DOES> OVER + + @ EXECUTE ;
: ;CASE [COMPILE] ; ; IMMEDIATE

CASE: DO(R1-R2*X)LOOP
    Re.LP DRe.LP X.LP DX.LP ;CASE

: }}R1-R2*X (r1 r2-- ::x--x)
    >R >R
    a{{ R> }row{ 0 }} DROP      \ M{{ r1 0 }}
    a{{ R@ }row{ 0 }} DROP      \ M{{ r2 0 }}
    Length I.STEP *      \ L*I.STEP (upper limit)
    R> I.STEP *          \ r2*I.STEP (lower limit)
    T DO(R1-R2*X)LOOP DDROP DDROP ;
\ We assume INITIALIZE has been invoked
```

Another word is needed to perform the same manipulation on the inhomogeneous term. Since this latter process runs in $O(N^2)$ time we need not concern ourselves unduly with efficiency.

```
: }V1-V2*X ( r1 r2-- ::x--)
    SWAP >R >R
    b{ 0.0 } DROP DDUP \V[0] V[O]
    R> }row{ ISTEP * + >FS G*
    R> }row{ ISTEP * + DDUP >FS GR- FS> ;
```

Note:

- After **}V1-V2*X** executes, the multiplier x is no longer needed, so we drop it here by using **G*** rather than **G*NP**.

We now combine the words **}SWAP**, **}}PIVOT**, **}}ROW*X**. **}R1-R2*X** and **}V1-V2*X** to implement the triangularization portion of pivotal elimination.

Since the Gaussian elimination method makes it very easy to compute the determinant of the matrix as we go, we might as well do so, especially as it is only an O(N) process. By evaluating the determinants of simple cases, we realize the determinant is simply the product of all the pivot elements, multiplied by $(-1)^{\#\mathbf{swaps}}$. Hence we need to keep track of swaps, as well as to multiply the determinant (at a given stage) by the next pivot element. The need to do these things has been anticipated in defining **}SWAP** above.

Computing the determinant also lets us test as we go, that the equations can be solved: if at any stage the determinant is zero, (at least) two of the equations must have been equivalent. Should it be necessary to test the condition[13] of the

[13] The **condition** of a system of linear equations refers to how accurately they can be solved. Equations can be hard to solve precisely if the inverse matrix A^{-1} has some large elements. Thus, the error vector for a calculated solution, $\delta = b - A \cdot X_{Calc}$ can be small, but the difference between

equations, this too can be found as we proceed, by computing the determinant.

Here is a simple recipe for computing the determinant, with checking to be sure it does not vanish identically. We use the intelligent fstack (ifstack) defined in Ch. 7.

```
DCOMPLEX SCALAR DET \ room for any type

: INIT\_DET T ’DET  ! \ set Type
    T G=1 DET G! ;    \ set det = 1

% 1.E-10 FCONSTANT CONDITION

: DETERMINANT ( -- :: x -- x)
    DET >FS G*NP
    ?SWAP IF GNEGATE THEN
    FS.DUP GABS FS>F CONDITION F<
    ABORT" determinant too small!" DET FS> ;
```

Now we define the word that triangularizes the matrix:

```
0 VAR b{        \ to keep the stack short
: INITIALIZE    ( M{{V{-- ::--)
    IS b{
    IS a{{
    a{{ D.TYPE IS T
    a{{ D.LEN IS Length
    a{{ D.TYPE #BYTES DROP IS ISTEP
    INIT.DET ;  \ set det=1

: }/PIVOT }row{ } DROP DDUP T >FS G* T FS> ;
    ( seg ofd t--::x--)

:TRIANGULARIZE ( M{{ V{ --)
    INITIAUZE
    Length 0 DO             \ loop 1 -- by rows
    a{{ I }}PIVOT           \ find pivot in col I
    ’}row{ I I.PIV }SWAP    \ exchange rows
    a{{ I }row{ I }} >FS    \ pivot->iistack
    DETERMINANT             \ calc det
    1/G a{{ I }}ROW*X       \ row I /pivot
    b{ I }/PIVOT            \ inhom. term /pivot
```

the exact solution and the calculated solution can be large, since (see §9§3 below)

$$X_{Exact} - X_{Calc} = A^{-1} \cdot \delta\,.$$

A test for ill-condition is whether the determinant gets small compared with the precision of the arithmetic. See, *e.g.*, A. Ralston,*A First Course in Numerical Analysis* (McGraw-Hill Book Co., New York, 1965).

```
    Length I 1 + DO         \ loop 2 - by rows
        a{{ I }row{ J }} >FS    \ x -> ifstack
        a{{ I J }}R1-R2*X
                \ row[i] = row[i]-row[j]*x
        b{ I J }V1-V2*X
                \ same for b{ and drop x
        LOOP            \ and loop 2
    LOOP ;              \ end loop 1
\Usege: A{{ B{ TRIANGULARIZE
```

Now at last we can back-solve the triangularized equations to find the unknown x's. The word for this is }BACK-SOLVE, defined as follows:

```
: }BACK-SOLVE ( -- )
    0 LENGTH 2- DO              \ outer loop
        T G=0                   \
        LENGTH I 1 + DO         \ inner loop
        a{{ J }row{ I }} > PS   \
        bi{ I }row{ } >FS G* G+ \
        LOOP                    \ inner loop
    b{ I }{ } DROP DDUP T >FS   \
    GR- T FS >                  \
    -1 +LOOP ;                  \outer loop J
```

Putting the entire program together we have the linear equation solver given in the file SOLVE1. Examples of solving dense 3 × 3 and 4 × 4 systems are included for testing purposes.

Timing

We should like to know how much time it will take to solve a given system. (Of course it is also useful to know whether the solution is correct!) We time the solution of 4 sets of N equations, with 4 different values of N. The running time can be expressed as a cubic polynomial in N with undetermined coefficients:

$$T_N = a_0 + a_1 N^1 + a_2 N^2 + a_3 N^3 \tag{1.19}$$

Evaluating 1.19 for 4 different values of N, and measuring the 4 times T_{N_i} , we produce 4 inhomogeneous linear equations in 4 unknowns: $a_i, i = 0, 1, 2, 3$. As luck would have it, we just happen to have on hand a linear equation solver, and are thus in a position to determine these coefficients numerically.

For this timing and testing chore we need an exactly soluble dense system of linear equations, of arbitrary order. The simplest such system involves a rank-1 matrix, that is, a matrix of the form [14]:

$$A = I - uv^{\dagger} \tag{1.20a}$$

In terms of matrix elements,

$$A_{ij} = \delta_{ij} - u_i v_j^* \tag{1.20b}$$

The solution of the system $A \cdot x = b$ is simple: using the standard notation

$$v^{\dagger} \cdot x \equiv (v, x) = \sum_i v_i^* x_i \tag{1.21}$$

we have

$$x_i = b_i + (v, x)\, u_i \tag{1.22}$$

The coefficient (v, x) is just a (generally complex) number; we compute it from 1.22*via*

$$(v, x) = (v, b) + (v, u)(v, x) \tag{1.23}$$

or

$$(v, x) = \frac{(v, b)}{1 - (v, u)} \quad . \tag{1.24}$$

Substituting 1.24back in1.22, we have

$$x_i = b_i + \frac{(v, b)}{1 - (v, u)} u_i \tag{1.25}$$

Everything in 1.25 is determined, hence the solution is known in closed form.

An example that embodies the above idea is given in the file EX.20, where we make the special choices

$$\mathbf{u} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ \dots \\ \dots \\ \dots \end{pmatrix}, \ \mathbf{v} = \frac{1}{2N}\mathbf{u}, \ \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ \dots \\ \dots \end{pmatrix}, \ \mathbf{x} = \begin{pmatrix} 1.5 \\ 0.5 \\ 1.5 \\ 0.5 \\ \dots \\ \dots \end{pmatrix}, N \text{ even}$$

We give the times only for the case of a highly optimized inner loop (written in assembler), for the type REAL*4:

The coefficients of the cubic polynomial extracted from the above are (in seconds, 9.54 MHz 8086 machine, machine-coded inner loop)

[14] We employ the standard notation that a vector u is a column and an adjoint vector $v^{\dagger}$ a row. Their outer product, $uv^{\dagger}$, is a matrix. Given a column v, we construct its adjoint (row) $v^{\dagger}$ by taking the complex conjugate of each element and placing it in the corresponding position in a row-vector.

Table 1.1: Execution times for various N (9.54 MHz 8086 + 8087 machine)

N	Time
20	1.20
50	7.75
75	19.6
100	38.8

$$\begin{aligned} a_0 &= 0.32727304 \\ a_1 &= -0.01084850 \\ a_2 &= 0.00241636 \\ a_3 &= 0.00001539 \end{aligned}$$

from them we can extrapolate the time to solve a 350 × 350 real system to be about 16 minutes. On a 25 MHz 80386/80387 machine with 32-bit addressing implemented, the time should decrease five- to tenfold [15].

It is interesting to explore a bit further the extracted value of a_3, which is $\frac{1}{3}$ the time needed to evaluate the inner loop (defined above as Re.LP and hand-coded in assembler for ultimate speed). Converting to clock cycles on an IBM-PC compatible (running at 9.54 MHz) we have an average of 440 clock cycles per traversal (of this loop). Recall the operations that are needed: a 32-bit memory fetch, a floating-point multiply, another 32-bit memory fetch, a floating-point subtraction, and a 32-bit store. The initial fetch-and-multiply can be be compressed into a single co-processor operation, as can the fetch-and-subtract. The times in cycles for these basic operations are[16]

We see from Table 1.2 above that the time computed from the cpu and coprocessor specifications (422 clocks) is close to the measured time (440 clocks). The slight difference doubtless comes both from measurement error and from the fact the timing of CISC chips is not an exact art (for example, there a ?? periodic interruptions for dynamic memory refreshment and f ?? the system clock).

[15] When I think that solving 100 × 100 systems – key to my Ph.D. research in 1966 – took the better path of an hour on an IBM 7094, these results seem incredible.

[16] See 8087P.

Table 1.2: Execution times of innermost loop operations

Operation	Time (cpu clocks)
memory FMUL	133
memory FSUBR	128
memory FSTP	100
overhead	61
Total	422

Finally, we confirm that little is to be gained by optimizing out loops. Suppose, *e.g.*, we could halve a_2 by clever programming then we should cut the N = 350 time from 16 to 13 minutes, and the time for N = 1000 by some 20 minutes in 5 hours, or 6%.

Matrix Inversion

We introduce this subject with a brief discourse on liners algebra of square matrices.

Linear transformations

Suppose A is a square $(N \times N)$ matrix and x is an N-dimensional vector (a column, or $(N \times 1)$ matrix). We can think of the symbolic operation

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x} \tag{1.26}$$

as a linear transformation of the column x to a new column y. In terms of matrix elements and components,

$$\mathrm{y_m} = \sum_{\mathrm{n}=0}^{\mathrm{N}-1} \mathrm{A_{mn} x_n} \tag{1.27}$$

The transformation is **linear** because if x is the sum of 2 columns (added component by component)

$$\mathrm{x} = \mathrm{x}^{(1)} + \mathrm{x}^{(2)} \,, \tag{1.28}$$

we can calculate $A \cdot x$ either by first adding the two vectors ant: then transforming, written

$$\mathrm{A} \cdot \mathrm{x} = \mathrm{A} \cdot \left(\mathrm{x}^{(1)} + \mathrm{x}^{(2)}\right) \tag{1.29}$$

or we could transform first and then add the transformed vectors. The identity of the results is called the **distributive law**

$$\mathrm{A} \cdot \mathrm{x} = \mathrm{A} \cdot \left(\mathrm{x}^{(1)} + \mathrm{x}^{(2)}\right) \equiv \mathrm{A} \cdot \mathrm{x}^{(1)} + \mathrm{A} \cdot \mathrm{x}^{(2)} \tag{1.30}$$

of linear transformations.

Matrix multiplication

Now, suppose we had several square matrices, **A, B, C,...** We could imagine performing successive linear transformations on a vector **x** *via*

$$\mathrm{y} = \mathrm{A} \cdot \mathrm{x} \tag{1.31a}$$

$$\mathrm{z} = \mathrm{B} \cdot \mathrm{y} \tag{1.31b}$$

$$\mathrm{w} = \mathrm{C} \cdot \mathrm{z} \tag{1.31c}$$

These can conveniently be written

$$\mathrm{w} = \mathrm{C} \cdot \left(\mathrm{B} \cdot (\mathrm{A} \cdot \mathrm{x})\right) . \tag{1.32}$$

The concept of successive transformations leads to the idea of multiplying two matrices to obtain a third:

$$\mathrm{D} = \mathrm{B} \cdot \mathrm{A} \quad \mathrm{E} = \mathrm{C} \cdot \mathrm{D} \tag{1.33}$$

In terms of matrix elements we have, for example

$$\mathrm{D}_{\mathrm{ik}} = \sum_{\mathrm{j}=0}^{\mathrm{N}-1} \mathrm{B}_{\mathrm{ij}} \mathrm{A}_{\mathrm{jk}} \tag{1.34}$$

The important point is that the (matrix) multiplications may be performed in any order, so long as the left-to-right ordering of the factors is maintained:

$$\mathrm{C} \cdot \left(\mathrm{B} \cdot \mathrm{A}\right) \equiv \left(\mathrm{C} \cdot \mathrm{B}\right) \cdot \mathrm{A} \tag{1.35}$$

Equation 1.35 is known as the associative law of matrix multiplication. Finally, we note that – as hinted above – the left-to-right order of factors in matrix multiplication is significant. That is, i general,

$$\mathrm{A} \cdot \mathrm{B} \neq \mathrm{B} \cdot \mathrm{A} \tag{1.36}$$

We say that, unlike with ordinary or even complex arithmetics matrix multiplication – even of square matrices – does not in general obey the commutative law [17].

Matrix Inversion

With this introduction, what does it mean to invert a matrix? First of all, the concept can apply only to a square matrix. Given an $N \times N$ matrix A, we seek another $N \times N$ matrix A^{-1} with the property that

$$\mathrm{A}^{-1} \cdot \mathrm{A} \equiv \mathrm{A} \cdot \mathrm{A}^{-1} \tag{1.37}$$

Err(2) where I is the unit matrix defined in the beginning of this chapter (Eq. [1.3]$_r$[2]–1's on the main diagonal, 0's everywhere else). We have implied in Eq. 1.37 that a matrix that is an inverse with respect to left-multiplication is also an inverse with respect to right-multiplication. Put another way, we imply that

$$\left(\mathrm{A}^{-1}\right)^{-1} \equiv \mathrm{A} \tag{1.38}$$

The condition that – given A – we can construct A^{-1} is the same as the condition that we should be able to solve the linear equation

$$\mathrm{A} \cdot \mathrm{x} = \mathrm{b}\,;$$

namely, $\det(\mathrm{A}) \neq 0$.

Why invert matrices, anyway?

We have developed a linear equation solver program already – why should we be interested in a matrix inversion program?

Here is why: The time needed to solve a single system (that is, with a given inhomogeneous term) of linear equations is conditioned by the number of floating-point multiplications required: about $N^3/3$. The number needed to invert the matrix is about N^3, roughly 3× as many, meaning roughly 3× as long to invert as to solve, if the matrix is large. Clearly there is no advantage to inverting unless we want to solve a number of equations with the same

[17] MISSING FOOTNOTE.Ref. in page 244, something about commutative law...

[2] Erratum! typo (original text was: "4")

coefficient matrix but with different inhomogeneous terms. In this case, we can write

$$\mathrm{x} = \mathrm{A}^{-1} \cdot \mathrm{b} \tag{1.39}$$

and just recalculate for each **b**. Clearly this breaks even - relative to solving 3 sets of equations - for 3 different **b**'s and is superior to re-solving for more than 3 **b**'s.

An Example

Let us now calculate the inverse of our 3×3 matrix from before. The equation is (let $C = A^{-1}$ be the inverse[18])

$$\begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.40}$$

It is easy to see that Eq. 1.40 is like 3 linear equations, the unknown being each column of the matrix C. The brute-force method to calculate A is then to work on the right-hand-side all at once as we triangularize, and back-solve. It is easy to see that triangularization by pivotal elimination leads to

$$\begin{pmatrix} 1 & {}^{2}/_{3} & {}^{4}/_{3} \\ 0 & 1 & {}^{-11}/_{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} 0 & {}^{1}/_{3} & 0 \\ {}^{-3}/_{2} & {}^{1}/_{2} & 0 \\ {}^{1}/_{13} & {}^{-1}/_{13} & {}^{2}/_{13} \end{pmatrix} \tag{1.41}$$

To construct the inverse we replace the right-hand side with the results of back-solving, column by column. This is N times as much work as back-solving a single set of equations, hence the total time required for brute-force inversion is $\approx \frac{4}{3}N^3$. By keeping track of zero elements we could reduce this time by another $\frac{1}{3}N^3$, thereby obtaining the theoretical minimum (for Gaussian elimination) of $O(N^3)$. However, the brute-force method is unsatisfactory for another reason: it takes twice the storage of more sophisticated algorithms. Modern matrix packages therefore use LU decomposition for both linear equations and matrix inversion.

LU decomposition

We now investigate the LU decomposition algorithm[19]. Suppose a given matrix A could be rewritten

$$\mathrm{A} = \begin{pmatrix} a_{00} & a_{01} & \dots \\ a_{10} & a_{11} & \dots \\ \dots & \dots & \dots \end{pmatrix} = \mathrm{L} \cdot \mathrm{U} = \begin{pmatrix} \lambda_{00} & 0 & 0 \\ \lambda_{10} & \lambda_{11} & 0 \\ \dots & \dots & \dots \end{pmatrix} \cdot \begin{pmatrix} \mu_{00} & \mu_{01} & \dots \\ 0 & \mu_{11} & \dots \\ 0 & 0 & \dots \end{pmatrix} \tag{1.42}$$

[18] Note: if C is a right-inverse, AC = I, it is also a left-inverse, CA= I.

[19] See, *e.g.*,William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery. Numerical Recipes (Cambridge University Press, Cambridge, 1986), p. 31ff.

then the solution of

$$(\mathrm{L}\cdot\mathrm{U})\cdot\mathrm{x} \equiv \mathrm{L}\cdot(\mathrm{U}\cdot\mathrm{x}) = \mathrm{b} \tag{1.43}$$

can be found in two steps: First, solve

$$\mathrm{L}\cdot\mathrm{y} = \mathrm{b} \tag{1.44}$$

for

$$\mathrm{y} \equiv \mathrm{U}\cdot\mathrm{x} \tag{1.45}$$

via

$$\begin{aligned} \lambda_{00}y_0 &= b_0 \\ \lambda_{10}y_0 + \lambda_{11}y_1 &= b_1 \\ \lambda_{20}y_0 + \lambda_{21}y_1 + \lambda_{22}y_2 &= b_2 \\ \ldots\ &etc.\ \ldots \end{aligned} \tag{1.46}$$

Err(3) which can be solved successively by forward [substitution]$^{3}_{r}$. Next solve 45 successively (by back-substitution) for **x**:

$$\begin{aligned} \mu_{N-1\,N-1}x_{N-1} &= y_{N-1} \\ \mu_{N-2\,N-2}x_{N-2} + \mu_{N-2\,N-1}x_{N-1} &= y_{N-2} \\ \ldots\ &etc.\ \ldots \end{aligned} \tag{1.47}$$

The n'th term of Eq. 1.46 requires n multiplications and n additions. Since we must sum n from 0 to N-1, we find $N(N-1)/2 \approx N^2/2$ multiplications and additions to solve all of 1.46. Similarly, solving 1.47 requires about $N^2/2$ additions and multiplications. Thus, the dominant time in solving must be the time to decompose according to Eq. 1.42.

The decomposition time is $\approx N^3/3$, and the method for decomposing is described clearly in *Numerical Recipes*. The equations to be solved are

$$\sum_{k=0}^{N-1} \lambda_{mk}\mu_{kn} = A_{mn} \tag{1.48}$$

constituting $N^2 + N$ equations for N^2 unknowns. Thus we may arbitrarily **choose** $\lambda_{kk} = 1$.

The equations 1.48 are easy to solve if we do so in a sensible order. Clearly,

$$\begin{aligned} \lambda_{mk} &\equiv 0\ ,\ m > k \\ \\ \mu_{kn} &\equiv 0\ ,\ k < n \end{aligned} \tag{1.49}$$

[3] Erratum! typo (original text was: "substtution")

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

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

$$\lambda_{mn} = \frac{1}{\mu_{nn}}\Big(A_{mn} - \sum_{k=0}^{n-1} \lambda_{mk}\mu_{kn}\Big),\; m = n+1, n+2, \ldots, N-1 \tag{1.50}$$

Inspection of Eq. 1.50 makes clear that the terms on the right side are always 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 diagonals we store μ_{nn} since $\lambda_{kk} = 1$ is known).

To limit roundoff error we again pivot, which amounts to permutl ing so the row with the largest diagonal element is the currenf one. Much of the code developed for the Gauss eliminatios method is applicable, as the file LU.FTH shows.