Linear Algebra

Contents

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.

§1 Simultaneous linear equations

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

§§1 Theory of linear algebraic equations

We begin by stating the problem. Given the equations

$$\sum_{n=0}^{N-1} A_{mn}\, x_n = b_m. \qquad (1)$$

—more compactly written $\vec{\mathbf{A}} \cdot \vec{\mathbf{x}} = \vec{\mathbf{b}}$— 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?

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 $\vec{\mathbf{A}}$ should not be 0. Contrariwise, if $\det(\vec{\mathbf{A}}) = 0$, a solution with $\vec{\mathbf{x}} \neq 0$ *can* be found when $\vec{\mathbf{b}} = 0$.

---

1. The determinant of an N'th-order square matrix **A** — denoted by det(**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 matrices of square submatrices of **A**. See §9§§2.1.

§§2 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

$$\vec{\mathbf{A}} \cdot \vec{x} = \lambda \vec{\beta} \cdot \vec{x} \qquad (2)$$

have solutions? Here λ is generally a complex number, called the **eigenvalue** (or **characteristic** value) of Eq. 2, and $\vec{\beta}$ is often called the **mass matrix**. Frequently $\vec{\beta}$ is the unit matrix **I**,

$$\mathbf{I}_{mn} = \begin{cases} 1, & m = n \\ 0, & m \neq n \end{cases}, \qquad (3)$$

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

$$\vec{\mathbf{A}} \cdot \vec{x} = 0 \qquad (4)$$

exists if and only if $\|\vec{\mathbf{A}}\| = 0$. This fact is useful in solving **eigenvalue** problems such as Eq. 2 above.

---

2. buildings, cars, airplanes, bridges ...

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

$$\| \mathbf{A} - \lambda \boldsymbol{\rho} \| = 0 \qquad (5)$$

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

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. 3 and its **complex conjugate**:

$$\sum_n A_{mn} x_n = \lambda \sum_n \rho_{mn} x_n \; ; \qquad (6a)$$

$$\sum_n x^*_n A^*_{nm} = \lambda^* \sum_n x^*_n \rho^*_{nm} \; ; \qquad (6b)$$

Equation 6b can be rewritten (using the fact that $\mathbf{A}$ and $\boldsymbol{\rho}$ are Hermitian)

$$\sum_m x^*_m A_{mn} = \lambda^* \sum_n x^*_m \rho_{mn} \qquad (6b')$$

Multiply 6b′ by x_m and 6a by x^*_m, sum both over m and subtract: this gives

$$0 = \left(\lambda^* - \lambda\right) \sum_{n,m} x^*_m \rho_{mn} x_n \, . \qquad (7)$$

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

---

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.

$$\mathbf{x}^{\dagger} \cdot \rho \cdot \mathbf{x} \equiv \sum_{n,m} x_m^* \rho_{mn} x_n > 0 \qquad (8)$$

for any non-zero vector[4] $\mathbf{x}$. Thus, Eq. 7 $\Rightarrow$ $\lambda^* \equiv \lambda$, 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

$$\begin{aligned} \sin(i\,\omega t) &= i \sinh(\omega t) \\ \cos(i\,\omega t) &= \cosh(\omega t) \end{aligned} \qquad (9)$$

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 $\mathbf{x}^{(0)}$ and let

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

Assuming the largest eigenvalue is unique,the sequence of vectors $\mathbf{x}^{(n)}$, $n = 1, 2, \ldots$, 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 5 for all N roots.

§2 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

4. For clarity we now omit the vector "$\rightarrow$" and dyad "$\leftrightarrow$" symbols from vectors and matrices.

more rapidly than with other languages. For me this is the chief benefit of the language.

§§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×N matrix **A**. Pretend for the moment we know how to compute the determinant of an (N-1)×(N-1) matrix. The determinant of **A** is defined to be

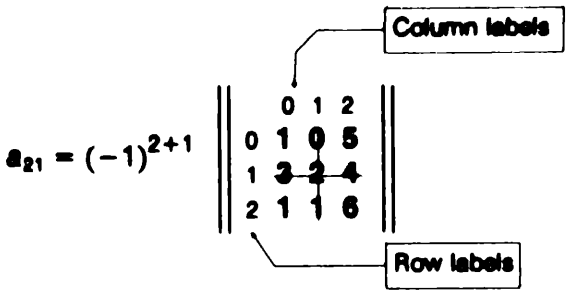
$$\det(A) = \sum_{n=0}^{N-1} A_{mn}\, a_{nm}$$

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)×(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)×(N-1) matrix). To illustrate, consider the 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix} \tag{11}$$

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



$$a_{21} = (-1)^{2+1} \quad \text{(12)}$$

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 zero[5]. What would happen to Eq. 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. 11 can be rewritten more generally

$$\sum_{n=0}^{N-1} A_{kn}\, a_{nm} = \| \mathbf{A} \|\, \delta_{km} \ . \quad \text{(13)}$$

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

$$x_n = \frac{1}{\det(A)} \sum_{m}^{N-1} a_{nm} b_m . \quad \text{(14)}$$

We see from Eq. 13 that Eq. 14 solves the equation. Equation 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

---

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

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

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 Eq. 11. We will write

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

$$a_{10} = \begin{Vmatrix} 2 & 4 \\ 1 & 6 \end{Vmatrix}$$

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

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

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

$$a_{00} = 2 \cdot 6 + (-1) \cdot 4 \cdot 1 = 8$$

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

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

$$\| \mathbf{A} \| = 1 \cdot 8 + 0 \cdot (-14) + 5 \cdot 1 = 13 \, .$$

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

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

It is easy to see[6,7] the solution to this is

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

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.

§2 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 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

---

6. Professors always say this, hee, hee! See R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Company, Reading, MA 1983).
7. N! means N×(N-1)×(N-2)×...×2×1. The base of the "natural" logarithms is e = 2.7182818...
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 close as possible.

elements. This will be clearer with a concrete illustration rather than further description: Consider the 3×3 system of equations:

$$\begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 2 \end{pmatrix} \tag{15}$$

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 **A** and in **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

$$\begin{pmatrix} 3 & 2 & 4 \\ 1 & 0 & 5 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \\ 2 \end{pmatrix} \tag{16}$$

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

$$\begin{pmatrix} 1 & 2/3 & 4/3 \\ 1 & 0 & 5 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4/3 \\ 0 \\ 2 \end{pmatrix} \tag{17}$$

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

$$\begin{pmatrix} 1 & 2/3 & 4/3 \\ 0 & -2/3 & 11/3 \\ 0 & 1/3 & 14/3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4/3 \\ -4/3 \\ 2/3 \end{pmatrix} \tag{18}$$

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

$$\begin{pmatrix} 1 & 2/3 & 4/3 \\ 0 & 1 & -1\frac{1}{2} \\ 0 & 1/3 & 1\frac{1}{3} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4/3 \\ 2 \\ 2/3 \end{pmatrix} \tag{19}$$

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

$$\begin{pmatrix} 1 & 2/3 & 4/3 \\ 0 & 1 & -1\frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4/3 \\ 2 \\ 0 \end{pmatrix} \tag{20}$$

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:

$$1\,x_0 + 2/3\,x_1 + 4/3\,x_2 = 4/3 \tag{21.0}$$

$$0\,x_0 + 1\,x_1 + (-1\frac{1}{2})\,x_2 = 2 \tag{21.1}$$

$$0\,x_0 + 0\,x_1 + 1\,x_2 = 0 \tag{21.2}$$

That is, from Eq. 21.2, $x_2 = 0$. We can back-substitute this in 21.1, then solve for x_1 to get

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

and similarly, from 21.0 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:

$$1\cdot 0 + 0\cdot 2 + 5\cdot 0 = 0$$
$$3\cdot 0 + 2\cdot 2 + 4\cdot 0 = 4$$
$$1\cdot 0 + 2\cdot 2 + 6\cdot 0 = 2$$

This works —we have indeed found the solution.

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 $\approx N/2$ 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 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 $k=0, 1, \ldots, N-1$, 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 $\tau eN!$ 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 because for large N they are overshadowed by the innermost loop[9].

---

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.

§§3 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
   0 DO  CR  DUP  I }{ 0 }  G@  G.  LOOP DROP ;
```

We define a word to put ASCII numbers into matrices:

```
: 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 **$->F** to convert the string to floating point format and put it on the 87stack.

---

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.

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 dimension 3 vector and 3x3 matrix from a table, then display them.
Now displaying vector:
V{ .V CR
 0.0000000
 4.0000000
 2.0000000
Now displaying 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 gracefully FORGET these words  ok
```

§§4 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. 9-1 below.

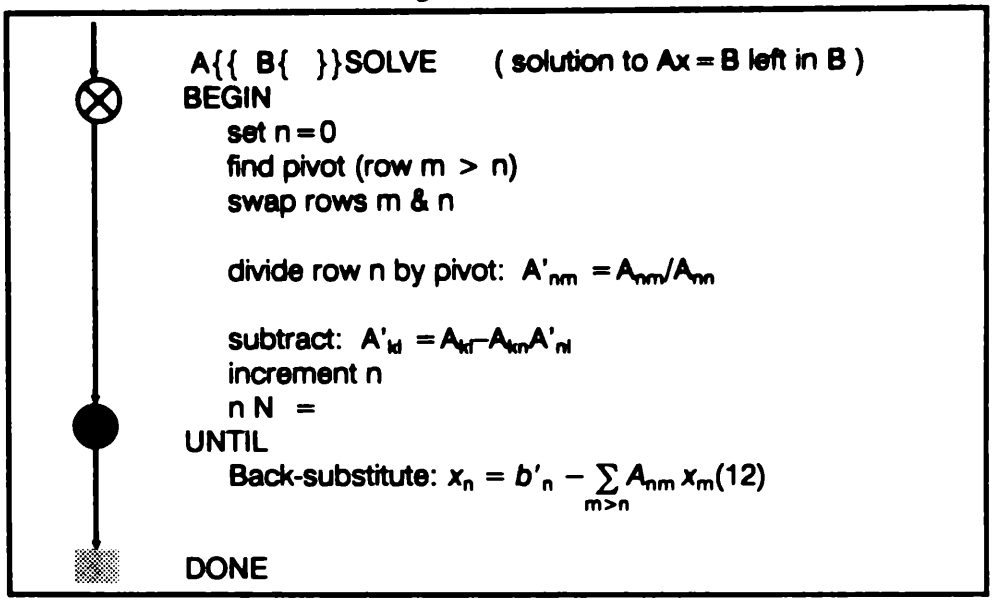


Fig. 9-1 *Pseudocode for pivotal elimination*

§§5 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 initilized 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;
```

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.

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    \ no swap – 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:

```
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. 9-2 on page 230 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

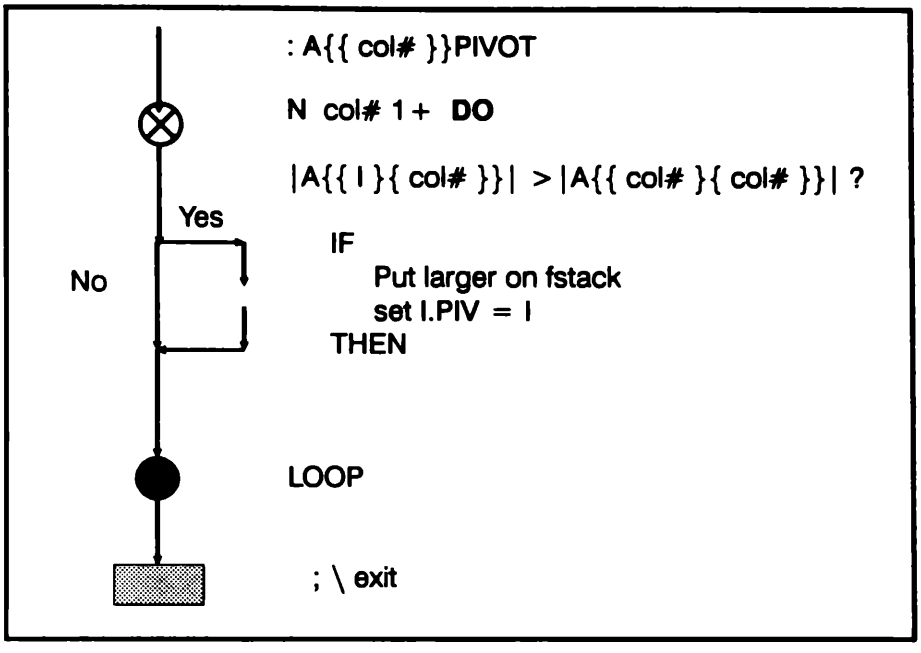


Fig. 9-2 *Pseudocode for finding the pivot element*

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
```

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.