```
: }}PIVOT              ( M{{  col - -  :: - -)
   IS COL   COL IS I.PIV
   INITIALIZE
   a{{  COL }}row{ COL }}    ( - - seg.off[M{{col,col}}] t )
   >FS  GABS  FS>F           ( 87: - - |1st.elt| )
   R>   COL 1 +              \ 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   FSWAP  I IS I.PIV   THEN   FDROP
   LOOP                      \ end loop
   FDROP   ;                 \ clean up fstack

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

Notes:

- We avoid filling up the parameter stack with addresses that have to be juggled, by putting arguments in named storage locations (VARs). We anticipate 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 .( find the pivot row for columns 0, 1, 2 )
CR
CR .( A{{ 0 }}PIVOT IPIV .) BL EMIT A{{ 0 }}PIVOT IPIV .
CR .( A{{ 1 }}PIVOT IPIV .) BL EMIT A{{ 1 }}PIVOT IPIV .
CR .( A{{ 2 }}PIVOT IPIV .) BL EMIT 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. 9-1, p. 227) 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  l.fin  l.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 $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$) 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 compute 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. 9-3 below.

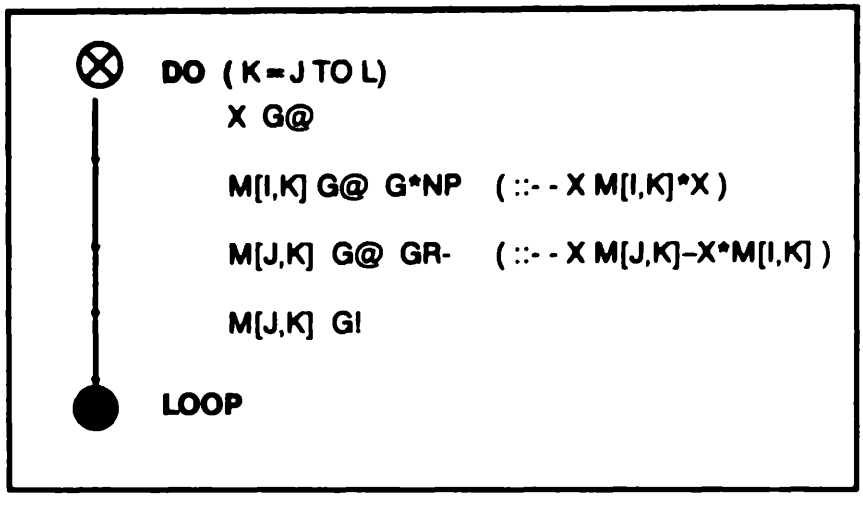

Fig. 9-3 *Steps in* }}R1-R2*X

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     ( r1 r2 - - :: 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           ( :: — x x*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 I + +   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[0]
   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 $(-1)^{\#swaps}$. 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 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> ;
```

---

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 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).

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 off t - - :: x - - )

: TRIANGULARIZE   ( M{{ V{  - - )
    INITIALIZE
    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- > ifstack
          DETERMINANT                      \ calc det
          1/G  a{{ I }}ROW*X               \ row I /pivot
          b{ I }/PIVOT                     \ inhom. term /pivot
          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                             \ end loop 2
    LOOP   ;                               \ end loop 1
\ Usage:  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 }}  >FS       \
         b{ I }row{  }  >FS  G* G+   \
         LOOP                        \ inner loop
   b{ I }{  }  DROP DDUP  T >FS      \
   GR-   T FS>                       \
   -1  +LOOP  ;                      \ outer loop
```

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.

§§6 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:

$$T_N = a_0 + a_1 N^1 + a_2 N^2 + a_3 N^3 \qquad (19)$$

Evaluating 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]:

---

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

$$\mathbf{A} = \mathbf{I} - \mathbf{u}\,\mathbf{v}^{\dagger} \tag{20}$$

In terms of matrix elements,

$$A_{ij} = \delta_{ij} - u_i\, v_j^{\,*} \tag{20'}$$

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

$$\mathbf{v}^{\dagger}\cdot x \equiv (\mathbf{v}, \mathbf{x}) = \sum_i v_i^{\,*} x_i \tag{21}$$

we have

$$x_i = b_i + (\mathbf{v}, \mathbf{x})\, u_i \tag{22}$$

The coefficient $(\mathbf{v}, \mathbf{x})$ is just a (generally complex) number; we compute it from 22 *via*

$$(\mathbf{v}, \mathbf{x}) = (\mathbf{v}, \mathbf{b}) + (\mathbf{v}, \mathbf{u})\,(\mathbf{v}, \mathbf{x}) \tag{23}$$

or

$$(\mathbf{v}, \mathbf{x}) = \frac{(\mathbf{v}, \mathbf{b})}{1 - (\mathbf{v}, \mathbf{u})} \quad . \tag{24}$$

Substituting 24 back in 22, we have

$$x_i = b_i + \frac{(\mathbf{v}, \mathbf{b})}{1 - (\mathbf{v}, \mathbf{u})}\, u_i \tag{25}$$

Everything in 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

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

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

Table 9-1 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

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

$a_0 = \quad 0.32727304$
$a_1 = -0.01084850$
$a_2 = \quad 0.00241636$
$a_3 = \quad 0.00001539$

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 innermost 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]

Table 9-2 Execution times of innermost loop operations

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

15. When I think that solving 100×100 systems – key to my Ph.D. research in 1966 – took the better part of an hour on an IBM 7094, these results seem incredible.
16. See 8087P.

We see from Table 9-2 above that the time computed from t cpu and coprocessor specifications (422 clocks) is close to t measured time (440 clocks). The slight difference doubtl comes both from measurement error and from the fact th timing of CISC chips is not an exact art (for example, there a periodic interruptions for dynamic memory refreshment and f the system clock).

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

§3 Matrix inversion

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

§§1 Linear transformations

Suppose **A** is a square (N×N) matrix and **x** is an N-dimensiona vector (a column, or N×1 matrix). We can think of the symbolic operation

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x} \tag{26}$$

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

$$y_m = \sum_{n=0}^{N-1} A_{mn} x_n \tag{27}$$

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

$$\mathbf{x} = \mathbf{x}^{(1)} + \mathbf{x}^{(2)}, \tag{28}$$

we can calculate $\mathbf{A} \cdot \mathbf{x}$ either by first adding the two vectors and *then* transforming, written

$$\mathbf{A}\cdot\mathbf{x} = \mathbf{A}\cdot\left(\mathbf{x}^{(1)} + \mathbf{x}^{(2)}\right) \tag{29}$$

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

$$\mathbf{A}\cdot\mathbf{x} = \mathbf{A}\cdot\left(\mathbf{x}^{(1)} + \mathbf{x}^{(2)}\right) \equiv \mathbf{A}\cdot\mathbf{x}^{(1)} + \mathbf{A}\cdot\mathbf{x}^{(2)} \tag{30}$$

of linear transformations.

§2 Matrix multiplication

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

$$\mathbf{y} = \mathbf{A}\cdot\mathbf{x} \tag{31a}$$

$$\mathbf{z} = \mathbf{B}\cdot\mathbf{y} \tag{31b}$$

$$\mathbf{w} = \mathbf{C}\cdot\mathbf{z} \tag{31c}$$

These can conveniently be written

$$\mathbf{w} = \mathbf{C}\cdot\left[\mathbf{B}\cdot\left(\mathbf{A}\cdot\mathbf{x}\right)\right]. \tag{32}$$

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

$$\mathbf{D} = \mathbf{B}\cdot\mathbf{A} \qquad \mathbf{E} = \mathbf{C}\cdot\mathbf{D} \tag{33}$$

In terms of matrix elements we have, for example

$$D_{ik} = \sum_{j=0}^{N-1} B_{ij}\,A_{jk} \tag{34}$$

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:

$$\mathbf{C} \cdot \left(\mathbf{B} \cdot \mathbf{A}\right) \equiv \left(\mathbf{C} \cdot \mathbf{B}\right) \cdot \mathbf{A} \tag{35}$$

Equation 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, in general,

$$\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A} \tag{36}$$

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

§§3 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×N matrix **A**, we seek another N×N matrix $\mathbf{A}^{-1}$ with the property that

$$\mathbf{A}^{-1} \cdot \mathbf{A} \equiv \mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I} \tag{37}$$

where **I** is the unit matrix defined in the beginning of this chapter (Eq. 4 — 1's on the main diagonal, 0's everywhere else). We have implied in Eq. 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

$$\left(\mathbf{A}^{-1}\right)^{-1} \equiv \mathbf{A} \tag{38}$$

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

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b};$$

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

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

$$\mathbf{x}=\mathbf{A}^{-1}\cdot\mathbf{b} \tag{39}$$

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.

§§5 An example

Let us now calculate the inverse of our 3x3 matrix from before. The equation is (let $\mathbf{C}=\mathbf{A}^{-1}$ be the inverse[18])

$$\begin{pmatrix}1&0&5\\3&2&4\\1&1&6\end{pmatrix}\begin{pmatrix}c_{00}&c_{01}&c_{02}\\c_{10}&c_{11}&c_{12}\\c_{20}&c_{12}&c_{22}\end{pmatrix}=\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix} \tag{40}$$

---

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

It is easy to see that Eq. 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

$$\begin{pmatrix} 1 & 2/3 & 1/3 \\ 0 & 1 & -1\,1/2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{12} & c_{22} \end{pmatrix} = \begin{pmatrix} 0 & 1/3 & 0 \\ -3/2 & 1/2 & 0 \\ 1/13 & -1/13 & 2/13 \end{pmatrix} \quad (41)$$

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.

§4 LU decomposition

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

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & \cdots \\ a_{10} & a_{11} & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix} = \mathbf{L}\cdot\mathbf{U} = \begin{pmatrix} \lambda_{00} & 0 & 0 \\ \lambda_{10} & \lambda_{11} & 0 \\ \cdots & \cdots & \cdots \end{pmatrix} \begin{pmatrix} \mu_{00} & \mu_{01} & \cdots \\ 0 & \mu_{11} & \cdots \\ 0 & 0 & \cdots \end{pmatrix} \quad (42)$$

then the solution of

19. 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.

$$\left(\mathbf{L}\cdot\mathbf{U}\right)\cdot\mathbf{x} \equiv \mathbf{L}\cdot\left(\mathbf{U}\cdot\mathbf{x}\right) = \mathbf{b} \tag{43}$$

can be found in two steps: First, solve

$$\mathbf{L}\cdot\mathbf{y} = \mathbf{b} \tag{44}$$

for

$$\mathbf{y} \equiv \mathbf{U}\cdot\mathbf{x} \tag{45}$$

via

$$\begin{aligned} \lambda_{00}\, y_0 &= b_0 \\ \lambda_{10}\, y_0 + \lambda_{11}\, y_1 &= b_1 \\ \lambda_{20}\, y_0 + \lambda_{21}\, y_1 + \lambda_{22}\, y_2 &= b_2 \\ &\ldots \textit{ etc. } \ldots \end{aligned} \tag{46}$$

which can be solved successively by forward substtution. Next solve 45 successively (by back-substitution) for $\mathbf{x}$:

$$\begin{aligned} \mu_{N-1\,N-1}\, x_{N-1} &= y_{N-1} \\ \mu_{N-2\,N-2}\, x_{N-2} + \mu_{N-2\,N-1}\, x_{N-1} &= y_{N-2} \\ &\ldots \textit{ etc. } \ldots \end{aligned} \tag{47}$$

The n'th term of Eq. 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 46. Similarly, solving 47 requires about $N^2/2$ additions and multiplications. Thus, the dominant time in solving must be the time to decompose according to Eq. 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

$$\sum_{k=0}^{N-1} \lambda_{mk}\,\mu_{kn} = A_{mn} \tag{48}$$

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

The equations 48 are easy to solve if we do so in a sensible order. Clearly,

$$\lambda_{mk} \equiv 0\,,\ m > k$$
$$\mu_{kn} \equiv 0\,,\ k < n \tag{49}$$

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

$$\mu_{mn} = A_{mn} - \sum_{k=0}^{m-1} \lambda_{mk}\mu_{kn}\,,\ m = 0, 1, \ldots, n$$
$$\lambda_{mn} = \frac{1}{\mu_{nn}}\left(A_{mn} - \sum_{k=0}^{n-1} \lambda_{mk}\mu_{kn}\right)\,,\ m = n+1, n+2, \ldots, N-1 \tag{50}$$

Inspection of Eq. 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 permuting so the row with the largest diagonal element is the current one. Much of the code developed for the Gauss elimination method is applicable, as the file LU.FTH shows.