GAUSSIAN ELIMINATION with EXTRA-PRECISE ACCUMULATION of PRODUCTS

-- is it worth the cost ? --

W. Kahan, April 23, 1983
Univ. of Calif. @ Berkeley

Issues:

1. How to do it?
   a. Extended precision sums in inner loops. (Fast & cheap)
   b. Extended precision temporary vectors. (Slowed by memory)

2. What good is it?
   a. More accuracy in "systematically ill-conditioned" cases, almost as good as if all data were stored with a few extra bits; but otherwise the improvement is small.
   b. Error and its bound grows less quickly with dimension, so improvement is most apparent when dimension is huge.

3. What does it cost?
   a. Hardware is more complicated, but not much slowed.
   b. Subexpression semantics harder to compile.
   c. Method 1a may stumble over paging problems; this can be largely circumvented by trickery and some use of 1b .

4. Examples and comparisons:
   a. On 8087-like architectures (INTEL 86/330, IBM PC FORTH)
   b. Using software floating-point (hp-85, APPLE III)
   c. High-performance machines (ELXSI 6400, ...)

5. Programs listed below:

   LUPA: Triangular Factorization with extra-precise accumulation of inner products (method 1a), and alternative column-oriented code using extra-precise vector to accumulate scalar*vector products (meth. 1b).

   LUXPB: Forward and back sustitution by two methods, like LUPA .

   RBAX: Residual by two methods, like LUPA .

   VNORM: Root-sum-squares norm with extended-range accumulation of squares (method 1a), and alternative code using no extended range but three times slower.

   RESYS: Solve system of linear equations and refine solution iteratively, using LUPA, LUXPB, RBAX and VNORM .

   HUPA: A faster version of LUPA , and
   HUXPB: a faster version of LUXPB , to be used together in situations where page faults seem to preclude extra-precise accumulation of products.

LUPA: W. Kahan, Apr. 23, 1983

Given a square matrix A , we seek triangular factors to satisfy

$$LU = PA ,$$

where

L = unit lower triangular matrix,
U = upper triangular matrix, and
P = permutation matrix represented by indices Ip[...] thus:

$$(Px)[i] = x[Ip[i]] ,$$
$$\text{inverse}(P) = \text{transpose}(P) ,$$
$$(\text{inverse}(P)y)[i] = y[j] \text{ where } Ip[j]=i .$$

If $i > j$ then

$$A[Ip[i],j] = L[i,j]*U[j,j] + \text{Sum}(k<j)(L[i.k]*U[k,j])$$

else

$$A[Ip[i],j] = 1*U[i,j] + \text{Sum}(k<i)(L[i,k]*U[k,j]) .$$

```
Subroutine LUPA( A, LU, Id, IP, N ):
   Integer values  Id, N ;  Integer variable  IP[N+] ;
   Real variables  A[Id,N+], LU[Id,N+] ; ... they may coincide.

   Integer  i, j, k, imax ;
   Logical  UnSav ; ... to save & restore Underflow flag.
   Real  cmax, dmax, rndf, undr,  U[N+] ;
   Tempreal tsum, tpmax, tsmax, T[N+] ; ... more precise than Real
   Equivalence  (U,T) ; ... Save storage by packing  U  inside  T .
   Common /L1BWSP/  T ; ... Shares workspace with other programs.

...Glossary:
...    A[Id,N+] is a square matrix dimensioned  A[Id, at least N ]
...    LU[Id,N+]  will hold  LU[i,j] = L[i,j]  for i>j ,
...                                  = U[i,j] otherwise .
...         ( The program allows  LU  to overwrite  A .)
...    IP[N+] will hold permuted indices  1, 2, 3, ..., N  thus:
...          (Px)[i] = x[IP[i]] .
...    j  is a column index that will run  1, 2, 3, ..., N .
...    U[N+]  will hold temporarily  column j  of  U .
...    T[N+]  will hold temporarily  (column j  of  L)*U[j,j] .
...    cmax holds the max. magnitude in column j  of  A .
...    dmax holds the max. subdiagonal magnitude in column j of PA
...    rndf = 1.000...0001 - 1  ,  measures roundoff among Reals.
...    undr = tiniest positive number ,  at or beyond underflow.
...    i  is a row index that will run  1, 2, 3, ..., N .
...    tsum = A[IP[i],j] - Sum(k)( L[i,k]*U[k,j] ) .
...    tsmax = max. |tsum|  in column j ;  if  tsmax/(8j) > cmax ,
...          U  has grown so big that roundoff may be excessive, so
...          columns 1 and j of  A  should be swapped. (Very rare.)
...    tpmax = max. subdiagonal |tsum|  in column j  for pivoting.
...    imax = row index where  tpmax  occurs.

   UnSav := UndrflowFlag( .false. ) ; ... to save & reset U-flag.
...    Gradual Underflow during factorization is ignorable.
```

```
   rndf := 1.0 ; rndf := nextafter(rndf, 2.0) - rndf ;
...    or else try  rndf := 4.0 ;  rndf := rndf/3.0 ;
...                 rndf := abs( (rndf - 5.0/4.0)*3.0 - 1.0/4.0 ) ;
   undr := 0.0 ;  undr := nextafter(undr, 1.0) ;
...    or else try  undr := underflow threshold for the  Reals .

...    Initialize  IP :
   For  i = 1 to N do  IP[i] := i  ;

...    Outer loop, traversed once per column (j) :
   For  j = 1 to N ;
        cmax := 0.0 ;  tsmax := 0.0 ;
        ... Compute column j of U :
        For  i = 1 to j-1 ;
             tsum := A[IP[i], j] ;  cmax := max{ cmax, abs(tsum) } ;
             For  k =1 to i-1  do  tsum := tsum - LU[i,k]*U[k] ;
             U[i] := tsum ;  tsmax := max{ tsmax, abs(tsum) } ;
             next i ;

        ... Compute potential pivots :
        dmax := 0.0 ;  tpmax := 0.0 ;  imax := j ;
        For  i = j to N ;
             tsum := A[IP[i], j] ;  dmax := max{ dmax, abs(tsum) } ;
             for  k = 1 to j-1  do  tsum := tsum - LU[i,k]*U[k] ;
             T[i] := tsum ;  tsum := abs(tsum) ;
             if  tsum > tpmax then { imax := i ;  tpmax := tsum } ;
             next i ;
        cmax := max{ cmax, dmax } ;  tsmax := max{ tsmax, tpmax } ;
        If  imax = j  then {
                  if  tpmax = 0.0  then {
                                T[j] := max{undr, rndf*dmax} ;
                                go to DivByPiv }
                             }
             else { ... exchange rows  j  and  imax .
                  for  k = 1 to j-1 ;  dmax := LU[imax,k] ;
                       LU[imax,k] := LU[j,k] ;  LU[j,k] := dmax ;
                       next k ;
                  k := IP[imax] ;  IP[imax] := IP[j] ;  IP[j] := k ;
                  }
        If tsmax/(8*j) > cmax  then {
                  Display {"Warning: Extraordinary growth of
                           intermediate results in  LUPA  may lose
                           too much accuracy.  To avoid this loss,
                           recompute after exchanging columns  1
                           and  ", j } ;
                  tsum := 0.0/0.0 ;  ... signals  Invalid Operation.
                                     }
   DivByPiv:  tsum := T[imax] ;  T[imax] := T[j] ;  U[j] := tsum ;
        for  k = 1 to j  do  LU[k,j] := U[k] ;          ... pivot.
        for  k = j+1 to N  do  LU[k,j] := T[k]/tsum ; ... = L[k,j].
        next  j ;
   UnSav := UndrflowFlag(UnSav) ; ... Restore Underflow flag.
   return ;
   end LUPA .
```

```
.....    Alternative Column-Oriented Code    .....

Subroutine LUPA( A, LU, Id, IP, N ):
   Integer values  Id, N ;  Integer variable  IP[N+1] ;
   Real variables  A[Id,N+1], LU[Id,N+1] ; ... they may coincide.

   Integer  i, j, k, imax ;
   Logical  UnSav ; ... to save & restore Underflow flag.
   Real  cmax, dmax, smax, rndf, undr, z ;
   Tempreal  t, tpmax,  T[N+1] ; ... more precise than Real
   Common /L1BWSP/  T ; ... Shares workspace with other programs.

...Glossary:
...    A[Id,N+1] is a square matrix dimensioned  A[Id, at least N ]
...    LU[Id,N+1]  will hold  LU[i,j] = L[i,j]  for i>j ,
...                                   = U[i,j] otherwise .
...         ( The program allows  LU  to overwrite  A .)
...    IP[N+1] will hold permuted indices  1, 2, 3, ..., N  thus:
...           (Px)[i] = x[IP[i]] .
...    j  is a column index that will run  1, 2, 3, ..., N .
...    T[N+1]  will hold temporarily  column j  of  U , and then it
...           will hold temporarily  (column j  of  L)*U[j,j] .
...    cmax holds the max. magnitude in column j  of  A .
...    dmax holds the max. subdiagonal magnitude in column j of PA
...    rndf = 1 000...0001 - 1  ,  measures roundoff among Reals.
...    undr = tiniest positive number ,  at or beyond underflow.
...    i  is a row index that will run  1, 2, 3, ..., N .
...    smax = max. |T[i]|  in column j ;  if  smax/(8j) > cmax ,
...         U  has grown so big that roundoff may be excessive, s
...         columns 1 and j of  A  should be swapped. (Very rare.)
...    tpmax = max. subdiagonal |T[i]|  in column j  for pivoting.
...    imax = row index where  tpmax  occurs.

   UnSav := UndrflowFlag( .false. ) ; ... to save & reset U-flag.
...    Gradual Underflow during factorization is ignorable.

   rndf := 1.0 ; rndf := nextafter(rndf, 2.0) - rndf ;
...    or else try  rndf := 4.0 ;  rndf := rndf/3.0 ;
...                 rndf := abs( (rndf - 5.0/4.0)*3.0 - 1.0/4.0 ) ;
   undr := 0.0 ;  undr := nextafter(undr, 1.0) ;
...    or else try  undr := underflow threshold for the  Reals .

...    Initialize  IP :
   For  i = 1 to N do  IP[i] := i  ;
```

```
...     Outer loop, traversed once per column (j) :
    For  j = 1 to N ;
        tpmax := cmax := dmax := smax := 0.0 ;
        ... Initialize column  T .
        For  i = 1 to N ;
            T[i] := z := A[IP[i], j] ;  z := abs(z) ;
            cmax := max{ cmax, z } ;
            if  i ≥ j  then  dmax := max{ dmax, z } ;
            next i ;

        For  k = 1 to j-1 ;  ... subtract U[k,j]*(col.k of L).
            LU[k,j] := z := T[k] ; ... = U[k,j] .
            smax := max{ smax, abs(z) } ;
            for  i = k+1 to N  do  T[i] := T[i] - LU[i,k]*z ;
            next k ;
        ...  Locate pivot  t ; it maximizes  |T[i]| .
        imax := j ;
        For  i = j to N ;
            t := abs(T[i]) ;
            if  t > tpmax then { imax := i ;  tpmax := t } ;
            next i ;
        If  imax = j  then {
                  if  tpmax = 0.0  then {
                              T[j] := max{undr, rndf*dmax} ;
                              go to DivByPiv }
                         }
            else { ... exchange rows  j  and  imax .
                 for  k = 1 to j-1 ;  dmax := LU[imax,k] ;
                      LU[imax,k] := LU[j,k] ;  LU[j,k] := dmax ;
                      next k ;
                 k := IP[imax] ;  IP[imax] := IP[j] ;  IP[j] := k ;
                  }
        If max{ smax, tpmax }/(8*j) > cmax  then {
                Display {"Warning: Extraordinary growth of
                         intermediate results in  LUPA  may lose
                         too much accuracy.  To avoid this loss,
                         recompute after exchanging columns  1
                         and  ", j } ;
                t := 0.0/0.0 ;  ...  signals  Invalid Operation.
                                                    }
    DivByPiv:  t := T[imax] ;  T[imax] := T[j] ;
        LU[j,j] := t ; ... = pivot U[j,j] .
        for  k = j+1 to N  do  LU[k,j] := T[k]/t ; ... = L[k,j] .
        next  j ;
    UnSav := UndrflowFlag(UnSav) ; ... Restore Underflow flag.
    return ;
    end LUPA .
```

The two LUPA codes should give identical results, including roundoff, but at different speeds depending upon the dimension N and details of the machine's memory management. On a machine that accumulates products in a fast-access register, the first code should be the faster while N is so small that all data fits in a few pages and cache-blocks; otherwise the second code should be the faster, the more so as N increases. (Cf. HUPA below.)

```
LUXPB:                                          W. Kahan,  Apr. 16, 1983
~~~~~
This program solves  LUX = PB  for  X  given matrices
   L = an unit lower triangular  NxN  matrix and
   U = an upper triangular  NxN  matrix stored in  LU  thus:
       if  i > j  then  LU[i,j] = L[i,j]  else  LU[i,j] = U[i,j].
   B = an  NxM  matrix, and
   P = an  NxN  permutation matrix represented by indices  Ip[i]
       thus:    (Px)[i] = x[Ip[i]] .

   X = an  NxM  matrix that will be calculated by solving in turn
       LC = PB ,   C[i,j] + Sum{k<i}( L[i,k]*C[k,j] ) = B[Ip[i],j]
       UX = C  ,   Sum{k≥i}( U[i,k]*X[k,j] ) = C[i,j] .
   The solution  X  may overwrite  B  but not  LU .


Subroutine  LUXPB( LU, Id, IP, N, B, X, M ):
   Integer values  Id, N, M ;  Integer variable  IP[N+] ;
   Real variables  LU[Id, N+],  B[Id, M+],  X[Id, M+] ;

   Integer  i, j, k ;
   Real  C[N+] ;
   Tempreal  tsum ; ... more precise than Reals .
   Common /L1BWSP/  C ;
   Logical UnSav ; ... Gradual Underflow matters only in  X .
   Unsav := UndrflowFlag(.false.) ;

   For  j = 1 to M ; ... solve for column  j  :
       for  i = 1 to N ;
           tsum := B[IP[i],j] ;
           for  k = 1 to i-1  do  tsum := tsum - LU[i,k]*C[k] ;
           C[i] := tsum ;
           next i ;
       for i = N to 1 step -1 ;
           tsum := C[i] ;
           for  k = i+1 to N  do  tsum := tsum - LU[i,k]*C[k] ;
           UnSav := UndrflowFlag(UnSav) ; ... Expose Underflow.
           X[i,j] := C[i] := tsum/LU[i,i] ;
           UnSav := UndrflowFlag(UnSav) ; ... Hide Underflow.
           next i ;
       next j ;
   UnSav := UndrflowFlag(UnSav) ; ... Reveal  X 's  Underflows.
   return ;
   end LUXPB .
```

```
........  Alternative Column-Oriented Code  ........

Subroutine  LUXPB( LU, Id, IP, N, B, X, M ):
   Integer values  Id, N, M ;  Integer variable  IP[N+] ;
   Real variables  LU[Id, N+],  B[Id, M+],  X[Id, M+] ;

   Integer  i, j, k ;
   Real  z ;                                                    ... *1
   Tempreal  C[N+] ; ... more precise than Reals .              ... *2
   Common /L1BWSP/  C ; ... shared workspace.
   Logical UnSav ; ... Gradual Underflow matters only in  X .
   Unsav := UndrflowFlag(.false.) ;

   For  j = 1 to M ; ... solve for column  j  :
       for  i = 1 to N  do  C[i] := B[IP[i],j] ;
       for  k = 1 to N ;
           z := C[k] ;  C[k] := z ;                             ... *3
           for  i = k+1 to N  do  C[i] := C[i] - LU[i,k]*z ;
           next k ;
       for k = N to 1 step -1 ;
           UnSav := UndrflowFlag(UnSav) ; ... Expose Underflow.
           X[k,j] := z := C[k]/LU[k,k] ;                        ... *4
           UnSav := UndrflowFlag(UnSav) ; ... Hide Underflow.
           for  i = 1 to k-1  do  C[i] := C[i] - LU[i,k]*z ;
           next k ;                                             ... *5
       next j ;
   UnSav := UndrflowFlag(UnSav) ; ... Reveal  X 's  Underflows.
   return ;
   end LUXPB .
```

*Notes: The foregoing two codes should produce identical results including the effects of roundoff. However, the second code can be modified slightly to give marginally more accurate results at no significant extra cost provided multiplication of Real by Tempreal costs at most negligibly more than Real by Real .
First merge declarations ... *1 and ... *2 to read

```
   Tempreal  z,  C[N+] ; ... more precise than Reals. ... 1* & 2*
```

Next simplify statement ... *3 to read

```
           z := C[k] ;                                          ... 3*
```

Finally, but only if references to UndrflowFlag() cost rather more than a handfull of memory references, replace ... *4 by

```
           C[k] := z := C[k]/LU[k,k] ;                          ... 4*
```

and move the adjacent statements to bracket a new statement inserted after ... *5 thus:

```
         next k ;                                               ... *5
       UnSav := UndrflowFlag(UnSav) ; ... Expose Underflow.
       for  i = 1 to N  do  X[i,j] := C[i] ;
       UnSav := UndrflowFlag(UnSav) ; ... Hide Underflow.
       next j ; ... etc.
```

(Cf. HUXPB below.)

RBAX: W. Kahan, Apr. 16, 1983

This program calculates a residual R = B - AX given matrices
B = an NxM matrix,
A = an NxN matrix, and
X = an NxM matrix.
R may overwrite B but not A nor X .

```
Subroutine  RBAX( A, Id, N, X, M, B, R ):
   Integer values  Id, N, M ;
   Real variables  A, X, B, R ;

   Integer  i, j, k ;
   Tempreal  tsum ; ... more precise than Reals .

   For  j = 1 to M ; ... compute column  j .
       for  i = 1 to N ; ... row i .
           tsum := B[i,j] ;
           for  k = 1 to N  do  tsum := tsum - A[i,k]*X[k,j] ;
           R[i,j] := tsum ;
           next i ;
       next j ;
   return ;
   end RBAX .
```

..... Alternative Column-Oriented Code .....

```
Subroutine  RBAX( A, Id, N, X, M, B, R ):
   Integer values  Id, N, M ;
   Real variables  A, X, B, R ;

   Integer  i, j, k ;
   Real  z ;
   Tempreal  T[N+] ; ... more precise than Reals .
   Common  /L1BWSP/  T ;  ... shared workspace.

   For  j = 1 to M ; ... compute column  j .
       for  i = 1 to N  do  T[i] := B[i,j] ;
       for  k = 1 to N ;  z := -X[k,j] ;
           for  i = 1 to N  do  T[i] := T[i] + A[i,k]*z ;
           next k ;
       for  i = 1 to N  do  R[i,j] := T[i] ;
       next j ;
   return ;
   end  RBAX .
```

VNORM: W. Kahan, Apr. 14, 1983

For any NxM matrix B ,
VNORM(B, Id, N, M) = || B || = SQRT(trace(B'B))
= SQRT(Sum{ 1≤j≤M, 1≤i≤n }(B[i,j]**2)
where B is dimensioned B[Id, M+] .

```
Real Function  VNORM( B, Id, N, M ):
   Integer values  Id, N, M ;
   Real variable  B[Id, M+] ;
   Tempreal  t ;
   t := 0.0 ;
   for j=1 to M do  for i=1 to N do  t := t + B[i,j]**2 ;
   return  VNORM := SQRT(t) ;
   end  VNORM .
```

..... Alternatively, ...
if Tempreal is unavailable then the following code avoids over/underflow at the cost of some speed and accuracy.

```
Real Function  VNORM( B, Id, N, M ):
   Integer values  Id, N, M ;
   Real variable  B[Id, M+] ;
   Real  s, d, z ;
   Logical UnSav ; ... to save & restore Underflow flag.

   UnSav := UndrflowFlag(.false.) ;
   d := s := 0.0 ;
   for  j = 1 to M ;  for  i = 1 to N ;
       z := abs( B[i,j] ) ;
       if  z > d  then {  s := s*(d/z)**2 + 1.0 ;  d := s  }
           else if  z > 0.0  then  s := s + (z/d)**2 ;
       next i ;  next j ;
   UnSav := UndrflowFlag(UnSav) ; ... Ignore Underflows.
   return  VNORM := d*SQRT(s) ;
   end  VNORM .
```

RESYS: W. Kahan, Apr. 14, 1983

This program uses iterative refinement to solve AX = B and returns RESYS = || B - AX || , where
A = an NxN matrix dimensioned A[Id, N+] ,
B = an NxM matrix dimensioned B[Id, M+] , and
X = an NxM matrix dimensioned X[Id, M+] .

```
Real Function  RESYS( A, Id, N, B, M, X ):
   Integer values  Id, N, M ;
   Real  A[Id, N+],  B[Id, M+],  X[Id, M+] ;

   Integer  i, j, k, L,   IP[N+] ;
   Real  Rold, Rnew, WS[ Id*(N+M+1)+ ] ;
   Common /L2BWSP/  WS ; ... shared work-space.
   Equivalence (IP,WS) ; ... packs  IP  inside  WS .

   Call LUPA( A, WS[Id+1], Id, IP, N ) ; ... LU = PA .
   Call LUXPB( WS[Id+1], Id, IP, N, B, X, M ) ; ... LUX = PB .
   Rold := 0.0 ;  L := 1+Id*(N+1) ;  Go to Residual ;

   Loop:  Rold := Rnew ;
       Call LUXPB( WS[Id+1], Id, IP, N, WS[L], WS[L], M ) ;
       ...   LUZ = PR , and  Z  overwrites  R  in  WS .
       For  j = 1 to M ; ...    do  X := X + Z .
           k := (N+j)*Id ;
           for i = 1 to N do  X[i,j] := X[i,j] + WS[k+i] ;
           next j ;
   Residual:  Call RBAX( A, ID, N, X, M, B, WS[L] ) ;
       ...   R = B - AX  in  WS .
       Rnew := VNORM( WS[L], Id, N, M ) ; ... = || R || .
       if ( Rold = 0.0  .or.  Rold > Rnew ) then go to Loop ;
   return RESYS :=  Rnew ;
   end RESYS .
```

Note: To make this code run faster on a paged machine when N is huge, replace LUPA and LUXPB respectively with HUPA and HUPXB respectively.

HUPA: W. Kahan, Apr. 17, 1983

Given an NxN matrix A , this program does the same as LUPA except faster when N is very large. It calculates factors

LU = PA ,

where

L = unit lower triangular matrix,
U = upper triangular matrix, and
P = permutation matrix represented by indices Ip[...] thus: (Px)[i] = x[Ip[i]] .

If i > j then
A[Ip[i],j] = L[i,j]*U[j,j] + Sum{k<j}(L[i.k]*U[k,j])
else
A[Ip[i],j] = 1*U[i,j] + Sum{k<i}(L[i,k]*U[k,j]) .

But, to diminish the performance degradation caused by page faults and other artifacts of memory management, HUPA packs L thus:

L[i,j] = HU[N+1-i+j, N+1-i] for i >j .

```
Subroutine HUPA( A, HU, Id, IP, N ):
   Integer values  Id, N ;  Integer variable  IP[N+] ;
   Real variables  A[Id,N+], HU[Id,N+] ; ... they must NOT overlap.

   Integer  i, j, k, imax, L ;
   Logical  UnSav ; ... to save & restore Underflow flag.
   Real  cmax, dmax, rndf, undr,  U[N+] ;
   Tempreal tsum, tpmax, tsmax, T[N+] ; ... more precise than Real
   Equivalence  (U,T) ; ... Save storage by packing  U  inside  T .
   Common /L1BWSP/  T ; ... Shares workspace with other programs.

...Glossary:
...    A[Id,N+] is a square matrix dimensioned  A[Id, at least N ]
...    HU[Id,N+]  will hold  HU[i,j] = L[N+1-j, i-j]  for  i > j ,
...                                  = U[i,j] otherwise .
...     ( The program expects  HU and A  NOT to overlap.)
...    IP[N+] will hold permuted indices  1, 2, 3, ..., N  thus:
...           (Px)[i] = x[IP[i]] .
...    j  is a column index that will run  1, 2, 3, ..., N .
...    U[N+]  will hold temporarily  column j  of  U .
...    T[N+]  will hold temporarily  (column j  of  L)*U[j,j] .
...    cmax holds the max. magnitude in column j  of  A .
...    dmax holds the max. subdiagonal magnitude in column j of PA
...    rndf = 1.000...0001 - 1  ,  measures roundoff among Reals.
...    undr = tiniest positive number ,  at or beyond underflow.
...    i  is a row index that will run  1, 2, 3, ..., N .
...    tsum = A[IP[i],j] - Sum{k}( L[i,k]*U[k,j] ) .
...    tsmax = max. |tsum|  in column j ;  if  tsmax/(8j) > cmax ,
...          U  has grown so big that roundoff may be excessive, so
...          columns 1 and j of  A  should be swapped. (Very rare.)
...    tpmax = max. subdiagonal |tsum|  in column j  for pivoting.
...    imax = row index where  tpmax  occurs.

   UnSav := UndrflowFlag( .false. ) ; ... to save & reset U-flag.
...    Gradual Underflow during factorization is ignorable.
```

```
   rndf := 1.0 ; rndf := nextafter(rndf, 2.0) - rndf ;
...    or else try  rndf := 4.0 ;  rndf := rndf/3.0 ;
...                 rndf := abs( (rndf - 5.0/4.0)*3.0 - 1.0/4.0 ) ,
   undr := 0.0 ;  undr := nextafter(undr, 1.0) ;
...    or else try  undr := underflow threshold for the  Reals .

...    Initialize  IP :
   For  i = 1 to N do  IP[i] := i  ;

...    Outer loop, traversed once per column (j) :
   For  j = 1 to N ;
       cmax := 0.0 ;  tsmax := 0.0 ;
       ... Compute column j of U :
       For  i = 1 to j-1 ;
           tsum := A[IP[i], j] ;  cmax := max{ cmax, abs(tsum) } ;
           L := N+1-i ;
           For  k =1 to i-1  do  tsum := tsum - HU[L+k,L]*U[k] ;
           HU[i,j] := U[i] := tsum ;
           tsmax := max{ tsmax, abs(tsum) } ;
           next i ;

       ... Compute potential pivots :
       dmax := 0.0 ;  tpmax := 0.0 ;  imax := j ;
       For  i = j to N ;
           tsum := A[IP[i], j] ;  dmax := max{ dmax, abs(tsum) } ;
           L := N+1-i ;
           for  k = 1 to j-1  do  tsum := tsum - HU[L+k,L]*U[k] ;
           T[i] := tsum ;  tsum := abs(tsum) ;
           if  tsum > tpmax then { imax := i ;  tpmax := tsum } ;
           next i ;
       cmax := max{ cmax, dmax } ;  tsmax := max{ tsmax, tpmax } ;
       If  imax = j  then {
                  if  tpmax = 0.0  then {
                              T[j] := max{undr, rndf*dmax} ;
                              go to DivByPiv }
                        }
           else { ... exchange rows  j  and  imax .
                L := N+1-imax ;  i := N+1-j ;
                for  k = 1 to j-1 ;  dmax := HU[L+k,L] ;
                     HU[L+k,L] := HU[i+k,i] ;  HU[i+k,i] := dmax ;
                     next k ;
                k := IP[imax] ;  IP[imax] := IP[j] ;  IP[j] := k ;
                 }
       If tsmax/(8*j) > cmax  then {
              Display {"Warning: Extraordinary growth of
                       intermediate results in  HUPA  may lose
                       too much accuracy.  To avoid this loss,
                       recompute after exchanging columns  1
                       and  ", j } ;
              tsum := 0.0/0.0 ;  ...  signals  Invalid Operation.
                                    }
   DivByPiv:  tsum := T[imax] ;  T[imax] := T[j] ;
       HU[j,j] := U[j] := tsum ; ... = pivot U[j,j] .
       for  k = 1 to N-j  do  HU[j+k,k] := T[N+1-k]/tsum ;
       next  j ;                          ... = L[N+1-k,j] .
   UnSav := UndrflowFlag(UnSav) ; ... Restore Underflow flag.
   return ;
   end HUPA .
```

HUXPB: W. Kahan, Apr. 18, 1983

This program solves LUX = PB for X given matrices
L = an unit lower triangular NxN matrix and
U = an upper triangular NxN matrix stored in HU thus:
if i > j then HU[i,j] = L[N+1-j,i-j]
else HU[i,j] = U[i,j] .
B = an NxM matrix, and
P = an NxN permutation matrix represented by indices Ip[i] thus: (Px)[i] = x[Ip[i]] .

X = an NxM matrix that will be calculated by solving in turn
LC = PB , C[i,j] + Sum{k<i}(L[i,k]*C[k,j]) = B[Ip[i],j]
UX = C , Sum{k≥i}(U[i,k]*X[k,j]) = C[i,j] .
The solution X may overwrite B but not HU .

```
Subroutine HUXPB( HU, Id, IP, N, B, X, M ):
   Integer values Id, N, M ; Integer variable IP[N+] ;
   Real variables HU[Id, N+], B[Id, M+], X[Id, M+] ;

   Integer i, j, k, L ;
   Real z, C[N+] ;
   Tempreal tsum, T[N+] ; ... more precise than Reals. ... *1
   Equivalence (C,T) ; ... Save storage by packing C inside T .
   Common /L1BWSP/ T ; ... shared workspace.
   Logical UnSav ; ... Gradual Underflow matters only in X .
   Unsav := UndrflowFlag(.false.) ;

   For j = 1 to M ; ... solve for column j :
       for i = 1 to N ;
           tsum := B[IP[i],j] ; L := N+1-i ;
           for k = 1 to i-1 do tsum := tsum - HU[L+k,L]*C[k] ;
           C[i] := tsum ;                                    ... *2
           next i ;
       for k = N to 1 step -1 do T[k] := C[k] ;              ... *3
       for k = N to 1 step -1 ;
           UnSav := UndrflowFlag(UnSav) ; ... Expose Underflow.
           X[k,j] := z := T[k]/HU[k,k] ;                     ... *4
           UnSav := UndrflowFlag(UnSav) ; ... Hide Underflow.
           for i = 1 to k-1 do T[i] := T[i] - HU[i,k]*z ;
           next k ;                                          ... *5
       next j ;
   UnSav := UndrflowFlag(UnSav) ; ... Reveal X 's Underflows.
   return ;
   end HUXPB .
```

*Notes: The foregoing code can be modified slightly to give marginally more accurate results at no significant extra cost provided multiplication of Real by Tempreal is only slightly slower than Real by Real . First merge declaration ... *1 with its two neighbors thus:

```
   Tempreal z, tsum, T[N+] ; ... more precise than Reals. ... 1*
```

Next replace two references to C[...] by T[...] in statement ... *2 and its predecessor; and delete statement ... *3 . Finally, but only if references to UndrflowFlag() cost rather

more than a handfull of memory references, replace ... *4 by
```
          T[k] := z := T[k]/HU[k,k] ;                    ... 4*
```
and move the adjacent statements to bracket a new statement inserted after ... *5 thus:
```
          next k ;                                       ... *5
      UnSav := UndrflowFlag(UnSav) ; ... Expose Underflow.
      for  i = 1 to N  do  X[i,j] := T[i] ;
      UnSav := UndrflowFlag(UnSav) ; ... Hide Underflow.
      next j ; ... etc.
```

Comparison of HU... with LU... :

Programs, like RESYS , that use LUPA and LUXPB can instead use HUPA and HUXPB respectively to get the same results but at different speeds. At first sight, two pairs of programs appear to be under consideration; actually there are three pairs:
 LU... accumulating scalar products extra precisely (method 1a).
 LU... alternative versions using column-oriented code (1b).
 HU... with L[i,j] = HU[N+1-i+j, N+1-i] .
The HU... codes should be never much slower than the first LU... codes, and always significantly faster than the second LU... codes, even on vectorized and pipelined parallel machines, unless compiled with an allegedly optimizing compiler that fails to recognize and optimize subscript references of the form HU[L+k,L] when L is fixed and k varies in an inner loop. Here we assume that arrays are stored by columns as prescribed for Fortran.

The extra-precise accumulation of scalar products is a practice in decline on the largest and fastest computers. Part of the decline is attributable to the omission, from the instruction sets of newer machines, of an instruction that evaluates a product to wider precision than the factors; that omission may be motivated by the belief that page faults and similar artifacts of memory management will drive numerical analysts to use column-oriented codes exclusively rather than sacrifice speed to achieve a little more accuracy. The HU... codes sacrifice neither speed nor accuracy, so perhaps the issues should be reconsidered.

Transpositions and Permutations: W. Kahan, May 8, 1983

There are two ways to keep track of the pivotal exchanges of rows during Gaussian Elimination. One way uses an array ip[.] of n indices ip[1], ip[2], ..., ip[n] to represent the n by n permutation matrix P directly thus:

row ip[i] of A is row i of PA .

Hence, { ip[1], ip[2], ..., ip[n] } is a permutation of the indices { 1, 2, ..., n } . The second way represents P as a product of n-1 transpositions thus:

P = (n-1, k[n-1]) (n-2, k[n-2]) (...) (3, k[3]) (2, k[2]) (1, k[1])

where each (i, k[i]) is a transposition (exchange) of the rows in positions i and k[i] ; moreover i ≤ k[i] . These indices k[.] are called "imax" in programs LUPA and HUPA , where they are encountered and applied in order k[1], k[2], k[3], ..., k[n-1] to produce the array ip[.] thus:

```
for i = 1 to n  do  ip[i] := i ; ... initialization
for i = 1 to n-1  do  swap( ip[i], ip[k[i]] ) ; ... build  ip[.]
```

Given this array ip[.] , can we reverse the process to recover the array k[.] ? Yes. But first the permutation iq[.] inverse to ip[.] must be calculated thus:

```
for i = 1 to n  do  iq[ip[i]] := i ; ... inversion .
```

Now row iq[i] of PA is row i of A . Next we gradually transforr ip[.] and iq[.] back to identity permutations while keeping them inverse to each other as k[.] is recovered thus:

```
for i = 1 to n-1  do begin
                    k[i] := ip[i] ; ... reversion
                    swap( ip[i], ip[iq[i]] ) ; ... so now ip[i]=i
                    swap( iq[i], iq[k[i]] ) ; ...  so now iq[i]=i
                    end ;
```

One application of the reversion is to reveal the sign of

det(A) = det(PA)/det(P) = det(U)/det(P) , where

det(P) = (-1)^(number of instances when k[i] > i) .

Another application is to the encoding of P within L to dispense with the bother of providing for the array IP[.] when the factors L and U are saved for subsequent re-use. The encode function E(x) maps the reals x with |x| ≤ 1 to |E(x)| ≥ 2 :

```
if  x=0  then  E(x) := Copysign(2, x)  else  E(x) := Scalb(x, K)
```

where K is an integer barely large enough that Scalb(1.0, K) overflows to infinity. K = 128 for Single, or 1024 for Double precision in the proposed IEEE standard p754. The decode function D(x) inverse to E(x) is

```
if  x is infinite  then  D(x) := Copysign(1, x)
else  if  |x| = 2  then  D(x) := Copysign(0, x)
else  D(x) := Scalb(x, -K) ; ... and ignore Underflow .
```

Then to encode IP[.] within L we revert IP[.] to k[.] and then replace L[k[j],j] by E(L[k[j],j) whenever k[j] > j . To recover k[.] later, we scan { L[i,j], j < i ≤ n } to find where |L[k[j],j]| > 1 , thereby determining k[j] > j ; otherwise k[j] = j .

The success of the reversion process above is tantamount to a

Theorem: Every permutation of n positions can be expressed uniquely as a product of n-1 transpositions (n-1,k[n-1]) (n-2,k[n-2]) (...) (2,k[2]) (1,k[1]) in which each k[i] ≥ i .

The theorem's validity can be confirmed by running the following program:

```
Program Proof(uptoN):
   procedure Nest(m):
      if m>0 then for j = m to n do begin
                                    k[m] := j ;  Nest(m-1)
                                    end
      else begin
           for i = 1 to n  do  ip[i] := i ;
           for i = 1 to n-1  do  swap( ip[i], ip[k[i]] ) ;
           for i = 1 to n  do  iq[ip[i]] := i ;
           for i = 1 to n-1  do begin
                               if ip[i]=k[i] then begin
                                         swap( ip[i], ip[iq[i]] );
                                         swap( iq[i], iq[k[i]] )
                                               end
                               else begin
                                    write( "Test fails at  n ="; n
                                           " with  i = "; i
                                           " and  k[.] = "; k[.]
                                           "      ip[.] = "; ip[.]
                                           "      iq[.] = "; iq );
                                    stop
                                    end
                             end;
           write( " n = "; n ; " tested successfully." )
           end end Nest ;
   for n = 1 to uptoN  do  Nest(n) ;  write( "End of test.)
   end Proof.
```

Inverting the Hilbert matrix: W. Kahan, Dec. 8, 1983

Floating-point matrix inversion programs are customarily tested on an $n \times n$ Hilbert matrix H whose elements are $H_{i,j} = 1/(i+j+p-1)$ for $1 \le i,j \le n$ and any integer $p \ge 0$. Because H becomes so ill-conditioned as n or p becomes big, its inverse $W = H^{-1}$ becomes difficult to compute accurately in the face of roundoff. None the less, a way exists to compute W exactly and easily; it uses a little-known formula $W = V H V$ where V is a diagonal matrix of integers $V_j = (-1)^j((n+j+p-1)!)/((n-j)!\,(j-1)!\,(j+p-1)!)$ obtained from a simple recurrence in which only integers appear:

$V_1 := -n$; for $k = 1$ to p do $V_1 := (V_1/k)(n+k)$;
for $j = 1$ to $n-1$ do $V_{j+1} := (((V_j/(j+p))(j-n))/j)(n+j-p)$.

Then $W_{i,j} := V_i V_j/(i+j+p-1)$. (S. Schechter, MTAC, 1959)

Since the elements of H are reciprocals of integers, they cannot be represented exactly in floating-point but must be rounded off. These initial rounding errors may do more damage to H^{-1} than the inversion program under test. To avoid them, we actually use

$A := mH$, where $m := LCM(\, p+1,\ p+2,\ p+3,\ \ldots,\ p+2n-1\,)$,

which has integer elements all representable exactly in floating-point provided n and p are not too big. Then the inversion program is tested by using it to solve $A X = mI$ numerically for X . Here I is the $n \times n$ identity matrix. Since ideally X should match W , the error introduced by the program under test is indicated by displaying a rough measure of the relative error in X :

$$r = \max_{i,j} |X_{i,j} - W_{i,j}| \,/\, |W_{i,j}| \quad .$$

This statistic makes no allowance for the ill-condition of H nor for the precision of the arithmetic in which X was calculated. The ill-condition of H can be gauged from

$$c = \max_i \sum_j |H_{i,j} W_{j,i}| \ ,$$

which exceeds 1 to an extent that indicates how severe is cancellation when $H W = I$ is evaluated. The precision is indicated by

$u = 1.000\ldots001 - 1.000\ldots000 = 0.000\ldots001$
= One unit in the last place carried in numbers near 1 .

Then one figure of merit for the program under test is

$$q = r/(uc) \ ;$$

the smaller is q , the better the program. Normally $r < 1$ and $q < n$; but when $r > 1$ the matrices A and H are so nearly singular that the program cannot be relied upon to get even one significant digit correct in X , and then the value of q becomes irrelevant. Another figure of merit is the largest value of n for which $r < 1$; the larger is this n , the better.

The error r , and therefore q , depend upon rounding errors that occur during the calculation of X , but rounding errors are not entirely dependable; they behave sometimes almost as if they were random. Therefore prudence demands that roundoff be sampled more than once before conclusions be drawn about a program's vulnerability to roundoff. For instance, most matrix inversion programs, and certainly those using LUPA and HUPA above, will generate different rounding errors if the column ordering of the matrix being inverted is changed. To be more specific, let S be the $n \times n$ permutation matrix that reverses order; that is,

$$S = \begin{vmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{vmatrix} \quad \text{when } n = 3 .$$

Then the inverse of S H S is S W S , but the computed solution Z of (S A S) Z = mI usually differs from S X S because of differences in the way roundoff occurs. Calculating r and q from Z instead of X gives a second opinion about the effect of roundoff upon the program under test.

To calculate m = LCM(p+1, p+2, p+3, ..., p+2n-1) , do thus:

```
GCD(x,y): while y ≠ 0  do { z := y ;  y := x rem z ;  x := z } ;
          return  GCD := |x| end GCD .

LCM(x,y): if x=0 then return LCM := 0
                 else return LCM := ( |y|/GCD(x,y) )*|x| end LCM.

m := p+1 ;  for k = p+2 to p+2*n-1  do  m := LCM(m,k) ;
  ...                                yields  m .
```

For example, when p = 1 , we find ...

n = 8	n = 9	n = 10 or 11	n = 12
m = 360360	m = 12252240	m = 232792560	m = 5354228880