Chapter B2. Solution of Linear Algebraic Equations

```
SUBROUTINE gaussj(a,b)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerand,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a,b
```

Linear equation solution by Gauss-Jordan elimination, equation (2.1.1). a is an $N \times N$ input coefficient matrix. b is an $N \times M$ input matrix containing M right-hand-side vectors. On output, a is replaced by its matrix inverse, and b is replaced by the corresponding set of solution vectors.

```
INTEGER(I4B), DIMENSION(size(a,1)) :: ipiv,indxr,indxc
```

These arrays are used for bookkeeping on the pivoting.

```
LOGICAL(LGT), DIMENSION(size(a,1)) :: lpiv
REAL(SP) :: pivinv
REAL(SP), DIMENSION(size(a,1)) :: dumc
INTEGER(I4B), TARGET :: irc(2)
INTEGER(I4B) :: i,l,n
INTEGER(I4B), POINTER :: irow,icol
n=assert_eq(size(a,1),size(a,2),size(b,1),'gaussj')
irow => irc(1)
icol => irc(2)
ipiv=0
do i=1,n                          Main loop over columns to be reduced.
    lpiv = (ipiv == 0)            Begin search for a pivot element.
    irc=maxloc(abs(a),outerand(lpiv,lpiv))
    ipiv(icol)=ipiv(icol)+1
    if (ipiv(icol) > 1) call nrerror('gaussj: singular matrix (1)')
```

We now have the pivot element, so we interchange rows, if needed, to put the pivot element on the diagonal. The columns are not physically interchanged, only relabeled: indxc(i), the column of the ith pivot element, is the ith column that is reduced, while indxr(i) is the row in which that pivot element was originally located. If indxr(i) $\neq$ indxc(i) there is an implied column interchange. With this form of bookkeeping, the solution b's will end up in the correct order, and the inverse matrix will be scrambled by columns.

```
    if (irow /= icol) then
        call swap(a(irow,:),a(icol,:))
        call swap(b(irow,:),b(icol,:))
    end if
    indxr(i)=irow                 We are now ready to divide the pivot row by the pivot
    indxc(i)=icol                     element, located at irow and icol.
    if (a(icol,icol) == 0.0) &
        call nrerror('gaussj: singular matrix (2)')
    pivinv=1.0_sp/a(icol,icol)
    a(icol,icol)=1.0
    a(icol,:)=a(icol,:)*pivinv
    b(icol,:)=b(icol,:)*pivinv
    dumc=a(:,icol)                Next, we reduce the rows, except for the pivot one, of
    a(:,icol)=0.0                     course.
```

```
    a(icol,icol)=pivinv
    a(1:icol-1,:)=a(1:icol-1,:)-outerprod(dumc(1:icol-1),a(icol,:))
    b(1:icol-1,:)=b(1:icol-1,:)-outerprod(dumc(1:icol-1),b(icol,:))
    a(icol+1:,:)=a(icol+1:,:)-outerprod(dumc(icol+1:),a(icol,:))
    b(icol+1:,:)=b(icol+1:,:)-outerprod(dumc(icol+1:),b(icol,:))
end do
```

It only remains to unscramble the solution in view of the column interchanges. We do this by interchanging pairs of columns in the reverse order that the permutation was built up.

```
do l=n,1,-1
    call swap(a(:,indxr(l)),a(:,indxc(l)))
end do
END SUBROUTINE gaussj
```

f90 `irow => irc(1) ... icol => irc(2)` The `maxloc` intrinsic returns the location of the maximum value of an array as an integer array, in this case of size 2. Pre-pointing pointer variables to components of the array that will be thus set makes possible convenient references to the desired row and column positions.

`irc=maxloc(abs(a),outerand(lpiv,lpiv))` The combination of `maxloc` and one of the `outer...` routines from `nrutil` allows for a very concise formulation. If this task is done with loops, it becomes the ungainly "flying vee,"

```
aa=0.0
do i=1,n
   if (lpiv(i)) then
      do j=1,n
         if (lpiv(j)) then
            if (abs(a(i,j)) > aa) then
               aa=abs(a(i,j))
               irow=i
               icol=j
            endif
         endif
      end do
   end do
end do
```

`call swap(a(irow,:),a(icol,:))` The `swap` routine (in `nrutil`) is concise and convenient. Fortran 90's ability to overload multiple routines onto a single name is vital here: Much of the convenience would vanish if we had to remember variant routine names for each variable type and rank of object that might be swapped.

Even better, here, than overloading would be if Fortran 90 allowed user-written *elemental* procedures (procedures with unspecified or arbitrary rank and shape), like the intrinsic elemental procedures built into the language. Fortran 95 will, but Fortran 90 doesn't.

One quick (if superficial) test for how much parallelism is achieved in a Fortran 90 routine is to count its do-loops, and compare that number to the number of do-loops in the Fortran 77 version of the same routine. Here, in `gaussj`, 13 do-loops are reduced to 2.

```
a(1:icol-1,:)=... b(1:icol-1,:)=...
a(icol+1:,:)=...  b(icol+1:,:)=...
```

Here the same operation is applied to every row of a, and to every row of b, *except* row number icol. On a massively multiprocessor (MMP) machine it would be better to use a logical mask and do all of a in a single statement, all of b in another one. For a small-scale parallel (SSP) machine, the lines as written should saturate the machine's concurrency, and they avoid the additional overhead of testing the mask.

This would be a good place to point out, however, that linear algebra routines written in Fortran 90 are likely *never* to be competitive with the hand-coded library routines that are generally supplied as part of MMP programming environments. If you are using our routines instead of library routines written specifically for your architecture, you are wasting cycles!

⋆ ⋆ ⋆

```
SUBROUTINE ludcmp(a,indx,d)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,nrerror,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
REAL(SP), INTENT(OUT) :: d
```

Given an $N \times N$ input matrix a, this routine replaces it by the LU decomposition of a rowwise permutation of itself. On output, a is arranged as in equation (2.3.14); indx is an output vector of length N that records the row permutation effected by the partial pivoting; d is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with lubksb to solve linear equations or invert a matrix.

```
REAL(SP), DIMENSION(size(a,1)) :: vv          vv stores the implicit scaling of each row.
REAL(SP), PARAMETER :: TINY=1.0e-20_sp        A small number.
INTEGER(I4B) :: j,n,imax
n=assert_eq(size(a,1),size(a,2),size(indx),'ludcmp')
d=1.0                                         No row interchanges yet.
vv=maxval(abs(a),dim=2)                       Loop over rows to get the implicit scaling
if (any(vv == 0.0)) call nrerror('singular matrix in ludcmp')      information.
  There is a row of zeros.
vv=1.0_sp/vv                                  Save the scaling.
do j=1,n
    imax=(j-1)+imaxloc(vv(j:n)*abs(a(j:n,j)))        Find the pivot row.
    if (j /= imax) then                       Do we need to interchange rows?
        call swap(a(imax,:),a(j,:))           Yes, do so...
        d=-d                                  ...and change the parity of d.
        vv(imax)=vv(j)                        Also interchange the scale factor.
    end if
    indx(j)=imax
    if (a(j,j) == 0.0) a(j,j)=TINY
      If the pivot element is zero the matrix is singular (at least to the precision of the al-
      gorithm). For some applications on singular matrices, it is desirable to substitute TINY
      for zero.
    a(j+1:n,j)=a(j+1:n,j)/a(j,j)              Divide by the pivot element.
    a(j+1:n,j+1:n)=a(j+1:n,j+1:n)-outerprod(a(j+1:n,j),a(j,j+1:n))
      Reduce remaining submatrix.
end do
END SUBROUTINE ludcmp
```

f90 vv=maxval(abs(a),dim=2) A single statement finds the maximum absolute value in each row. Fortran 90 intrinsics like maxval generally "do their thing" *in* the dimension specified by *dim* and return a result with a shape corresponding to the *other* dimensions. Thus, here, vv's size is that of the *first* dimension of a.

`imax=(j-1)+imaxloc(vv(j:n)*abs(a(j:n,j))` Here we see why the `nrutil` routine `imaxloc` is handy: We want the index, in the range `1:n` of a quantity to be searched for only in the limited range `j:n`. Using `imaxloc`, we just add back the proper offset of `j-1`. (Using only Fortran 90 intrinsics, we could write `imax=(j-1)+sum(maxloc(vv(j:n)*abs(a(j:n,j))))`, but the use of `sum` just to turn an array of length 1 into a scalar seems sufficiently confusing as to be avoided.)

`a(j+1:n,j+1:n)=a(j+1:n,j+1:n)-outerprod(a(j+1:n,j),a(j,j+1:n))` The Fortran 77 version of `ludcmp`, using Crout's algorithm for the reduction, does not parallelize well: The elements are updated by $O(N^2)$ separate dot product operations in a particular order. Here we use a slightly different reduction, termed "outer product Gaussian elimination" by Golub and Van Loan [1], that requires just N steps of matrix-parallel reduction. (See their §3.2.3 and §3.2.9 for the algorithm, and their §3.4.1 to understand how the pivoting is performed.)

We use `nrutil`'s routine `outerprod` instead of the more cumbersome pure Fortran 90 construction:

```
spread(a(j+1:n,j),dim=2,ncopies=n-j)*spread(a(j,j+1:n),dim=1,ncopies=n-j)
```

```
SUBROUTINE lubksb(a,indx,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
```

Solves the set of N linear equations $A \cdot X = B$. Here the $N \times N$ matrix `a` is input, not as the original matrix A, but rather as its LU decomposition, determined by the routine `ludcmp`. `indx` is input as the permutation vector of length N returned by `ludcmp`. `b` is input as the right-hand-side vector B, also of length N, and returns with the solution vector X. `a` and `indx` are not modified by this routine and can be left in place for successive calls with different right-hand sides `b`. This routine takes into account the possibility that `b` will begin with many zero elements, so it is efficient for use in matrix inversion.

```
INTEGER(I4B) :: i,n,ii,ll
REAL(SP) :: summ
n=assert_eq(size(a,1),size(a,2),size(indx),'lubksb')
ii=0                          When ii is set to a positive value, it will become the in-
do i=1,n                          dex of the first nonvanishing element of b. We now do
    ll=indx(i)                    the forward substitution, equation (2.3.6). The only new
    summ=b(ll)                    wrinkle is to unscramble the permutation as we go.
    b(ll)=b(i)
    if (ii /= 0) then
        summ=summ-dot_product(a(i,ii:i-1),b(ii:i-1))
    else if (summ /= 0.0) then
        ii=i                  A nonzero element was encountered, so from now on we will
    end if                        have to do the dot product above.
    b(i)=summ
end do
do i=n,1,-1                   Now we do the backsubstitution, equation (2.3.7).
    b(i) = (b(i)-dot_product(a(i,i+1:n),b(i+1:n)))/a(i,i)
end do
END SUBROUTINE lubksb
```

Conceptually, the search for the first nonvanishing element of b (index `ii`) should be moved out of the first do-loop. However, in practice, the need to unscramble the permutation, and also considerations of performance

on scalar machines, cause us to write this very scalar-looking code. The performance penalty on parallel machines should be minimal.

⋆ ⋆ ⋆

Serial and parallel algorithms for tridiagonal problems are quite different. We therefore provide separate routines `tridag_ser` and `tridag_par`. In the `MODULE nr` interface file, one or the other of these (your choice) is given the generic name `tridag`. Of course, *either* version will work correctly on any computer; it is only a question of efficiency. See §22.2 for the numbering of the equation coefficients, and for a description of the parallel algorithm.

```
SUBROUTINE tridag_ser(a,b,c,r,u)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u
    Solves for a vector u of size N the tridiagonal linear set given by equation (2.4.1) using a
    serial algorithm. Input vectors b (diagonal elements) and r (right-hand sides) have size N,
    while a and c (off-diagonal elements) are size N − 1.
REAL(SP), DIMENSION(size(b)) :: gam        One vector of workspace, gam is needed.
INTEGER(I4B) :: n,j
REAL(SP) :: bet
n=assert_eq((/size(a)+1,size(b),size(c)+1,size(r),size(u)/),'tridag_ser')
bet=b(1)
if (bet == 0.0) call nrerror('tridag_ser: Error at code stage 1')
  If this happens then you should rewrite your equations as a set of order N − 1, with u2
  trivially eliminated.
u(1)=r(1)/bet
do j=2,n                                   Decomposition and forward substitution.
    gam(j)=c(j-1)/bet
    bet=b(j)-a(j-1)*gam(j)
    if (bet == 0.0) &                      Algorithm fails; see below routine in Vol. 1.
        call nrerror('tridag_ser: Error at code stage 2')
    u(j)=(r(j)-a(j-1)*u(j-1))/bet
end do
do j=n-1,1,-1                              Backsubstitution.
    u(j)=u(j)-gam(j+1)*u(j+1)
end do
END SUBROUTINE tridag_ser
```

```
RECURSIVE SUBROUTINE tridag_par(a,b,c,r,u)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : tridag_ser
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u
    Solves for a vector u of size N the tridiagonal linear set given by equation (2.4.1) using a
    parallel algorithm. Input vectors b (diagonal elements) and r (right-hand sides) have size
    N, while a and c (off-diagonal elements) are size N − 1.
INTEGER(I4B), PARAMETER :: NPAR_TRIDAG=4      Determines when serial algorithm is in-
INTEGER(I4B) :: n,n2,nm,nx                        voked.
REAL(SP), DIMENSION(size(b)/2) :: y,q,piva
REAL(SP), DIMENSION(size(b)/2-1) :: x,z
REAL(SP), DIMENSION(size(a)/2) :: pivc
n=assert_eq((/size(a)+1,size(b),size(c)+1,size(r),size(u)/),'tridag_par')
if (n < NPAR_TRIDAG) then
    call tridag_ser(a,b,c,r,u)
else
    if (maxval(abs(b(1:n))) == 0.0) &         Algorithm fails; see below routine in Vol. 1.
```

```
        call nrerror('tridag_par: possible singular matrix')
    n2=size(y)
    nm=size(pivc)
    nx=size(x)
    piva = a(1:n-1:2)/b(1:n-1:2)              Zero the odd a's and even c's, giving x,
    pivc = c(2:n-1:2)/b(3:n:2)                    y, z, q.
    y(1:nm) = b(2:n-1:2)-piva(1:nm)*c(1:n-2:2)-pivc*a(2:n-1:2)
    q(1:nm) = r(2:n-1:2)-piva(1:nm)*r(1:n-2:2)-pivc*r(3:n:2)
    if (nm < n2) then
        y(n2) = b(n)-piva(n2)*c(n-1)
        q(n2) = r(n)-piva(n2)*r(n-1)
    end if
    x = -piva(2:n2)*a(2:n-2:2)
    z = -pivc(1:nx)*c(3:n-1:2)
    call tridag_par(x,y,z,q,u(2:n:2))         Recurse and get even u's.
    u(1) = (r(1)-c(1)*u(2))/b(1)              Substitute and get odd u's.
    u(3:n-1:2) = (r(3:n-1:2)-a(2:n-2:2)*u(2:n-2:2) &
        -c(3:n-1:2)*u(4:n:2))/b(3:n-1:2)
    if (nm == n2) u(n)=(r(n)-a(n-1)*u(n-1))/b(n)
end if
END SUBROUTINE tridag_par
```

The serial version `tridag_ser` is called when the routine has recursed its way down to sufficiently small subproblems. The point at which this occurs is determined by the parameter `NPAR_TRIDAG` whose optimal value is likely machine-dependent. Notice that `tridag_ser` must here be called by its specific name, not by the generic `tridag` (which might itself be overloaded with either `tridag_ser` or `tridag_par`).

⋆ ⋆ ⋆

```
SUBROUTINE banmul(a,m1,m2,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,arth
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
INTEGER(I4B), INTENT(IN) :: m1,m2
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(OUT) :: b
```

Matrix multiply $\mathbf{b} = \mathbf{A} \cdot \mathbf{x}$, where $\mathbf{A}$ is band diagonal with `m1` rows below the diagonal and `m2` rows above. If the input vector $\mathbf{x}$ and output vector $\mathbf{b}$ are of length N, then the array `a(1:`N`,1:m1+m2+1)` stores $\mathbf{A}$ as follows: The diagonal elements are in `a(1:`N`,m1+1)`. Subdiagonal elements are in `a(`j`:`N`,1:m1)` (with $j > 1$ appropriate to the number of elements on each subdiagonal). Superdiagonal elements are in `a(1:`j`,m1+2:m1+m2+1)` with $j < N$ appropriate to the number of elements on each superdiagonal.

```
INTEGER(I4B) :: m,n
n=assert_eq(size(a,1),size(b),size(x),'banmul: n')
m=assert_eq(size(a,2),m1+m2+1,'banmul: m')
b=sum(a*eoshift(spread(x,dim=2,ncopies=m), &
    dim=1,shift=arth(-m1,1,m)),dim=2)
END SUBROUTINE banmul
```

```
b=sum(a*eoshift(spread(x,dim=2,ncopies=m), &
    dim=1,shift=arth(-m1,1,m)),dim=2)
```

This is a good example of Fortran 90 at both its best and its worst: best, because it allows quite subtle combinations of fully parallel operations to be built up; worst, because the resulting code is virtually incomprehensible!

What is going on becomes clearer if we imagine a temporary array y with a declaration like `REAL(SP), DIMENSION(size(a,1),size(a,2)) :: y`. Then, the above single line decomposes into

```
y=spread(x,dim=2,ncopies=m)              [Duplicate x into columns of y.]
y=eoshift(y,dim=1,shift=arth(-m1,1,m))   [Shift columns by a linear progression.]
b=sum(a*y,dim=2)                         [Multiply by the band-diagonal elements,
                                              and sum.]
```

We use here a relatively rare subcase of the `eoshift` intrinsic, using a vector value for the `shift` argument to accomplish the simultaneous shifting of a bunch of columns, by different amounts (here specified by the linear progression returned by `arth`).

If you still don't see how this accomplishes the multiplication of a band diagonal matrix by a vector, work through a simple example by hand.

```
SUBROUTINE bandec(a,m1,m2,al,indx,d)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,swap,arth
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: m1,m2
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: al
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
REAL(SP), INTENT(OUT) :: d
REAL(SP), PARAMETER :: TINY=1.0e-20_sp
    Given an N × N band diagonal matrix A with m1 subdiagonal rows and m2 superdiagonal
    rows, compactly stored in the array a(1:N,1:m1+m2+1) as described in the comment for
    routine banmul, this routine constructs an LU decomposition of a rowwise permutation of
    A. The upper triangular matrix replaces a, while the lower triangular matrix is returned in
    al(1:N,1:m1). indx is an output vector of length N that records the row permutation
    effected by the partial pivoting; d is output as ±1 depending on whether the number of
    row interchanges was even or odd, respectively. This routine is used in combination with
    banbks to solve band-diagonal sets of equations.
INTEGER(I4B) :: i,k,l,mdum,mm,n
REAL(SP) :: dum
n=assert_eq(size(a,1),size(al,1),size(indx),'bandec: n')
mm=assert_eq(size(a,2),m1+m2+1,'bandec: mm')
mdum=assert_eq(size(al,2),m1,'bandec: mdum')
a(1:m1,:)=eoshift(a(1:m1,:),dim=2,shift=arth(m1,-1,m1))     Rearrange the storage a
d=1.0                                                            bit.
do k=1,n                                  For each row...
    l=min(m1+k,n)
    i=imaxloc(abs(a(k:l,1)))+k-1          Find the pivot element.
    dum=a(i,1)
    if (dum == 0.0) a(k,1)=TINY
      Matrix is algorithmically singular, but proceed anyway with TINY pivot (desirable in some
      applications).
    indx(k)=i
    if (i /= k) then                      Interchange rows.
        d=-d
        call swap(a(k,1:mm),a(i,1:mm))
    end if
    do i=k+1,l                            Do the elimination.
        dum=a(i,1)/a(k,1)
        al(k,i-k)=dum
        a(i,1:mm-1)=a(i,2:mm)-dum*a(k,2:mm)
        a(i,mm)=0.0
    end do
end do
END SUBROUTINE bandec
```

f90 `a(1:m1,:)=eoshift(a(1:m1,:),...` See similar discussion of `eoshift` for `banmul`, just above.

`i=imaxloc(abs(a(k:l,1)))+k-1` See discussion of `imaxloc` on p. 1017.

Notice that the above is *not* well parallelized for MMP machines: the outer do-loop is done N times, where N, the diagonal length, is potentially the largest dimension in the problem. Small-scale parallel (SSP) machines, and scalar machines, are not disadvantaged, because the parallelism of order `mm=m1+m2+1` in the inner loops can be enough to saturate their concurrency.

We don't know of an N-parallel algorithm for decomposing band diagonal matrices, at least one that has any reasonably concise expression in Fortran 90. Conceptually, one can view a band diagonal matrix as a *block tridiagonal* matrix, and then apply the same recursive strategy as was used in `tridag_par`. However, the implementation details of this are daunting. (We would welcome a user-contributed routine, clear, concise, and with parallelism of order N.)

```
SUBROUTINE banbks(a,m1,m2,al,indx,b)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,al
INTEGER(I4B), INTENT(IN) :: m1,m2
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
    Given the arrays a, al, and indx as returned from bandec, and given a right-hand-side
    vector b, solves the band diagonal linear equations A·x = b. The solution vector x overwrites
    b. The other input arrays are not modified, and can be left in place for successive calls with
    different right-hand sides.
INTEGER(I4B) :: i,k,l,mdum,mm,n
n=assert_eq(size(a,1),size(al,1),size(b),size(indx),'banbks: n')
mm=assert_eq(size(a,2),m1+m2+1,'banbks: mm')
mdum=assert_eq(size(al,2),m1,'banbks: mdum')
do k=1,n                    Forward substitution, unscrambling the permuted rows as we
    l=min(n,m1+k)                 go.
    i=indx(k)
    if (i /= k) call swap(b(i),b(k))
    b(k+1:l)=b(k+1:l)-al(k,1:l-k)*b(k)
end do
do i=n,1,-1                 Backsubstitution.
    l=min(mm,n-i+1)
    b(i)=(b(i)-dot_product(a(i,2:l),b(1+i:i+l-1)))/a(i,1)
end do
END SUBROUTINE banbks
```

As for `bandec`, the routine `banbks` is not parallelized on the large dimension N, though it does give the compiler the opportunity for ample small-scale parallelization inside the loops.

⋆ ⋆ ⋆

```
SUBROUTINE mprove(a,alud,indx,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : lubksb
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,alud
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(IN) :: b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
```

Improves a solution vector x of the linear set of equations $A \cdot X = B$. The $N \times N$ matrix a and the N-dimensional vectors b and x are input. Also input is alud, the LU decomposition of a as returned by ludcmp, and the N-dimensional vector indx also returned by that routine. On output, only x is modified, to an improved set of values.

```
INTEGER(I4B) :: ndum
REAL(SP), DIMENSION(size(a,1)) :: r
ndum=assert_eq((/size(a,1),size(a,2),size(alud,1),size(alud,2),size(b),&
    size(x),size(indx)/),'mprove')
r=matmul(real(a,dp),real(x,dp))-real(b,dp)
```

Calculate the right-hand side, accumulating the residual in double precision.

```
call lubksb(alud,indx,r)        Solve for the error term,
x=x-r                           and subtract it from the old solution.
END SUBROUTINE mprove
```

f90 `assert_eq((/.../),'mprove')` This overloaded version of the `nrutil` routine `assert_eq` makes use of a trick for passing a variable number of scalar arguments to a routine: Put them into an array constructor, `(/.../)`, and pass the array. The receiving routine can use the `size` intrinsic to count them. The technique has some obvious limitations: All the arguments in the array must be of the same type; and the arguments are passed, in effect, by *value*, not by address, so they must be, in effect, `INTENT(IN)`.

`r=matmul(real(a,dp),real(x,dp))-real(b,dp)` Since Fortran 90's elemental intrinsics operate with the type of their arguments, we can use the `real(...,dp)`'s to force the `matmul` matrix multiplication to be done in double precision, which is what we want. In Fortran 77, we would have to do the matrix multiplication with temporary double precision variables, both inconvenient and (since Fortran 77 has no dynamic memory allocation) a waste of memory.

⋆ ⋆ ⋆

```
SUBROUTINE svbksb_sp(u,w,v,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
REAL(SP), DIMENSION(:,:), INTENT(IN) :: u,v
REAL(SP), DIMENSION(:), INTENT(IN) :: w,b
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
```

Solves $A \cdot X = B$ for a vector X, where A is specified by the arrays u, v, w as returned by svdcmp. Here u is $M \times N$, v is $N \times N$, and w is of length N. b is the M-dimensional input right-hand side. x is the N-dimensional output solution vector. No input quantities are destroyed, so the routine may be called sequentially with different b's.

```
INTEGER(I4B) :: mdum,ndum
REAL(SP), DIMENSION(size(x)) :: tmp
mdum=assert_eq(size(u,1),size(b),'svbksb_sp: mdum')
ndum=assert_eq((/size(u,2),size(v,1),size(v,2),size(w),size(x)/),&
    'svbksb_sp: ndum')
where (w /= 0.0)
    tmp=matmul(b,u)/w           Calculate diag$(1/w_j)U^T B$,
elsewhere
    tmp=0.0                     but replace $1/w_j$ by zero if $w_j = 0$.
end where
```

```
x=matmul(v,tmp)                Matrix multiply by V to get answer.
END SUBROUTINE svbksb_sp
```

f90 `where (w /= 0.0)...tmp=...elsewhere...tmp=` Normally, when a `where ...elsewhere` construction is used to set a variable (here `tmp`) to one or another value, we like to replace it with a `merge` expression. Here, however, the `where` is required to guarantee that a division by zero doesn't occur. The rule is that `where` will *never* evaluate expressions that are excluded by the mask in the `where` line, but other constructions, like `merge`, *might* perform speculative evaluation of more than one possible outcome before selecting the applicable one.

Because singular value decomposition is something that one often wants to do in double precision, we include a double-precision version. In `nr`, the single- and double-precision versions are overloaded onto the name `svbksb`.

```
SUBROUTINE svbksb_dp(u,w,v,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,v
REAL(DP), DIMENSION(:), INTENT(IN) :: w,b
REAL(DP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B) :: mdum,ndum
REAL(DP), DIMENSION(size(x)) :: tmp
mdum=assert_eq(size(u,1),size(b),'svbksb_dp: mdum')
ndum=assert_eq((/size(u,2),size(v,1),size(v,2),size(w),size(x)/),&
    'svbksb_dp: ndum')
where (w /= 0.0)
    tmp=matmul(b,u)/w
elsewhere
    tmp=0.0
end where
x=matmul(v,tmp)
END SUBROUTINE svbksb_dp
```

```
SUBROUTINE svdcmp_sp(a,w,v)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerprod
USE nr, ONLY : pythag
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: w
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
```

Given an $M \times N$ matrix `a`, this routine computes its singular value decomposition, $A = U \cdot W \cdot V^T$. The matrix U replaces `a` on output. The diagonal matrix of singular values W is output as the N-dimensional vector `w`. The $N \times N$ matrix V (not the transpose V^T) is output as `v`.

```
INTEGER(I4B) :: i,its,j,k,l,m,n,nm
REAL(SP) :: anorm,c,f,g,h,s,scale,x,y,z
REAL(SP), DIMENSION(size(a,1)) :: tempm
REAL(SP), DIMENSION(size(a,2)) :: rv1,tempn
m=size(a,1)
n=assert_eq(size(a,2),size(v,1),size(v,2),size(w),'svdcmp_sp')
g=0.0
scale=0.0
do i=1,n                                Householder reduction to bidiagonal form.
    l=i+1
    rv1(i)=scale*g
    g=0.0
    scale=0.0
    if (i <= m) then
```

```
        scale=sum(abs(a(i:m,i)))
        if (scale /= 0.0) then
            a(i:m,i)=a(i:m,i)/scale
            s=dot_product(a(i:m,i),a(i:m,i))
            f=a(i,i)
            g=-sign(sqrt(s),f)
            h=f*g-s
            a(i,i)=f-g
            tempn(l:n)=matmul(a(i:m,i),a(i:m,l:n))/h
            a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),tempn(l:n))
            a(i:m,i)=scale*a(i:m,i)
        end if
    end if
    w(i)=scale*g
    g=0.0
    scale=0.0
    if ((i <= m) .and. (i /= n)) then
        scale=sum(abs(a(i,l:n)))
        if (scale /= 0.0) then
            a(i,l:n)=a(i,l:n)/scale
            s=dot_product(a(i,l:n),a(i,l:n))
            f=a(i,l)
            g=-sign(sqrt(s),f)
            h=f*g-s
            a(i,l)=f-g
            rv1(l:n)=a(i,l:n)/h
            tempm(l:m)=matmul(a(l:m,l:n),a(i,l:n))
            a(l:m,l:n)=a(l:m,l:n)+outerprod(tempm(l:m),rv1(l:n))
            a(i,l:n)=scale*a(i,l:n)
        end if
    end if
end do
anorm=maxval(abs(w)+abs(rv1))
do i=n,1,-1                             Accumulation of right-hand transformations.
    if (i < n) then
        if (g /= 0.0) then
            v(l:n,i)=(a(i,l:n)/a(i,l))/g       Double division to avoid possible under-
            tempn(l:n)=matmul(a(i,l:n),v(l:n,l:n))        flow.
            v(l:n,l:n)=v(l:n,l:n)+outerprod(v(l:n,i),tempn(l:n))
        end if
        v(i,l:n)=0.0
        v(l:n,i)=0.0
    end if
    v(i,i)=1.0
    g=rv1(i)
    l=i
end do
do i=min(m,n),1,-1                      Accumulation of left-hand transformations.
    l=i+1
    g=w(i)
    a(i,l:n)=0.0
    if (g /= 0.0) then
        g=1.0_sp/g
        tempn(l:n)=(matmul(a(l:m,i),a(l:m,l:n))/a(i,i))*g
        a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),tempn(l:n))
        a(i:m,i)=a(i:m,i)*g
    else
        a(i:m,i)=0.0
    end if
    a(i,i)=a(i,i)+1.0_sp
end do
do k=n,1,-1                             Diagonalization of the bidiagonal form: Loop over
    do its=1,30                             singular values, and over allowed iterations.
        do l=k,1,-1                     Test for splitting.
```

```
    nm=l-1
    if ((abs(rv1(l))+anorm) == anorm) exit
      Note that rv1(1) is always zero, so can never fall through bottom of loop.
    if ((abs(w(nm))+anorm) == anorm) then
        c=0.0                   Cancellation of rv1(l), if l > 1.
        s=1.0
        do i=l,k
            f=s*rv1(i)
            rv1(i)=c*rv1(i)
            if ((abs(f)+anorm) == anorm) exit
            g=w(i)
            h=pythag(f,g)
            w(i)=h
            h=1.0_sp/h
            c= (g*h)
            s=-(f*h)
            tempm(1:m)=a(1:m,nm)
            a(1:m,nm)=a(1:m,nm)*c+a(1:m,i)*s
            a(1:m,i)=-tempm(1:m)*s+a(1:m,i)*c
        end do
        exit
    end if
end do
z=w(k)
if (l == k) then                Convergence.
    if (z < 0.0) then           Singular value is made nonnegative.
        w(k)=-z
        v(1:n,k)=-v(1:n,k)
    end if
    exit
end if
if (its == 30) call nrerror('svdcmp_sp: no convergence in svdcmp')
x=w(l)                          Shift from bottom 2-by-2 minor.
nm=k-1
y=w(nm)
g=rv1(nm)
h=rv1(k)
f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0_sp*h*y)
g=pythag(f,1.0_sp)
f=((x-z)*(x+z)+h*((y/(f+sign(g,f)))-h))/x
c=1.0                           Next QR transformation:
s=1.0
do j=l,nm
    i=j+1
    g=rv1(i)
    y=w(i)
    h=s*g
    g=c*g
    z=pythag(f,h)
    rv1(j)=z
    c=f/z
    s=h/z
    f= (x*c)+(g*s)
    g=-(x*s)+(g*c)
    h=y*s
    y=y*c
    tempn(1:n)=v(1:n,j)
    v(1:n,j)=v(1:n,j)*c+v(1:n,i)*s
    v(1:n,i)=-tempn(1:n)*s+v(1:n,i)*c
    z=pythag(f,h)
    w(j)=z                      Rotation can be arbitrary if z = 0.
    if (z /= 0.0) then
        z=1.0_sp/z
        c=f*z
```

```
            s=h*z
        end if
        f= (c*g)+(s*y)
        x=-(s*g)+(c*y)
        tempm(1:m)=a(1:m,j)
        a(1:m,j)=a(1:m,j)*c+a(1:m,i)*s
        a(1:m,i)=-tempm(1:m)*s+a(1:m,i)*c
    end do
    rv1(l)=0.0
    rv1(k)=f
    w(k)=x
  end do
end do
END SUBROUTINE svdcmp_sp
```

The SVD algorithm implemented above does not parallelize very well. There are two parts to the algorithm. The first, reduction to bidiagonal form, can be parallelized. The second, the iterative diagonalization of the bidiagonal form, uses QR transformations that are intrinsically serial. There have been proposals for parallel SVD algorithms [2], but we do not have sufficient experience with them yet to recommend them over the well-established serial algorithm.

`tempn(l:n)=matmul...a(i:m,l:n)=...outerprod...` Here is an example of an update as in equation (22.1.6). In this case b_i is independent of i: It is simply `1/h`. The lines beginning `tempm(l:m)=matmul` about 16 lines down are of a similar form, but with the terms in the opposite order in the `matmul`.

As with `svbksb`, single- and double-precision versions of the routines are overloaded onto the name `svdcmp` in `nr`.

```
SUBROUTINE svdcmp_dp(a,w,v)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerprod
USE nr, ONLY : pythag
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(DP), DIMENSION(:), INTENT(OUT) :: w
REAL(DP), DIMENSION(:,:), INTENT(OUT) :: v
INTEGER(I4B) :: i,its,j,k,l,m,n,nm
REAL(DP) :: anorm,c,f,g,h,s,scale,x,y,z
REAL(DP), DIMENSION(size(a,1)) :: tempm
REAL(DP), DIMENSION(size(a,2)) :: rv1,tempn
m=size(a,1)
n=assert_eq(size(a,2),size(v,1),size(v,2),size(w),'svdcmp_dp')
g=0.0
scale=0.0
do i=1,n
    l=i+1
    rv1(i)=scale*g
    g=0.0
    scale=0.0
    if (i <= m) then
        scale=sum(abs(a(i:m,i)))
        if (scale /= 0.0) then
            a(i:m,i)=a(i:m,i)/scale
            s=dot_product(a(i:m,i),a(i:m,i))
            f=a(i,i)
            g=-sign(sqrt(s),f)
```

```
            h=f*g-s
            a(i,i)=f-g
            tempn(l:n)=matmul(a(i:m,i),a(i:m,l:n))/h
            a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),tempn(l:n))
            a(i:m,i)=scale*a(i:m,i)
        end if
    end if
    w(i)=scale*g
    g=0.0
    scale=0.0
    if ((i <= m) .and. (i /= n)) then
        scale=sum(abs(a(i,l:n)))
        if (scale /= 0.0) then
            a(i,l:n)=a(i,l:n)/scale
            s=dot_product(a(i,l:n),a(i,l:n))
            f=a(i,l)
            g=-sign(sqrt(s),f)
            h=f*g-s
            a(i,l)=f-g
            rv1(l:n)=a(i,l:n)/h
            tempm(l:m)=matmul(a(l:m,l:n),a(i,l:n))
            a(l:m,l:n)=a(l:m,l:n)+outerprod(tempm(l:m),rv1(l:n))
            a(i,l:n)=scale*a(i,l:n)
        end if
    end if
end do
anorm=maxval(abs(w)+abs(rv1))
do i=n,1,-1
    if (i < n) then
        if (g /= 0.0) then
            v(l:n,i)=(a(i,l:n)/a(i,l))/g
            tempn(l:n)=matmul(a(i,l:n),v(l:n,l:n))
            v(l:n,l:n)=v(l:n,l:n)+outerprod(v(l:n,i),tempn(l:n))
        end if
        v(i,l:n)=0.0
        v(l:n,i)=0.0
    end if
    v(i,i)=1.0
    g=rv1(i)
    l=i
end do
do i=min(m,n),1,-1
    l=i+1
    g=w(i)
    a(i,l:n)=0.0
    if (g /= 0.0) then
        g=1.0_dp/g
        tempn(l:n)=(matmul(a(l:m,i),a(l:m,l:n))/a(i,i))*g
        a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),tempn(l:n))
        a(i:m,i)=a(i:m,i)*g
    else
        a(i:m,i)=0.0
    end if
    a(i,i)=a(i,i)+1.0_dp
end do
do k=n,1,-1
    do its=1,30
        do l=k,1,-1
            nm=l-1
            if ((abs(rv1(l))+anorm) == anorm) exit
            if ((abs(w(nm))+anorm) == anorm) then
                c=0.0
                s=1.0
                do i=l,k
```

```
            f=s*rv1(i)
            rv1(i)=c*rv1(i)
            if ((abs(f)+anorm) == anorm) exit
            g=w(i)
            h=pythag(f,g)
            w(i)=h
            h=1.0_dp/h
            c= (g*h)
            s=-(f*h)
            tempm(1:m)=a(1:m,nm)
            a(1:m,nm)=a(1:m,nm)*c+a(1:m,i)*s
            a(1:m,i)=-tempm(1:m)*s+a(1:m,i)*c
        end do
        exit
    end if
end do
z=w(k)
if (l == k) then
    if (z < 0.0) then
        w(k)=-z
        v(1:n,k)=-v(1:n,k)
    end if
    exit
end if
if (its == 30) call nrerror('svdcmp_dp: no convergence in svdcmp')
x=w(l)
nm=k-1
y=w(nm)
g=rv1(nm)
h=rv1(k)
f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0_dp*h*y)
g=pythag(f,1.0_dp)
f=((x-z)*(x+z)+h*((y/(f+sign(g,f)))-h))/x
c=1.0
s=1.0
do j=l,nm
    i=j+1
    g=rv1(i)
    y=w(i)
    h=s*g
    g=c*g
    z=pythag(f,h)
    rv1(j)=z
    c=f/z
    s=h/z
    f= (x*c)+(g*s)
    g=-(x*s)+(g*c)
    h=y*s
    y=y*c
    tempn(1:n)=v(1:n,j)
    v(1:n,j)=v(1:n,j)*c+v(1:n,i)*s
    v(1:n,i)=-tempn(1:n)*s+v(1:n,i)*c
    z=pythag(f,h)
    w(j)=z
    if (z /= 0.0) then
        z=1.0_dp/z
        c=f*z
        s=h*z
    end if
    f= (c*g)+(s*y)
    x=-(s*g)+(c*y)
    tempm(1:m)=a(1:m,j)
    a(1:m,j)=a(1:m,j)*c+a(1:m,i)*s
    a(1:m,i)=-tempm(1:m)*s+a(1:m,i)*c
```

```
        end do
        rv1(l)=0.0
        rv1(k)=f
        w(k)=x
    end do
end do
END SUBROUTINE svdcmp_dp
```

```
FUNCTION pythag_sp(a,b)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: pythag_sp
```

Computes $(a^2 + b^2)^{1/2}$ without destructive underflow or overflow.

```
REAL(SP) :: absa,absb
absa=abs(a)
absb=abs(b)
if (absa > absb) then
    pythag_sp=absa*sqrt(1.0_sp+(absb/absa)**2)
else
    if (absb == 0.0) then
        pythag_sp=0.0
    else
        pythag_sp=absb*sqrt(1.0_sp+(absa/absb)**2)
    end if
end if
END FUNCTION pythag_sp
```

```
FUNCTION pythag_dp(a,b)
USE nrtype
IMPLICIT NONE
REAL(DP), INTENT(IN) :: a,b
REAL(DP) :: pythag_dp
REAL(DP) :: absa,absb
absa=abs(a)
absb=abs(b)
if (absa > absb) then
    pythag_dp=absa*sqrt(1.0_dp+(absb/absa)**2)
else
    if (absb == 0.0) then
        pythag_dp=0.0
    else
        pythag_dp=absb*sqrt(1.0_dp+(absa/absb)**2)
    end if
end if
END FUNCTION pythag_dp
```

⋆ ⋆ ⋆

```
SUBROUTINE cyclic(a,b,c,alpha,beta,r,x)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : tridag
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN):: a,b,c,r
REAL(SP), INTENT(IN) :: alpha,beta
REAL(SP), DIMENSION(:), INTENT(OUT):: x
```

Solves the "cyclic" set of linear equations given by equation (2.7.9). a, b, c, and r are input vectors, while x is the output solution vector, all of the same size. alpha and beta are the corner entries in the matrix. The input is not modified.

```
INTEGER(I4B) :: n
REAL(SP) :: fact,gamma
REAL(SP), DIMENSION(size(x)) :: bb,u,z
n=assert_eq((/size(a),size(b),size(c),size(r),size(x)/),'cyclic')
call assert(n > 2, 'cyclic arg')
gamma=-b(1)                                  Avoid subtraction error in forming bb(1).
bb(1)=b(1)-gamma                             Set up the diagonal of the modified tridiag-
bb(n)=b(n)-alpha*beta/gamma                      onal system.
bb(2:n-1)=b(2:n-1)
call tridag(a(2:n),bb,c(1:n-1),r,x)          Solve A · x = r.
u(1)=gamma                                   Set up the vector u.
u(n)=alpha
u(2:n-1)=0.0
call tridag(a(2:n),bb,c(1:n-1),u,z)          Solve A · z = u.
fact=(x(1)+beta*x(n)/gamma)/(1.0_sp+z(1)+beta*z(n)/gamma)      Form v·x/(1+v·z).
x=x-fact*z                                   Now get the solution vector x.
END SUBROUTINE cyclic
```

The parallelism in cyclic is in tridag. Users with multiprocessor machines will want to be sure that, in nrutil, they have set the name tridag to be overloaded with tridag_par instead of tridag_ser.

⋆ ⋆ ⋆

The routines sprsin, sprsax, sprstx, sprstp, and sprsdiag give roughly equivalent functionality to the corresponding Fortran 77 routines, but they are *not* plug compatible. Instead, they take advantage of (and illustrate) several Fortran 90 features that are not present in Fortran 77.

In the module nrtype we define a TYPE sprs2_sp for two-dimensional sparse, square, matrices, in single precision, as follows

```
TYPE sprs2_sp
   INTEGER(I4B) :: n,len
   REAL(SP), DIMENSION(:), POINTER :: val
   INTEGER(I4B), DIMENSION(:), POINTER :: irow
   INTEGER(I4B), DIMENSION(:), POINTER :: jcol
END TYPE sprs2_sp
```

This has much less structure to it than the "row-indexed sparse storage mode" used in Volume 1. Here, a sparse matrix is just a list of values, and corresponding lists giving the row and column number that each value is in. Two integers n and len give, respectively, the underlying size (number of rows or columns) in the full matrix, and the number of stored nonzero values. While the previously used row-indexed scheme can be somewhat more efficient for serial machines, it does not parallelize conveniently, while this one does (though with some caveats; see below).

```
SUBROUTINE sprsin_sp(a,thresh,sa)
USE nrtype; USE nrutil, ONLY : arth,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), INTENT(IN) :: thresh
TYPE(sprs2_sp), INTENT(OUT) :: sa
```

Converts a square matrix a to sparse storage format as sa. Only elements of a with magnitude $\geq$ thresh are retained.

```
INTEGER(I4B) :: n,len
LOGICAL(LGT), DIMENSION(size(a,1),size(a,2)) :: mask
n=assert_eq(size(a,1),size(a,2),'sprsin_sp')
mask=abs(a)>thresh
len=count(mask)              How many elements to store?
allocate(sa%val(len),sa%irow(len),sa%jcol(len))
sa%n=n
sa%len=len
sa%val=pack(a,mask)          Grab the values, row, and column numbers.
sa%irow=pack(spread(arth(1,1,n),2,n),mask)
sa%jcol=pack(spread(arth(1,1,n),1,n),mask)
END SUBROUTINE sprsin_sp
```

```
SUBROUTINE sprsin_dp(a,thresh,sa)
USE nrtype; USE nrutil, ONLY : arth,assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: a
REAL(DP), INTENT(IN) :: thresh
TYPE(sprs2_dp), INTENT(OUT) :: sa
INTEGER(I4B) :: n,len
LOGICAL(LGT), DIMENSION(size(a,1),size(a,2)) :: mask
n=assert_eq(size(a,1),size(a,2),'sprsin_dp')
mask=abs(a)>thresh
len=count(mask)
allocate(sa%val(len),sa%irow(len),sa%jcol(len))
sa%n=n
sa%len=len
sa%val=pack(a,mask)
sa%irow=pack(spread(arth(1,1,n),2,n),mask)
sa%jcol=pack(spread(arth(1,1,n),1,n),mask)
END SUBROUTINE sprsin_dp
```

f90 Note that the routines sprsin_sp and sprsin_dp — single and double precision versions of the same algorithm — are overloaded onto the name sprsin in module nr. We supply both forms because the routine linbcg, below, works in double precision.

sa%irow=pack(spread(arth(1,1,n),2,n),mask) The trick here is to use the same mask, abs(a)>thresh, in three consecutive pack expressions, thus guaranteeing that the corresponding elements of the array argument get selected for packing. The first time, we get the desired matrix element values. The second time (above code fragment), we construct a matrix with each element having the value of its *row* number. The third time, we construct a matrix with each element having the value of its *column* number.

```
SUBROUTINE sprsax_sp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,scatter_add
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
```

Multiply a matrix sa in sparse matrix format by a vector x, giving a vector b.

```
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprsax_sp')
b=0.0_sp
call scatter_add(b,sa%val*x(sa%jcol),sa%irow)
```

Each sparse matrix entry adds a term to some component of b.

```
END SUBROUTINE sprsax_sp
```

```
SUBROUTINE sprsax_dp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,scatter_add
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(IN) :: x
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprsax_dp')
b=0.0_dp
call scatter_add(b,sa%val*x(sa%jcol),sa%irow)
END SUBROUTINE sprsax_dp
```

`call scatter_add(b,sa%val*x(sa%jcol),sa%irow)` Since more than one component of the middle vector argument will, in general, need to be added into the same component of b, we must resort to a call to the `nrutil` routine `scatter_add` to achieve parallelism. *However*, this parallelism is achieved only if a parallel version of `scatter_add` is available! As we have discussed previously (p. 984), Fortran 90 does not provide any scatter-with-combine (here, scatter-with-add) facility, insisting instead that indexed operations yield non-colliding addresses. Luckily, almost all parallel machines do provide such a facility as a library program. In HPF, for example, the equivalent of `scatter_add` is `SUM_SCATTER`.

The call to `scatter_add` above is equivalent to the do-loop

```
b=0.0
do k=1,sa%len
   b(sa%irow(k))=b(sa%irow(k))+sa%val(k)*x(sa%jcol(k))
end do
```

```
SUBROUTINE sprstx_sp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,scatter_add
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
```

Multiply the transpose of a matrix sa in sparse matrix format by a vector x, giving a vector b.

```
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprstx_sp')
b=0.0_sp
call scatter_add(b,sa%val*x(sa%irow),sa%jcol)
```

Each sparse matrix entry adds a term to some component of b.

```
END SUBROUTINE sprstx_sp
```

```
SUBROUTINE sprstx_dp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,scatter_add
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(IN) :: x
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprstx_dp')
b=0.0_dp
call scatter_add(b,sa%val*x(sa%irow),sa%jcol)
END SUBROUTINE sprstx_dp
```

Precisely the same comments as for sprsax apply to sprstx. The call to scatter_add is here equivalent to

```
b=0.0
do k=1,sa%len
   b(sa%jcol(k))=b(sa%jcol(k))+sa%val(k)*x(sa%irow(k))
end do
```

```
SUBROUTINE sprstp(sa)
USE nrtype
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(INOUT) :: sa
   Replaces sa, in sparse matrix format, by its transpose.
INTEGER(I4B), DIMENSION(:), POINTER :: temp
temp=>sa%irow                We need only swap the row and column pointers.
sa%irow=>sa%jcol
sa%jcol=>temp
END SUBROUTINE sprstp
```

```
SUBROUTINE sprsdiag_sp(sa,b)
USE nrtype; USE nrutil, ONLY : array_copy,assert_eq
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION(:), INTENT(OUT) :: b
   Extracts the diagonal of a matrix sa in sparse matrix format into a vector b.
REAL(SP), DIMENSION(size(b)) :: val
INTEGER(I4B) :: k,l,ndum,nerr
INTEGER(I4B), DIMENSION(size(b)) :: i
LOGICAL(LGT), DIMENSION(:), ALLOCATABLE :: mask
ndum=assert_eq(sa%n,size(b),'sprsdiag_sp')
l=sa%len
allocate(mask(l))
mask = (sa%irow(1:l) == sa%jcol(1:l))      Find diagonal elements.
call array_copy(pack(sa%val(1:l),mask),val,k,nerr)      Grab the values...
i(1:k)=pack(sa%irow(1:l),mask)             ...and their locations.
deallocate(mask)
b=0.0                                      Zero b because zero values not stored in sa.
b(i(1:k))=val(1:k)                         Scatter values into correct slots.
END SUBROUTINE sprsdiag_sp
```

```
SUBROUTINE sprsdiag_dp(sa,b)
USE nrtype; USE nrutil, ONLY : array_copy,assert_eq
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION(:), INTENT(OUT) :: b
REAL(DP), DIMENSION(size(b)) :: val
INTEGER(I4B) :: k,l,ndum,nerr
INTEGER(I4B), DIMENSION(size(b)) :: i
LOGICAL(LGT), DIMENSION(:), ALLOCATABLE :: mask
ndum=assert_eq(sa%n,size(b),'sprsdiag_dp')
l=sa%len
allocate(mask(l))
mask = (sa%irow(1:l) == sa%jcol(1:l))
call array_copy(pack(sa%val(1:l),mask),val,k,nerr)
i(1:k)=pack(sa%irow(1:l),mask)
deallocate(mask)
b=0.0
b(i(1:k))=val(1:k)
END SUBROUTINE sprsdiag_dp
```

f90 `call array_copy(pack(sa%val(1:l),mask),val,k,nerr)` We use the `nrutil` routine `array_copy` because we don't know in advance how many nonzero diagonal elements will be selected by `mask`. Of course we could count them with a `count(mask)`, but this is an extra step, and inefficient on scalar machines.

`i(1:k)=pack(sa%irow(1:l),mask)` Using the same mask, we pick out the corresponding locations of the diagonal elements. No need to use `array_copy` now, since we know the value of `k`.

`b(i(1:k))=val(1:k)` Finally, we can put each element in the right place. Notice that if the sparse matrix is ill-formed, with more than one value stored for the same diagonal element (which should not happen!) then the vector subscript `i(1:k)` is a "many-one section" and its use on the left-hand side is illegal.

⋆ ⋆ ⋆

```
SUBROUTINE linbcg(b,x,itol,tol,itmax,iter,err)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : atimes,asolve,snrm
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: b
REAL(DP), DIMENSION(:), INTENT(INOUT) :: x
INTEGER(I4B), INTENT(IN) :: itol,itmax
REAL(DP), INTENT(IN) :: tol
INTEGER(I4B), INTENT(OUT) :: iter
REAL(DP), INTENT(OUT) :: err
REAL(DP), PARAMETER :: EPS=1.0e-14_dp
```

Double precision is a good idea in this routine.

Solves $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for `x`, given `b` of the same length, by the iterative biconjugate gradient method. On input `x` should be set to an initial guess of the solution (or all zeros); `itol` is 1,2,3, or 4, specifying which convergence test is applied (see text); `itmax` is the maximum number of allowed iterations; and `tol` is the desired convergence tolerance. On output, `x` is reset to the improved solution, `iter` is the number of iterations actually taken, and `err` is the estimated error. The matrix $\mathbf{A}$ is referenced only through the user-supplied routines `atimes`, which computes the product of either $\mathbf{A}$ or its transpose on a vector; and `asolve`,

which solves $\widetilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$ or $\widetilde{\mathbf{A}}^T \cdot \mathbf{x} = \mathbf{b}$ for some preconditioner matrix $\widetilde{\mathbf{A}}$ (possibly the trivial diagonal part of $\mathbf{A}$).

```
INTEGER(I4B) :: n
REAL(DP) :: ak,akden,bk,bkden,bknum,bnrm,dxnrm,xnrm,zm1nrm,znrm
REAL(DP), DIMENSION(size(b)) :: p,pp,r,rr,z,zz
n=assert_eq(size(b),size(x),'linbcg')
iter=0
call atimes(x,r,0)                          Calculate initial residual. Input to atimes is
r=b-r                                           x(1:n), output is r(1:n); the final 0
rr=r                                            indicates that the matrix (not its trans-
!   call atimes(r,rr,0)                         pose) is to be used.
  Uncomment this line to get the "minimum residual" variant of the algorithm.
select case(itol)                           Calculate norms for use in stopping criterion,
    case(1)                                     and initialize z.
        bnrm=snrm(b,itol)
        call asolve(r,z,0)                  Input to asolve is r(1:n), output is z(1:n);
    case(2)                                     the final 0 indicates that the matrix $\widetilde{\mathbf{A}}$
        call asolve(b,z,0)                      (not its transpose) is to be used.
        bnrm=snrm(z,itol)
        call asolve(r,z,0)
    case(3:4)
        call asolve(b,z,0)
        bnrm=snrm(z,itol)
        call asolve(r,z,0)
        znrm=snrm(z,itol)
    case default
        call nrerror('illegal itol in linbcg')
end select
do                                          Main loop.
    if (iter > itmax) exit
    iter=iter+1
    call asolve(rr,zz,1)                    Final 1 indicates use of transpose matrix $\widetilde{\mathbf{A}}^T$.
    bknum=dot_product(z,rr)                 Calculate coefficient bk and direction vectors
    if (iter == 1) then                         p and pp.
        p=z
        pp=zz
    else
        bk=bknum/bkden
        p=bk*p+z
        pp=bk*pp+zz
    end if
    bkden=bknum                             Calculate coefficient ak, new iterate x, and
    call atimes(p,z,0)                          new residuals r and rr.
    akden=dot_product(z,pp)
    ak=bknum/akden
    call atimes(pp,zz,1)
    x=x+ak*p
    r=r-ak*z
    rr=rr-ak*zz
    call asolve(r,z,0)                      Solve $\widetilde{\mathbf{A}} \cdot \mathbf{z} = \mathbf{r}$ and check stopping criterion.
    select case(itol)
        case(1)
            err=snrm(r,itol)/bnrm
        case(2)
            err=snrm(z,itol)/bnrm
        case(3:4)
            zm1nrm=znrm
            znrm=snrm(z,itol)
            if (abs(zm1nrm-znrm) > EPS*znrm) then
                dxnrm=abs(ak)*snrm(p,itol)
                err=znrm/abs(zm1nrm-znrm)*dxnrm
            else
                err=znrm/bnrm               Error may not be accurate, so loop again.
                cycle
```

```
            end if
            xnrm=snrm(x,itol)
            if (err <= 0.5_dp*xnrm) then
                err=err/xnrm
            else
                err=znrm/bnrm                Error may not be accurate, so loop again.
                cycle
            end if
    end select
    write (*,*) ' iter=',iter,' err=',err
    if (err <= tol) exit
end do
END SUBROUTINE linbcg
```

f90 `case default...call nrerror('illegal itol in linbcg')` It's *always* a good idea to trap errors when the value of a case construction is supplied externally to the routine, as here.

```
FUNCTION snrm(sx,itol)
USE nrtype
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: sx
INTEGER(I4B), INTENT(IN) :: itol
REAL(DP) :: snrm
    Compute one of two norms for a vector sx, as signaled by itol. Used by linbcg.
if (itol <= 3) then
    snrm=sqrt(dot_product(sx,sx))            Vector magnitude norm.
else
    snrm=maxval(abs(sx))                     Largest component norm.
end if
END FUNCTION snrm
```

```
SUBROUTINE atimes(x,r,itrnsp)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : sprsax,sprstx        DOUBLE PRECISION versions of sprsax and sprstx.
USE xlinbcg_data                    The matrix is accessed through this module.
REAL(DP), DIMENSION(:), INTENT(IN) :: x
REAL(DP), DIMENSION(:), INTENT(OUT) :: r
INTEGER(I4B), INTENT(IN) :: itrnsp
INTEGER(I4B) :: n
n=assert_eq(size(x),size(r),'atimes')
if (itrnsp == 0) then
    call sprsax(sa,x,r)
else
    call sprstx(sa,x,r)
end if
END SUBROUTINE atimes
```

```
SUBROUTINE asolve(b,x,itrnsp)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : sprsdiag          DOUBLE PRECISION version of sprsdiag.
USE xlinbcg_data                 The matrix is accessed through this module.
REAL(DP), DIMENSION(:), INTENT(IN) :: b
REAL(DP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B), INTENT(IN) :: itrnsp
INTEGER(I4B) :: ndum
ndum=assert_eq(size(b),size(x),'asolve')
call sprsdiag(sa,x)
```

The matrix $\tilde{\mathbf{A}}$ is taken to be the diagonal part of $\mathbf{A}$. Since the transpose matrix has the same diagonal, the flag `itrnsp` is not used.

```
if (any(x == 0.0)) call nrerror('asolve: singular diagonal matrix')
x=b/x
END SUBROUTINE asolve
```

f90 The routines `atimes` and `asolve` are examples of user-supplied routines that interface `linbcg` to a user-supplied method for multiplying the user's sparse matrix by a vector, and for solving the preconditioner matrix equation. Here, we have used these routines to connect `linbcg` to the sparse matrix machinery developed above. If we were instead using the different sparse matrix machinery of Volume 1, we would modify `atimes` and `asolve` accordingly.

`USE xlinbcg_data` This user-supplied module is assumed to have `sa` (the sparse matrix) in it.

⋆ ⋆ ⋆

```
FUNCTION vander(x,q)
USE nrtype; USE nrutil, ONLY : assert_eq,outerdiff
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: x,q
REAL(DP), DIMENSION(size(x)) :: vander
```

Solves the Vandermonde linear system $\sum_{i=1}^{N} x_i^{k-1} w_i = q_k$ $(k = 1, \ldots, N)$. Input consists of the vectors `x` and `q` of length N. The solution w (also of length N) is returned in `vander`.

```
REAL(DP), DIMENSION(size(x)) :: c
REAL(DP), DIMENSION(size(x),size(x)) :: a
INTEGER(I4B) :: i,n
n=assert_eq(size(x),size(q),'vander')
if (n == 1) then
    vander(1)=q(1)
else
    c(:)=0.0                                  Initialize array.
    c(n)=-x(1)                                Coefficients of the master polynomial are found
    do i=2,n                                      by recursion.
        c(n+1-i:n-1)=c(n+1-i:n-1)-x(i)*c(n+2-i:n)
        c(n)=c(n)-x(i)
    end do
    a(:,:)=outerdiff(x,x)                     Make vector w_j = ∏_{n≠j}(x_j − x_n).
    vander(:)=product(a,dim=2,mask=(a /= 0.0))
```

Now do synthetic division by $x - x_j$. The division for all x_j can be done in parallel (on a parallel machine), since the : in the loop below is over j.

```
    a(:,1)=-c(1)/x(:)
    do i=2,n
        a(:,i)=-(c(i)-a(:,i-1))/x(:)
    end do
    vander(:)=matmul(a,q)/vander(:)           Solve linear system and supply denomina-
end if                                            tor.
END FUNCTION vander
```

f90 `a=outerdiff...w=product...` Here is an example of the coding of equation (22.1.4). Since in this case the product is over the second index (n in $x_j - x_n$), we have `dim=2` in the `product`.

```
FUNCTION toeplz(r,y)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: r,y
REAL(SP), DIMENSION(size(y)) :: toeplz
```

Solves the Toeplitz system $\sum_{j=1}^{N} R_{(N+i-j)} x_j = y_i$ $(i = 1, \ldots, N)$. The Toeplitz matrix need not be symmetric. `y` (of length N) and `r` (of length $2N-1$) are input arrays; the solution x (of length N) is returned in `toeplz`.

```
INTEGER(I4B) :: m,m1,n,ndum
REAL(SP) :: sd,sgd,sgn,shn,sxn
REAL(SP), DIMENSION(size(y)) :: g,h,t
n=size(y)
ndum=assert_eq(2*n-1,size(r),'toeplz: ndum')
if (r(n) == 0.0) call nrerror('toeplz: initial singular minor')
toeplz(1)=y(1)/r(n)                          Initialize for the recursion.
if (n == 1) RETURN
g(1)=r(n-1)/r(n)
h(1)=r(n+1)/r(n)
do m=1,n                                     Main loop over the recursion.
    m1=m+1
    sxn=-y(m1)+dot_product(r(n+1:n+m),toeplz(m:1:-1))
      Compute numerator and denominator for x,
    sd=-r(n)+dot_product(r(n+1:n+m),g(1:m))
    if (sd == 0.0) exit
    toeplz(m1)=sxn/sd                        whence x.
    toeplz(1:m)=toeplz(1:m)-toeplz(m1)*g(m:1:-1)
    if (m1 == n) RETURN
    sgn=-r(n-m1)+dot_product(r(n-m:n-1),g(1:m))      Compute numerator and denom-
    shn=-r(n+m1)+dot_product(r(n+m:n+1:-1),h(1:m))        inator for G and H,
    sgd=-r(n)+dot_product(r(n-m:n-1),h(m:1:-1))
    if (sd == 0.0 .or. sgd == 0.0) exit
    g(m1)=sgn/sgd                            whence G and H.
    h(m1)=shn/sd
    t(1:m)=g(1:m)
    g(1:m)=g(1:m)-g(m1)*h(m:1:-1)
    h(1:m)=h(1:m)-h(m1)*t(m:1:-1)
end do                                       Back for another recurrence.
if (m > n) call nrerror('toeplz: sanity check failed in routine')
call nrerror('toeplz: singular principal minor')
END FUNCTION toeplz
```

⋆ ⋆ ⋆

```
SUBROUTINE choldc(a,p)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: p
```

Given an $N \times N$ positive-definite symmetric matrix `a`, this routine constructs its Cholesky decomposition, $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T$. On input, only the upper triangle of `a` need be given; it is not modified. The Cholesky factor $\mathbf{L}$ is returned in the lower triangle of `a`, except for its diagonal elements, which are returned in `p`, a vector of length N.

```
INTEGER(I4B) :: i,n
REAL(SP) :: summ
n=assert_eq(size(a,1),size(a,2),size(p),'choldc')
do i=1,n
```

```
    summm=a(i,i)-dot_product(a(i,1:i-1),a(i,1:i-1))
    if (summ <= 0.0) call nrerror('choldc failed')      a, with rounding errors, is
    p(i)=sqrt(summ)                                      not positive definite.
    a(i+1:n,i)=(a(i,i+1:n)-matmul(a(i+1:n,1:i-1),a(i,1:i-1)))/p(i)
end do
END SUBROUTINE choldc
```

```
SUBROUTINE cholsl(a,p,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: p,b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
```

Solves the set of N linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where a is a positive-definite symmetric matrix. a ($N \times N$) and p (of length N) are input as the output of the routine choldc. Only the lower triangle of a is accessed. b is the input right-hand-side vector, of length N. The solution vector, also of length N, is returned in x. a and p are not modified and can be left in place for successive calls with different right-hand sides b. b is not modified unless you identify b and x in the calling sequence, which is allowed.

```
INTEGER(I4B) :: i,n
n=assert_eq((/size(a,1),size(a,2),size(p),size(b),size(x)/),'cholsl')
do i=1,n                        Solve L · y = b, storing y in x.
    x(i)=(b(i)-dot_product(a(i,1:i-1),x(1:i-1)))/p(i)
end do
do i=n,1,-1                     Solve L^T · x = y.
    x(i)=(x(i)-dot_product(a(i+1:n,i),x(i+1:n)))/p(i)
end do
END SUBROUTINE cholsl
```

⋆ ⋆ ⋆

```
SUBROUTINE qrdcmp(a,c,d,sing)
USE nrtype; USE nrutil, ONLY : assert_eq,outerprod,vabs
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: c,d
LOGICAL(LGT), INTENT(OUT) :: sing
```

Constructs the QR decomposition of the $n \times n$ matrix a. The upper triangular matrix $\mathbf{R}$ is returned in the upper triangle of a, except for the diagonal elements of $\mathbf{R}$, which are returned in the n-dimensional vector d. The orthogonal matrix $\mathbf{Q}$ is represented as a product of $n-1$ Householder matrices $\mathbf{Q}_1 \ldots \mathbf{Q}_{n-1}$, where $\mathbf{Q}_j = \mathbf{1} - \mathbf{u}_j \otimes \mathbf{u}_j / c_j$. The ith component of $\mathbf{u}_j$ is zero for $i = 1, \ldots, j-1$ while the nonzero components are returned in a(i,j) for $i = j, \ldots, n$. sing returns as true if singularity is encountered during the decomposition, but the decomposition is still completed in this case.

```
INTEGER(I4B) :: k,n
REAL(SP) :: scale,sigma
n=assert_eq(size(a,1),size(a,2),size(c),size(d),'qrdcmp')
sing=.false.
do k=1,n-1
    scale=maxval(abs(a(k:n,k)))
    if (scale == 0.0) then          Singular case.
        sing=.true.
        c(k)=0.0
        d(k)=0.0
    else                            Form Q_k and Q_k · A.
        a(k:n,k)=a(k:n,k)/scale
        sigma=sign(vabs(a(k:n,k)),a(k,k))
        a(k,k)=a(k,k)+sigma
        c(k)=sigma*a(k,k)
```

```
        d(k)=-scale*sigma
        a(k:n,k+1:n)=a(k:n,k+1:n)-outerprod(a(k:n,k),&
            matmul(a(k:n,k),a(k:n,k+1:n)))/c(k)
    end if
end do
d(n)=a(n,n)
if (d(n) == 0.0) sing=.true.
END SUBROUTINE qrdcmp
```

f90 `a(k:n,k+1:n)=a(k:n,k+1:n)-outerprod...matmul...` See discussion of equation (22.1.6).

```
SUBROUTINE qrsolv(a,c,d,b)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : rsolv
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: c,d
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
```

Solves the set of n linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. The $n \times n$ matrix a and the n-dimensional vectors c and d are input as the output of the routine qrdcmp and are not modified. b is input as the right-hand-side vector of length n, and is overwritten with the solution vector on output.

```
INTEGER(I4B) :: j,n
REAL(SP) :: tau
n=assert_eq((/size(a,1),size(a,2),size(b),size(c),size(d)/),'qrsolv')
do j=1,n-1                    Form Q^T · b.
    tau=dot_product(a(j:n,j),b(j:n))/c(j)
    b(j:n)=b(j:n)-tau*a(j:n,j)
end do
call rsolv(a,d,b)             Solve R · x = Q^T · b.
END SUBROUTINE qrsolv
```

```
SUBROUTINE rsolv(a,d,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
```

Solves the set of n linear equations $\mathbf{R} \cdot \mathbf{x} = \mathbf{b}$, where $\mathbf{R}$ is an upper triangular matrix stored in a and d. The $n \times n$ matrix a and the vector d of length n are input as the output of the routine qrdcmp and are not modified. b is input as the right-hand-side vector of length n, and is overwritten with the solution vector on output.

```
INTEGER(I4B) :: i,n
n=assert_eq(size(a,1),size(a,2),size(b),size(d),'rsolv')
b(n)=b(n)/d(n)
do i=n-1,1,-1
    b(i)=(b(i)-dot_product(a(i,i+1:n),b(i+1:n)))/d(i)
end do
END SUBROUTINE rsolv
```

⋆ ⋆ ⋆

```
SUBROUTINE qrupdt(r,qt,u,v)
USE nrtype; USE nrutil, ONLY : assert_eq,ifirstloc
USE nr, ONLY : rotate,pythag
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: r,qt
REAL(SP), DIMENSION(:), INTENT(INOUT) :: u
REAL(SP), DIMENSION(:), INTENT(IN) :: v
```

Given the QR decomposition of some $n \times n$ matrix, calculates the QR decomposition of the matrix $\mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v})$. Here r and qt are $n \times n$ matrices, u and v are n-dimensional vectors. Note that $\mathbf{Q}^T$ is input and returned in qt.

```
INTEGER(I4B) :: i,k,n
n=assert_eq((/size(r,1),size(r,2),size(qt,1),size(qt,2),size(u),&
    size(v)/),'qrupdt')
k=n+1-ifirstloc(u(n:1:-1) /= 0.0)          Find largest k such that u(k) ≠ 0.
if (k < 1) k=1
do i=k-1,1,-1                              Transform R + u ⊗ v to upper Hessenberg.
    call rotate(r,qt,i,u(i),-u(i+1))
    u(i)=pythag(u(i),u(i+1))
end do
r(1,:)=r(1,:)+u(1)*v
do i=1,k-1                                 Transform upper Hessenberg matrix to upper
    call rotate(r,qt,i,r(i,i),-r(i+1,i))       triangular.
end do
END SUBROUTINE qrupdt
```

f90 `k=n+1-ifirstloc(u(n:1:-1) /= 0.0)` The function `ifirstloc` in `nrutil` returns the first occurrence of `.true.` in a logical vector. See the discussion of the analogous routine `imaxloc` on p. 1017.

```
SUBROUTINE rotate(r,qt,i,a,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), TARGET, INTENT(INOUT) :: r,qt
INTEGER(I4B), INTENT(IN) :: i
REAL(SP), INTENT(IN) :: a,b
```

Given $n \times n$ matrices r and qt, carry out a Jacobi rotation on rows i and i$+1$ of each matrix. a and b are the parameters of the rotation: $\cos\theta = a/\sqrt{a^2+b^2}$, $\sin\theta = b/\sqrt{a^2+b^2}$.

```
REAL(SP), DIMENSION(size(r,1)) :: temp
INTEGER(I4B) :: n
REAL(SP) :: c,fact,s
n=assert_eq(size(r,1),size(r,2),size(qt,1),size(qt,2),'rotate')
if (a == 0.0) then          Avoid unnecessary overflow or underflow.
    c=0.0
    s=sign(1.0_sp,b)
else if (abs(a) > abs(b)) then
    fact=b/a
    c=sign(1.0_sp/sqrt(1.0_sp+fact**2),a)
    s=fact*c
else
    fact=a/b
    s=sign(1.0_sp/sqrt(1.0_sp+fact**2),b)
    c=fact*s
end if
temp(i:n)=r(i,i:n)          Premultiply r by Jacobi rotation.
r(i,i:n)=c*temp(i:n)-s*r(i+1,i:n)
r(i+1,i:n)=s*temp(i:n)+c*r(i+1,i:n)
temp=qt(i,:)                Premultiply qt by Jacobi rotation.
qt(i,:)=c*temp-s*qt(i+1,:)
qt(i+1,:)=s*temp+c*qt(i+1,:)
END SUBROUTINE rotate
```

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press). [1]

Gu, M., Demmel, J., and Dhillon, I. 1994, LAPACK Working Note #88 (Computer Science Department, University of Tennessee at Knoxville, Preprint UT-CS-94-257; available from Netlib, or as http://www.cs.utk.edu/∼library/TechReports/1994/ut-cs-94-257.ps.Z). [2] See also discussion after `tqli` in Chapter B11.