C1. Listing of Utility Modules (nrtype and nrutil)

C1.1 Numerical Recipes Types (nrtype)

The file supplied as `nrtype.f90` contains a single module named `nrtype`, which in turn contains definitions for a number of named constants (that is, `PARAMETER`s), and a couple of elementary derived data types used by the sparse matrix routines in this book. Of the named constants, by far the most important are those that define the `KIND` types of virtually all the variables used in this book: `I4B`, `I2B`, and `I1B` for integer variables, `SP` and `DP` for real variables (and `SPC` and `DPC` for the corresponding complex cases), and `LGT` for the default logical type.

```
MODULE nrtype
    Symbolic names for kind types of 4-, 2-, and 1-byte integers:
INTEGER, PARAMETER :: I4B = SELECTED_INT_KIND(9)
INTEGER, PARAMETER :: I2B = SELECTED_INT_KIND(4)
INTEGER, PARAMETER :: I1B = SELECTED_INT_KIND(2)
    Symbolic names for kind types of single- and double-precision reals:
INTEGER, PARAMETER :: SP = KIND(1.0)
INTEGER, PARAMETER :: DP = KIND(1.0D0)
    Symbolic names for kind types of single- and double-precision complex:
INTEGER, PARAMETER :: SPC = KIND((1.0,1.0))
INTEGER, PARAMETER :: DPC = KIND((1.0D0,1.0D0))
    Symbolic name for kind type of default logical:
INTEGER, PARAMETER :: LGT = KIND(.true.)
    Frequently used mathematical constants (with precision to spare):
REAL(SP), PARAMETER :: PI=3.141592653589793238462643383279502884197_sp
REAL(SP), PARAMETER :: PIO2=1.57079632679489661923132169163975144209858_sp
REAL(SP), PARAMETER :: TWOPI=6.283185307179586476925286766559005768394_sp
REAL(SP), PARAMETER :: SQRT2=1.41421356237309504880168872420969807856967_sp
REAL(SP), PARAMETER :: EULER=0.5772156649015328606065120900824024310422_sp
REAL(DP), PARAMETER :: PI_D=3.141592653589793238462643383279502884197_dp
REAL(DP), PARAMETER :: PIO2_D=1.57079632679489661923132169163975144209858_dp
REAL(DP), PARAMETER :: TWOPI_D=6.283185307179586476925286766559005768394_dp
    Derived data types for sparse matrices, single and double precision (see use in Chapter B2):
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
TYPE sprs2_dp
    INTEGER(I4B) :: n,len
    REAL(DP), DIMENSION(:), POINTER :: val
```

```
    INTEGER(I4B), DIMENSION(:), POINTER :: irow
    INTEGER(I4B), DIMENSION(:), POINTER :: jcol
END TYPE sprs2_dp
END MODULE nrtype
```

About Converting to Higher Precision

You might hope that changing all the Numerical Recipes routines from single precision to double precision would be as simple as redefining the values of SP and DP in nrtype. Well ... not quite.

Converting algorithms to a higher precision is not a purely mechanical task because of the distinction between "roundoff error" and "truncation error." (Please see Volume 1, §1.2, if you are not familiar with these concepts.) While increasing the precision implied by the kind values SP and DP will indeed reduce a routine's roundoff error, it will not reduce any truncation error that may be intrinsic to the algorithm. Sometimes, a routine contains "accuracy parameters" that can be adjusted to reduce the truncation error to the new, desired level. In other cases, however, the truncation error cannot be so easily reduced; then, a whole new algorithm is needed. Clearly such new algorithms are beyond the scope of a simple mechanical "conversion."

If, despite these cautionary words, you want to proceed with converting some routines to a higher precision, here are some hints:

If your machine has a kind type that is distinct from, and has equal or greater precision than, the kind type that we use for DP, then, in nrtype, you can simply redefine DP to this new highest precision and redefine SP to what was previously DP. For example, DEC machines usually have a "quadruple precision" real type available, which can be used in this way. You should not need to make any further edits of nrtype or nrutil.

If, on the other hand, the kind type that we already use for DP is the highest precision available, then you must leave DP defined as it is, and redefine SP in nrtype to be this same kind type. Now, however, you will also have to edit nrutil, because some overloaded routines that were previously distinguishable (by the different kind types) will now be seen by the compiler as indistinguishable — and it will object strenuously. Simply delete all the "_dp" function names from the list of overloaded procedures (i.e., from the MODULE PROCEDURE statements). Note that it is not necessary to delete the routines from the MODULE itself. Similarly, in the interface file nr.f90 you must delete the "_dp" interfaces, *except* for the sprs... routines. (Since they have TYPE(sprs2_dp) or TYPE(sprs2_sp), they are treated as distinct even though they have functionally equivalent kind types.)

Finally, the following table gives some suggestions for changing the accuracy parameters, or constants, in some of the routines. Please note that this table is not necessarily complete, and that higher-precision performance is not guaranteed for all the routines, *even if* you make all the changes indicated. The above edits, and these suggestions, do, however, work in the majority of cases.

In routine...	*change...*	*to...*
beschb	NUSE1=5,NUSE2=5	NUSE1=7,NUSE2=8
bessi	IACC=40	IACC=200
bessik	EPS=1.0e-10_dp	EPS=epsilon(x)
bessj	IACC=40	IACC=160
bessjy	EPS=1.0e-10_dp	EPS=epsilon(x)
broydn	TOLF=1.0e-4_sp	TOLF=1.0e-8_sp
	TOLMIN=1.0e-6_sp	TOLMIN=1.0e-12_sp
fdjac	EPS=1.0e-4_sp	EPS=1.0e-8_sp
frprmn	EPS=1.0e-10_sp	EPS=1.0e-18_sp
gauher	EPS=3.0e-13_dp	EPS=1.0e-14_dp
gaujac	EPS=3.0e-14_dp	EPS=1.0e-14_dp
gaulag	EPS=3.0e-13_dp	EPS=1.0e-14_dp
gauleg	EPS=3.0e-14_dp	EPS=1.0e-14_dp
hypgeo	EPS=1.0e-6_sp	EPS=1.0e-14_sp
linmin	TOL=1.0e-4_sp	TOL=1.0e-8_sp
newt	TOLF=1.0e-4_sp	TOLF=1.0e-8_sp
	TOLMIN=1.0e-6_sp	TOLMIN=1.0e-12_sp
probks	EPS1=0.001_sp	EPS1=1.0e-6_sp
	EPS2=1.0e-8_sp	EPS2=1.0e-16_sp
qromb	EPS=1.0e-6_sp	EPS=1.0e-10_sp
qromo	EPS=1.0e-6_sp	EPS=1.0e-10_sp
qroot	TINY=1.0e-6_sp	TINY=1.0e-14_sp
qsimp	EPS=1.0e-6_sp	EPS=1.0e-10_sp
qtrap	EPS=1.0e-6_sp	EPS=1.0e-10_sp
rc	ERRTOL=0.04_sp	ERRTOL=0.0012_sp
rd	ERRTOL=0.05_sp	ERRTOL=0.0015_sp
rf	ERRTOL=0.08_sp	ERRTOL=0.0025_sp
rj	ERRTOL=0.05_sp	ERRTOL=0.0015_sp
sfroid	conv=5.0e-6_sp	conv=1.0e-14_sp
shoot	EPS=1.0e-6_sp	EPS=1.0e-14_sp
shootf	EPS=1.0e-6_sp	EPS=1.0e-14_sp
simplx	EPS=1.0e-6_sp	EPS=1.0e-14_sp
sncndn	CA=0.0003_sp	CA=1.0e-8_sp
sor	EPS=1.0e-5_dp	EPS=1.0e-13_dp
sphfpt	DXX=1.0e-4_sp	DXX=1.0e-8_sp
sphoot	dx=1.0e-4_sp	dx=1.0e-8_sp
svdfit	TOL=1.0e-5_sp	TOL=1.0e-13_sp
zroots	EPS=1.0e-6_sp	EPS=1.0e-14_sp

C1.2 Numerical Recipes Utilities (nrutil)

The file supplied as `nrutil.f90` contains a single module named `nrutil`, which contains specific implementations for all the Numerical Recipes utility functions described in detail in Chapter 23.

The specific implementations given are something of a compromise between demonstrating parallel techniques (when they can be achieved in Fortran 90) and running efficiently on conventional, serial machines. The parameters at the beginning of the module (names beginning with `NPAR_`) are typically related to array lengths *below which* the implementations revert to serial operations. On a purely serial machine, these can be set to large values to suppress many parallel constructions.

The length and repetitiveness of the `nrutil.f90` file stems in large part from its extensive use of overloading. Indeed, the file would be even longer if we overloaded versions for all the applicable data types that each utility could, in principle, instantiate. The descriptions in Chapter 23 detail both the full set of intended data types and shapes for each routine, and also the types and shapes actually here implemented (which can also be gleaned by examining the file). The intended result of all this overloading is, in essence, to give the utility routines the desirable properties of many of the Fortran 90 intrinsic functions, namely, to be both *generic* (apply to many data types) and *elemental* (apply element-by-element to arbitrary shapes). Fortran 95's provision of user-defined elemental functions will reduce the multiplicity of overloading in some of our routines; unfortunately the necessity to overload for multiple data types will still be present.

Finally, it is worth reemphasizing the following point, already made in Chapter 23: The purpose of the `nrutil` utilities is to remove from the Numerical Recipes programs just those programming tasks and "idioms" whose efficient implementation is *most* hardware and compiler dependent, so as to allow for specific, efficient implementations on different machines. One should therefore not expect the utmost in efficiency from the general purpose, one-size-fits-all, implementation listed here.

Correspondingly, we would encourage the incorporation of efficient `nrutil` implementations, and/or comparable capabilities under different names, with as broad as possible a set of overloaded data types, in libraries associated with specific compilers or machines. In support of this goal, we have specifically put this Appendix C1, and the files `nrtype.f90` and `nrutil.f90`, into the public domain.

```
MODULE nrutil
    TABLE OF CONTENTS OF THE NRUTIL MODULE:
        routines that move data:
            array_copy, swap, reallocate
        routines returning a location as an integer value
            ifirstloc, imaxloc, iminloc
        routines for argument checking and error handling:
            assert, assert_eq, nrerror
        routines relating to polynomials and recurrences
            arth, geop, cumsum, cumprod, poly, polyterm,
            zroots_unity
        routines for "outer" operations on vectors
            outerand, outersum, outerdiff, outerprod, outerdiv
        routines for scatter-with-combine
            scatter_add, scatter_max
        routines for skew operations on matrices
            diagadd, diagmult, get_diag, put_diag,
```

unit_matrix, lower_triangle, upper_triangle
miscellaneous routines
vabs

```
USE nrtype
```

Parameters for crossover from serial to parallel algorithms (these are used only within this nrutil module):

```
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NPAR_ARTH=16,NPAR2_ARTH=8     Each NPAR2 must be ≤ the
INTEGER(I4B), PARAMETER :: NPAR_GEOP=4,NPAR2_GEOP=2      corresponding NPAR.
INTEGER(I4B), PARAMETER :: NPAR_CUMSUM=16
INTEGER(I4B), PARAMETER :: NPAR_CUMPROD=8
INTEGER(I4B), PARAMETER :: NPAR_POLY=8
INTEGER(I4B), PARAMETER :: NPAR_POLYTERM=8
```

Next, generic interfaces for routines with overloaded versions. Naming conventions for appended codes in the names of overloaded routines are as follows: r=real, d=double precision, i=integer, c=complex, z=double-precision complex, h=character, l=logical. Any of r,d,i,c,z,h,l may be followed by v=vector or m=matrix (v,m suffixes are used only when needed to resolve ambiguities).

Routines that move data:

```
INTERFACE array_copy
    MODULE PROCEDURE array_copy_r, array_copy_d, array_copy_i
END INTERFACE
INTERFACE swap
    MODULE PROCEDURE swap_i,swap_r,swap_rv,swap_c, &
        swap_cv,swap_cm,swap_z,swap_zv,swap_zm, &
        masked_swap_rs,masked_swap_rv,masked_swap_rm
END INTERFACE
INTERFACE reallocate
    MODULE PROCEDURE reallocate_rv,reallocate_rm,&
        reallocate_iv,reallocate_im,reallocate_hv
END INTERFACE
```

Routines returning a location as an integer value (ifirstloc, iminloc are not currently overloaded and so do not have a generic interface here):

```
INTERFACE imaxloc
    MODULE PROCEDURE imaxloc_r,imaxloc_i
END INTERFACE
```

Routines for argument checking and error handling (nrerror is not currently overloaded):

```
INTERFACE assert
    MODULE PROCEDURE assert1,assert2,assert3,assert4,assert_v
END INTERFACE
INTERFACE assert_eq
    MODULE PROCEDURE assert_eq2,assert_eq3,assert_eq4,assert_eqn
END INTERFACE
```

Routines relating to polynomials and recurrences (cumprod, zroots_unity are not currently overloaded):

```
INTERFACE arth
    MODULE PROCEDURE arth_r, arth_d, arth_i
END INTERFACE
INTERFACE geop
    MODULE PROCEDURE geop_r, geop_d, geop_i, geop_c, geop_dv
END INTERFACE
INTERFACE cumsum
    MODULE PROCEDURE cumsum_r,cumsum_i
END INTERFACE
INTERFACE poly
    MODULE PROCEDURE poly_rr,poly_rrv,poly_dd,poly_ddv,&
        poly_rc,poly_cc,poly_msk_rrv,poly_msk_ddv
END INTERFACE
INTERFACE poly_term
    MODULE PROCEDURE poly_term_rr,poly_term_cc
END INTERFACE
```

Routines for “outer” operations on vectors (outerand, outersum, outerdiv are not currently overloaded):

```
INTERFACE outerprod
```

```
    MODULE PROCEDURE outerprod_r,outerprod_d
END INTERFACE
INTERFACE outerdiff
    MODULE PROCEDURE outerdiff_r,outerdiff_d,outerdiff_i
END INTERFACE
```

Routines for scatter-with-combine, scatter_add, scatter_max:

```
INTERFACE scatter_add
    MODULE PROCEDURE scatter_add_r,scatter_add_d
END INTERFACE
INTERFACE scatter_max
    MODULE PROCEDURE scatter_max_r,scatter_max_d
END INTERFACE
```

Routines for skew operations on matrices (unit_matrix, lower_triangle, upper_triangle not currently overloaded):

```
INTERFACE diagadd
    MODULE PROCEDURE diagadd_rv,diagadd_r
END INTERFACE
INTERFACE diagmult
    MODULE PROCEDURE diagmult_rv,diagmult_r
END INTERFACE
INTERFACE get_diag
    MODULE PROCEDURE get_diag_rv, get_diag_dv
END INTERFACE
INTERFACE put_diag
    MODULE PROCEDURE put_diag_rv, put_diag_r
END INTERFACE
```

Other routines (vabs is not currently overloaded):

CONTAINS

Routines that move data:

```
SUBROUTINE array_copy_r(src,dest,n_copied,n_not_copied)
```

Copy array where size of source not known in advance.

```
REAL(SP), DIMENSION(:), INTENT(IN) :: src
REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
n_copied=min(size(src),size(dest))
n_not_copied=size(src)-n_copied
dest(1:n_copied)=src(1:n_copied)
END SUBROUTINE array_copy_r

SUBROUTINE array_copy_d(src,dest,n_copied,n_not_copied)
REAL(DP), DIMENSION(:), INTENT(IN) :: src
REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
n_copied=min(size(src),size(dest))
n_not_copied=size(src)-n_copied
dest(1:n_copied)=src(1:n_copied)
END SUBROUTINE array_copy_d

SUBROUTINE array_copy_i(src,dest,n_copied,n_not_copied)
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: src
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: dest
INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
n_copied=min(size(src),size(dest))
n_not_copied=size(src)-n_copied
dest(1:n_copied)=src(1:n_copied)
END SUBROUTINE array_copy_i

SUBROUTINE swap_i(a,b)
```

Swap the contents of a and b.

```
INTEGER(I4B), INTENT(INOUT) :: a,b
INTEGER(I4B) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_i
```

```
SUBROUTINE swap_r(a,b)
REAL(SP), INTENT(INOUT) :: a,b
REAL(SP) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_r

SUBROUTINE swap_rv(a,b)
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
REAL(SP), DIMENSION(SIZE(a)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_rv

SUBROUTINE swap_c(a,b)
COMPLEX(SPC), INTENT(INOUT) :: a,b
COMPLEX(SPC) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_c

SUBROUTINE swap_cv(a,b)
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: a,b
COMPLEX(SPC), DIMENSION(SIZE(a)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_cv

SUBROUTINE swap_cm(a,b)
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: a,b
COMPLEX(SPC), DIMENSION(size(a,1),size(a,2)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_cm

SUBROUTINE swap_z(a,b)
COMPLEX(DPC), INTENT(INOUT) :: a,b
COMPLEX(DPC) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_z

SUBROUTINE swap_zv(a,b)
COMPLEX(DPC), DIMENSION(:), INTENT(INOUT) :: a,b
COMPLEX(DPC), DIMENSION(SIZE(a)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_zv

SUBROUTINE swap_zm(a,b)
COMPLEX(DPC), DIMENSION(:,:), INTENT(INOUT) :: a,b
COMPLEX(DPC), DIMENSION(size(a,1),size(a,2)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_zm

SUBROUTINE masked_swap_rs(a,b,mask)
REAL(SP), INTENT(INOUT) :: a,b
LOGICAL(LGT), INTENT(IN) :: mask
REAL(SP) :: swp
```

```
if (mask) then
    swp=a
    a=b
    b=swp
end if
END SUBROUTINE masked_swap_rs

SUBROUTINE masked_swap_rv(a,b,mask)
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(a)) :: swp
where (mask)
    swp=a
    a=b
    b=swp
end where
END SUBROUTINE masked_swap_rv

SUBROUTINE masked_swap_rm(a,b,mask)
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a,b
LOGICAL(LGT), DIMENSION(:,:), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(a,1),size(a,2)) :: swp
where (mask)
    swp=a
    a=b
    b=swp
end where
END SUBROUTINE masked_swap_rm

FUNCTION reallocate_rv(p,n)
    Reallocate a pointer to a new size, preserving its previous contents.
REAL(SP), DIMENSION(:), POINTER :: p, reallocate_rv
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B) :: nold,ierr
allocate(reallocate_rv(n),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_rv: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p)
reallocate_rv(1:min(nold,n))=p(1:min(nold,n))
deallocate(p)
END FUNCTION reallocate_rv

FUNCTION reallocate_iv(p,n)
INTEGER(I4B), DIMENSION(:), POINTER :: p, reallocate_iv
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B) :: nold,ierr
allocate(reallocate_iv(n),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_iv: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p)
reallocate_iv(1:min(nold,n))=p(1:min(nold,n))
deallocate(p)
END FUNCTION reallocate_iv

FUNCTION reallocate_hv(p,n)
CHARACTER(1), DIMENSION(:), POINTER :: p, reallocate_hv
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B) :: nold,ierr
allocate(reallocate_hv(n),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_hv: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p)
reallocate_hv(1:min(nold,n))=p(1:min(nold,n))
```

```
deallocate(p)
END FUNCTION reallocate_hv

FUNCTION reallocate_rm(p,n,m)
REAL(SP), DIMENSION(:,:), POINTER :: p, reallocate_rm
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B) :: nold,mold,ierr
allocate(reallocate_rm(n,m),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_rm: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p,1)
mold=size(p,2)
reallocate_rm(1:min(nold,n),1:min(mold,m))=&
    p(1:min(nold,n),1:min(mold,m))
deallocate(p)
END FUNCTION reallocate_rm

FUNCTION reallocate_im(p,n,m)
INTEGER(I4B), DIMENSION(:,:), POINTER :: p, reallocate_im
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B) :: nold,mold,ierr
allocate(reallocate_im(n,m),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_im: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p,1)
mold=size(p,2)
reallocate_im(1:min(nold,n),1:min(mold,m))=&
    p(1:min(nold,n),1:min(mold,m))
deallocate(p)
END FUNCTION reallocate_im
```

Routines returning a location as an integer value:

```
FUNCTION ifirstloc(mask)
```

Index of first occurrence of .true. in a logical vector.

```
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
INTEGER(I4B) :: ifirstloc
INTEGER(I4B), DIMENSION(1) :: loc
loc=maxloc(merge(1,0,mask))
ifirstloc=loc(1)
if (.not. mask(ifirstloc)) ifirstloc=size(mask)+1
END FUNCTION ifirstloc

FUNCTION imaxloc_r(arr)
```

Index of maxloc on an array.

```
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B) :: imaxloc_r
INTEGER(I4B), DIMENSION(1) :: imax
imax=maxloc(arr(:))
imaxloc_r=imax(1)
END FUNCTION imaxloc_r

FUNCTION imaxloc_i(iarr)
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iarr
INTEGER(I4B), DIMENSION(1) :: imax
INTEGER(I4B) :: imaxloc_i
imax=maxloc(iarr(:))
imaxloc_i=imax(1)
END FUNCTION imaxloc_i

FUNCTION iminloc(arr)
```

Index of minloc on an array.

```
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(1) :: imin
INTEGER(I4B) :: iminloc
imin=minloc(arr(:))
```

```
iminloc=imin(1)
END FUNCTION iminloc
```

Routines for argument checking and error handling:

```
SUBROUTINE assert1(n1,string)
```

Report and die if any logical is false (used for arg range checking).

```
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1
if (.not. n1) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert1'
end if
END SUBROUTINE assert1

SUBROUTINE assert2(n1,n2,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2
if (.not. (n1 .and. n2)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert2'
end if
END SUBROUTINE assert2

SUBROUTINE assert3(n1,n2,n3,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2,n3
if (.not. (n1 .and. n2 .and. n3)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert3'
end if
END SUBROUTINE assert3

SUBROUTINE assert4(n1,n2,n3,n4,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2,n3,n4
if (.not. (n1 .and. n2 .and. n3 .and. n4)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert4'
end if
END SUBROUTINE assert4

SUBROUTINE assert_v(n,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, DIMENSION(:), INTENT(IN) :: n
if (.not. all(n)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert_v'
end if
END SUBROUTINE assert_v

FUNCTION assert_eq2(n1,n2,string)
```

Report and die if integers not all equal (used for size checking).

```
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2
INTEGER :: assert_eq2
if (n1 == n2) then
    assert_eq2=n1
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
    STOP 'program terminated by assert_eq2'
end if
```

```
END FUNCTION assert_eq2

FUNCTION assert_eq3(n1,n2,n3,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2,n3
INTEGER :: assert_eq3
if (n1 == n2 .and. n2 == n3) then
    assert_eq3=n1
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
    STOP 'program terminated by assert_eq3'
end if
END FUNCTION assert_eq3

FUNCTION assert_eq4(n1,n2,n3,n4,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2,n3,n4
INTEGER :: assert_eq4
if (n1 == n2 .and. n2 == n3 .and. n3 == n4) then
    assert_eq4=n1
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
    STOP 'program terminated by assert_eq4'
end if
END FUNCTION assert_eq4

FUNCTION assert_eqn(nn,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, DIMENSION(:), INTENT(IN) :: nn
INTEGER :: assert_eqn
if (all(nn(2:) == nn(1))) then
    assert_eqn=nn(1)
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
    STOP 'program terminated by assert_eqn'
end if
END FUNCTION assert_eqn

SUBROUTINE nrerror(string)
   Report a message, then die.
CHARACTER(LEN=*), INTENT(IN) :: string
write (*,*) 'nrerror: ',string
STOP 'program terminated by nrerror'
END SUBROUTINE nrerror
```

Routines relating to polynomials and recurrences:

```
FUNCTION arth_r(first,increment,n)
   Array function returning an arithmetic progression.
REAL(SP), INTENT(IN) :: first,increment
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: arth_r
INTEGER(I4B) :: k,k2
REAL(SP) :: temp
if (n > 0) arth_r(1)=first
if (n <= NPAR_ARTH) then
    do k=2,n
        arth_r(k)=arth_r(k-1)+increment
    end do
else
    do k=2,NPAR2_ARTH
        arth_r(k)=arth_r(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
```

```
    do
        if (k >= n) exit
        k2=k+k
        arth_r(k+1:min(k2,n))=temp+arth_r(1:min(k,n-k))
        temp=temp+temp
        k=k2
    end do
end if
END FUNCTION arth_r

FUNCTION arth_d(first,increment,n)
REAL(DP), INTENT(IN) :: first,increment
INTEGER(I4B), INTENT(IN) :: n
REAL(DP), DIMENSION(n) :: arth_d
INTEGER(I4B) :: k,k2
REAL(DP) :: temp
if (n > 0) arth_d(1)=first
if (n <= NPAR_ARTH) then
    do k=2,n
        arth_d(k)=arth_d(k-1)+increment
    end do
else
    do k=2,NPAR2_ARTH
        arth_d(k)=arth_d(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
    do
        if (k >= n) exit
        k2=k+k
        arth_d(k+1:min(k2,n))=temp+arth_d(1:min(k,n-k))
        temp=temp+temp
        k=k2
    end do
end if
END FUNCTION arth_d

FUNCTION arth_i(first,increment,n)
INTEGER(I4B), INTENT(IN) :: first,increment,n
INTEGER(I4B), DIMENSION(n) :: arth_i
INTEGER(I4B) :: k,k2,temp
if (n > 0) arth_i(1)=first
if (n <= NPAR_ARTH) then
    do k=2,n
        arth_i(k)=arth_i(k-1)+increment
    end do
else
    do k=2,NPAR2_ARTH
        arth_i(k)=arth_i(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
    do
        if (k >= n) exit
        k2=k+k
        arth_i(k+1:min(k2,n))=temp+arth_i(1:min(k,n-k))
        temp=temp+temp
        k=k2
    end do
end if
END FUNCTION arth_i

FUNCTION geop_r(first,factor,n)
   Array function returning a geometric progression.
REAL(SP), INTENT(IN) :: first,factor
```

```
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: geop_r
INTEGER(I4B) :: k,k2
REAL(SP) :: temp
if (n > 0) geop_r(1)=first
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_r(k)=geop_r(k-1)*factor
    end do
else
    do k=2,NPAR2_GEOP
        geop_r(k)=geop_r(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_r(k+1:min(k2,n))=temp*geop_r(1:min(k,n-k))
        temp=temp*temp
        k=k2
    end do
end if
END FUNCTION geop_r

FUNCTION geop_d(first,factor,n)
REAL(DP), INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
REAL(DP), DIMENSION(n) :: geop_d
INTEGER(I4B) :: k,k2
REAL(DP) :: temp
if (n > 0) geop_d(1)=first
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_d(k)=geop_d(k-1)*factor
    end do
else
    do k=2,NPAR2_GEOP
        geop_d(k)=geop_d(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_d(k+1:min(k2,n))=temp*geop_d(1:min(k,n-k))
        temp=temp*temp
        k=k2
    end do
end if
END FUNCTION geop_d

FUNCTION geop_i(first,factor,n)
INTEGER(I4B), INTENT(IN) :: first,factor,n
INTEGER(I4B), DIMENSION(n) :: geop_i
INTEGER(I4B) :: k,k2,temp
if (n > 0) geop_i(1)=first
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_i(k)=geop_i(k-1)*factor
    end do
else
    do k=2,NPAR2_GEOP
        geop_i(k)=geop_i(k-1)*factor
    end do
```

```
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_i(k+1:min(k2,n))=temp*geop_i(1:min(k,n-k))
        temp=temp*temp
        k=k2
    end do
end if
END FUNCTION geop_i

FUNCTION geop_c(first,factor,n)
COMPLEX(SP), INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
COMPLEX(SP), DIMENSION(n) :: geop_c
INTEGER(I4B) :: k,k2
COMPLEX(SP) :: temp
if (n > 0) geop_c(1)=first
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_c(k)=geop_c(k-1)*factor
    end do
else
    do k=2,NPAR2_GEOP
        geop_c(k)=geop_c(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_c(k+1:min(k2,n))=temp*geop_c(1:min(k,n-k))
        temp=temp*temp
        k=k2
    end do
end if
END FUNCTION geop_c

FUNCTION geop_dv(first,factor,n)
REAL(DP), DIMENSION(:), INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
REAL(DP), DIMENSION(size(first),n) :: geop_dv
INTEGER(I4B) :: k,k2
REAL(DP), DIMENSION(size(first)) :: temp
if (n > 0) geop_dv(:,1)=first(:)
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_dv(:,k)=geop_dv(:,k-1)*factor(:)
    end do
else
    do k=2,NPAR2_GEOP
        geop_dv(:,k)=geop_dv(:,k-1)*factor(:)
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_dv(:,k+1:min(k2,n))=geop_dv(:,1:min(k,n-k))*&
            spread(temp,2,size(geop_dv(:,1:min(k,n-k)),2))
        temp=temp*temp
        k=k2
    end do
end if
```

```
END FUNCTION geop_dv

RECURSIVE FUNCTION cumsum_r(arr,seed) RESULT(ans)
   Cumulative sum on an array, with optional additive seed.
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP), OPTIONAL, INTENT(IN) :: seed
REAL(SP), DIMENSION(size(arr)) :: ans
INTEGER(I4B) :: n,j
REAL(SP) :: sd
n=size(arr)
if (n == 0_i4b) RETURN
sd=0.0_sp
if (present(seed)) sd=seed
ans(1)=arr(1)+sd
if (n < NPAR_CUMSUM) then
    do j=2,n
        ans(j)=ans(j-1)+arr(j)
    end do
else
    ans(2:n:2)=cumsum_r(arr(2:n:2)+arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)+arr(3:n:2)
end if
END FUNCTION cumsum_r

RECURSIVE FUNCTION cumsum_i(arr,seed) RESULT(ans)
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), OPTIONAL, INTENT(IN) :: seed
INTEGER(I4B), DIMENSION(size(arr)) :: ans
INTEGER(I4B) :: n,j,sd
n=size(arr)
if (n == 0_i4b) RETURN
sd=0_i4b
if (present(seed)) sd=seed
ans(1)=arr(1)+sd
if (n < NPAR_CUMSUM) then
    do j=2,n
        ans(j)=ans(j-1)+arr(j)
    end do
else
    ans(2:n:2)=cumsum_i(arr(2:n:2)+arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)+arr(3:n:2)
end if
END FUNCTION cumsum_i

RECURSIVE FUNCTION cumprod(arr,seed) RESULT(ans)
   Cumulative product on an array, with optional multiplicative seed.
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP), OPTIONAL, INTENT(IN) :: seed
REAL(SP), DIMENSION(size(arr)) :: ans
INTEGER(I4B) :: n,j
REAL(SP) :: sd
n=size(arr)
if (n == 0_i4b) RETURN
sd=1.0_sp
if (present(seed)) sd=seed
ans(1)=arr(1)*sd
if (n < NPAR_CUMPROD) then
    do j=2,n
        ans(j)=ans(j-1)*arr(j)
    end do
else
    ans(2:n:2)=cumprod(arr(2:n:2)*arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)*arr(3:n:2)
end if
END FUNCTION cumprod
```

```
FUNCTION poly_rr(x,coeffs)
   Polynomial evaluation.
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs
REAL(SP) :: poly_rr
REAL(SP) :: pow
REAL(SP), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
    poly_rr=0.0_sp
else if (n < NPAR_POLY) then
    poly_rr=coeffs(n)
    do i=n-1,1,-1
        poly_rr=x*poly_rr+coeffs(i)
    end do
else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
        vec(n+1)=0.0_sp
        nn=ishft(n+1,-1)
        vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
        if (nn == 1) exit
        pow=pow*pow
        n=nn
    end do
    poly_rr=vec(1)
    deallocate(vec)
end if
END FUNCTION poly_rr

FUNCTION poly_dd(x,coeffs)
REAL(DP), INTENT(IN) :: x
REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs
REAL(DP) :: poly_dd
REAL(DP) :: pow
REAL(DP), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
    poly_dd=0.0_dp
else if (n < NPAR_POLY) then
    poly_dd=coeffs(n)
    do i=n-1,1,-1
        poly_dd=x*poly_dd+coeffs(i)
    end do
else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
        vec(n+1)=0.0_dp
        nn=ishft(n+1,-1)
        vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
        if (nn == 1) exit
        pow=pow*pow
        n=nn
    end do
    poly_dd=vec(1)
    deallocate(vec)
end if
END FUNCTION poly_dd
```

```
FUNCTION poly_rc(x,coeffs)
COMPLEX(SPC), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs
COMPLEX(SPC) :: poly_rc
COMPLEX(SPC) :: pow
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
    poly_rc=0.0_sp
else if (n < NPAR_POLY) then
    poly_rc=coeffs(n)
    do i=n-1,1,-1
        poly_rc=x*poly_rc+coeffs(i)
    end do
else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
        vec(n+1)=0.0_sp
        nn=ishft(n+1,-1)
        vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
        if (nn == 1) exit
        pow=pow*pow
        n=nn
    end do
    poly_rc=vec(1)
    deallocate(vec)
end if
END FUNCTION poly_rc

FUNCTION poly_cc(x,coeffs)
COMPLEX(SPC), INTENT(IN) :: x
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: coeffs
COMPLEX(SPC) :: poly_cc
COMPLEX(SPC) :: pow
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
    poly_cc=0.0_sp
else if (n < NPAR_POLY) then
    poly_cc=coeffs(n)
    do i=n-1,1,-1
        poly_cc=x*poly_cc+coeffs(i)
    end do
else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
        vec(n+1)=0.0_sp
        nn=ishft(n+1,-1)
        vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
        if (nn == 1) exit
        pow=pow*pow
        n=nn
    end do
    poly_cc=vec(1)
    deallocate(vec)
end if
END FUNCTION poly_cc

FUNCTION poly_rrv(x,coeffs)
```

```
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs,x
REAL(SP), DIMENSION(size(x)) :: poly_rrv
INTEGER(I4B) :: i,n,m
m=size(coeffs)
n=size(x)
if (m <= 0) then
    poly_rrv=0.0_sp
else if (m < n .or. m < NPAR_POLY) then
    poly_rrv=coeffs(m)
    do i=m-1,1,-1
        poly_rrv=x*poly_rrv+coeffs(i)
    end do
else
    do i=1,n
        poly_rrv(i)=poly_rr(x(i),coeffs)
    end do
end if
END FUNCTION poly_rrv

FUNCTION poly_ddv(x,coeffs)
REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs,x
REAL(DP), DIMENSION(size(x)) :: poly_ddv
INTEGER(I4B) :: i,n,m
m=size(coeffs)
n=size(x)
if (m <= 0) then
    poly_ddv=0.0_dp
else if (m < n .or. m < NPAR_POLY) then
    poly_ddv=coeffs(m)
    do i=m-1,1,-1
        poly_ddv=x*poly_ddv+coeffs(i)
    end do
else
    do i=1,n
        poly_ddv(i)=poly_dd(x(i),coeffs)
    end do
end if
END FUNCTION poly_ddv

FUNCTION poly_msk_rrv(x,coeffs,mask)
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs,x
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(x)) :: poly_msk_rrv
poly_msk_rrv=unpack(poly_rrv(pack(x,mask),coeffs),mask,0.0_sp)
END FUNCTION poly_msk_rrv

FUNCTION poly_msk_ddv(x,coeffs,mask)
REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs,x
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
REAL(DP), DIMENSION(size(x)) :: poly_msk_ddv
poly_msk_ddv=unpack(poly_ddv(pack(x,mask),coeffs),mask,0.0_dp)
END FUNCTION poly_msk_ddv

RECURSIVE FUNCTION poly_term_rr(a,b) RESULT(u)
   Tabulate cumulants of a polynomial.
REAL(SP), DIMENSION(:), INTENT(IN) :: a
REAL(SP), INTENT(IN) :: b
REAL(SP), DIMENSION(size(a)) :: u
INTEGER(I4B) :: n,j
n=size(a)
if (n <= 0) RETURN
u(1)=a(1)
if (n < NPAR_POLYTERM) then
    do j=2,n
        u(j)=a(j)+b*u(j-1)
    end do
```

```
else
    u(2:n:2)=poly_term_rr(a(2:n:2)+a(1:n-1:2)*b,b*b)
    u(3:n:2)=a(3:n:2)+b*u(2:n-1:2)
end if
END FUNCTION poly_term_rr

RECURSIVE FUNCTION poly_term_cc(a,b) RESULT(u)
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
COMPLEX(SPC), INTENT(IN) :: b
COMPLEX(SPC), DIMENSION(size(a)) :: u
INTEGER(I4B) :: n,j
n=size(a)
if (n <= 0) RETURN
u(1)=a(1)
if (n < NPAR_POLYTERM) then
    do j=2,n
        u(j)=a(j)+b*u(j-1)
    end do
else
    u(2:n:2)=poly_term_cc(a(2:n:2)+a(1:n-1:2)*b,b*b)
    u(3:n:2)=a(3:n:2)+b*u(2:n-1:2)
end if
END FUNCTION poly_term_cc

FUNCTION zroots_unity(n,nn)
```

Complex function returning nn powers of the nth root of unity.

```
INTEGER(I4B), INTENT(IN) :: n,nn
COMPLEX(SPC), DIMENSION(nn) :: zroots_unity
INTEGER(I4B) :: k
REAL(SP) :: theta
zroots_unity(1)=1.0
theta=TWOPI/n
k=1
do
    if (k >= nn) exit
    zroots_unity(k+1)=cmplx(cos(k*theta),sin(k*theta),SPC)
    zroots_unity(k+2:min(2*k,nn))=zroots_unity(k+1)*&
        zroots_unity(2:min(k,nn-k))
    k=2*k
end do
END FUNCTION zroots_unity
```

Routines for "outer" operations on vectors. The order convention is: result(i,j) = first_operand(i) (op) second_operand(j).

```
FUNCTION outerprod_r(a,b)
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a),size(b)) :: outerprod_r
outerprod_r = spread(a,dim=2,ncopies=size(b)) * &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerprod_r

FUNCTION outerprod_d(a,b)
REAL(DP), DIMENSION(:), INTENT(IN) :: a,b
REAL(DP), DIMENSION(size(a),size(b)) :: outerprod_d
outerprod_d = spread(a,dim=2,ncopies=size(b)) * &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerprod_d

FUNCTION outerdiv(a,b)
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a),size(b)) :: outerdiv
outerdiv = spread(a,dim=2,ncopies=size(b)) / &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiv

FUNCTION outersum(a,b)
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
```

```
REAL(SP), DIMENSION(size(a),size(b)) :: outersum
outersum = spread(a,dim=2,ncopies=size(b)) + &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outersum

FUNCTION outerdiff_r(a,b)
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a),size(b)) :: outerdiff_r
outerdiff_r = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_r

FUNCTION outerdiff_d(a,b)
REAL(DP), DIMENSION(:), INTENT(IN) :: a,b
REAL(DP), DIMENSION(size(a),size(b)) :: outerdiff_d
outerdiff_d = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_d

FUNCTION outerdiff_i(a,b)
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: a,b
INTEGER(I4B), DIMENSION(size(a),size(b)) :: outerdiff_i
outerdiff_i = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_i

FUNCTION outerand(a,b)
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: a,b
LOGICAL(LGT), DIMENSION(size(a),size(b)) :: outerand
outerand = spread(a,dim=2,ncopies=size(b)) .and. &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerand
```

Routines for scatter-with-combine.

```
SUBROUTINE scatter_add_r(dest,source,dest_index)
REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
REAL(SP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_add_r')
m=size(dest)
do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
END SUBROUTINE scatter_add_r
SUBROUTINE scatter_add_d(dest,source,dest_index)
REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
REAL(DP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_add_d')
m=size(dest)
do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
END SUBROUTINE scatter_add_d
SUBROUTINE scatter_max_r(dest,source,dest_index)
REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
REAL(SP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_max_r')
m=size(dest)
do j=1,n
    i=dest_index(j)
```

```
    if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
end do
END SUBROUTINE scatter_max_r
SUBROUTINE scatter_max_d(dest,source,dest_index)
REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
REAL(DP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_max_d')
m=size(dest)
do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
end do
END SUBROUTINE scatter_max_d
```

Routines for skew operations on matrices:

```
SUBROUTINE diagadd_rv(mat,diag)
```

Adds vector or scalar diag to the diagonal of matrix mat.

```
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
REAL(SP), DIMENSION(:), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = assert_eq2(size(diag),min(size(mat,1),size(mat,2)),'diagadd_rv')
do j=1,n
    mat(j,j)=mat(j,j)+diag(j)
end do
END SUBROUTINE diagadd_rv

SUBROUTINE diagadd_r(mat,diag)
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
REAL(SP), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = min(size(mat,1),size(mat,2))
do j=1,n
    mat(j,j)=mat(j,j)+diag
end do
END SUBROUTINE diagadd_r

SUBROUTINE diagmult_rv(mat,diag)
```

Multiplies vector or scalar diag into the diagonal of matrix mat.

```
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
REAL(SP), DIMENSION(:), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = assert_eq2(size(diag),min(size(mat,1),size(mat,2)),'diagmult_rv')
do j=1,n
    mat(j,j)=mat(j,j)*diag(j)
end do
END SUBROUTINE diagmult_rv

SUBROUTINE diagmult_r(mat,diag)
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
REAL(SP), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = min(size(mat,1),size(mat,2))
do j=1,n
    mat(j,j)=mat(j,j)*diag
end do
END SUBROUTINE diagmult_r

FUNCTION get_diag_rv(mat)
```

Return as a vector the diagonal of matrix mat.

```
REAL(SP), DIMENSION(:,:), INTENT(IN) :: mat
REAL(SP), DIMENSION(size(mat,1)) :: get_diag_rv
INTEGER(I4B) :: j
j=assert_eq2(size(mat,1),size(mat,2),'get_diag_rv')
do j=1,size(mat,1)
    get_diag_rv(j)=mat(j,j)
```

```
end do
END FUNCTION get_diag_rv

FUNCTION get_diag_dv(mat)
REAL(DP), DIMENSION(:,:), INTENT(IN) :: mat
REAL(DP), DIMENSION(size(mat,1)) :: get_diag_dv
INTEGER(I4B) :: j
j=assert_eq2(size(mat,1),size(mat,2),'get_diag_dv')
do j=1,size(mat,1)
    get_diag_dv(j)=mat(j,j)
end do
END FUNCTION get_diag_dv

SUBROUTINE put_diag_rv(diagv,mat)
   Set the diagonal of matrix mat to the values of a vector or scalar.
REAL(SP), DIMENSION(:), INTENT(IN) :: diagv
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
INTEGER(I4B) :: j,n
n=assert_eq2(size(diagv),min(size(mat,1),size(mat,2)),'put_diag_rv')
do j=1,n
    mat(j,j)=diagv(j)
end do
END SUBROUTINE put_diag_rv

SUBROUTINE put_diag_r(scal,mat)
REAL(SP), INTENT(IN) :: scal
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
INTEGER(I4B) :: j,n
n = min(size(mat,1),size(mat,2))
do j=1,n
    mat(j,j)=scal
end do
END SUBROUTINE put_diag_r

SUBROUTINE unit_matrix(mat)
   Set the matrix mat to be a unit matrix (if it is square).
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: mat
INTEGER(I4B) :: i,n
n=min(size(mat,1),size(mat,2))
mat(:,:)=0.0_sp
do i=1,n
    mat(i,i)=1.0_sp
end do
END SUBROUTINE unit_matrix

FUNCTION upper_triangle(j,k,extra)
   Return an upper triangular logical mask.
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: upper_triangle
INTEGER(I4B) :: n
n=0
if (present(extra)) n=extra
upper_triangle=(outerdiff(arth_i(1,1,j),arth_i(1,1,k)) < n)
END FUNCTION upper_triangle

FUNCTION lower_triangle(j,k,extra)
   Return a lower triangular logical mask.
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: lower_triangle
INTEGER(I4B) :: n
n=0
if (present(extra)) n=extra
lower_triangle=(outerdiff(arth_i(1,1,j),arth_i(1,1,k)) > -n)
END FUNCTION lower_triangle
```

Other routines:

```
FUNCTION vabs(v)
    Return the length (ordinary L2 norm) of a vector.
REAL(SP), DIMENSION(:), INTENT(IN) :: v
REAL(SP) :: vabs
vabs=sqrt(dot_product(v,v))
END FUNCTION vabs
```

END MODULE nrutil