Chapter 23. Numerical Recipes Utility Functions for Fortran 90

23.0 Introduction and Summary Listing

This chapter describes and summarizes the Numerical Recipes utility routines that are used throughout the rest of this volume. A complete implementation of these routines in Fortran 90 is listed in Appendix C1.

Why do we need utility routines? Aren't there already enough of them built into the language as Fortran 90 intrinsics? The answers lie in this volume's dual purpose: to implement the Numerical Recipes routines in Fortran 90 code that runs efficiently on fast serial machines, *and* to implement them, wherever possible, with efficient parallel code for multiprocessor machines that will become increasingly common in the future. We have found three kinds of situations where additional utility routines seem desirable:

1. Fortran 90 is a big language, with many high-level constructs — single statements that actually result in a lot of computing. We like this; it gives the language the potential for expressing algorithms very readably, getting them "out of the mud" of microscopic coding. In coding the 350+ Recipes for this volume, we kept a systematic watch for bits of microscopic coding that were repeated in many routines, and that seemed to be at a lower level of coding than that aspired to by good Fortran 90 style. Once these bits were identified, we pulled them out and substituted calls to new utility routines. These are the utilities that arguably ought to be new language intrinsics, equally useful for serial and parallel machines. (A prime example is `swap`.)

2. Fortran 90 contains many highly parallelizable language constructions. But, as we have seen in §22.5, it is also missing a few important constructions. Most parallel machines will provide these missing elements as machine-coded library subroutines. Some of our utility routines are provided simply as a standard interface to these common, but nonstandard, functionalities. Note that it is the nature of these routines that our specific implementation, in Appendix C1, will be serial, and therefore inefficient on parallel machines. If you have a parallel machine, you will need to recode these; this often involves no more than substituting a one-line library function call for the body of our implementation. Utilities in this category will likely become unnecessary over time, either as machine-dependent libraries converge to standard interfaces, or as the utilities get added to future Fortran

versions. (Indeed, some routines in this category will be unnecessary in Fortran 95, once it is available; see §23.7.)

3. Some tasks should just be done differently in serial, versus parallel, implementation. Linear recurrence relations are a good example (§22.2). These are trivially coded with a do-loop on serial machines, but require a fairly elaborate recursive construction for good parallelization. Rather than provide separate serial and parallel versions of the Numerical Recipes, we have chosen to pull out of the Recipes, and into utility routines, some identifiable tasks of this kind. These are cases where some recoding of our implementation in Appendix C1 might result in improved performance on your particular hardware. Unfortunately, it is not so simple as providing a single "serial implementation" and another single "parallel implementation," because even the seemingly simple word "serial" hides, at the hardware level, a variety of different degrees of pipelining, wide instructions, and so on. Appendix C1 therefore provides only a single implementation, although with some adjustable parameters that you can customize (by experiment) to maximize performance on your hardware.

The above three cases are not really completely distinct, and it is therefore not possible to assign any single utility routine to exactly one situation. Instead, we organize the rest of this chapter as follows: first, an alphabetical list, with short summary, of all the new utility routines; next, a series of short sections, grouped by functionality, that contain the detailed descriptions of the routines.

Alphabetical Listing

The following list gives an abbreviated mnemonic for the type, rank, and/or shape of the returned values (as in §21.4), the routine's calling sequence (optional arguments shown in italics), and a brief, often incomplete, description. The complete description of the routine is given in the later section shown in square brackets.

For each entry, the number shown in parentheses is the approximate number of distinct Recipes in this book that make use of that particular utility function, and is thus a rough guide to that utility's importance. (There may be multiple invocations of the utility in each such Recipe.) Where this number is small or zero, it is usually because the utility routine is a member of a related family of routines whose total usage was deemed significant enough to include, and we did not want users to have to "guess" which family members were instantiated.

`call array_copy(src,dest,n_copied,n_not_copied)`
Copy one-dimensional array (whose size is not necessarily known). [23.1] (9)

[Arr] `arth(first,increment,n)`
Return an arithmetic progression as an array. [23.4] (55)

`call assert(n1,`*`n2,...,`*`string)`
Exit with error message if any logical arguments are false. [23.3] (50)

[Int] `assert_eq(n1,`*`n2,...,`*`string)`
Exit with error message if all integer arguments are not equal; otherwise return common value. [23.3] (133)

[argTS] `cumprod(arr,`*`seed`*`)`

Cumulative products of one-dimensional array, with optional seed value. [23.4] (3)

[argTS] `cumsum(arr,`*`seed`*`)`
Cumulative sums of one-dimensional array, with optional seed value. [23.4] (9)

`call diagadd(mat,diag)`
Adds vector to diagonal of a matrix. [23.7] (4)

`call diagmult(mat,diag)`
Multiplies vector into diagonal of a matrix. [23.7] (2)

[Arr] `geop(first,factor,n)`
Return a geometrical progression as an array. [23.4] (7)

[Arr] `get_diag(mat)`
Gets diagonal of a matrix. [23.7] (2)

[Int] `ifirstloc(arr)`
Location of first true value in a logical array, returned as an integer. [23.2] (3)

[Int] `imaxloc(arr)`
Location of array maximum, returned as an integer. [23.2] (11)

[Int] `iminloc(arr)`
Location of array minimum, returned as an integer. [23.2] (8)

[Mat] `lower_triangle(j,k,`*`extra`*`)`
Returns a lower triangular logical mask. [23.7] (1)

`call nrerror(string)`
Exit with error message. [23.3] (96)

[Mat] `outerand(a,b)`
Returns the outer logical and of two vectors. [23.5] (1)

[Mat] `outerdiff(a,b)`
Returns the outer difference of two vectors. [23.5] (4)

[Mat] `outerdiv(a,b)`
Returns the outer quotient of two vectors. [23.5] (0)

[Mat] `outerprod(a,b)`
Returns the outer product of two vectors. [23.5] (14)

[Mat] `outersum(a,b)`
Returns the outer sum of two vectors. [23.5] (0)

[argTS] `poly(x,coeffs,`*`mask`*`)`
Evaluate a polynomial $P(x)$ for one or more values x, with optional mask. [23.4] (15)

[argTS] `poly_term(a,x)`
Returns partial cumulants of a polynomial, equivalent to synthetic

division. [23.4] (4)

`call put_diag(diag,mat)`
Sets diagonal of a matrix. [23.7] (0)

[Ptr] `reallocate(p,n,`*`m,...`*`)`
Reallocate pointer to new size, preserving its contents. [23.1] (5)

`call scatter_add(dest,source,dest_index)`
Scatter-adds source vector to specified components of destination vector. [23.6] (2)

`call scatter_max(dest,source,dest_index)`
Scatter-max source vector to specified components of destination vector. [23.6] (0)

`call swap(a,b,`*`mask`*`)`
Swap corresponding elements of `a` and `b`. [23.1] (24)

`call unit_matrix(mat)`
Sets matrix to be a unit matrix. [23.7] (6)

[Mat] `upper_triangle(j,k,`*`extra`*`)`
Returns an upper triangular logical mask. [23.7] (4)

[Real] `vabs(v)`
Length of a vector in L_2 norm. [23.8] (6)

[CArr] `zroots_unity(n,nn)`
Returns `nn` consecutive powers of the complex `n`th root of unity. [23.4] (4)

Comment on Relative Frequencies of Use

We find it interesting to compare our frequency of using the `nrutil` utility routines, with our most used language intrinsics (see §21.4). On this basis, the following routines are as useful to us as the *top 10* language intrinsics: `arth`, `assert`, `assert_eq`, `outerprod`, `poly`, and `swap`. We strongly recommend that the X3J3 standards committee, as well as individual compiler library implementors, consider the inclusion of new language intrinsics (or library routines) that subsume the capabilities of at least these routines. In the next tier of importance, we would put some further cumulative operations (`geop`, `cumsum`), some other "outer" operations on vectors (e.g., `outerdiff`), basic operations on the diagonals of matrices (`get_diag`, `put_diag`, `diag_add`), and some means of access to an array of unknown size (`array_copy`).

23.1 Routines That Move Data

To describe our utility routines, we introduce two items of Fortran 90 pseudocode: We use the symbol **T** to denote some type and rank declaration (including scalar rank, i.e., zero); and when we append a colon to a type specification, as in `INTEGER(I4B)(:)`, for example, we denote an array of the given type.

The routines `swap`, `array_copy`, and `reallocate` simply move data around in useful ways.

⋆ ⋆ ⋆

swap (swaps corresponding elements)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE swap(a,b,mask)
T, INTENT(INOUT) :: a,b
LOGICAL(LGT), INTENT(IN), OPTIONAL :: mask
END SUBROUTINE swap
```

Applicable types and ranks:

T ≡ *any type, any rank*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP), REAL(SP)(:), REAL(DP),`
`COMPLEX(SPC), COMPLEX(SPC)(:), COMPLEX(SPC)(:,:),`
`COMPLEX(DPC), COMPLEX(DPC)(:), COMPLEX(DPC)(:,:)`

Action:

Swaps the corresponding elements of `a` and `b`. If `mask` is present, performs the swap only where `mask` is true. (Following code is the unmasked case. For speed at run time, the masked case is implemented by overloading, not by testing for the optional argument.)

Reference implementation:

```
T :: dum
dum=a
a=b
b=dum
```

⋆ ⋆ ⋆

array_copy (copy one-dimensional array)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE array_copy(src,dest,n_copied,n_not_copied)
T, INTENT(IN) :: src
T, INTENT(OUT) :: dest
INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
END SUBROUTINE array_copy
```

Applicable types and ranks:

T ≡ *any type, rank 1*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B)(:), REAL(SP)(:), REAL(DP)(:)`

Action:

Copies to a destination array `dest` the one-dimensional array `src`, or as much of `src` as will fit in `dest`. Returns the number of components copied as `n_copied`, and the number of components not copied as `n_not_copied`.

The main use of this utility is where `src` is an expression that returns an array whose size is not known in advance, for example, the value returned by the `pack` intrinsic.

Reference implementation:

```
n_copied=min(size(src),size(dest))
n_not_copied=size(src)-n_copied
dest(1:n_copied)=src(1:n_copied)
```

⋆ ⋆ ⋆

reallocate (reallocate a pointer, preserving contents)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION reallocate(p,n[,m,...])
T, POINTER :: p, reallocate
INTEGER(I4B), INTENT(IN) :: n[,m,...]
END FUNCTION reallocate
```

Applicable types and ranks:

T ≡ *any type, rank 1 or greater*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B)(:)`, `INTEGER(I4B)(:,:)`, `REAL(SP)(:)`, `REAL(SP)(:,:)`, `CHARACTER(1)(:)`

Action:

Allocates storage for a new array with shape specified by the integer(s) `n`, `m`, . . . (equal in number to the rank of pointer `p`). Then, copies the contents of `p`'s target (or as much as will fit) into the new storage. Then, deallocates `p` and returns a pointer to the new storage.

The typical use is `p=reallocate(p,n`$[, m, \ldots]$`)`, which has the effect of changing the allocated size of `p` while preserving the contents.

The reference implementation, below, shows only the case of rank 1.

Reference implementation:

```
INTEGER(I4B) :: nold,ierr
allocate(reallocate(n),stat=ierr)
if (ierr /= 0) call &
   nrerror('reallocate: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p)
reallocate(1:min(nold,n))=p(1:min(nold,n))
deallocate(p)
```

23.2 Routines Returning a Location

Fortran 90's intrinsics `maxloc` and `minloc` return rank-one arrays. When, in the most frequent usage, their argument is a one-dimensional array, the answer comes back in the inconvenient form of an array containing a single component, which cannot be itself used in a subscript calculation. While there are workaround tricks (e.g., use of the `sum` intrinsic to convert the array to a scalar), it seems clearer to define routines `imaxloc` and `iminloc` that return integers directly.

The routine `ifirstloc` adds a related facility missing among the intrinsics: Return the first true location in a logical array.

⋆ ⋆ ⋆

imaxloc (location of array maximum as an integer)

User interface (or, “`USE nrutil`*”):*

```
FUNCTION imaxloc(arr)
T, INTENT(IN) :: arr
INTEGER(I4B) :: imaxloc
END FUNCTION imaxloc
```

Applicable types and ranks:

T ≡ *any integer or real type, rank 1*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B)(:), REAL(SP)(:)`

Action:

For one-dimensional arrays, identical to the `maxloc` intrinsic, except returns its answer as an integer rather than as `maxloc`’s somewhat awkward rank-one array containing a single component.

Reference implementation:

```
INTEGER(I4B), DIMENSION(1) :: imax
imax=maxloc(arr(:))
imaxloc=imax(1)
```

⋆ ⋆ ⋆

iminloc (location of array minimum as an integer)

User interface (or, “`USE nrutil`*”):*

```
FUNCTION iminloc(arr)
T, INTENT(IN) :: arr
INTEGER(I4B) :: iminloc
END FUNCTION iminloc
```

Applicable types and ranks:

T ≡ *any integer or real type, rank 1*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)(:)`

Action:

For one-dimensional arrays, identical to the `minloc` intrinsic, except returns its answer as an integer rather than as `minloc`’s somewhat awkward rank-one array containing a single component.

Reference implementation:

```
INTEGER(I4B), DIMENSION(1) :: imin
imin=minloc(arr(:))
iminloc=imin(1)
```

⋆ ⋆ ⋆

ifirstloc (returns location of first “true” in a logical vector)

User interface (or, “`USE nrutil`*”):*

```
FUNCTION ifirstloc(mask)
T, INTENT(IN) :: mask
INTEGER(I4B) :: ifirstloc
END FUNCTION ifirstloc
```

Applicable types and ranks:

T ≡ *any logical type, rank 1*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `LOGICAL(LGT)`

Action:

Returns the index (subscript value) of the first location, in a one-dimensional logical `mask`, that has the value `.TRUE.`, or returns `size(mask)+1` if all components of `mask` are `.FALSE.`

Note that while the reference implementation uses a do-loop, the function is parallelized in `nrutil` by instead using the `merge` and `maxloc` intrinsics.

Reference implementation:

```
INTEGER(I4B) :: i
do i=1,size(mask)
   if (mask(i)) then
      ifirstloc=i
      return
   end if
end do
ifirstloc=i
```

23.3 Argument Checking and Error Handling

It is good programming practice for a routine to check the assumptions ("assertions") that it makes about the sizes of input arrays, allowed range of numerical arguments, and so forth. The routines `assert` and `assert_eq` are meant for this kind of use. The routine `nrerror` is our default error reporting routine.

⋆ ⋆ ⋆

assert (exit with error message if any assertion is false)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE assert(n1,n2,...,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2,...
END SUBROUTINE assert
```

Action:

Embedding program dies gracefully with an error message if any of the logical arguments are false. Typical use is with logical expressions as the actual arguments. `nrutil` implements and overloads forms with 1, 2, 3, and 4 logical arguments, plus a form with a vector logical argument,

```
LOGICAL, DIMENSION(:), INTENT(IN) ::  n
```

that is checked by the `all(n)` intrinsic.

Reference implementation:

```
if (.not. (n1.and.n2.and...)) then
   write (*,*) 'nrerror: an assertion failed with this tag:', string
   STOP 'program terminated by assert'
end if
```

⋆ ⋆ ⋆

assert_eq (exit with error message if integer arguments not all equal)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION assert_eq(n1,n2,n3,...,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2,n3,...
INTEGER :: assert_eq
END FUNCTION assert_eq
```

Action:

Embedding program dies gracefully with an error message if any of the integer arguments are not equal to the first. Otherwise, return the value of the first argument. Typical use is for enforcing equality on the sizes of arrays passed to a subprogram. `nrutil` implements and overloads forms with 1, 2, 3, and 4 integer arguments, plus a form with a vector integer argument,

```
INTEGER, DIMENSION(:), INTENT(IN) ::  n
```

that is checked by the conditional `if (all(nn(2:)==nn(1)))`.

Reference implementation:

```
if (n1==n2.and.n2==n3.and...) then
   assert_eq=n1
else
   write (*,*) 'nrerror: an assert_eq failed with this tag:', string
   STOP 'program terminated by assert_eq'
end if
```

⋆ ⋆ ⋆

nrerror (report error message and stop)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE nrerror(string)
CHARACTER(LEN=*), INTENT(IN) :: string
END SUBROUTINE nrerror
```

Action:

This is the minimal error handler used in this book. In applications of any complexity, it is intended only as a placeholder for a user's more complicated error handling strategy.

Reference implementation:

```
write (*,*) 'nrerror: ',string
STOP 'program terminated by nrerror'
```

23.4 Routines for Polynomials and Recurrences

Apart from programming convenience, these routines are designed to allow for nontrivial parallel implementations, as discussed in §22.2 and §22.3.

⋆ ⋆ ⋆

arth (returns arithmetic progression as an array)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION arth(first,increment,n)
T, INTENT(IN) :: first,increment
INTEGER(I4B), INTENT(IN) :: n
T, DIMENSION(n) [or, 1 rank higher than T]:: arth
END FUNCTION arth
```

Applicable types and ranks:

T ≡ *any numerical type, any rank*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP), REAL(DP)`

Action:

Returns an array of length `n` containing an arithmetic progression whose first value is `first` and whose increment is `increment`. If `first` and `increment` have rank greater than zero, returns an array of one larger rank, with the last subscript having size `n` and indexing the progressions. Note that the following reference implementation (for the scalar case) is definitional only, and neither parallelized nor optimized for roundoff error. See §22.2 and Appendix C1 for implementation by subvector scaling.

Reference implementation:

```
INTEGER(I4B) :: k
if (n > 0) arth(1)=first
do k=2,n
   arth(k)=arth(k-1)+increment
end do
```

⋆ ⋆ ⋆

geop (returns geometric progression as an array)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION geop(first,factor,n)
T, INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
T, DIMENSION(n) [or, 1 rank higher than T]:: geop
END FUNCTION geop
```

Applicable types and ranks:

T ≡ *any numerical type, any rank*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP), REAL(DP), REAL(DP)(:),`
`COMPLEX(SPC)`

Action:

Returns an array of length `n` containing a geometric progression whose first value is `first` and whose multiplier is `factor`. If `first` and `factor` have rank greater than zero, returns an array of one larger rank, with the last subscript having size `n` and indexing the progression. Note that the following reference implementation (for the scalar case) is definitional only, and neither parallelized nor optimized for roundoff error. See §22.2 and Appendix C1 for implementation by subvector scaling.

Reference implementation:

```
INTEGER(I4B) :: k
if (n > 0) geop(1)=first
do k=2,n
   geop(k)=geop(k-1)*factor
end do
```

⋆ ⋆ ⋆

cumsum (cumulative sum on an array, with optional additive seed)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION cumsum(arr,seed)
T, DIMENSION(:), INTENT(IN) :: arr
T, OPTIONAL, INTENT(IN) :: seed
T, DIMENSION(size(arr)), INTENT(OUT) :: cumsum
END FUNCTION cumsum
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP)`

Action:

Given the rank 1 array `arr` of type **T**, returns an array of identical type and size containing the cumulative sums of `arr`. If the optional argument `seed` is present, it is added to the first component (and therefore, by cumulation, all components) of the result. See §22.2 for parallelization ideas.

Reference implementation:

```
INTEGER(I4B) :: n,j
T :: sd
n=size(arr)
if (n == 0) return
sd=0.0
if (present(seed)) sd=seed
cumsum(1)=arr(1)+sd
do j=2,n
   cumsum(j)=cumsum(j-1)+arr(j)
end do
```

⋆ ⋆ ⋆

cumprod (cumulative prod on an array, with optional multiplicative seed)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION cumprod(arr,seed)
T, DIMENSION(:), INTENT(IN) :: arr
T, OPTIONAL, INTENT(IN) :: seed
T, DIMENSION(size(arr)), INTENT(OUT) :: cumprod
END FUNCTION cumprod
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Given the rank 1 array `arr` of type **T**, returns an array of identical type and size containing the cumulative products of `arr`. If the optional argument `seed` is present, it is multiplied into the first component (and therefore, by cumulation, into all components) of the result. See §22.2 for parallelization ideas.

Reference implementation:

```
INTEGER(I4B) :: n,j
T :: sd
n=size(arr)
if (n == 0) return
sd=1.0
if (present(seed)) sd=seed
cumprod(1)=arr(1)*sd
do j=2,n
   cumprod(j)=cumprod(j-1)*arr(j)
end do
```

⋆ ⋆ ⋆

poly (polynomial evaluation)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION poly(x,coeffs,mask)
T,, DIMENSION(:,...), INTENT(IN) :: x
T, DIMENSION(:), INTENT(IN) :: coeffs
LOGICAL(LGT), DIMENSION(:,...), OPTIONAL, INTENT(IN) :: mask
T :: poly
END FUNCTION poly
```

Applicable types and ranks:

T ≡ *any numerical type (*`x` *may be scalar or have any rank;* `x` *and* `coeffs` *may have different numerical types)*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ *various combinations of* `REAL(SP)`, `REAL(SP)(:)`, `REAL(DP)`, `REAL(DP)(:)`, `COMPLEX(SPC)` *(see Appendix C1 for details)*

Action:

Returns a scalar value or array with the same type and shape as `x`, containing the result of evaluating the polynomial with one-dimensional coefficient vector `coeffs` on each component of `x`. The optional argument `mask`, if present, has the same shape as `x`, and suppresses evaluation of the polynomial where its components are `.false.`. The following reference code shows the case where `mask` is not present. (The other case can be included by overloading.)

Reference implementation:

```
INTEGER(I4B) :: i,n
n=size(coeffs)
if (n <= 0) then
   poly=0.0
else
   poly=coeffs(n)
   do i=n-1,1,-1
      poly=x*poly+coeffs(i)
   end do
end if
```

⋆ ⋆ ⋆

poly_term (partial cumulants of a polynomial)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION poly_term(a,x)
T, DIMENSION(:), INTENT(IN) :: a
T, INTENT(IN) :: x
T, DIMENSION(size(a)) :: poly_term
END FUNCTION poly_term
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP), COMPLEX(SPC)`

Action:

Returns an array of type and size the same as the one-dimensional array `a`, containing the partial cumulants of the polynomial with coefficients `a` (arranged from highest-order to lowest-order coefficients, n.b.) evaluated at `x`. This is equivalent to synthetic division, and can be parallelized. See §22.3. Note that the order of arguments is reversed in `poly` and `poly_term` — each routine returns a value with the size and shape of the *first* argument, the usual Fortran 90 convention.

Reference implementation:

```
INTEGER(I4B) :: n,j
n=size(a)
if (n <= 0) return
poly_term(1)=a(1)
do j=2,n
   poly_term(j)=a(j)+x*poly_term(j-1)
end do
```

⋆ ⋆ ⋆

zroots_unity (returns powers of complex nth root of unity)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION zroots_unity(n,nn)
INTEGER(I4B), INTENT(IN) :: n,nn
COMPLEX(SPC), DIMENSION(nn) :: zroots_unity
END FUNCTION zroots_unity
```

Action:

Returns a complex array containing `nn` consecutive powers of the `n`th complex root of unity. Note that the following reference implementation is definitional only, and neither parallelized nor optimized for roundoff error. See Appendix C1 for implementation by subvector scaling.

Reference implementation:

```
INTEGER(I4B) :: k
REAL(SP) :: theta
if (nn==0) return
zroots_unity(1)=1.0
if (nn==1) return
theta=TWOPI/n
zroots_unity(2)=cmplx(cos(theta),sin(theta))
do k=3,nn
   zroots_unity(k)=zroots_unity(k-1)*zroots_unity(2)
end do
```

23.5 Routines for Outer Operations on Vectors

Outer operations on vectors take two vectors as input, and return a matrix as output. One dimension of the matrix is the size of the first vector, the other is the size of the second vector. Our convention is always the standard one,

$$\texttt{result(i,j)} = \texttt{first\_operand(i)}\ (op)\ \texttt{second\_operand(j)}$$

where (op) is any of addition, subtraction, multiplication, division, and logical `and`. The reason for coding these as utility routines is that Fortran 90's native construction, with two spreads (cf. §22.1), is difficult to read and thus prone to programmer errors.

⋆ ⋆ ⋆

outerprod (outer product)

User interface (or, "`USE nrutil`"):

```
FUNCTION outerprod(a,b)
T, DIMENSION(:), INTENT(IN) :: a,b
T, DIMENSION(size(a),size(b)) :: outerprod
END FUNCTION outerprod
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`:

T ≡ `REAL(SP), REAL(DP)`

Action:

Returns a matrix that is the outer product of two vectors.

Reference implementation:

```
outerprod = spread(a,dim=2,ncopies=size(b)) * &
   spread(b,dim=1,ncopies=size(a))
```

⋆ ⋆ ⋆

outerdiv (outer quotient)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION outerdiv(a,b)
T, DIMENSION(:), INTENT(IN) :: a,b
T, DIMENSION(size(a),size(b)) :: outerdiv
END FUNCTION outerdiv
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Returns a matrix that is the outer quotient of two vectors.

Reference implementation:

```
outerdiv = spread(a,dim=2,ncopies=size(b)) / &
   spread(b,dim=1,ncopies=size(a))
```

⋆ ⋆ ⋆

outersum (outer sum)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION outersum(a,b)
T, DIMENSION(:), INTENT(IN) :: a,b
T, DIMENSION(size(a),size(b)) :: outersum
END FUNCTION outersum
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Returns a matrix that is the outer sum of two vectors.

Reference implementation:

```
outersum = spread(a,dim=2,ncopies=size(b)) + &
   spread(b,dim=1,ncopies=size(a))
```

⋆ ⋆ ⋆

outerdiff (outer difference)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION outerdiff(a,b)
T, DIMENSION(:), INTENT(IN) :: a,b
T, DIMENSION(size(a),size(b)) :: outerdiff
END FUNCTION outerdiff
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP), REAL(DP)`

Action:

Returns a matrix that is the outer difference of two vectors.

Reference implementation:

```
outerdiff = spread(a,dim=2,ncopies=size(b)) - &
   spread(b,dim=1,ncopies=size(a))
```

⋆ ⋆ ⋆

outerand (outer logical and)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION outerand(a,b)
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: a,b
LOGICAL(LGT), DIMENSION(size(a),size(b)) :: outerand
END FUNCTION outerand
```

Applicable types and ranks:

T ≡ *any logical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `LOGICAL(LGT)`

Action:

Returns a matrix that is the outer logical and of two vectors.

Reference implementation:

```
outerand = spread(a,dim=2,ncopies=size(b)) .and. &
   spread(b,dim=1,ncopies=size(a))
```

23.6 Routines for Scatter with Combine

These are common parallel functions that Fortran 90 simply doesn't provide a means for implementing. If you have a parallel machine, you should substitute library routines specific to your hardware.

⋆ ⋆ ⋆

scatter_add (scatter-add source to specified components of destination)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE scatter_add(dest,source,dest_index)
T, DIMENSION(:), INTENT(OUT) :: dest
T, DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
END SUBROUTINE scatter_add
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP), REAL(DP)`

Action:

Adds each component of the array `source` into a component of `dest` specified by the index array `dest_index`. (The user will usually have zeroed `dest` before the call to this routine.) Note that `dest_index` has the size of `source`, but must contain values in the range from 1 to `size(dest)`, inclusive. Out-of-range values are ignored. There is no parallel implementation of this routine accessible from Fortran 90; most parallel machines supply an implementation as a library routine.

Reference implementation:

```
INTEGER(I4B) :: m,n,j,i
n=assert_eq(size(source),size(dest_index),'scatter_add')
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
```

⋆ ⋆ ⋆

scatter_max (scatter-max source to specified components of destination)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE scatter_max(dest,source,dest_index)
T, DIMENSION(:), INTENT(OUT) :: dest
T, DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
END SUBROUTINE scatter_max
```

Applicable types and ranks:

T ≡ *any integer or real type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP), REAL(DP)`

Action:

Takes the `max` operation between each component of the array `source` and a component of `dest` specified by the index array `dest_index`, replacing that component of `dest` with the value obtained ("maxing into" operation). (The user will often want to fill the array `dest` with the value −`huge` before the call to this routine.) Note that `dest_index` has the size of `source`, but must contain values in the range from 1 to `size(dest)`, inclusive. Out-of-range values are ignored. There is no parallel implementation of this routine accessible from Fortran 90; most parallel machines supply an implementation as a library routine.

Reference implementation:

```
INTEGER(I4B) :: m,n,j,i
n=assert_eq(size(source),size(dest_index),'scatter_max')
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
```

23.7 Routines for Skew Operations on Matrices

These are also missing parallel capabilities in Fortran 90. In Appendix C1 they are coded serially, with one or more do-loops.

⋆ ⋆ ⋆

diagadd (adds vector to diagonal of a matrix)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE diagadd(mat,diag)
T, DIMENSION(:,:), INTENT(INOUT) :: mat
T, DIMENSION(:), INTENT(IN) :: diag
END SUBROUTINE diagadd
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

The argument `diag`, either a scalar or else a vector whose size must be the smaller of the two dimensions of matrix `mat`, is added to the diagonal of the matrix `mat`. The following shows an implementation where `diag` is a vector; the scalar case can be overloaded (see Appendix C1).

Reference implementation:

```
INTEGER(I4B) :: j,n
n = assert_eq(size(diag),min(size(mat,1),size(mat,2)),'diagadd')
do j=1,n
   mat(j,j)=mat(j,j)+diag(j)
end do
```

⋆ ⋆ ⋆

diagmult (multiplies vector into diagonal of a matrix)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE diagmult(mat,diag)
T, DIMENSION(:,:), INTENT(INOUT) :: mat
T, DIMENSION(:), INTENT(IN) :: diag
END SUBROUTINE diagmult
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

The argument `diag`, either a scalar or else a vector whose size must be the smaller of the two dimensions of matrix `mat`, is multiplied onto the diagonal of the matrix `mat`. The following shows an implementation where `diag` is a vector; the scalar case can be overloaded (see Appendix C1).

Reference implementation:

```
INTEGER(I4B) :: j,n
n = assert_eq(size(diag),min(size(mat,1),size(mat,2)),'diagmult')
do j=1,n
   mat(j,j)=mat(j,j)*diag(j)
end do
```

⋆ ⋆ ⋆

get_diag (gets diagonal of matrix)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION get_diag(mat)
T, DIMENSION(:,:), INTENT(IN) :: mat
T, DIMENSION(min(size(mat,1),size(mat,2))) :: get_diag
END FUNCTION get_diag
```

Applicable types and ranks:

T ≡ *any type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP), REAL(DP)`

Action:

Returns a vector containing the diagonal values of the matrix `mat`.

Reference implementation:

```
INTEGER(I4B) :: j
do j=1,min(size(mat,1),size(mat,2))
   get_diag(j)=mat(j,j)
end do
```

⋆ ⋆ ⋆

put_diag (sets the diagonal elements of a matrix)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE put_diag(diag,mat)
T, DIMENSION(:), INTENT(IN) :: diag
T, DIMENSION(:,:), INTENT(INOUT) :: mat
END SUBROUTINE put_diag
```

Applicable types and ranks:

T ≡ *any type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Sets the diagonal of matrix `mat` equal to the argument `diag`, either a scalar or else a vector whose size must be the smaller of the two dimensions of matrix `mat`. The following shows an implementation where `diag` is a vector; the scalar case can be overloaded (see Appendix C1).

Reference implementation:

```
INTEGER(I4B) :: j,n
n=assert_eq(size(diag),min(size(mat,1),size(mat,2)),'put_diag')
do j=1,n
   mat(j,j)=diag(j)
end do
```

⋆ ⋆ ⋆

unit_matrix (returns a unit matrix)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE unit_matrix(mat)
T, DIMENSION(:,:), INTENT(OUT) :: mat
END SUBROUTINE unit_matrix
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Sets the diagonal components of `mat` to unity, all other components to zero. When `mat` is square, this will be the unit matrix; otherwise, a unit matrix with appended rows or columns of zeros.

Reference implementation:

```
INTEGER(I4B) :: i,n
n=min(size(mat,1),size(mat,2))
mat(:,:)=0.0
do i=1,n
   mat(i,i)=1.0
end do
```

⋆ ⋆ ⋆

upper_triangle (returns an upper triangular mask)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION upper_triangle(j,k,extra)
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: upper_triangle
END FUNCTION upper_triangle
```

Action:

When the optional argument `extra` is zero or absent, returns a logical mask of shape $(\mathtt{j},\mathtt{k})$ whose values are true above and to the right of the diagonal, false elsewhere (including on the diagonal). When `extra` is present and positive, a corresponding number of additional (sub-)diagonals are returned as true. ($\mathtt{extra} = 1$ makes the main diagonal return true.) When `extra` is present and negative, it suppresses a corresponding number of superdiagonals.

Reference implementation:

```
INTEGER(I4B) :: n,jj,kk
n=0
if (present(extra)) n=extra
do jj=1,j
   do kk=1,k
      upper_triangle(jj,kk)= (jj-kk < n)
   end do
end do
```

⋆ ⋆ ⋆

lower_triangle (returns a lower triangular mask)

User interface (or, “`USE nrutil`*”):*

```
FUNCTION lower_triangle(j,k,extra)
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: lower_triangle
END FUNCTION lower_triangle
```

Action:

When the optional argument `extra` is zero or absent, returns a logical mask of shape (`j`, `k`) whose values are true below and to the left of the diagonal, false elsewhere (including on the diagonal). When `extra` is present and positive, a corresponding number of additional (super-)diagonals are returned as true. (`extra` $= 1$ makes the main diagonal return true.) When `extra` is present and negative, it suppresses a corresponding number of subdiagonals.

Reference implementation:

```
INTEGER(I4B) :: n,jj,kk
n=0
if (present(extra)) n=extra
do jj=1,j
   do kk=1,k
      lower_triangle(jj,kk)= (kk-jj < n)
   end do
end do
```

Fortran 95’s `forall` construction will make the parallel implementation of all our skew operations utilities extremely simple. For example, the do-loop in `diagadd` will collapse to

```
forall (j=1:n) mat(j,j)=mat(j,j)+diag(j)
```

In fact, this implementation is so simple as to raise the question of whether a separate utility like `diagadd` will be needed at all. There are valid arguments on both sides of this question: The “con” argument, against a routine like `diagadd`, is that it is just another reserved name that you have to remember (if you want to use it). The “pro” argument is that a separate routine avoids the “index pollution” (the opposite disease from “index loss” discussed in §22.1) of introducing a superfluous variable `j`, and that a separate utility allows for additional error checking on the sizes and compatibility of its arguments. We expect that different programmers will have differing tastes.

The argument for keeping a routine like `upper_triangle` or `lower_triangle`, once Fortran 95’s *masked* `forall` constructions become available, is less persuasive. We recommend that you consider these two routines as placeholders for “remember to recode this in Fortran 95, someday.”

23.8 Other Routine(s)

You might argue that we don’t really need a routine for the idiom

```
sqrt(dot_product(v,v))
```

You might be right. The ability to overload the complex case, with its additional complex conjugate, is an argument in its favor, however.

⋆ ⋆ ⋆

vabs (L_2 norm of a vector)

User interface (or, “`USE nrutil`*”):*

```
FUNCTION vabs(v)
T, DIMENSION(:), INTENT(IN) :: v
T :: vabs
END FUNCTION vabs
```

Applicable types and ranks:

T ≡ *any real or complex type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Returns the length of a vector `v` in L_2 norm, that is, the square root of the sum of the squares of the components. (For complex types, the `dot_product` should be between the vector and its complex conjugate.)

Reference implementation:

```
vabs=sqrt(dot_product(v,v))
```