Chapter B12. Fast Fourier Transform

The algorithms underlying the parallel routines in this chapter are described in §22.4. As described there, the basic building block is a routine for simultaneously taking the FFT of each row of a two-dimensional matrix:

```
SUBROUTINE fourrow_sp(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
    Replaces each row (constant first index) of data(1:M,1:N) by its discrete Fourier trans-
    form (transform on second index), if isign is input as 1; or replaces each row of data
    by N times its inverse discrete Fourier transform, if isign is input as -1. N must be an
    integer power of 2. Parallelism is M-fold on the first index of data.
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,1)) :: temp
COMPLEX(DPC) :: w,wp                    Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,2)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_sp')
n2=n/2
j=n2
  This is the bit-reversal section of the routine.
do i=1,n-2
    if (j > i) call swap(data(:,j+1),data(:,i+1))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
  Here begins the Danielson-Lanczos section of the routine.
do                                      Outer loop executed log_2 N times.
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)             Initialize for the trigonometric recurrence.
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax                         Here are the two nested inner loops.
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(:,j)           This is the Danielson-Lanczos formula.
            data(:,j)=data(:,i)-temp
            data(:,i)=data(:,i)+temp
        end do
        w=w*wp+w                        Trigonometric recurrence.
    end do
    mmax=istep
```

```
end do
END SUBROUTINE fourrow_sp
```

f90 `call assert(iand(n,n-1)==0 ...` All the Fourier routines in this chapter require the dimension N of the data to be a power of 2. This is easily tested for by AND'ing N and $N-1$: N should have the binary representation $10000\ldots$, in which case $N-1 = 01111\ldots$.

```
SUBROUTINE fourrow_dp(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(DPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(DPC), DIMENSION(size(data,1)) :: temp
COMPLEX(DPC) :: w,wp
COMPLEX(DPC) :: ws
n=size(data,2)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_dp')
n2=n/2
j=n2
do i=1,n-2
    if (j > i) call swap(data(:,j+1),data(:,i+1))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
do
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(:,j)
            data(:,j)=data(:,i)-temp
            data(:,i)=data(:,i)+temp
        end do
        w=w*wp+w
    end do
    mmax=istep
end do
END SUBROUTINE fourrow_dp
```

```
SUBROUTINE fourrow_3d(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

If `isign` is input as 1, replaces each third-index section (constant first and second indices) of `data(1:`L`,1:`M`,1:`N`)` by its discrete Fourier transform (transform on third index); or

```
    replaces each third-index section of data by N times its inverse discrete Fourier transform,
    if isign is input as -1. N must be an integer power of 2. Parallelism is L × M-fold on
    the first and second indices of data.
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,1),size(data,2)) :: temp
COMPLEX(DPC) :: w,wp                    Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,3)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_3d')
n2=n/2
j=n2
  This is the bit-reversal section of the routine.
do i=1,n-2
    if (j > i) call swap(data(:,:,j+1),data(:,:,i+1))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
  Here begins the Danielson-Lanczos section of the routine.
do                                      Outer loop executed log2 N times.
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)             Initialize for the trigonometric recurrence.
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax                         Here are the two nested inner loops.
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(:,:,j)         This is the Danielson-Lanczos formula.
            data(:,:,j)=data(:,:,i)-temp
            data(:,:,i)=data(:,:,i)+temp
        end do
        w=w*wp+w                        Trigonometric recurrence.
    end do
    mmax=istep
end do
END SUBROUTINE fourrow_3d
```

⋆ ⋆ ⋆

Exactly as in the preceding routines, we can take the FFT of each *column* of a two-dimensional matrix, and for each *first-index* section of a three-dimensional array.

```
SUBROUTINE fourcol(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
    Replaces each column (constant second index) of data(1:N,1:M) by its discrete Fourier
    transform (transform on first index), if isign is input as 1; or replaces each row of data
```

by N times its inverse discrete Fourier transform, if `isign` is input as -1. N must be an integer power of 2. Parallelism is M-fold on the second index of `data`.

```
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,2)) :: temp
COMPLEX(DPC) :: w,wp                     Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,1)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourcol')
n2=n/2
j=n2
  This is the bit-reversal section of the routine.
do i=1,n-2
    if (j > i) call swap(data(j+1,:),data(i+1,:))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
  Here begins the Danielson-Lanczos section of the routine.
do                                       Outer loop executed log2 N times.
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)              Initialize for the trigonometric recurrence.
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax                          Here are the two nested inner loops.
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(j,:)            This is the Danielson-Lanczos formula.
            data(j,:)=data(i,:)-temp
            data(i,:)=data(i,:)+temp
        end do
        w=w*wp+w                         Trigonometric recurrence.
    end do
    mmax=istep
end do
END SUBROUTINE fourcol
```

```
SUBROUTINE fourcol_3d(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

If `isign` is input as 1, replaces each first-index section (constant second and third indices) of `data(1:`N`,1:`M`,1:`L`)` by its discrete Fourier transform (transform on first index); or replaces each first-index section of `data` by N times its inverse discrete Fourier transform, if `isign` is input as -1. N must be an integer power of 2. Parallelism is $M \times L$-fold on the second and third indices of `data`.

```
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,2),size(data,3)) :: temp
COMPLEX(DPC) :: w,wp                     Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,1)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourcol_3d')
n2=n/2
```

```
j=n2
  This is the bit-reversal section of the routine.
do i=1,n-2
    if (j > i) call swap(data(j+1,:,:),data(i+1,:,:))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
  Here begins the Danielson-Lanczos section of the routine.
do                                  Outer loop executed log_2 N times.
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)         Initialize for the trigonometric recurrence.
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax                     Here are the two nested inner loops.
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(j,:,:)     This is the Danielson-Lanczos formula.
            data(j,:,:)=data(i,:,:)-temp
            data(i,:,:)=data(i,:,:)+temp
        end do
        w=w*wp+w                    Trigonometric recurrence.
    end do
    mmax=istep
end do
END SUBROUTINE fourcol_3d
```

⋆ ⋆ ⋆

Here now are implementations of the method of §22.4 for the FFT of one-dimensional single- and double-precision complex arrays:

```
SUBROUTINE four1_sp(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
    Replaces a complex array data by its discrete Fourier transform, if isign is input as 1;
    or replaces data by its inverse discrete Fourier transform times the size of data, if isign
    is input as -1. The size of data must be an integer power of 2. Parallelism is achieved
    by internally reshaping the input array to two dimensions. (Use this version if fourrow is
    faster than fourcol on your machine.)
COMPLEX(SPC), DIMENSION(:,:), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_sp')
  Find dimensions as close to square as possible, allocate space, and reshape the input array.
m1=2**ceiling(0.5_sp*log(real(n,sp))/0.693147_sp)
m2=n/m1
allocate(dat(m1,m2),theta(m1),w(m1),wp(m1),temp(m2,m1))
dat=reshape(data,shape(dat))
call fourrow(dat,isign)             Transform on second index.
```

```
theta=arth(0,isign,m1)*TWOPI_D/n       Set up recurrence.
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
w=cmplx(1.0_dp,0.0_dp,kind=dpc)
do j=2,m2                              Multiply by the extra phase factor.
    w=w*wp+w
    dat(:,j)=dat(:,j)*w
end do
temp=transpose(dat)                    Transpose, and transform on (original) first in-
call fourrow(temp,isign)                   dex.
data=reshape(temp,shape(data))         Reshape the result back to one dimension.
deallocate(dat,w,wp,theta,temp)
END SUBROUTINE four1_sp
```

```
SUBROUTINE four1_dp(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(DPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(DPC), DIMENSION(:,:), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_dp')
m1=2**ceiling(0.5_sp*log(real(n,sp))/0.693147_sp)
m2=n/m1
allocate(dat(m1,m2),theta(m1),w(m1),wp(m1),temp(m2,m1))
dat=reshape(data,shape(dat))
call fourrow(dat,isign)
theta=arth(0,isign,m1)*TWOPI_D/n
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
w=cmplx(1.0_dp,0.0_dp,kind=dpc)
do j=2,m2
    w=w*wp+w
    dat(:,j)=dat(:,j)*w
end do
temp=transpose(dat)
call fourrow(temp,isign)
data=reshape(temp,shape(data))
deallocate(dat,w,wp,theta,temp)
END SUBROUTINE four1_dp
```

The above routines use `fourrow` exclusively, on the assumption that it is faster than its sibling `fourcol`. When that is the case (as we typically find), it is likely that `four1_sp` is also faster than Volume 1's scalar `four1`. The reason, on scalar machines, is that `fourrow`'s parallelism is taking better advantage of cache memory locality.

If `fourrow` is *not* faster than `fourcol` on your machine, then you should instead try the following alternative FFT version that uses `fourcol` only.

```
SUBROUTINE four1_alt(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : fourcol
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
    Replaces a complex array data by its discrete Fourier transform, if isign is input as 1; or
    replaces data by its inverse discrete Fourier transform times the size of data, if isign is
```

input as -1. The size of `data` must be an integer power of 2. Parallelism is achieved by internally reshaping the input array to two dimensions. (Use this version *only* if `fourcol` is faster than `fourrow` on your machine.)

```
COMPLEX(SPC), DIMENSION(:,:), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_alt')
```

Find dimensions as close to square as possible, allocate space, and reshape the input array.

```
m1=2**ceiling(0.5_sp*log(real(n,sp))/0.693147_sp)
m2=n/m1
allocate(dat(m1,m2),theta(m1),w(m1),wp(m1),temp(m2,m1))
dat=reshape(data,shape(dat))
temp=transpose(dat)                 Transpose and transform on (original) second in-
call fourcol(temp,isign)               dex.
theta=arth(0,isign,m1)*TWOPI_D/n    Set up recurrence.
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
w=cmplx(1.0_dp,0.0_dp,kind=dpc)
do j=2,m2                           Multiply by the extra phase factor.
    w=w*wp+w
    temp(j,:)=temp(j,:)*w
end do
dat=transpose(temp)                 Transpose, and transform on (original) first in-
call fourcol(dat,isign)                dex.
temp=transpose(dat)                 Transpose and then reshape the result back to
data=reshape(temp,shape(data))         one dimension.
deallocate(dat,w,wp,theta,temp)
END SUBROUTINE four1_alt
```

⋆ ⋆ ⋆

With all the machinery of `fourrow` and `fourcol`, two-dimensional FFTs are extremely straightforward. Again there is an alternative version provided in case your hardware favors `fourcol` (which would be, we think, unusual).

```
SUBROUTINE four2(data,isign)
USE nrtype
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a 2-d complex array `data` by its discrete 2-d Fourier transform, if `isign` is input as 1; or replaces `data` by its inverse 2-d discrete Fourier transform times the product of its two sizes, if `isign` is input as -1. Both of `data`'s sizes must be integer powers of 2 (this is checked for in `fourrow`). Parallelism is by use of `fourrow`.

```
COMPLEX(SPC), DIMENSION(size(data,2),size(data,1)) :: temp
call fourrow(data,isign)        Transform in second dimension.
temp=transpose(data)            Tranpose.
call fourrow(temp,isign)        Transform in (original) first dimension.
data=transpose(temp)            Transpose into data.
END SUBROUTINE four2
```

```
SUBROUTINE four2_alt(data,isign)
USE nrtype
USE nr, ONLY : fourcol
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
    Replaces a 2-d complex array data by its discrete 2-d Fourier transform, if isign is input
    as 1; or replaces data by its inverse 2-d discrete Fourier transform times the product of
    its two sizes, if isign is input as -1. Both of data's sizes must be integer powers of 2
    (this is checked for in fourcol). Parallelism is by use of fourcol. (Use this version only
    if fourcol is faster than fourrow on your machine.)
COMPLEX(SPC), DIMENSION(size(data,2),size(data,1)) :: temp
temp=transpose(data)            Tranpose.
call fourcol(temp,isign)        Transform in (original) second dimension.
data=transpose(temp)            Transpose.
call fourcol(data,isign)        Transform in (original) first dimension.
END SUBROUTINE four2_alt
```

⋆ ⋆ ⋆

Most of the remaining routines in this chapter simply call one or another of the above FFT routines, with a small amount of auxiliary computation, so they are fairly straightforward conversions from their Volume 1 counterparts.

```
SUBROUTINE twofft(data1,data2,fft1,fft2)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: fft1,fft2
    Given two real input arrays data1 and data2 of length N, this routine calls four1 and
    returns two complex output arrays, fft1 and fft2, each of complex length N, that contain
    the discrete Fourier transforms of the respective data arrays. N must be an integer power
    of 2.
INTEGER(I4B) :: n,n2
COMPLEX(SPC), PARAMETER :: C1=(0.5_sp,0.0_sp), C2=(0.0_sp,-0.5_sp)
COMPLEX, DIMENSION(size(data1)/2+1) :: h1,h2
n=assert_eq(size(data1),size(data2),size(fft1),size(fft2),'twofft')
call assert(iand(n,n-1)==0, 'n must be a power of 2 in twofft')
fft1=cmplx(data1,data2,kind=spc)        Pack the two real arrays into one complex array.
call four1(fft1,1)                      Transform the complex array.
fft2(1)=cmplx(aimag(fft1(1)),0.0_sp,kind=spc)
fft1(1)=cmplx(real(fft1(1)),0.0_sp,kind=spc)
n2=n/2+1
h1(2:n2)=C1*(fft1(2:n2)+conjg(fft1(n:n2:-1)))       Use symmetries to separate the
h2(2:n2)=C2*(fft1(2:n2)-conjg(fft1(n:n2:-1)))           two transforms.
fft1(2:n2)=h1(2:n2)                     Ship them out in two complex arrays.
fft1(n:n2:-1)=conjg(h1(2:n2))
fft2(2:n2)=h2(2:n2)
fft2(n:n2:-1)=conjg(h2(2:n2))
END SUBROUTINE twofft
```

⋆ ⋆ ⋆

```
SUBROUTINE realft_sp(data,isign,zdata)
USE nrtype; USE nrutil, ONLY : assert,assert_eq,zroots_unity
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(SPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
```

When isign $= 1$, calculates the Fourier transform of a set of N real-valued data points, input in the array data. If the optional argument zdata is not present, the data are replaced by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned as elements data(1) and data(2), respectively. If the complex array zdata of length $N/2$ is present, data is unchanged and the transform is returned in zdata. N must be a power of 2. If isign $= -1$, this routine calculates the inverse transform of a complex data array if it is the transform of real data. (Result in this case must be multiplied by $2/N$.) The data can be supplied either in data, with zdata absent, or in zdata.

```
INTEGER(I4B) :: n,ndum,nh,nq
COMPLEX(SPC), DIMENSION(size(data)/4) :: w
COMPLEX(SPC), DIMENSION(size(data)/4-1) :: h1,h2
COMPLEX(SPC), DIMENSION(:), POINTER :: cdata      Used for internal complex computa-
COMPLEX(SPC) :: z                                     tions.
REAL(SP) :: c1=0.5_sp,c2
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in realft_sp')
nh=n/2
nq=n/4
if (present(zdata)) then
    ndum=assert_eq(n/2,size(zdata),'realft_sp')
    cdata=>zdata                                  Use zdata as cdata.
    if (isign == 1) cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
else
    allocate(cdata(n/2))                          Have to allocate storage ourselves.
    cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
end if
if (isign == 1) then
    c2=-0.5_sp
    call four1(cdata,+1)                          The forward transform is here.
else                                              Otherwise set up for an inverse trans-
    c2=0.5_sp                                         form.
end if
w=zroots_unity(sign(n,isign),n/4)
w=cmplx(-aimag(w),real(w),kind=spc)
h1=c1*(cdata(2:nq)+conjg(cdata(nh:nq+2:-1)))      The two separate transforms are sep-
h2=c2*(cdata(2:nq)-conjg(cdata(nh:nq+2:-1)))          arated out of cdata.
```

Next they are recombined to form the true transform of the original real data:

```
cdata(2:nq)=h1+w(2:nq)*h2
cdata(nh:nq+2:-1)=conjg(h1-w(2:nq)*h2)
z=cdata(1)                                        Squeeze the first and last data to-
if (isign == 1) then                                  gether to get them all within the
    cdata(1)=cmplx(real(z)+aimag(z),real(z)-aimag(z),kind=spc)      original array.
else
    cdata(1)=cmplx(c1*(real(z)+aimag(z)),c1*(real(z)-aimag(z)),kind=spc)
    call four1(cdata,-1)                          This is the inverse transform for the
end if                                                case isign=-1.
if (present(zdata)) then                          Ship out answer in data if required.
    if (isign /= 1) then
        data(1:n-1:2)=real(cdata)
        data(2:n:2)=aimag(cdata)
    end if
else
    data(1:n-1:2)=real(cdata)
    data(2:n:2)=aimag(cdata)
    deallocate(cdata)
end if
```

```
END SUBROUTINE realft_sp
```

```
SUBROUTINE realft_dp(data,isign,zdata)
USE nrtype; USE nrutil, ONLY : assert,assert_eq,zroots_unity
USE nr, ONLY : four1
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(DPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
INTEGER(I4B) :: n,ndum,nh,nq
COMPLEX(DPC), DIMENSION(size(data)/4) :: w
COMPLEX(DPC), DIMENSION(size(data)/4-1) :: h1,h2
COMPLEX(DPC), DIMENSION(:), POINTER :: cdata
COMPLEX(DPC) :: z
REAL(DP) :: c1=0.5_dp,c2
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in realft_dp')
nh=n/2
nq=n/4
if (present(zdata)) then
    ndum=assert_eq(n/2,size(zdata),'realft_dp')
    cdata=>zdata
    if (isign == 1) cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
else
    allocate(cdata(n/2))
    cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
end if
if (isign == 1) then
    c2=-0.5_dp
    call four1(cdata,+1)
else
    c2=0.5_dp
end if
w=zroots_unity(sign(n,isign),n/4)
w=cmplx(-aimag(w),real(w),kind=dpc)
h1=c1*(cdata(2:nq)+conjg(cdata(nh:nq+2:-1)))
h2=c2*(cdata(2:nq)-conjg(cdata(nh:nq+2:-1)))
cdata(2:nq)=h1+w(2:nq)*h2
cdata(nh:nq+2:-1)=conjg(h1-w(2:nq)*h2)
z=cdata(1)
if (isign == 1) then
    cdata(1)=cmplx(real(z)+aimag(z),real(z)-aimag(z),kind=dpc)
else
    cdata(1)=cmplx(c1*(real(z)+aimag(z)),c1*(real(z)-aimag(z)),kind=dpc)
    call four1(cdata,-1)
end if
if (present(zdata)) then
    if (isign /= 1) then
        data(1:n-1:2)=real(cdata)
        data(2:n:2)=aimag(cdata)
    end if
else
    data(1:n-1:2)=real(cdata)
    data(2:n:2)=aimag(cdata)
    deallocate(cdata)
end if
END SUBROUTINE realft_dp
```

⋆ ⋆ ⋆

```
SUBROUTINE sinft(y)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
```

Calculates the sine transform of a set of N real-valued data points stored in array y. The number N must be a power of 2. On exit y is replaced by its transform. This program, without changes, also calculates the inverse sine transform, but in this case the output array should be multiplied by $2/N$.

```
REAL(SP), DIMENSION(size(y)/2+1) :: wi
REAL(SP), DIMENSION(size(y)/2) :: y1,y2
INTEGER(I4B) :: n,nh
n=size(y)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in sinft')
nh=n/2
wi=aimag(zroots_unity(n+n,nh+1))       Calculate the sine for the auxiliary array.
y(1)=0.0
y1=wi(2:nh+1)*(y(2:nh+1)+y(n:nh+1:-1))
  Construct the two pieces of the auxiliary array.
y2=0.5_sp*(y(2:nh+1)-y(n:nh+1:-1))     Put them together to make the auxiliary ar-
y(2:nh+1)=y1+y2                           ray.
y(n:nh+1:-1)=y1-y2
call realft(y,+1)                      Transform the auxiliary array.
y(1)=0.5_sp*y(1)                       Initialize the sum used for odd terms.
y(2)=0.0
y1=cumsum(y(1:n-1:2))                  Odd terms are determined by this running sum.
y(1:n-1:2)=y(2:n:2)                    Even terms in the transform are determined di-
y(2:n:2)=y1                               rectly.
END SUBROUTINE sinft
```

```
SUBROUTINE cosft1(y)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
```

Calculates the cosine transform of a set of $N+1$ real-valued data points y. The transformed data replace the original data in array y. N must be a power of 2. This program, without changes, also calculates the inverse cosine transform, but in this case the output array should be multiplied by $2/N$.

```
COMPLEX(SPC), DIMENSION((size(y)-1)/2) :: w
REAL(SP), DIMENSION((size(y)-1)/2-1) :: y1,y2
REAL(SP) :: summ
INTEGER(I4B) :: n,nh
n=size(y)-1
call assert(iand(n,n-1)==0, 'n must be a power of 2 in cosft1')
nh=n/2
w=zroots_unity(n+n,nh)
summ=0.5_sp*(y(1)-y(n+1))
y(1)=0.5_sp*(y(1)+y(n+1))
y1=0.5_sp*(y(2:nh)+y(n:nh+2:-1))       Construct the two pieces of the auxiliary array.
y2=y(2:nh)-y(n:nh+2:-1)
summ=summ+sum(real(w(2:nh))*y2)        Carry along this sum for later use in unfolding
y2=y2*aimag(w(2:nh))                      the transform.
y(2:nh)=y1-y2                          Calculate the auxiliary function.
y(n:nh+2:-1)=y1+y2
call realft(y(1:n),1)                  Calculate the transform of the auxiliary function.
y(n+1)=y(2)
y(2)=summ                              summ is the value of F_1 in equation (12.3.21).
y(2:n:2)=cumsum(y(2:n:2))              Equation (12.3.20).
END SUBROUTINE cosft1
```

```
SUBROUTINE cosft2(y,isign)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
INTEGER(I4B), INTENT(IN) :: isign
```

Calculates the "staggered" cosine transform of a set of N real-valued data points y. The transformed data replace the original data in array y. N must be a power of 2. Set isign to $+1$ for a transform, and to -1 for an inverse transform. For an inverse transform, the output array should be multiplied by $2/N$.

```
COMPLEX(SPC), DIMENSION(size(y)) :: w
REAL(SP), DIMENSION(size(y)/2) :: y1,y2
REAL(SP) :: ytemp
INTEGER(I4B) :: n,nh
n=size(y)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in cosft2')
nh=n/2
w=zroots_unity(4*n,n)
if (isign == 1) then                        Forward transform.
    y1=0.5_sp*(y(1:nh)+y(n:nh+1:-1))        Calculate the auxiliary function.
    y2=aimag(w(2:n:2))*(y(1:nh)-y(n:nh+1:-1))
    y(1:nh)=y1+y2
    y(n:nh+1:-1)=y1-y2
    call realft(y,1)                        Calculate transform of the auxiliary function.
    y1(1:nh-1)=y(3:n-1:2)*real(w(3:n-1:2)) &      Even terms.
        -y(4:n:2)*aimag(w(3:n-1:2))
    y2(1:nh-1)=y(4:n:2)*real(w(3:n-1:2)) &
        +y(3:n-1:2)*aimag(w(3:n-1:2))
    y(3:n-1:2)=y1(1:nh-1)
    y(4:n:2)=y2(1:nh-1)
    ytemp=0.5_sp*y(2)                       Initialize recurrence for odd terms with 1/2 R_N/2.
    y(n-2:2:-2)=cumsum(y(n:4:-2),ytemp)         Recurrence for odd terms.
    y(n)=ytemp
else if (isign == -1) then                  Inverse transform.
    ytemp=y(n)
    y(4:n:2)=y(2:n-2:2)-y(4:n:2)            Form difference of odd terms.
    y(2)=2.0_sp*ytemp
    y1(1:nh-1)=y(3:n-1:2)*real(w(3:n-1:2)) &      Calculate R_k and I_k.
        +y(4:n:2)*aimag(w(3:n-1:2))
    y2(1:nh-1)=y(4:n:2)*real(w(3:n-1:2)) &
        -y(3:n-1:2)*aimag(w(3:n-1:2))
    y(3:n-1:2)=y1(1:nh-1)
    y(4:n:2)=y2(1:nh-1)
    call realft(y,-1)
    y1=y(1:nh)+y(n:nh+1:-1)                 Invert auxiliary array.
    y2=(0.5_sp/aimag(w(2:n:2)))*(y(1:nh)-y(n:nh+1:-1))
    y(1:nh)=0.5_sp*(y1+y2)
    y(n:nh+1:-1)=0.5_sp*(y1-y2)
end if
END SUBROUTINE cosft2
```

⋆ ⋆ ⋆

```
SUBROUTINE four3(data,isign)
USE nrtype
USE nr, ONLY : fourrow_3d
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a 3-d complex array data by its discrete 3-d Fourier transform, if isign is input as 1; or replaces data by its inverse 3-d discrete Fourier transform times the product of its

```
    three sizes, if isign is input as −1. All three of data's sizes must be integer powers of 2
    (this is checked for in fourrow_3d). Parallelism is by use of fourrow_3d.
COMPLEX(SPC), DIMENSION(:,:,:), ALLOCATABLE :: dat2,dat3
call fourrow_3d(data,isign)                   Transform in third dimension.
allocate(dat2(size(data,2),size(data,3),size(data,1)))
dat2=reshape(data,shape=shape(dat2),order=(/3,1,2/))     Transpose.
call fourrow_3d(dat2,isign)                   Transform in (original) first dimension.
allocate(dat3(size(data,3),size(data,1),size(data,2)))
dat3=reshape(dat2,shape=shape(dat3),order=(/3,1,2/))     Transpose.
deallocate(dat2)
call fourrow_3d(dat3,isign)                   Transform in (original) second dimension.
data=reshape(dat3,shape=shape(data),order=(/3,1,2/))     Transpose back to output or-
deallocate(dat3)                                             der.
END SUBROUTINE four3
```

f90 The `reshape` intrinsic, used with an `order=` parameter, is the multidimensional generalization of the two-dimensional `transpose` operation. The line

```
dat2=reshape(data,shape=shape(dat2),order=(/3,1,2/))
```

is equivalent to the do-loop

```
do j=1,size(data,1)
   dat2(:,:,j)=data(j,:,:)
end do
```

Incidentally, we have found some Fortran 90 compilers that (for scalar machines) are significantly *slower* executing the `reshape` than executing the equivalent do-loop. This, of course, shouldn't happen, since the `reshape` basically *is* an implicit do-loop. If you find such inefficient behavior on your compiler, you should report it as a bug to your compiler vendor! (Only thus will Fortran 90 compilers be brought to mature states of efficiency.)

```
SUBROUTINE four3_alt(data,isign)
USE nrtype
USE nr, ONLY : fourcol_3d
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
    Replaces a 3-d complex array data by its discrete 2-d Fourier transform, if isign is input
    as 1; or replaces data by its inverse 3-d discrete Fourier transform times the product of
    its three sizes, if isign is input as −1. All three of data's sizes must be integer powers
    of 2 (this is checked for in fourcol_3d). Parallelism is by use of fourcol_3d. (Use this
    version only if fourcol_3d is faster than fourrow_3d on your machine.)
COMPLEX(SPC), DIMENSION(:,:,:), ALLOCATABLE :: dat2,dat3
call fourcol_3d(data,isign)                   Transform in first dimension.
allocate(dat2(size(data,2),size(data,3),size(data,1)))
dat2=reshape(data,shape=shape(dat2),order=(/3,1,2/))     Transpose.
call fourcol_3d(dat2,isign)                   Transform in (original) second dimension.
allocate(dat3(size(data,3),size(data,1),size(data,2)))
dat3=reshape(dat2,shape=shape(dat3),order=(/3,1,2/))     Transpose.
deallocate(dat2)
call fourcol_3d(dat3,isign)                   Transform in (original) third dimension.
data=reshape(dat3,shape=shape(data),order=(/3,1,2/))     Transpose back to output or-
deallocate(dat3)                                             der.
END SUBROUTINE four3_alt
```

Note that `four3` uses `fourrow_3d`, the three-dimensional counterpart of `fourrow`, while `four3_alt` uses `fourcol_3d`, the three-dimensional counterpart of `fourcol`. You may want to time these programs to see which is faster on your machine.

⋆ ⋆ ⋆

In Volume 1, a single routine named `rlft3` was able to serve both as a three-dimensional real FFT, and as a two-dimensional real FFT. The trick is that the Fortran 77 version doesn't care whether the input array `data` is dimensioned as two- or three-dimensional. Fortran 90 is not so indifferent, and better programming practice is to have two separate versions of the algorithm:

```
SUBROUTINE rlft2(data,spec,speq,isign)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four2
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: data
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: spec
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: speq
INTEGER(I4B), INTENT(IN) :: isign
```

Given a two-dimensional real array `data(1:`M`,1:`N`)`, this routine returns (for `isign=1`) the complex fast Fourier transform as two complex arrays: On output, `spec(1:`$M/2$`,1:`N`)` contains the zero and positive frequency values of the first frequency component, while `speq(1:`N`)` contains the Nyquist critical frequency values of the first frequency component. The second frequency components are stored for zero, positive, and negative frequencies, in standard wrap-around order. For `isign=-1`, the inverse transform (times $M \times N/2$ as a constant multiplicative factor) is performed, with output `data` deriving from input `spec` and `speq`. For inverse transforms on data not generated first by a forward transform, make sure the complex input data array satisfies property (12.5.2). The size of all arrays must always be integer powers of 2.

```
INTEGER :: i1,j1,nn1,nn2
REAL(DP) :: theta
COMPLEX(SPC) :: c1=(0.5_sp,0.0_sp),c2,h1,h2,w
COMPLEX(SPC), DIMENSION(size(data,2)-1) :: h1a,h2a
COMPLEX(DPC) :: ww,wp
nn1=assert_eq(size(data,1),2*size(spec,1),'rlft2: nn1')
nn2=assert_eq(size(data,2),size(spec,2),size(speq),'rlft2: nn2')
call assert(iand((/nn1,nn2/),(/nn1,nn2/)-1)==0, &
    'dimensions must be powers of 2 in rlft2')
c2=cmplx(0.0_sp,-0.5_sp*isign,kind=spc)
theta=TWOPI_D/(isign*nn1)
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=spc)
if (isign == 1) then                        Case of forward transform.
    spec(:,:)=cmplx(data(1:nn1:2,:),data(2:nn1:2,:),kind=spc)
    call four2(spec,isign)                  Here is where most all of the compute time
    speq=spec(1,:)                              is spent.
end if
h1=c1*(spec(1,1)+conjg(speq(1)))
h1a=c1*(spec(1,2:nn2)+conjg(speq(nn2:2:-1)))
h2=c2*(spec(1,1)-conjg(speq(1)))
h2a=c2*(spec(1,2:nn2)-conjg(speq(nn2:2:-1)))
spec(1,1)=h1+h2
spec(1,2:nn2)=h1a+h2a
speq(1)=conjg(h1-h2)
speq(nn2:2:-1)=conjg(h1a-h2a)
ww=cmplx(1.0_dp,0.0_dp,kind=dpc)            Initialize trigonometric recurrence.
do i1=2,nn1/4+1
    j1=nn1/2-i1+2                           Corresponding negative frequency.
    ww=ww*wp+ww                             Do the trig recurrence.
    w=ww
    h1=c1*(spec(i1,1)+conjg(spec(j1,1)))        Equation (12.3.5).
    h1a=c1*(spec(i1,2:nn2)+conjg(spec(j1,nn2:2:-1)))
```

```
    h2=c2*(spec(i1,1)-conjg(spec(j1,1)))
    h2a=c2*(spec(i1,2:nn2)-conjg(spec(j1,nn2:2:-1)))
    spec(i1,1)=h1+w*h2
    spec(i1,2:nn2)=h1a+w*h2a
    spec(j1,1)=conjg(h1-w*h2)
    spec(j1,nn2:2:-1)=conjg(h1a-w*h2a)
end do
if (isign == -1) then                       Case of reverse transform.
    call four2(spec,isign)
    data(1:nn1:2,:)=real(spec)
    data(2:nn1:2,:)=aimag(spec)
end if
END SUBROUTINE rlft2
```

`call assert(iand((/nn1,nn2/),(/nn1,nn2/)-1)==0 ...` Here an overloaded version of `assert` that takes vector arguments is used to check that each dimension is a power of 2. Note that `iand` acts element-by-element on an array.

```
SUBROUTINE rlft3(data,spec,speq,isign)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four3
REAL(SP), DIMENSION(:,:,:), INTENT(INOUT) :: data
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: spec
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: speq
INTEGER(I4B), INTENT(IN) :: isign
```

Given a three-dimensional real array `data(1:`L`,1:`M`,1:`N`)`, this routine returns (for `isign=1`) the complex Fourier transform as two complex arrays: On output, the zero and positive frequency values of the first frequency component are in `spec(1:`$L/2$`,1:`M`,1:`N`)`, while `speq(1:`M`,1:`N`)` contains the Nyquist critical frequency values of the first frequency component. The second and third frequency components are stored for zero, positive, and negative frequencies, in standard wrap-around order. For `isign=-1`, the inverse transform (times $L \times M \times N/2$ as a constant multiplicative factor) is performed, with output `data` deriving from input `spec` and `speq`. For inverse transforms on data not generated first by a forward transform, make sure the complex input data array satisfies property (12.5.2). The size of all arrays must always be integer powers of 2.

```
INTEGER :: i1,i3,j1,j3,nn1,nn2,nn3
REAL(DP) :: theta
COMPLEX(SPC) :: c1=(0.5_sp,0.0_sp),c2,h1,h2,w
COMPLEX(SPC), DIMENSION(size(data,2)-1) :: h1a,h2a
COMPLEX(DPC) :: ww,wp
c2=cmplx(0.0_sp,-0.5_sp*isign,kind=spc)
nn1=assert_eq(size(data,1),2*size(spec,1),'rlft2: nn1')
nn2=assert_eq(size(data,2),size(spec,2),size(speq,1),'rlft2: nn2')
nn3=assert_eq(size(data,3),size(spec,3),size(speq,2),'rlft2: nn3')
call assert(iand((/nn1,nn2,nn3/),(/nn1,nn2,nn3/)-1)==0, &
    'dimensions must be powers of 2 in rlft3')
theta=TWOPI_D/(isign*nn1)
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
if (isign == 1) then                        Case of forward transform.
    spec(:,:,:)=cmplx(data(1:nn1:2,:,:),data(2:nn1:2,:,:),kind=spc)
    call four3(spec,isign)                  Here is where most all of the compute time
    speq=spec(1,:,:)                            is spent.
end if
do i3=1,nn3
    j3=1
    if (i3 /= 1) j3=nn3-i3+2
    h1=c1*(spec(1,1,i3)+conjg(speq(1,j3)))
    h1a=c1*(spec(1,2:nn2,i3)+conjg(speq(nn2:2:-1,j3)))
    h2=c2*(spec(1,1,i3)-conjg(speq(1,j3)))
    h2a=c2*(spec(1,2:nn2,i3)-conjg(speq(nn2:2:-1,j3)))
    spec(1,1,i3)=h1+h2
    spec(1,2:nn2,i3)=h1a+h2a
```

```
    speq(1,j3)=conjg(h1-h2)
    speq(nn2:2:-1,j3)=conjg(h1a-h2a)
    ww=cmplx(1.0_dp,0.0_dp,kind=dpc)        Initialize trigonometric recurrence.
    do i1=2,nn1/4+1
        j1=nn1/2-i1+2                       Corresponding negative frequency.
        ww=ww*wp+ww                         Do the trig recurrence.
        w=ww
        h1=c1*(spec(i1,1,i3)+conjg(spec(j1,1,j3)))      Equation (12.3.5).
        h1a=c1*(spec(i1,2:nn2,i3)+conjg(spec(j1,nn2:2:-1,j3)))
        h2=c2*(spec(i1,1,i3)-conjg(spec(j1,1,j3)))
        h2a=c2*(spec(i1,2:nn2,i3)-conjg(spec(j1,nn2:2:-1,j3)))
        spec(i1,1,i3)=h1+w*h2
        spec(i1,2:nn2,i3)=h1a+w*h2a
        spec(j1,1,j3)=conjg(h1-w*h2)
        spec(j1,nn2:2:-1,j3)=conjg(h1a-w*h2a)
    end do
end do
if (isign == -1) then                       Case of reverse transform.
    call four3(spec,isign)
    data(1:nn1:2,:,:)=real(spec)
    data(2:nn1:2,:,:)=aimag(spec)
end if
END SUBROUTINE rlft3
```

⋆ ⋆ ⋆

Referring back to the discussion of parallelism, §22.4, that led to four1's implementation with $\sqrt{N}$ parallelism, you might wonder whether Fortran 90 provides sufficiently powerful high-level constructs to enable an FFT routine with N-fold parallelism. The answer is, "*It does*, but you wouldn't want to use them!" Access to arbitrary interprocessor communication in Fortran 90 is through the mechanism of the "vector subscript" (one-dimensional array of indices in arbitrary order). When a vector subscript is on the right-hand side of an assignment statement, the operation performed is effectively a "gather"; when it is on the left-hand side, the operation is effectively a "scatter."

It is quite possible to write the classic FFT algorithm in terms of gather and scatter operations. In fact, we do so now. The problem is efficiency: The computations involved in constructing the vector subscripts for the scatter/gather operations, and the actual scatter/gather operations themselves, tend to swamp the underlying very lean FFT algorithm. The result is very slow, though theoretically perfectly parallelizable, code. Since small-scale parallel (SSP) machines can saturate their processors with $\sqrt{N}$ parallelism, while massively multiprocessor (MMP) machines inevitably come with architecture-optimized FFT library calls, there is really no niche for these routines, except as pedagogical demonstrations. We give here a one-dimensional routine, and also an arbitrary-dimensional routine modeled on Volume 1's fourn. Note the complete absence of do-loops of size N; the loops that remain are over $\log N$ stages, or over the number of dimensions.

```
SUBROUTINE four1_gather(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a complex array data by its discrete Fourier transform, if isign is input as 1; or replaces data by size(data) times its inverse discrete Fourier transform, if isign is input as -1. The size of data must be an integer power of 2. This routine demonstrates coding the FFT algorithm in high-level Fortran 90 constructs. Generally the result is *very* *much slower* than library routines coded for specific architectures, and also *significantly slower* than the parallelization-by-rows method used in the routine `four1`.

```
INTEGER(I4B) :: n,n2,m,mm
REAL(DP) :: theta
COMPLEX(SPC) :: wp
INTEGER(I4B), DIMENSION(size(data)) :: jarr
INTEGER(I4B), DIMENSION(:), ALLOCATABLE :: jrev
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: wtab,dtemp
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_gather')
if (n <= 1) RETURN
allocate(jrev(n))                       Begin bit-reversal section of the routine.
jarr=arth(0,1,n)
jrev=0
n2=n/2
m=n2
do                                      Construct an array of pointers from an index
    where (iand(jarr,1) /= 0) jrev=jrev+m     to its bit-reverse.
    jarr=jarr/2
    m=m/2
    if (m == 0) exit
end do
data=data(jrev+1)                       Move all data to bit-reversed location by a
deallocate(jrev)                            single gather/scatter.
allocate(dtemp(n),wtab(n2))             Begin Danielson-Lanczos section of the rou-
jarr=arth(0,1,n)                            tine.
m=1
mm=n2
wtab(1)=(1.0_sp,0.0_sp)                 Seed the roots-of-unity table.
do                                      Outer loop executed log_2 N times.
    where (iand(jarr,m) /= 0)
          The basic idea is to address the correct root-of-unity for each Danielson-Lanczos
          multiplication by tricky bit manipulations.
        dtemp=data*wtab(mm*iand(jarr,m-1)+1)
        data=eoshift(data,-m)-dtemp     This is half of Danielson-Lanczos.
    elsewhere
        data=data+eoshift(dtemp,m)      This is the other half. The referenced ele-
    end where                               ments of dtemp will have been set in the
    m=m*2                                   where clause.
    if (m >= n) exit
    mm=mm/2
    theta=PI_D/(isign*m)                Ready for trigonometry?
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2, sin(theta),kind=spc)
      Add entries to the table for the next iteration.
    wtab(mm+1:n2:2*mm)=wtab(1:n2-mm:2*mm)*wp+wtab(1:n2-mm:2*mm)
end do
deallocate(dtemp,wtab)
END SUBROUTINE four1_gather
```

```
SUBROUTINE fourn_gather(data,nn,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), DIMENSION(:) :: nn
INTEGER(I4B), INTENT(IN) :: isign
```

For `data` a one-dimensional complex array containing the values (in Fortran normal ordering) of an M-dimensional complex arrray, this routine replaces `data` by its M-dimensional discrete Fourier transform, if `isign` is input as 1. `nn(1:`M`)` is an integer array containing the lengths of each dimension (number of complex values), each of which must be a power of 2. If `isign` is input as -1, `data` is replaced by its inverse transform times the product of the lengths of all dimensions. This routine demonstrates coding the multidimensional FFT algorithm in high-level Fortran 90 constructs. Generally the result is *very much slower* than

```
    library routines coded for specific architectures, and significantly slower than routines four2
    and four3 for the two- and three-dimensional cases.
INTEGER(I4B), DIMENSION(:), ALLOCATABLE :: jarr
INTEGER(I4B) :: ndim,idim,ntot,nprev,n,n2,msk0,msk1,msk2,m,mm,mn
REAL(DP) :: theta
COMPLEX(SPC) :: wp
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: wtab,dtemp
call assert(iand(nn,nn-1)==0, &
    'each dimension must be a power of 2 in fourn_gather')
ndim=size(nn)
ntot=product(nn)
nprev=1
allocate(jarr(ntot))
do idim=1,ndim                                  Loop over the dimensions.
    jarr=arth(0,1,ntot)                         We begin the bit-reversal section of the
    n=nn(idim)                                      routine.
    n2=n/2
    msk0=nprev
    msk1=nprev*n2
    msk2=msk0+msk1
    do                                          Construct an array of pointers from an
        if (msk1 <= msk0) exit                      index to its bit-reverse.
        where (iand(jarr,msk0) == 0 .neqv. iand(jarr,msk1) == 0) &
            jarr=ieor(jarr,msk2)
        msk0=msk0*2
        msk1=msk1/2
        msk2=msk0+msk1
    end do
    data=data(jarr+1)                           Move all data to bit-reversed location by
    allocate(dtemp(ntot),wtab(n2))                  a single gather/scatter.
      We begin the Danielson-Lanczos section of the routine.
    jarr=iand(n-1,arth(0,1,ntot)/nprev)
    m=1
    mm=n2
    mn=m*nprev
    wtab(1)=(1.0_sp,0.0_sp)                     Seed the roots-of-unity table.
    do                                          This loop executed log_2 N times.
        if (mm == 0) exit
        where (iand(jarr,m) /= 0)
              The basic idea is to address the correct root-of-unity for each Danielson-Lanczos
              multiplication by tricky bit manipulations.
            dtemp=data*wtab(mm*iand(jarr,m-1)+1)
            data=eoshift(data,-mn)-dtemp        This is half of Danielson-Lanczos.
        elsewhere
            data=data+eoshift(dtemp,mn)         This is the other half. The referenced el-
        end where                                   ements of dtemp will have been set
        m=m*2                                       in the where clause.
        if (m >= n) exit
        mn=m*nprev
        mm=mm/2
        theta=PI_D/(isign*m)                    Ready for trigonometry?
        wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
          Add entries to the table for the next iteration.
        wtab(mm+1:n2:2*mm)=wtab(1:n2-mm:2*mm)*wp &
            +wtab(1:n2-mm:2*mm)
    end do
    deallocate(dtemp,wtab)
    nprev=n*nprev
end do
deallocate(jarr)
END SUBROUTINE fourn_gather
```

f90 `call assert(iand(nn,nn-1)==0 ...` Once again the vector version of `assert` is used to test all the dimensions stored in `nn` simultaneously.