Chapter B13. Fourier and Spectral Applications

```
FUNCTION convlv(data,respns,isign)
USE nrtype; USE nrutil, ONLY : assert,nrerror
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
REAL(SP), DIMENSION(:), INTENT(IN) :: respns
INTEGER(I4B), INTENT(IN) :: isign
REAL(SP), DIMENSION(size(data)) :: convlv
```

Convolves or deconvolves a real data set `data` (of length N, including any user-supplied zero padding) with a response function `respns`, stored in wrap-around order in a real array of length $M \leq N$. (M should be an odd integer, N a power of 2.) Wrap-around order means that the first half of the array `respns` contains the impulse response function at positive times, while the second half of the array contains the impulse response function at negative times, counting down from the highest element `respns`(M). On input `isign` is $+1$ for convolution, -1 for deconvolution. The answer is returned as the function `convlv`, an array of length N. `data` has `INTENT(INOUT)` for consistency with `realft`, but is actually unchanged.

```
INTEGER(I4B) :: no2,n,m
COMPLEX(SPC), DIMENSION(size(data)/2) :: tmpd,tmpr
n=size(data)
m=size(respns)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in convlv')
call assert(mod(m,2)==1, 'm must be odd in convlv')
convlv(1:m)=respns(:)                   Put respns in array of length n.
convlv(n-(m-3)/2:n)=convlv((m+3)/2:m)
convlv((m+3)/2:n-(m-1)/2)=0.0           Pad with zeros.
no2=n/2
call realft(data,1,tmpd)                FFT both arrays.
call realft(convlv,1,tmpr)
if (isign == 1) then                    Multiply FFTs to convolve.
    tmpr(1)=cmplx(real(tmpd(1))*real(tmpr(1))/no2, &
        aimag(tmpd(1))*aimag(tmpr(1))/no2, kind=spc)
    tmpr(2:)=tmpd(2:)*tmpr(2:)/no2
else if (isign == -1) then              Divide FFTs to deconvolve.
    if (any(abs(tmpr(2:)) == 0.0) .or. real(tmpr(1)) == 0.0 &
        .or. aimag(tmpr(1)) == 0.0) call nrerror &
        ('deconvolving at response zero in convlv')
    tmpr(1)=cmplx(real(tmpd(1))/real(tmpr(1))/no2, &
        aimag(tmpd(1))/aimag(tmpr(1))/no2, kind=spc)
    tmpr(2:)=tmpd(2:)/tmpr(2:)/no2
else
    call nrerror('no meaning for isign in convlv')
end if
call realft(convlv,-1,tmpr)             Inverse transform back to time domain.
END FUNCTION convlv
```

f90 `tmpr(1)=cmplx(...kind=spc)` The intrinsic function `cmplx` returns a quantity of type default complex unless the `kind` argument is present. It is therefore a good idea always to include this argument. The intrinsic functions `real` and `aimag`, on the other hand, when called with a complex argument, return the same kind as their argument. So it is a good idea *not* to put in a `kind` argment for these. (In fact, `aimag` doesn't allow one.) Don't confuse these situations, regarding complex variables, with the completely unrelated use of `real` to convert a real or integer variable to a real value of specified kind. In this latter case, `kind` should be specified.

⋆ ⋆ ⋆

```
FUNCTION correl(data1,data2)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data1,data2
REAL(SP), DIMENSION(size(data1)) :: correl
```

Computes the correlation of two real data sets `data1` and `data2` of length N (including any user-supplied zero padding). N must be an integer power of 2. The answer is returned as the function `correl`, an array of length N. The answer is stored in wrap-around order, i.e., correlations at increasingly negative lags are in `correl`(N) on down to `correl`($N/2+1$), while correlations at increasingly positive lags are in `correl(1)` (zero lag) on up to `correl`($N/2$). Sign convention of this routine: if `data1` lags `data2`, i.e., is shifted to the right of it, then `correl` will show a peak at positive lags.

```
COMPLEX(SPC), DIMENSION(size(data1)/2) :: cdat1,cdat2
INTEGER(I4B) :: no2,n                          Normalization for inverse FFT.
n=assert_eq(size(data1),size(data2),'correl')
call assert(iand(n,n-1)==0, 'n must be a power of 2 in correl')
no2=n/2
call realft(data1,1,cdat1)                     Transform both data vectors.
call realft(data2,1,cdat2)
cdat1(1)=cmplx(real(cdat1(1))*real(cdat2(1))/no2, &     Multiply to find FFT of their
    aimag(cdat1(1))*aimag(cdat2(1))/no2, kind=spc)          correlation.
cdat1(2:)=cdat1(2:)*conjg(cdat2(2:))/no2
call realft(correl,-1,cdat1)                   Inverse transform gives correlation.
END FUNCTION correl
```

f90 `cdat1(1)=cmplx(...kind=spc)` See just above for why we use the explicit kind type parameter `spc` for `cmplx`, but omit `sp` for `real`.

⋆ ⋆ ⋆

```
SUBROUTINE spctrm(p,k,ovrlap,unit,n_window)
USE nrtype; USE nrutil, ONLY : arth,nrerror
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: p
INTEGER(I4B), INTENT(IN) :: k
LOGICAL(LGT), INTENT(IN) :: ovrlap          True for overlapping segments, false other-
INTEGER(I4B), OPTIONAL, INTENT(IN) :: n_window,unit     wise.
```

Reads data from input unit 9, or if the optional argument `unit` is present, from that input unit. The output is an array `p` of length M that contains the data's power (mean square amplitude) at frequency $(j-1)/2M$ cycles per grid point, for $j = 1, 2, \ldots, M$, based on `(2*k+1)*`M data points (if `ovrlap` is set `.true.`) or `4*k*`M data points (if `ovrlap` is set `.false.`). The number of segments of the data is `2*k` in both cases: The routine calls `four1` `k` times, each call with 2 partitions each of $2M$ real data points. If the optional argument `n_window` is present, the routine uses the Bartlett window, the square window,

or the Welch window for n_window $= 1, 2, 3$ respectively. If n_window is not present, the Bartlett window is used.

```
INTEGER(I4B) :: j,joff,joffn,kk,m,m4,m43,m44,mm,iunit,nn_window
REAL(SP) :: den,facm,facp,sumw
REAL(SP), DIMENSION(2*size(p)) :: w
REAL(SP), DIMENSION(4*size(p)) :: w1
REAL(SP), DIMENSION(size(p)) :: w2
COMPLEX(SPC), DIMENSION(2*size(p)) :: cw1
m=size(p)
if (present(n_window)) then
    nn_window=n_window
else
    nn_window=1
end if
if (present(unit)) then
    iunit=unit
else
    iunit=9
end if
mm=m+m                                  Useful factors.
m4=mm+mm
m44=m4+4
m43=m4+3
den=0.0
facm=m                                  Factors used by the window function.
facp=1.0_sp/m
w1(1:mm)=window(arth(1,1,mm),facm,facp,nn_window)
sumw=dot_product(w1(1:mm),w1(1:mm))     Accumulate the squared sum of the weights.
p(:)=0.0                                Initialize the spectrum to zero.
if (ovrlap) read (iunit,*) (w2(j),j=1,m)    Initialize the "save" half-buffer.
do kk=1,k                               Loop over data segments in groups of two.
    do joff=-1,0,1                      Get two complete segments into workspace.
        if (ovrlap) then
            w1(joff+2:joff+mm:2)=w2(1:m)
            read (iunit,*) (w2(j),j=1,m)
            joffn=joff+mm
            w1(joffn+2:joffn+mm:2)=w2(1:m)
        else
            read (iunit,*) (w1(j),j=joff+2,m4,2)
        end if
    end do
    w=window(arth(1,1,mm),facm,facp,nn_window)      Apply the window to the data.
    w1(2:m4:2)=w1(2:m4:2)*w
    w1(1:m4:2)=w1(1:m4:2)*w
    cw1(1:mm)=cmplx(w1(1:m4:2),w1(2:m4:2),kind=spc)
    call four1(cw1(1:mm),1)             Fourier transform the windowed data.
    w1(1:m4:2)=real(cw1(1:mm))
    w1(2:m4:2)=aimag(cw1(1:mm))
    p(1)=p(1)+w1(1)**2+w1(2)**2         Sum results into previous segments.
    p(2:m)=p(2:m)+w1(4:2*m:2)**2+w1(3:2*m-1:2)**2+&
        w1(m44-4:m44-2*m:-2)**2+w1(m43-4:m43-2*m:-2)**2
    den=den+sumw
end do
p(:)=p(:)/(m4*den)                      Normalize the output.
CONTAINS

FUNCTION window(j,facm,facp,nn_window)
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: j
INTEGER(I4B), INTENT(IN) :: nn_window
REAL(SP), INTENT(IN) :: facm,facp
REAL(SP), DIMENSION(size(j)) :: window
select case(nn_window)
    case(1)
```

```
        window(j)=(1.0_sp-abs(((j-1)-facm)*facp))        Bartlett window.
    case(2)
        window(j)=1.0                                    Square window.
    case(3)
        window(j)=(1.0_sp-(((j-1)-facm)*facp)**2)        Welch window.
    case default
        call nrerror('unimplemented window function in spctrm')
end select
END FUNCTION window
END SUBROUTINE spctrm
```

f90 The Fortran 90 optional argument feature allows us to make unit 9 the default output unit in this routine, but leave the user the option of specifying a different output unit by supplying an actual argument for unit. We also use an optional argument to allow the user the option of overriding the default selection of the Bartlett window function.

FUNCTION window(j,facm,facp,nn_window) In Fortran 77 we coded this as a statement function. Here the internal function is equivalent, but allows full specification of the interface and so is preferred.

⋆ ⋆ ⋆

```
SUBROUTINE memcof(data,xms,d)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: xms
REAL(SP), DIMENSION(:), INTENT(IN) :: data
REAL(SP), DIMENSION(:), INTENT(OUT) :: d
```

Given a real vector data of length N, this routine returns M linear prediction coefficients in a vector d of length M, and returns the mean square discrepancy as xms.

```
INTEGER(I4B) :: k,m,n
REAL(SP) :: denom,pneum
REAL(SP), DIMENSION(size(data)) :: wk1,wk2,wktmp
REAL(SP), DIMENSION(size(d)) :: wkm
m=size(d)
n=size(data)
xms=dot_product(data,data)/n
wk1(1:n-1)=data(1:n-1)
wk2(1:n-1)=data(2:n)
do k=1,m
    pneum=dot_product(wk1(1:n-k),wk2(1:n-k))
    denom=dot_product(wk1(1:n-k),wk1(1:n-k))+ &
        dot_product(wk2(1:n-k),wk2(1:n-k))
    d(k)=2.0_sp*pneum/denom
    xms=xms*(1.0_sp-d(k)**2)
    d(1:k-1)=wkm(1:k-1)-d(k)*wkm(k-1:1:-1)
```

The algorithm is recursive, although it is implemented as an iteration. It builds up the answer for larger and larger values of m until the desired value is reached. At this point in the algorithm, one could return the vector d and scalar xms for a set of LP coefficients with k (rather than m) terms.

```
    if (k == m) RETURN
    wkm(1:k)=d(1:k)
    wktmp(2:n-k)=wk1(2:n-k)
    wk1(1:n-k-1)=wk1(1:n-k-1)-wkm(k)*wk2(1:n-k-1)
    wk2(1:n-k-1)=wk2(2:n-k)-wkm(k)*wktmp(2:n-k)
end do
call nrerror('never get here in memcof')
END SUBROUTINE memcof
```

⋆ ⋆ ⋆

```
SUBROUTINE fixrts(d)
USE nrtype
USE nr, ONLY : zroots
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
```
Given the LP coefficients d, this routine finds all roots of the characteristic polynomial (13.6.14), reflects any roots that are outside the unit circle back inside, and then returns a modified set of coefficients in d.
```
INTEGER(I4B) :: i,m
LOGICAL(LGT) :: polish
COMPLEX(SPC), DIMENSION(size(d)+1) :: a
COMPLEX(SPC), DIMENSION(size(d)) :: roots
m=size(d)
a(m+1)=cmplx(1.0_sp,kind=spc)              Set up complex coefficients for polynomial
a(m:1:-1)=cmplx(-d(1:m),kind=spc)              root finder.
polish=.true.
call zroots(a(1:m+1),roots,polish)         Find all the roots.
where (abs(roots) > 1.0) roots=1.0_sp/conjg(roots)
```
Reflect all roots outside the unit circle back inside.
```
a(1)=-roots(1)                             Now reconstruct the polynomial coefficients,
a(2:m+1)=cmplx(1.0_sp,kind=spc)
do i=2,m                                   by looping over the roots
    a(2:i)=a(1:i-1)-roots(i)*a(2:i)        and synthetically multiplying.
    a(1)=-roots(i)*a(1)
end do
d(m:1:-1)=-real(a(1:m))                    The polynomial coefficients are guaranteed
END SUBROUTINE fixrts                          to be real, so we need only return the
                                               real part as new LP coefficients.
```

f90 `a(m+1)=cmplx(1.0_sp,kind=spc)` See after convlv on p. 1254 to review why we use the explicit kind type parameter spc for cmplx.

⋆ ⋆ ⋆

```
FUNCTION predic(data,d,nfut)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data,d
INTEGER(I4B), INTENT(IN) :: nfut
REAL(SP), DIMENSION(nfut) :: predic
```
Given an array data, and given the data's LP coefficients d in an array of length M, this routine applies equation (13.6.11) to predict the next nfut data points, which it returns in an array as the function value predic. Note that the routine references only the last M values of data, as initial values for the prediction.
```
INTEGER(I4B) :: j,ndata,m
REAL(SP) :: discrp,sm
REAL(SP), DIMENSION(size(d)) :: reg
m=size(d)
ndata=size(data)
reg(1:m)=data(ndata:ndata+1-m:-1)
do j=1,nfut
    discrp=0.0
```
This is where you would put in a known discrepancy if you were reconstructing a function by linear predictive coding rather than extrapolating a function by linear prediction. See text.
```
    sm=discrp+dot_product(d,reg)
    reg=eoshift(reg,-1,sm)          [If you want to implement circular arrays, you can
    predic(j)=sm                        avoid this shifting of coefficients!]
end do
END FUNCTION predic
```

⋆ ⋆ ⋆

```
FUNCTION evlmem(fdt,d,xms)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: fdt,xms
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP) :: evlmem
```

Given d and xms as returned by memcof, this function returns the power spectrum estimate $P(f)$ as a function of fdt $= f\Delta$.

```
COMPLEX(SPC) :: z,zz
REAL(DP) :: theta                Trigonometric recurrences in double precision.
theta=TWOPI_D*fdt
z=cmplx(cos(theta),sin(theta),kind=spc)
zz=1.0_sp-z*poly(z,d)
evlmem=xms/abs(zz)**2            Equation (13.7.4).
END FUNCTION evlmem
```

f90 zz=...poly(z,d) The poly function in nrutil returns the value of the polynomial with coefficients d(:) at z. Here a version that takes real coefficients and a complex argument is actually invoked, but all the different versions have been overloaded onto the same name poly.

⋆ ⋆ ⋆

```
SUBROUTINE period(x,y,ofac,hifac,px,py,jmax,prob)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc
USE nr, ONLY : avevar
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: jmax
REAL(SP), INTENT(IN) :: ofac,hifac
REAL(SP), INTENT(OUT) :: prob
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), DIMENSION(:), POINTER :: px,py
```

Input is a set of N data points with abscissas x (which need not be equally spaced) and ordinates y, and a desired oversampling factor ofac (a typical value being 4 or larger). The routine returns pointers to internally allocated arrays px and py. px is filled with an increasing sequence of frequencies (not angular frequencies) up to hifac times the "average" Nyquist frequency, and py is filled with the values of the Lomb normalized periodogram at those frequencies. The length of these arrays is 0.5*ofac*hifac*N. The arrays x and y are not altered. The routine also returns jmax such that py(jmax) is the maximum element in py, and prob, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of prob indicates that a significant periodic signal is present.

```
INTEGER(I4B) :: i,n,nout
REAL(SP) :: ave,cwtau,effm,expy,pnow,sumc,sumcy,&
    sums,sumsh,sumsy,swtau,var,wtau,xave,xdif,xmax,xmin
REAL(DP), DIMENSION(size(x)) :: tmp1,tmp2,wi,wpi,wpr,wr
LOGICAL(LGT), SAVE :: init=.true.
n=assert_eq(size(x),size(y),'period')
if (init) then
    init=.false.
    nullify(px,py)
else
    if (associated(px)) deallocate(px)
    if (associated(py)) deallocate(py)
end if
nout=0.5_sp*ofac*hifac*n
allocate(px(nout),py(nout))
call avevar(y(:),ave,var)                Get mean and variance of the input data.
xmax=maxval(x(:))                        Go through data to get the range of abscis-
xmin=minval(x(:))                            sas.
xdif=xmax-xmin
```

```
    xave=0.5_sp*(xmax+xmin)
    pnow=1.0_sp/(xdif*ofac)                    Starting frequency.
    tmp1(:)=TWOPI_D*((x(:)-xave)*pnow)         Initialize values for the trigonometric recur-
    wpr(:)=-2.0_dp*sin(0.5_dp*tmp1)**2             rences at each data point. The recur-
    wpi(:)=sin(tmp1(:))                            rences are done in double precision.
    wr(:)=cos(tmp1(:))
    wi(:)=wpi(:)
    do i=1,nout                                Main loop over the frequencies to be evalu-
        px(i)=pnow                                 ated.
        sumsh=dot_product(wi,wr)               First, loop over the data to get τ and related
        sumc=dot_product(wr(:)-wi(:),wr(:)+wi(:))      quantities.
        wtau=0.5_sp*atan2(2.0_sp*sumsh,sumc)
        swtau=sin(wtau)
        cwtau=cos(wtau)
        tmp1(:)=wi(:)*cwtau-wr(:)*swtau        Then, loop over the data again to get the
        tmp2(:)=wr(:)*cwtau+wi(:)*swtau            periodogram value.
        sums=dot_product(tmp1,tmp1)
        sumc=dot_product(tmp2,tmp2)
        sumsy=dot_product(y(:)-ave,tmp1)
        sumcy=dot_product(y(:)-ave,tmp2)
        tmp1(:)=wr(:)                          Update the trigonometric recurrences.
        wr(:)=(wr(:)*wpr(:)-wi(:)*wpi(:))+wr(:)
        wi(:)=(wi(:)*wpr(:)+tmp1(:)*wpi(:))+wi(:)
        py(i)=0.5_sp*(sumcy**2/sumc+sumsy**2/sums)/var
        pnow=pnow+1.0_sp/(ofac*xdif)           The next frequency.
    end do
    jmax=imaxloc(py(1:nout))
    expy=exp(-py(jmax))                        Evaluate statistical significance of the maxi-
    effm=2.0_sp*nout/ofac                          mum.
    prob=effm*expy
    if (prob > 0.01_sp) prob=1.0_sp-(1.0_sp-expy)**effm
    END SUBROUTINE period
```

f90 This routine shows another example of how to return arrays whose size is not known in advance (cf. `zbrac` in Chapter B9). The coding is explained in the subsection on pointers in §21.5. The size of the output arrays, `nout` in the code, is available as `size(px)`.

`jmax=imaxloc...` See discussion of `imaxloc` on p. 1017.

```
SUBROUTINE fasper(x,y,ofac,hifac,px,py,jmax,prob)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,imaxloc,nrerror
USE nr, ONLY : avevar,realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(IN) :: ofac,hifac
INTEGER(I4B), INTENT(OUT) :: jmax
REAL(SP), INTENT(OUT) :: prob
REAL(SP), DIMENSION(:), POINTER :: px,py
INTEGER(I4B), PARAMETER :: MACC=4
```

Input is a set of N data points with abscissas `x` (which need not be equally spaced) and ordinates `y`, and a desired oversampling factor `ofac` (a typical value being 4 or larger). The routine returns pointers to internally allocated arrays `px` and `py`. `px` is filled with an increasing sequence of frequencies (not angular frequencies) up to `hifac` times the "average" Nyquist frequency, and `py` is filled with the values of the Lomb normalized periodogram at those frequencies. The length of these arrays is `0.5*ofac*hifac*`N. The arrays `x` and `y` are not altered. The routine also returns `jmax` such that `py(jmax)` is the maximum element in `py`, and `prob`, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of `prob` indicates that a significant

```
    periodic signal is present.
    Parameter: MACC is the number of interpolation points per 1/4 cycle of highest frequency.
INTEGER(I4B) :: j,k,n,ndim,nfreq,nfreqt,nout
REAL(SP) :: ave,ck,ckk,cterm,cwt,den,df,effm,expy,fac,fndim,hc2wt,&
    hs2wt,hypo,sterm,swt,var,xdif,xmax,xmin
REAL(SP), DIMENSION(:), ALLOCATABLE :: wk1,wk2
LOGICAL(LGT), SAVE :: init=.true.
n=assert_eq(size(x),size(y),'fasper')
if (init) then
    init=.false.
    nullify(px,py)
else
    if (associated(px)) deallocate(px)
    if (associated(py)) deallocate(py)
end if
nfreqt=ofac*hifac*n*MACC
nfreq=64
do                                  Size the FFT as next power of 2 above nfreqt.
    if (nfreq >= nfreqt) exit
    nfreq=nfreq*2
end do
ndim=2*nfreq
allocate(wk1(ndim),wk2(ndim))
call avevar(y(1:n),ave,var)         Compute the mean, variance, and range of the data.
xmax=maxval(x(:))
xmin=minval(x(:))
xdif=xmax-xmin
wk1(1:ndim)=0.0                     Zero the workspaces.
wk2(1:ndim)=0.0
fac=ndim/(xdif*ofac)
fndim=ndim
do j=1,n                            Extirpolate the data into the workspaces.
    ck=1.0_sp+mod((x(j)-xmin)*fac,fndim)
    ckk=1.0_sp+mod(2.0_sp*(ck-1.0_sp),fndim)
    call spreadval(y(j)-ave,wk1,ck,MACC)
    call spreadval(1.0_sp,wk2,ckk,MACC)
end do
call realft(wk1(1:ndim),1)          Take the fast Fourier transforms.
call realft(wk2(1:ndim),1)
df=1.0_sp/(xdif*ofac)
nout=0.5_sp*ofac*hifac*n
allocate(px(nout),py(nout))
k=3
do j=1,nout                         Compute the Lomb value for each frequency.
    hypo=sqrt(wk2(k)**2+wk2(k+1)**2)
    hc2wt=0.5_sp*wk2(k)/hypo
    hs2wt=0.5_sp*wk2(k+1)/hypo
    cwt=sqrt(0.5_sp+hc2wt)
    swt=sign(sqrt(0.5_sp-hc2wt),hs2wt)
    den=0.5_sp*n+hc2wt*wk2(k)+hs2wt*wk2(k+1)
    cterm=(cwt*wk1(k)+swt*wk1(k+1))**2/den
    sterm=(cwt*wk1(k+1)-swt*wk1(k))**2/(n-den)
    px(j)=j*df
    py(j)=(cterm+sterm)/(2.0_sp*var)
    k=k+2
end do
deallocate(wk1,wk2)
jmax=imaxloc(py(1:nout))
expy=exp(-py(jmax))                 Estimate significance of largest peak value.
effm=2.0_sp*nout/ofac
prob=effm*expy
if (prob > 0.01_sp) prob=1.0_sp-(1.0_sp-expy)**effm
CONTAINS
```

```
SUBROUTINE spreadval(y,yy,x,m)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: y,x
REAL(SP), DIMENSION(:), INTENT(INOUT) :: yy
INTEGER(I4B), INTENT(IN) :: m
```

Given an array yy of length N, extirpolate (spread) a value y into m actual array elements that best approximate the "fictional" (i.e., possibly noninteger) array element number x. The weights used are coefficients of the Lagrange interpolating polynomial.

```
INTEGER(I4B) :: ihi,ilo,ix,j,nden,n
REAL(SP) :: fac
INTEGER(I4B), DIMENSION(10) :: nfac = (/ &
    1,1,2,6,24,120,720,5040,40320,362880 /)
if (m > 10) call nrerror('factorial table too small in spreadval')
n=size(yy)
ix=x
if (x == real(ix,sp)) then
    yy(ix)=yy(ix)+y
else
    ilo=min(max(int(x-0.5_sp*m+1.0_sp),1),n-m+1)
    ihi=ilo+m-1
    nden=nfac(m)
    fac=product(x-arth(ilo,1,m))
    yy(ihi)=yy(ihi)+y*fac/(nden*(x-ihi))
    do j=ihi-1,ilo,-1
        nden=(nden/(j+1-ilo))*(j-ihi)
        yy(j)=yy(j)+y*fac/(nden*(x-j))
    end do
end if
END SUBROUTINE spreadval
END SUBROUTINE fasper
```

f90 This routine shows another example of how to return arrays whose size is not known in advance (cf. zbrac in Chapter B9). The coding is explained in the subsection on pointers in §21.5. The size of the output arrays, nout in the code, is available as size(px).

`jmax=imaxloc...` See discussion of imaxloc on p. 1017.

`if (x == real(ix,sp)) then` Without the explicit kind type parameter sp, real returns a value of type default real for an integer argument. This prevents automatic conversion of the routine from single to double precision. Here all you have to do is redefine sp in nrtype to get double precision.

⋆ ⋆ ⋆

```
SUBROUTINE dftcor(w,delta,a,b,endpts,corre,corim,corfac)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: w,delta,a,b
REAL(SP), INTENT(OUT) :: corre,corim,corfac
REAL(SP), DIMENSION(:), INTENT(IN) :: endpts
```

For an integral approximated by a discrete Fourier transform, this routine computes the correction factor that multiplies the DFT and the endpoint correction to be added. Input is the angular frequency w, stepsize delta, lower and upper limits of the integral a and b, while the array endpts of length 8 contains the first 4 and last 4 function values. The

correction factor $W(\theta)$ is returned as corfac, while the real and imaginary parts of the endpoint correction are returned as corre and corim.

```
REAL(SP) :: a0i,a0r,a1i,a1r,a2i,a2r,a3i,a3r,arg,c,cl,cr,s,sl,sr,t,&
    t2,t4,t6
REAL(DP) :: cth,ctth,spth2,sth,sth4i,stth,th,th2,th4,&
    tmth2,tth4i
th=w*delta
call assert(a < b, th >= 0.0, th <= PI_D, 'dftcor args')
if (abs(th) < 5.0e-2_dp) then          Use series.
    t=th
    t2=t*t
    t4=t2*t2
    t6=t4*t2
    corfac=1.0_sp-(11.0_sp/720.0_sp)*t4+(23.0_sp/15120.0_sp)*t6
    a0r=(-2.0_sp/3.0_sp)+t2/45.0_sp+(103.0_sp/15120.0_sp)*t4-&
        (169.0_sp/226800.0_sp)*t6
    a1r=(7.0_sp/24.0_sp)-(7.0_sp/180.0_sp)*t2+(5.0_sp/3456.0_sp)*t4&
        -(7.0_sp/259200.0_sp)*t6
    a2r=(-1.0_sp/6.0_sp)+t2/45.0_sp-(5.0_sp/6048.0_sp)*t4+t6/64800.0_sp
    a3r=(1.0_sp/24.0_sp)-t2/180.0_sp+(5.0_sp/24192.0_sp)*t4-t6/259200.0_sp
    a0i=t*(2.0_sp/45.0_sp+(2.0_sp/105.0_sp)*t2-&
        (8.0_sp/2835.0_sp)*t4+(86.0_sp/467775.0_sp)*t6)
    a1i=t*(7.0_sp/72.0_sp-t2/168.0_sp+(11.0_sp/72576.0_sp)*t4-&
        (13.0_sp/5987520.0_sp)*t6)
    a2i=t*(-7.0_sp/90.0_sp+t2/210.0_sp-(11.0_sp/90720.0_sp)*t4+&
        (13.0_sp/7484400.0_sp)*t6)
    a3i=t*(7.0_sp/360.0_sp-t2/840.0_sp+(11.0_sp/362880.0_sp)*t4-&
        (13.0_sp/29937600.0_sp)*t6)
else                                   Use trigonometric formulas in double precision.
    cth=cos(th)
    sth=sin(th)
    ctth=cth**2-sth**2
    stth=2.0_dp*sth*cth
    th2=th*th
    th4=th2*th2
    tmth2=3.0_dp-th2
    spth2=6.0_dp+th2
    sth4i=1.0_sp/(6.0_dp*th4)
    tth4i=2.0_dp*sth4i
    corfac=tth4i*spth2*(3.0_sp-4.0_dp*cth+ctth)
    a0r=sth4i*(-42.0_dp+5.0_dp*th2+spth2*(8.0_dp*cth-ctth))
    a0i=sth4i*(th*(-12.0_dp+6.0_dp*th2)+spth2*stth)
    a1r=sth4i*(14.0_dp*tmth2-7.0_dp*spth2*cth)
    a1i=sth4i*(30.0_dp*th-5.0_dp*spth2*sth)
    a2r=tth4i*(-4.0_dp*tmth2+2.0_dp*spth2*cth)
    a2i=tth4i*(-12.0_dp*th+2.0_dp*spth2*sth)
    a3r=sth4i*(2.0_dp*tmth2-spth2*cth)
    a3i=sth4i*(6.0_dp*th-spth2*sth)
end if
cl=a0r*endpts(1)+a1r*endpts(2)+a2r*endpts(3)+a3r*endpts(4)
sl=a0i*endpts(1)+a1i*endpts(2)+a2i*endpts(3)+a3i*endpts(4)
cr=a0r*endpts(8)+a1r*endpts(7)+a2r*endpts(6)+a3r*endpts(5)
sr=-a0i*endpts(8)-a1i*endpts(7)-a2i*endpts(6)-a3i*endpts(5)
arg=w*(b-a)
c=cos(arg)
s=sin(arg)
corre=cl+c*cr-s*sr
corim=sl+s*cr+c*sr
END SUBROUTINE dftcor
```

```
SUBROUTINE dftint(func,a,b,w,cosint,sinint)
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : dftcor,polint,realft
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,w
REAL(SP), INTENT(OUT) :: cosint,sinint
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: M=64,NDFT=1024,MPOL=6
```

Example subroutine illustrating how to use the routine dftcor. The user supplies an external function func that returns the quantity $h(t)$. The routine then returns $\int_a^b \cos(\omega t)h(t)\,dt$ as cosint and $\int_a^b \sin(\omega t)h(t)\,dt$ as sinint.

Parameters: The values of M, NDFT, and MPOL are merely illustrative and should be optimized for your particular application. M is the number of subintervals, NDFT is the length of the FFT (a power of 2), and MPOL is the degree of polynomial interpolation used to obtain the desired frequency from the FFT.

```
INTEGER(I4B) :: nn
INTEGER(I4B), SAVE :: init=0
INTEGER(I4B), DIMENSION(MPOL) :: nnmpol
REAL(SP) :: c,cdft,cerr,corfac,corim,corre,en,s,sdft,serr
REAL(SP), SAVE :: delta
REAL(SP), DIMENSION(MPOL) :: cpol,spol,xpol
REAL(SP), DIMENSION(NDFT), SAVE :: data
REAL(SP), DIMENSION(8), SAVE :: endpts
REAL(SP), SAVE :: aold=-1.0e30_sp,bold=-1.0e30_sp
if (init /= 1 .or. a /= aold .or. b /= bold) then
    init=1
    aold=a
    bold=b
    delta=(b-a)/M
    data(1:M+1)=func(a+arth(0,1,M+1)*delta)
      Load the function values into the data array.
    data(M+2:NDFT)=0.0
    endpts(1:4)=data(1:4)
    endpts(5:8)=data(M-2:M+1)
    call realft(data(1:NDFT),1)
      realft returns the unused value corresponding to \omega_{N/2} in data(2). We actually want
      this element to contain the imaginary part corresponding to \omega_0, which is zero.
    data(2)=0.0
end if
```

Do we need to initialize, or is only ω changed?

Zero pad the rest of the data array.

Load the endpoints.

Now interpolate on the DFT result for the desired frequency. If the frequency is an ω_n, i.e., the quantity en is an integer, then cdft=data(2*en-1), sdft=data(2*en), and you could omit the interpolation.

```
en=w*delta*NDFT/TWOPI+1.0_sp
nn=min(max(int(en-0.5_sp*MPOL+1.0_sp),1),NDFT/2-MPOL+1)
nnmpol=arth(nn,1,MPOL)
cpol(1:MPOL)=data(2*nnmpol(:)-1)
spol(1:MPOL)=data(2*nnmpol(:))
xpol(1:MPOL)=nnmpol(:)
call polint(xpol,cpol,en,cdft,cerr)
call polint(xpol,spol,en,sdft,serr)
call dftcor(w,delta,a,b,endpts,corre,corim,corfac)
cdft=cdft*corfac+corre
sdft=sdft*corfac+corim
c=delta*cos(w*a)
s=delta*sin(w*a)
cosint=c*cdft-s*sdft
```

Leftmost point for the interpolation.

Now get the endpoint correction and the multiplicative factor $W(\theta)$.

Finally multiply by Δ and $\exp(i\omega a)$.

```
sinint=s*cdft+c*sdft
END SUBROUTINE dftint
```

⋆ ⋆ ⋆

```
SUBROUTINE wt1(a,isign,wtstep)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
INTERFACE
    SUBROUTINE wtstep(a,isign)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE wtstep
END INTERFACE
```

One-dimensional discrete wavelet transform. This routine implements the pyramid algorithm, replacing a by its wavelet transform (for isign=1), or performing the inverse operation (for isign=−1). The length of a is N, which must be an integer power of 2. The subroutine wtstep, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.

```
INTEGER(I4B) :: n,nn
n=size(a)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in wt1')
if (n < 4) RETURN
if (isign >= 0) then          Wavelet transform.
    nn=n                      Start at largest hierarchy,
    do
        if (nn < 4) exit
        call wtstep(a(1:nn),isign)
        nn=nn/2               and work towards smallest.
    end do
else                          Inverse wavelet transform.
    nn=4                      Start at smallest hierarchy,
    do
        if (nn > n) exit
        call wtstep(a(1:nn),isign)
        nn=nn*2               and work towards largest.
    end do
end if
END SUBROUTINE wt1
```

```
SUBROUTINE daub4(a,isign)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
```

Applies the Daubechies 4-coefficient wavelet filter to data vector a (for isign=1) or applies its transpose (for isign=−1). Used hierarchically by routines wt1 and wtn.

```
REAL(SP), DIMENSION(size(a)) :: wksp
REAL(SP), PARAMETER :: C0=0.4829629131445341_sp,&
    C1=0.8365163037378079_sp,C2=0.2241438680420134_sp,&
    C3=-0.1294095225512604_sp
INTEGER(I4B) :: n,nh,nhp,nhm
n=size(a)
if (n < 4) RETURN
nh=n/2
nhp=nh+1
nhm=nh-1
```

```
if (isign >= 0) then          Apply filter.
    wksp(1:nhm) = C0*a(1:n-3:2)+C1*a(2:n-2:2) &
        +C2*a(3:n-1:2)+C3*a(4:n:2)
    wksp(nh)=C0*a(n-1)+C1*a(n)+C2*a(1)+C3*a(2)
    wksp(nhp:n-1) = C3*a(1:n-3:2)-C2*a(2:n-2:2) &
        +C1*a(3:n-1:2)-C0*a(4:n:2)
    wksp(n)=C3*a(n-1)-C2*a(n)+C1*a(1)-C0*a(2)
else                          Apply transpose filter.
    wksp(1)=C2*a(nh)+C1*a(n)+C0*a(1)+C3*a(nhp)
    wksp(2)=C3*a(nh)-C0*a(n)+C1*a(1)-C2*a(nhp)
    wksp(3:n-1:2) = C2*a(1:nhm)+C1*a(nhp:n-1) &
        +C0*a(2:nh)+C3*a(nh+2:n)
    wksp(4:n:2) = C3*a(1:nhm)-C0*a(nhp:n-1) &
        +C1*a(2:nh)-C2*a(nh+2:n)
end if
a(1:n)=wksp(1:n)
END SUBROUTINE daub4
```

```
MODULE pwtcom
USE nrtype
INTEGER(I4B), SAVE :: ncof=0,ioff,joff        These module variables communicate the
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: cc,cr      filter to pwt.
END MODULE pwtcom
```

```
SUBROUTINE pwtset(n)
USE nrtype; USE nrutil, ONLY : nrerror
USE pwtcom
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
```

Initializing routine for pwt, here implementing the Daubechies wavelet filters with 4, 12, and 20 coefficients, as selected by the input value n. Further wavelet filters can be included in the obvious manner. This routine must be called (once) before the first use of pwt. (For the case n=4, the specific routine daub4 is considerably faster than pwt.)

```
REAL(SP) :: sig
REAL(SP), PARAMETER :: &
    c4(4)=(/&
    0.4829629131445341_sp, 0.8365163037378079_sp, &
    0.2241438680420134_sp,-0.1294095225512604_sp /), &
    c12(12)=(/&
    0.111540743350_sp, 0.494623890398_sp, 0.751133908021_sp, &
    0.315250351709_sp,-0.226264693965_sp,-0.129766867567_sp, &
    0.097501605587_sp, 0.027522865530_sp,-0.031582039318_sp, &
    0.000553842201_sp, 0.004777257511_sp,-0.001077301085_sp /), &
    c20(20)=(/&
    0.026670057901_sp, 0.188176800078_sp, 0.527201188932_sp, &
    0.688459039454_sp, 0.281172343661_sp,-0.249846424327_sp, &
    -0.195946274377_sp, 0.127369340336_sp, 0.093057364604_sp, &
    -0.071394147166_sp,-0.029457536822_sp, 0.033212674059_sp, &
    0.003606553567_sp,-0.010733175483_sp, 0.001395351747_sp, &
    0.001992405295_sp,-0.000685856695_sp,-0.000116466855_sp, &
    0.000093588670_sp,-0.000013264203_sp /)
if (allocated(cc)) deallocate(cc)
if (allocated(cr)) deallocate(cr)
allocate(cc(n),cr(n))
ncof=n
ioff=-n/2                     These values center the "support" of the wavelets at each
joff=-n/2                         level. Alternatively, the "peaks" of the wavelets can
sig=-1.0                          be approximately centered by the choices ioff=-2
select case(n)                    and joff=-n+2. Note that daub4 and pwtset with
    case(4)                       n=4 use different default centerings.
```

```
        cc=c4
    case(12)
        cc=c12
    case(20)
        cc=c20
    case default
        call nrerror('unimplemented value n in pwtset')
end select
cr(n:1:-1) = cc
cr(n:1:-2) = -cr(n:1:-2)
END SUBROUTINE pwtset
```

f90 Here we need to have as global variables arrays whose dimensions are known only at run time. At first sight the situation is the same as with the module `fminln` in `newt` on p. 1197. If you review the discussion there and in §21.5, you will recall that there are two good ways to implement this: with allocatable arrays ("Method 1") or with pointers ("Method 2"). There is a difference here that makes allocatable arrays simpler. We do not wish to deallocate the arrays on exiting `pwtset`. On the contrary, the values in `cc` and `cr` need to be preserved for use in `pwt`. Since allocatable arrays are born in the well-defined state of "not currently allocated," we can declare the arrays here as

```
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: cc,cr
```

and test whether they were used on a previous call with

```
if (allocated(cc)) deallocate(cc)
if (allocated(cr)) deallocate(cr)
```

We are then ready to allocate the new storage:

```
allocate(cc(n),cr(n))
```

With pointers, we would need the additional machinery of nullifying the pointers on the initial call, since pointers are born in an undefined state (see §21.5).

There is an additional important point in this example. The module variables need to be used by a "sibling" routine, `pwt`. We need to be sure that they do not become undefined when we exit `pwtset`. We could ensure this by putting a `USE pwtcom` in the main program that calls both `pwtset` and `pwt`, but it's easy to forget to do this. It is preferable to put explicit `SAVE`s on all the module variables.

```
SUBROUTINE pwt(a,isign)
USE nrtype; USE nrutil, ONLY : arth,nrerror
USE pwtcom
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
```

Partial wavelet transform: applies an arbitrary wavelet filter to data vector `a` (for `isign=1`) or applies its transpose (for `isign=-1`). Used hierarchically by routines `wt1` and `wtn`. The actual filter is determined by a preceding (and required) call to `pwtset`, which initializes the module `pwtcom`.

```
REAL(SP), DIMENSION(size(a)) :: wksp
INTEGER(I4B), DIMENSION(size(a)/2) :: jf,jr
INTEGER(I4B) :: k,n,nh,nmod
n=size(a)
if (n < 4) RETURN
if (ncof == 0) call nrerror('pwt: must call pwtset before pwt')
nmod=ncof*n
```

A positive constant equal to zero mod `n`.

```
nh=n/2
wksp(:)=0.0
jf=iand(n-1,arth(2+nmod+ioff,2,nh))        Use bitwise AND to wrap-around the point-
jr=iand(n-1,arth(2+nmod+joff,2,nh))            ers. n-1 is a mask of all bits, since n is
do k=1,ncof                                    a power of 2.
    if (isign >= 0) then                   Apply filter.
        wksp(1:nh)=wksp(1:nh)+cc(k)*a(jf+1)
        wksp(nh+1:n)=wksp(nh+1:n)+cr(k)*a(jr+1)
    else                                   Apply transpose filter.
        wksp(jf+1)=wksp(jf+1)+cc(k)*a(1:nh)
        wksp(jr+1)=wksp(jr+1)+cr(k)*a(nh+1:n)
    end if
    if (k == ncof) exit
    jf=iand(n-1,jf+1)
    jr=iand(n-1,jr+1)
end do
a(:)=wksp(:)                               Copy the results back from workspace.
END SUBROUTINE pwt
```

⋆ ⋆ ⋆

```
SUBROUTINE wtn(a,nn,isign,wtstep)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nn
INTEGER(I4B), INTENT(IN) :: isign
INTERFACE
    SUBROUTINE wtstep(a,isign)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE wtstep
END INTERFACE
```

Replaces a by its N-dimensional discrete wavelet transform, if isign is input as 1. nn is an integer array of length N, containing the lengths of each dimension (number of real values), which must all be powers of 2. a is a real array of length equal to the product of these lengths, in which the data are stored as in a multidimensional real FORTRAN array. If isign is input as -1, a is replaced by its inverse wavelet transform. The subroutine wtstep, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.

```
INTEGER(I4B) :: i1,i2,i3,idim,n,ndim,nnew,nprev,nt,ntot
REAL(SP), DIMENSION(:), ALLOCATABLE :: wksp
call assert(iand(nn,nn-1)==0, 'each dimension must be a power of 2 in wtn')
allocate(wksp(maxval(nn)))
ndim=size(nn)
ntot=product(nn(:))
nprev=1
do idim=1,ndim                                 Main loop over the dimensions.
    n=nn(idim)
    nnew=n*nprev
    if (n > 4) then
        do i2=0,ntot-1,nnew
            do i1=1,nprev
                i3=i1+i2
                wksp(1:n)=a(arth(i3,nprev,n))  Copy the relevant row or column
                i3=i3+n*nprev                      or etc. into workspace.
                if (isign >= 0) then           Do one-dimensional wavelet trans-
                    nt=n                           form.
                    do
```

```
                    if (nt < 4) exit
                    call wtstep(wksp(1:nt),isign)
                    nt=nt/2
                end do
            else                                    Or inverse transform.
                nt=4
                do
                    if (nt > n) exit
                    call wtstep(wksp(1:nt),isign)
                    nt=nt*2
                end do
            end if
            i3=i1+i2
            a(arth(i3,nprev,n))=wksp(1:n)           Copy back from workspace.
            i3=i3+n*nprev
        end do
    end do
  end if
  nprev=nnew
end do
deallocate(wksp)
END SUBROUTINE wtn
```