Chapter B17. Two Point Boundary Value Problems

```
!   FUNCTION shoot(v) is named "funcv" for use with "newt"
    FUNCTION funcv(v)
    USE nrtype
    USE nr, ONLY : odeint,rkqs
    USE sphoot_caller, ONLY : nvar,x1,x2; USE ode_path, ONLY : xp,yp
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: v
    REAL(SP), DIMENSION(size(v)) :: funcv
    REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Routine for use with `newt` to solve a two point boundary value problem for N coupled ODEs by shooting from `x1` to `x2`. Initial values for the ODEs at `x1` are generated from the n_2 input coefficients `v`, using the user-supplied routine `load`. The routine integrates the ODEs to `x2` using the Runge-Kutta method with tolerance `EPS`, initial stepsize `h1`, and minimum stepsize `hmin`. At `x2` it calls the user-supplied subroutine `score` to evaluate the n_2 functions `funcv` that ought to be zero to satisfy the boundary conditions at `x2`. The functions `funcv` are returned on output. `newt` uses a globally convergent Newton's method to adjust the values of `v` until the functions `funcv` are zero. The user-supplied subroutine `derivs(x,y,dydx)` supplies derivative information to the ODE integrator (see Chapter 16). The module `sphoot_caller` receives its values from the main program so that `funcv` can have the syntax required by `newt`. Set `nvar` $= N$ in the main program.

```
    REAL(SP) :: h1,hmin
    REAL(SP), DIMENSION(nvar) :: y
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs

        SUBROUTINE load(x1,v,y)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x1
        REAL(SP), DIMENSION(:), INTENT(IN) :: v
        REAL(SP), DIMENSION(:), INTENT(OUT) :: y
        END SUBROUTINE load

        SUBROUTINE score(x2,y,f)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x2
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: f
        END SUBROUTINE score
    END INTERFACE
    h1=(x2-x1)/100.0_sp
```

```
    hmin=0.0
    call load(x1,v,y)
    if (associated(xp)) deallocate(xp,yp)        Prevent memory leak if save_steps set
    call odeint(y,x1,x2,EPS,h1,hmin,derivs,rkqs)     to .true.
    call score(x2,y,funcv)
    END FUNCTION funcv
```

⋆ ⋆ ⋆

```
!   FUNCTION shootf(v) is named "funcv" for use with "newt"
    FUNCTION funcv(v)
    USE nrtype
    USE nr, ONLY : odeint,rkqs
    USE sphfpt_caller, ONLY : x1,x2,xf,nn2; USE ode_path, ONLY : xp,yp
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: v
    REAL(SP), DIMENSION(size(v)) :: funcv
    REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Routine for use with `newt` to solve a two point boundary value problem for N coupled ODEs by shooting from `x1` and `x2` to a fitting point `xf`. Initial values for the ODEs at `x1` (`x2`) are generated from the n_2 (n_1) coefficients V_1 (V_2), using the user-supplied routine `load1` (`load2`). The coefficients V_1 and V_2 should be stored in a single array `v` of length N in the main program, and referenced by pointers as `v1=>v(1:`n_2`)`, `v2=>v(`n_2+1`:`N`)`. Here $N = n_1 + n_2$. The routine integrates the ODEs to `xf` using the Runge-Kutta method with tolerance `EPS`, initial stepsize `h1`, and minimum stepsize `hmin`. At `xf` it calls the user-supplied subroutine `score` to evaluate the N functions `f1` and `f2` that ought to match at `xf`. The differences `funcv` are returned on output. `newt` uses a globally convergent Newton's method to adjust the values of `v` until the functions `funcv` are zero. The user-supplied subroutine `derivs(x,y,dydx)` supplies derivative information to the ODE integrator (see Chapter 16). The module `sphfpt_caller` receives its values from the main program so that `funcv` can have the syntax required by `newt`. Set `nn2` $= n_2$ in the main program.

```
    REAL(SP) :: h1,hmin
    REAL(SP), DIMENSION(size(v)) :: f1,f2,y
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs

        SUBROUTINE load1(x1,v1,y)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x1
        REAL(SP), DIMENSION(:), INTENT(IN) :: v1
        REAL(SP), DIMENSION(:), INTENT(OUT) :: y
        END SUBROUTINE load1

        SUBROUTINE load2(x2,v2,y)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x2
        REAL(SP), DIMENSION(:), INTENT(IN) :: v2
        REAL(SP), DIMENSION(:), INTENT(OUT) :: y
        END SUBROUTINE load2

        SUBROUTINE score(x2,y,f)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x2
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
```

```
    REAL(SP), DIMENSION(:), INTENT(OUT) :: f
    END SUBROUTINE score
END INTERFACE
h1=(x2-x1)/100.0_sp
hmin=0.0
call load1(x1,v,y)                          Path from x1 to xf with best trial values V_1.
if (associated(xp)) deallocate(xp,yp)       Prevent memory leak if save_steps set
call odeint(y,x1,xf,EPS,h1,hmin,derivs,rkqs)    to .true.
call score(xf,y,f1)
call load2(x2,v(nn2+1:),y)                  Path from x2 to xf with best trial values V_2.
call odeint(y,x2,xf,EPS,h1,hmin,derivs,rkqs)
call score(xf,y,f2)
funcv(:)=f1(:)-f2(:)
END FUNCTION funcv
```

★ ★ ★

```
SUBROUTINE solvde(itmax,conv,slowc,scalv,indexv,nb,y)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,nrerror
USE nr, ONLY : difeq
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: itmax,nb
REAL(SP), INTENT(IN) :: conv,slowc
REAL(SP), DIMENSION(:), INTENT(IN) :: scalv
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: y
```

Driver routine for solution of two point boundary value problems with N equations by relaxation. `itmax` is the maximum number of iterations. `conv` is the convergence criterion (see text). `slowc` controls the fraction of corrections actually used after each iteration. `scalv`, a vector of length N, contains typical sizes for each dependent variable, used to weight errors. `indexv`, also of length N, lists the column ordering of variables used to construct the matrix `s` of derivatives. (The `nb` boundary conditions at the first mesh point must contain some dependence on the first `nb` variables listed in `indexv`.) There are a total of M mesh points. `y` is the $N \times M$ array that contains the initial guess for all the dependent variables at each mesh point. On each iteration, it is updated by the calculated correction.

```
INTEGER(I4B) :: ic1,ic2,ic3,ic4,it,j,j1,j2,j3,j4,j5,j6,j7,j8,&
    j9,jc1,jcf,jv,k,k1,k2,km,kp,m,ne,nvars
INTEGER(I4B), DIMENSION(size(scalv)) :: kmax
REAL(SP) :: err,fac
REAL(SP), DIMENSION(size(scalv)) :: ermax
REAL(SP), DIMENSION(size(scalv),2*size(scalv)+1) :: s
REAL(SP), DIMENSION(size(scalv),size(scalv)-nb+1,size(y,2)+1) :: c
ne=assert_eq(size(scalv),size(indexv),size(y,1),'solvde: ne')
m=size(y,2)
k1=1                                        Set up row and column markers.
k2=m
nvars=ne*m
j1=1
j2=nb
j3=nb+1
j4=ne
j5=j4+j1
j6=j4+j2
j7=j4+j3
j8=j4+j4
j9=j8+j1
ic1=1
ic2=ne-nb
ic3=ic2+1
ic4=ne
jc1=1
jcf=ic3
do it=1,itmax                               Primary iteration loop.
    k=k1                                    Boundary conditions at first point.
```

```
    call difeq(k,k1,k2,j9,ic3,ic4,indexv,s,y)
    call pinvs(ic3,ic4,j5,j9,jc1,k1,c,s)
    do k=k1+1,k2                                  Finite difference equations at all point
        kp=k-1                                        pairs.
        call difeq(k,k1,k2,j9,ic1,ic4,indexv,s,y)
        call red(ic1,ic4,j1,j2,j3,j4,j9,ic3,jc1,jcf,kp,c,s)
        call pinvs(ic1,ic4,j3,j9,jc1,k,c,s)
    end do
    k=k2+1                                        Final boundary conditions.
    call difeq(k,k1,k2,j9,ic1,ic2,indexv,s,y)
    call red(ic1,ic2,j5,j6,j7,j8,j9,ic3,jc1,jcf,k2,c,s)
    call pinvs(ic1,ic2,j7,j9,jcf,k2+1,c,s)
    call bksub(ne,nb,jcf,k1,k2,c)                 Backsubstitution.
    do j=1,ne                                     Convergence check, accumulate average
        jv=indexv(j)                                  error.
        km=imaxloc(abs(c(jv,1,k1:k2)))+k1-1
          Find point with largest error, for each dependent variable.
        ermax(j)=c(jv,1,km)
        kmax(j)=km
    end do
    ermax(:)=ermax(:)/scalv(:)                    Weighting for each dependent variable.
    err=sum(sum(abs(c(indexv(:),1,k1:k2)),dim=2)/scalv(:))/nvars
    fac=slowc/max(slowc,err)
      Reduce correction applied when error is large.
    y(:,k1:k2)=y(:,k1:k2)-fac*c(indexv(:),1,k1:k2)        Apply corrections.
    write(*,'(1x,i4,2f12.6)') it,err,fac
      Summary of corrections for this step. Point with largest error for each variable can be
      monitored by writing out kmax and ermax.
    if (err < conv) RETURN
end do
call nrerror('itmax exceeded in solvde')          Convergence failed.
CONTAINS

SUBROUTINE bksub(ne,nb,jf,k1,k2,c)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ne,nb,jf,k1,k2
REAL(SP), DIMENSION(:,:,:), INTENT(INOUT) :: c
   Backsubstitution, used internally by solvde.
INTEGER(I4B) :: im,k,nbf
nbf=ne-nb
im=1
do k=k2,k1,-1
      Use recurrence relations to eliminate remaining dependences.
    if (k == k1) im=nbf+1                         Special handling of first point.
    c(im:ne,jf,k)=c(im:ne,jf,k)-matmul(c(im:ne,1:nbf,k),c(1:nbf,jf,k+1))
end do
c(1:nb,1,k1:k2)=c(1+nbf:nb+nbf,jf,k1:k2)          Reorder corrections to be in column 1.
c(1+nb:nbf+nb,1,k1:k2)=c(1:nbf,jf,k1+1:k2+1)
END SUBROUTINE bksub

SUBROUTINE pinvs(ie1,ie2,je1,jsf,jc1,k,c,s)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ie1,ie2,je1,jsf,jc1,k
REAL(SP), DIMENSION(:,:,:), INTENT(OUT) :: c
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: s
   Diagonalize the square subsection of the s matrix, and store the recursion coefficients in
   c; used internally by solvde.
INTEGER(I4B) :: i,icoff,id,ipiv,jcoff,je2,jp,jpiv,js1
INTEGER(I4B), DIMENSION(ie2) :: indxr
REAL(SP) :: big,piv,pivinv
REAL(SP), DIMENSION(ie2) :: pscl
je2=je1+ie2-ie1
js1=je2+1
pscl(ie1:ie2)=maxval(abs(s(ie1:ie2,je1:je2)),dim=2)
  Implicit pivoting, as in §2.1.
```

```
if (any(pscl(ie1:ie2) == 0.0)) &
    call nrerror('singular matrix, row all 0 in pinvs')
pscl(ie1:ie2)=1.0_sp/pscl(ie1:ie2)
indxr(ie1:ie2)=0
do id=ie1,ie2
    piv=0.0
    do i=ie1,ie2                                  Find pivot element.
        if (indxr(i) == 0) then
            jp=imaxloc(abs(s(i,je1:je2)))+je1-1
            big=abs(s(i,jp))
            if (big*pscl(i) > piv) then
                ipiv=i
                jpiv=jp
                piv=big*pscl(i)
            end if
        end if
    end do
    if (s(ipiv,jpiv) == 0.0) call nrerror('singular matrix in pinvs')
    indxr(ipiv)=jpiv                              In place reduction. Save column order-
    pivinv=1.0_sp/s(ipiv,jpiv)                        ing.
    s(ipiv,je1:jsf)=s(ipiv,je1:jsf)*pivinv        Normalize pivot row.
    s(ipiv,jpiv)=1.0
    do i=ie1,ie2                                  Reduce nonpivot elements in column.
        if (indxr(i) /= jpiv .and. s(i,jpiv) /= 0.0) then
            s(i,je1:jsf)=s(i,je1:jsf)-s(i,jpiv)*s(ipiv,je1:jsf)
            s(i,jpiv)=0.0
        end if
    end do
end do
jcoff=jc1-js1                                     Sort and store unreduced coefficients.
icoff=ie1-je1
c(indxr(ie1:ie2)+icoff,js1+jcoff:jsf+jcoff,k)=s(ie1:ie2,js1:jsf)
END SUBROUTINE pinvs

SUBROUTINE red(iz1,iz2,jz1,jz2,jm1,jm2,jmf,ic1,jc1,jcf,kc,c,s)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iz1,iz2,jz1,jz2,jm1,jm2,jmf,ic1,jc1,jcf,kc
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: s
REAL(SP), DIMENSION(:,:,:), INTENT(IN) :: c
    Reduce columns jz1-jz2 of the s matrix, using previous results as stored in the c matrix.
    Only columns jm1-jm2,jmf are affected by the prior results. red is used internally by
    solvde.
INTEGER(I4B) :: ic,l,loff
loff=jc1-jm1
ic=ic1
do j=jz1,jz2                                      Loop over columns to be zeroed.
    do l=jm1,jm2                                  Loop over columns altered.
        s(iz1:iz2,l)=s(iz1:iz2,l)-s(iz1:iz2,j)*c(ic,l+loff,kc)
          Loop over rows.
    end do
    s(iz1:iz2,jmf)=s(iz1:iz2,jmf)-s(iz1:iz2,j)*c(ic,jcf,kc)     Plus final element.
    ic=ic+1
end do
END SUBROUTINE red
END SUBROUTINE solvde
```

f90 km=imaxloc... See discussion of imaxloc on p. 1017.

★ ★ ★

```
MODULE sfroid_data                          Communicates with difeq.
USE nrtype
INTEGER(I4B), PARAMETER :: M=41
INTEGER(I4B) :: mm,n
REAL(SP) :: anorm,c2,h
REAL(SP), DIMENSION(M) :: x
END MODULE sfroid_data
```

```
PROGRAM sfroid
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : plgndr,solvde
USE sfroid_data
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NE=3,NB=1
    Sample program using solvde. Computes eigenvalues of spheroidal harmonics S_mn(x; c)
    for m ≥ 0 and n ≥ m. In the program, m is mm, c^2 is c2, and γ of equation (17.4.20)
    is anorm.
INTEGER(I4B) :: itmax
INTEGER(I4B), DIMENSION(NE) :: indexv
REAL(SP) :: conv,slowc
REAL(SP), DIMENSION(M) :: deriv,fac1,fac2
REAL(SP), DIMENSION(NE) :: scalv
REAL(SP), DIMENSION(NE,M) :: y
itmax=100
conv=5.0e-6_sp
slowc=1.0
h=1.0_sp/(M-1)
c2=0.0
write(*,*) 'ENTER M,N'
read(*,*) mm,n
indexv(1:3)=merge( (/ 1, 2, 3 /), (/ 2, 1, 3 /), (mod(n+mm,2) == 1) )
  No interchanges necessary if n+mm is odd; otherwise interchange y_1 and y_2.
anorm=1.0                                   Compute γ.
if (mm /= 0) then
    anorm=(-0.5_sp)**mm*product(&
        arth(n+1,1,mm)*arth(real(n,sp),-1.0_sp,mm)/arth(1,1,mm))
end if
x(1:M-1)=arth(0,1,M-1)*h
fac1(1:M-1)=1.0_sp-x(1:M-1)**2              Compute initial guess.
fac2(1:M-1)=fac1(1:M-1)**(-mm/2.0_sp)
y(1,1:M-1)=plgndr(n,mm,x(1:M-1))*fac2(1:M-1)      P_n^m from §6.8.
deriv(1:M-1)=-((n-mm+1)*plgndr(n+1,mm,x(1:M-1))-(n+1)*&
    x(1:M-1)*plgndr(n,mm,x(1:M-1)))/fac1(1:M-1)
      Derivative of P_n^m from a recurrence relation.
y(2,1:M-1)=mm*x(1:M-1)*y(1,1:M-1)/fac1(1:M-1)+deriv(1:M-1)*fac2(1:M-1)
y(3,1:M-1)=n*(n+1)-mm*(mm+1)
x(M)=1.0                                    Initial guess at x = 1 done separately.
y(1,M)=anorm
y(3,M)=n*(n+1)-mm*(mm+1)
y(2,M)=(y(3,M)-c2)*y(1,M)/(2.0_sp*(mm+1.0_sp))
scalv(1:3)=(/ abs(anorm), max(abs(anorm),y(2,M)), max(1.0_sp,y(3,M)) /)
do
    write (*,*) 'ENTER C**2 OR 999 TO END'
    read (*,*) c2
    if (c2 == 999.0) exit
    call solvde(itmax,conv,slowc,scalv,indexv,NB,y)
    write (*,*) ' M = ',mm,' N = ',n,&
        ' C**2 = ',c2,' LAMBDA = ',y(3,1)+mm*(mm+1)
end do                                      Go back for another value of c^2.
END PROGRAM sfroid
```

f90 `MODULE sfroid_data` This module functions just like a common block to communicate variables with `difeq`. The advantage of a module is that it allows complete specification of the variables.

`anorm=(-0.5_sp)**mm*product(...` This statement computes equation (17.4.20) by direct multiplication.

⋆ ⋆ ⋆

```
SUBROUTINE difeq(k,k1,k2,jsf,is1,isf,indexv,s,y)
USE nrtype
USE sfroid_data
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: is1,isf,jsf,k,k1,k2
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: s
REAL(SP), DIMENSION(:,:), INTENT(IN) :: y
   Returns matrix s(i,j) for solvde.
REAL(SP) :: temp,temp2
INTEGER(I4B), DIMENSION(3) :: indexv3
indexv3(1:3)=3+indexv(1:3)
if (k == k1) then           Boundary condition at first point.
    if (mod(n+mm,2) == 1) then
        s(3,indexv3(1:3))= (/ 1.0_sp, 0.0_sp, 0.0_sp /)        Equation (17.4.32).
        s(3,jsf)=y(1,1)                                        Equation (17.4.31).
    else
        s(3,indexv3(1:3))= (/ 0.0_sp, 1.0_sp, 0.0_sp /)        Equation (17.4.32).
        s(3,jsf)=y(2,1)                                        Equation (17.4.31).
    end if
else if (k > k2) then       Boundary conditions at last point.
    s(1,indexv3(1:3))= (/ -(y(3,M)-c2)/(2.0_sp*(mm+1.0_sp)),&
        1.0_sp, -y(1,M)/(2.0_sp*(mm+1.0_sp)) /)                Equation (17.4.35).
    s(1,jsf)=y(2,M)-(y(3,M)-c2)*y(1,M)/(2.0_sp*(mm+1.0_sp))    Equation (17.4.33).
    s(2,indexv3(1:3))=(/ 1.0_sp, 0.0_sp, 0.0_sp /)             Equation (17.4.36).
    s(2,jsf)=y(1,M)-anorm                                      Equation (17.4.34).
else                        Interior point.
    s(1,indexv(1:3))=(/ -1.0_sp, -0.5_sp*h, 0.0_sp /)          Equation (17.4.28).
    s(1,indexv3(1:3))=(/ 1.0_sp, -0.5_sp*h, 0.0_sp /)
    temp=h/(1.0_sp-(x(k)+x(k-1))**2*0.25_sp)
    temp2=0.5_sp*(y(3,k)+y(3,k-1))-c2*0.25_sp*(x(k)+x(k-1))**2
    s(2,indexv(1:3))=(/ temp*temp2*0.5_sp,&                    Equation (17.4.29).
        -1.0_sp-0.5_sp*temp*(mm+1.0_sp)*(x(k)+x(k-1)),&
        0.25_sp*temp*(y(1,k)+y(1,k-1)) /)
    s(2,indexv3(1:3))=s(2,indexv(1:3))
    s(2,indexv3(2))=s(2,indexv3(2))+2.0_sp
    s(3,indexv(1:3))=(/ 0.0_sp, 0.0_sp, -1.0_sp /)             Equation (17.4.30).
    s(3,indexv3(1:3))=(/ 0.0_sp, 0.0_sp, 1.0_sp /)
    s(1,jsf)=y(1,k)-y(1,k-1)-0.5_sp*h*(y(2,k)+y(2,k-1))        Equation (17.4.23).
    s(2,jsf)=y(2,k)-y(2,k-1)-temp*((x(k)+x(k-1))*&             Equation (17.4.24).
        0.5_sp*(mm+1.0_sp)*(y(2,k)+y(2,k-1))-temp2*&
        0.5_sp*(y(1,k)+y(1,k-1)))
    s(3,jsf)=y(3,k)-y(3,k-1)                                   Equation (17.4.27).
end if
END SUBROUTINE difeq
```

⋆ ⋆ ⋆

```
MODULE sphoot_data                          Communicates with load, score, and derivs.
USE nrtype
INTEGER(I4B) :: m,n
REAL(SP) :: c2,dx,gamma
END MODULE sphoot_data
```

```
MODULE sphoot_caller                        Communicates with shoot.
USE nrtype
INTEGER(I4B) :: nvar
REAL(SP) :: x1,x2
END MODULE sphoot_caller
```

```
PROGRAM sphoot
    Sample program using shoot. Computes eigenvalues of spheroidal harmonics S_mn(x;c) for
    m ≥ 0 and n ≥ m. Be sure that routine funcv for newt is provided by shoot (§17.1).
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : newt
USE sphoot_data
USE sphoot_caller
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NV=3,N2=1
REAL(SP), DIMENSION(N2) :: v
LOGICAL(LGT) :: check
nvar=NV                                     Number of equations.
dx=1.0e-4_sp                                Avoid evaluating derivatives exactly at x =
do                                              −1.
    write(*,*) 'input m,n,c-squared (999 to end)'
    read(*,*) m,n,c2
    if (c2 == 999.0) exit
    if ((n < m) .or. (m < 0)) cycle
    gamma=(-0.5_sp)**m*product(&            Compute γ of equation (17.4.20).
        arth(n+1,1,m)*(arth(real(n,sp),-1.0_sp,m)/arth(1,1,m)))
    v(1)=n*(n+1)-m*(m+1)+c2/2.0_sp          Initial guess for eigenvalue.
    x1=-1.0_sp+dx                           Set range of integration.
    x2=0.0
    call newt(v,check)                      Find v that zeros function f in score.
    if (check) then
        write(*,*)'shoot failed; bad initial guess'
        exit
    else
        write(*,'(1x,t6,a)') 'mu(m,n)'
        write(*,'(1x,f12.6)') v(1)
    end if
end do
END PROGRAM sphoot
```

```
SUBROUTINE load(x1,v,y)
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1
REAL(SP), DIMENSION(:), INTENT(IN) :: v
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
    Supplies starting values for integration at x = −1 + dx.
REAL(SP) :: y1
y(3)=v(1)
y1=merge(gamma,-gamma, mod(n-m,2) == 0 )
y(2)=-(y(3)-c2)*y1/(2*(m+1))
y(1)=y1+y(2)*dx
END SUBROUTINE load
```

```
SUBROUTINE score(x2,y,f)
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: f
    Tests whether boundary condition at x = 0 is satisfied.
f(1)=merge(y(2),y(1), mod(n-m,2) == 0 )
END SUBROUTINE score
```

f90 `MODULE sphoot_data...MODULE sphoot_caller` These modules function just like common blocks to communicate variables from `sphoot` to the various subsidiary routines. The advantage of a module is that it allows complete specification of the variables.

```
SUBROUTINE derivs(x,y,dydx)
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    Evaluates derivatives for odeint.
dydx(1)=y(2)
dydx(2)=(2.0_sp*x*(m+1.0_sp)*y(2)-(y(3)-c2*x*x)*y(1))/(1.0_sp-x*x)
dydx(3)=0.0
END SUBROUTINE derivs
```

⋆ ⋆ ⋆

```
MODULE sphfpt_data
USE nrtype
INTEGER(I4B) :: m,n
REAL(SP) :: c2,dx,gamma
END MODULE sphfpt_data
```

Communicates with `load1`, `load2`, `score`, and `derivs`.

```
MODULE sphfpt_caller
USE nrtype
INTEGER(I4B) :: nn2
REAL(SP) :: x1,x2,xf
END MODULE sphfpt_caller
```

Communicates with `shootf`.

```
PROGRAM sphfpt
    Sample program using shootf. Computes eigenvalues of spheroidal harmonics S_mn(x;c)
    for m ≥ 0 and n ≥ m. Be sure that routine funcv for newt is provided by shootf (§17.2).
    The routine derivs is the same as for sphoot.
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : newt
USE sphfpt_data
USE sphfpt_caller
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: N1=2,N2=1,NTOT=N1+N2
REAL(SP), PARAMETER :: DXX=1.0e-4_sp
REAL(SP), DIMENSION(:), POINTER :: v1,v2
REAL(SP), DIMENSION(NTOT), TARGET :: v
LOGICAL(LGT) :: check
v1=>v(1:N2)
v2=>v(N2+1:NTOT)
nn2=N2
dx=DXX                                      Avoid evaluating derivatives exactly at x =
do                                              ±1.
    write(*,*) 'input m,n,c-squared (999 to end)'
    read(*,*) m,n,c2
    if (c2 == 999.0) exit
    if ((n < m) .or. (m < 0)) cycle
    gamma=(-0.5_sp)**m*product(&                Compute γ of equation (17.4.20).
        arth(n+1,1,m)*(arth(real(n,sp),-1.0_sp,m)/arth(1,1,m)))
    v1(1)=n*(n+1)-m*(m+1)+c2/2.0_sp             Initial guess for eigenvalue and function value.
    v2(2)=v1(1)
    v2(1)=gamma*(1.0_sp-(v2(2)-c2)*dx/(2*(m+1)))
    x1=-1.0_sp+dx                               Set range of integration.
    x2=1.0_sp-dx
    xf=0.0                                      Fitting point.
    call newt(v,check)                          Find v that zeros function f in score.
    if (check) then
        write(*,*) 'shootf failed; bad initial guess'
        exit
    else
        write(*,'(1x,t6,a)') 'mu(m,n)'
        write(*,'(1x,f12.6)') v1(1)
    end if
end do
END PROGRAM sphfpt
```

```
SUBROUTINE load1(x1,v1,y)
USE nrtype
USE sphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1
REAL(SP), DIMENSION(:), INTENT(IN) :: v1
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
    Supplies starting values for integration at x = −1 + dx.
REAL(SP) :: y1
y(3)=v1(1)
y1=merge(gamma,-gamma,mod(n-m,2) == 0)
y(2)=-(y(3)-c2)*y1/(2*(m+1))
y(1)=y1+y(2)*dx
END SUBROUTINE load1
```

```
SUBROUTINE load2(x2,v2,y)
USE nrtype
USE sphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:), INTENT(IN) :: v2
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
    Supplies starting values for integration at x = 1 − dx.
y(3)=v2(2)
y(1)=v2(1)
y(2)=(y(3)-c2)*y(1)/(2*(m+1))
END SUBROUTINE load2



SUBROUTINE score(xf,y,f)
USE nrtype
USE sphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xf
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: f
    Tests whether solutions match at fitting point x = 0.
f(1:3)=y(1:3)
END SUBROUTINE score
```

f90 `MODULE sphfpt_data...MODULE sphfpt_caller` These modules function just like common blocks to communicate variables from `sphfpt` to the various subsidiary routines. The advantage of a module is that it allows complete specification of the variables.