Chapter B19. Partial Differential Equations

```
SUBROUTINE sor(a,b,c,d,e,f,u,rjac)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: a,b,c,d,e,f
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), INTENT(IN) :: rjac
INTEGER(I4B), PARAMETER :: MAXITS=1000
REAL(DP), PARAMETER :: EPS=1.0e-5_dp
```

Successive overrelaxation solution of equation (19.5.25) with Chebyshev acceleration. a, b, c, d, e, and f are input as the coefficients of the equation, each dimensioned to the grid size $J \times J$. u is input as the initial guess to the solution, usually zero, and returns with the final value. rjac is input as the spectral radius of the Jacobi iteration, or an estimate of it. Double precision is a good idea for J bigger than about 25.

```
REAL(DP), DIMENSION(size(a,1),size(a,1)) :: resid
INTEGER(I4B) :: jmax,jm1,jm2,jm3,n
REAL(DP) :: anorm,anormf,omega
jmax=assert_eq((/size(a,1),size(a,2),size(b,1),size(b,2), &
    size(c,1),size(c,2),size(d,1),size(d,2),size(e,1), &
    size(e,2),size(f,1),size(f,2),size(u,1),size(u,2)/),'sor')
jm1=jmax-1
jm2=jmax-2
jm3=jmax-3
anormf=sum(abs(f(2:jm1,2:jm1)))
```

Compute initial norm of residual and terminate iteration when norm has been reduced by a factor EPS. This computation assumes initial u is zero.

```
omega=1.0
do n=1,MAXITS
```

First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:

```
    resid(2:jm1:2,2:jm1:2)=a(2:jm1:2,2:jm1:2)*u(3:jmax:2,2:jm1:2)+&
        b(2:jm1:2,2:jm1:2)*u(1:jm2:2,2:jm1:2)+&
        c(2:jm1:2,2:jm1:2)*u(2:jm1:2,3:jmax:2)+&
        d(2:jm1:2,2:jm1:2)*u(2:jm1:2,1:jm2:2)+&
        e(2:jm1:2,2:jm1:2)*u(2:jm1:2,2:jm1:2)-f(2:jm1:2,2:jm1:2)
    u(2:jm1:2,2:jm1:2)=u(2:jm1:2,2:jm1:2)-omega*&
        resid(2:jm1:2,2:jm1:2)/e(2:jm1:2,2:jm1:2)
    resid(3:jm2:2,3:jm2:2)=a(3:jm2:2,3:jm2:2)*u(4:jm1:2,3:jm2:2)+&
        b(3:jm2:2,3:jm2:2)*u(2:jm3:2,3:jm2:2)+&
        c(3:jm2:2,3:jm2:2)*u(3:jm2:2,4:jm1:2)+&
        d(3:jm2:2,3:jm2:2)*u(3:jm2:2,2:jm3:2)+&
        e(3:jm2:2,3:jm2:2)*u(3:jm2:2,3:jm2:2)-f(3:jm2:2,3:jm2:2)
    u(3:jm2:2,3:jm2:2)=u(3:jm2:2,3:jm2:2)-omega*&
        resid(3:jm2:2,3:jm2:2)/e(3:jm2:2,3:jm2:2)
    omega=merge(1.0_dp/(1.0_dp-0.5_dp*rjac**2), &
        1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega), n == 1)
```

Now do even-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:

```
    resid(3:jm2:2,2:jm1:2)=a(3:jm2:2,2:jm1:2)*u(4:jm1:2,2:jm1:2)+&
        b(3:jm2:2,2:jm1:2)*u(2:jm3:2,2:jm1:2)+&
```

```
        c(3:jm2:2,2:jm1:2)*u(3:jm2:2,3:jmax:2)+&
        d(3:jm2:2,2:jm1:2)*u(3:jm2:2,1:jm2:2)+&
        e(3:jm2:2,2:jm1:2)*u(3:jm2:2,2:jm1:2)-f(3:jm2:2,2:jm1:2)
    u(3:jm2:2,2:jm1:2)=u(3:jm2:2,2:jm1:2)-omega*&
        resid(3:jm2:2,2:jm1:2)/e(3:jm2:2,2:jm1:2)
    resid(2:jm1:2,3:jm2:2)=a(2:jm1:2,3:jm2:2)*u(3:jmax:2,3:jm2:2)+&
        b(2:jm1:2,3:jm2:2)*u(1:jm2:2,3:jm2:2)+&
        c(2:jm1:2,3:jm2:2)*u(2:jm1:2,4:jm1:2)+&
        d(2:jm1:2,3:jm2:2)*u(2:jm1:2,2:jm3:2)+&
        e(2:jm1:2,3:jm2:2)*u(2:jm1:2,3:jm2:2)-f(2:jm1:2,3:jm2:2)
    u(2:jm1:2,3:jm2:2)=u(2:jm1:2,3:jm2:2)-omega*&
        resid(2:jm1:2,3:jm2:2)/e(2:jm1:2,3:jm2:2)
    omega=1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega)
    anorm=sum(abs(resid(2:jm1,2:jm1)))
    if (anorm < EPS*anormf) exit
end do
if (n > MAXITS) call nrerror('MAXITS exceeded in sor')
END SUBROUTINE sor
```

Red-black iterative schemes like the one used in `sor` are easily parallelizable. Updating the red grid points requires information only from the black grid points, so they can all be updated independently. Similarly the black grid points can all be updated independently. Since nearest neighbors are involved in the updating, communication costs can be kept to a minimum.

There are several possibilities for coding the red-black iteration in a data parallel way using only Fortran 90 and no parallel language extensions. One way is to define an $N \times N$ logical mask `red` that is true on the red grid points and false on the black. Then each iteration consists of an update governed by a `where(red)...end where` block and a `where(.not. red)...end where` block. We have chosen a more direct coding that avoids the need for storage of the array `red`. The red update corresponds to the even-even and odd-odd grid points, the black to the even-odd and odd-even points. We can code each of these four cases directly with array sections, as in the routine above.

The array section notation used in `sor` is rather dense and hard to read. We could use pointer aliases to try to simplify things, but since each array section is different, we end up merely giving names to each term that was there all along. Pointer aliases do help if we code `sor` using a logical mask. Since there may be machines on which this version is faster, and since it is of some pedagogic interest, we give the alternative code:

```
SUBROUTINE sor_mask(a,b,c,d,e,f,u,rjac)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), TARGET, INTENT(IN) :: a,b,c,d,e,f
REAL(DP), DIMENSION(:,:), TARGET, INTENT(INOUT) :: u
REAL(DP), INTENT(IN) :: rjac
INTEGER(I4B), PARAMETER :: MAXITS=1000
REAL(DP), PARAMETER :: EPS=1.0e-5_dp
REAL(DP), DIMENSION(:,:), ALLOCATABLE :: resid
REAL(DP), DIMENSION(:,:), POINTER :: u_int,u_down,u_up,u_left,&
    u_right,a_int,b_int,c_int,d_int,e_int,f_int
INTEGER(I4B) :: jmax,jm1,jm2,jm3,n
REAL(DP) anorm,anormf,omega
LOGICAL, DIMENSION(:,:), ALLOCATABLE :: red
jmax=assert_eq((/size(a,1),size(a,2),size(b,1),size(b,2), &
```

```
    size(c,1),size(c,2),size(d,1),size(d,2),size(e,1), &
    size(e,2),size(f,1),size(f,2),size(u,1),size(u,2)/),'sor')
jm1=jmax-1
jm2=jmax-2
jm3=jmax-3
allocate(resid(jm2,jm2),red(jm2,jm2))          Interior is (jmax − 2) × (jmax − 2).
red=.false.
red(1:jm2:2,1:jm2:2)=.true.
red(2:jm3:2,2:jm3:2)=.true.
u_int=>u(2:jm1,2:jm1)
u_down=>u(3:jmax,2:jm1)
u_up=>u(1:jm2,2:jm1)
u_left=>u(2:jm1,1:jm2)
u_right=>u(2:jm1,3:jmax)
a_int=>a(2:jm1,2:jm1)
b_int=>b(2:jm1,2:jm1)
c_int=>c(2:jm1,2:jm1)
d_int=>d(2:jm1,2:jm1)
e_int=>e(2:jm1,2:jm1)
f_int=>f(2:jm1,2:jm1)
anormf=sum(abs(f_int))
omega=1.0
do n=1,MAXITS
    where(red)
        resid=a_int*u_down+b_int*u_up+c_int*u_right+&
            d_int*u_left+e_int*u_int-f_int
        u_int=u_int-omega*resid/e_int
    end where
    omega=merge(1.0_dp/(1.0_dp-0.5_dp*rjac**2), &
        1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega), n == 1)
    where(.not.red)
        resid=a_int*u_down+b_int*u_up+c_int*u_right+&
            d_int*u_left+e_int*u_int-f_int
        u_int=u_int-omega*resid/e_int
    end where
    omega=1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega)
    anorm=sum(abs(resid))
    if(anorm < EPS*anormf)exit
end do
deallocate(resid,red)
if (n > MAXITS) call nrerror('MAXITS exceeded in sor')
END SUBROUTINE sor_mask
```

⋆ ⋆ ⋆

```
SUBROUTINE mglin(u,ncycle)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : interp,rstrct,slvsml
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: ncycle
```

Full Multigrid Algorithm for solution of linear elliptic equation, here the model problem (19.0.6). On input u contains the right-hand side ρ in an $N \times N$ array, while on output it returns the solution. The dimension N is related to the number of grid levels used in the solution, ng below, by $N =$ 2**ng+1. ncycle is the number of V-cycles to be used at each level.

```
INTEGER(I4B) :: j,jcycle,n,ng,ngrid,nn
TYPE ptr2d                                  Define a type so we can have an array of pointers
    REAL(DP), POINTER :: a(:,:)                 to arrays of grid variables.
END TYPE ptr2d
TYPE(ptr2d), ALLOCATABLE :: rho(:)
```

```
REAL(DP), DIMENSION(:,:), POINTER :: uj,uj_1
n=assert_eq(size(u,1),size(u,2),'mglin')
ng=nint(log(n-1.0)/log(2.0))
if (n /= 2**ng+1) call nrerror('n-1 must be a power of 2 in mglin')
allocate(rho(ng))
nn=n
ngrid=ng
allocate(rho(ngrid)%a(nn,nn))          Allocate storage for r.h.s. on grid ng,
rho(ngrid)%a=u                         and fill it with the input r.h.s.
do                                     Similarly allocate storage and fill r.h.s. on all coarse
    if (nn <= 3) exit                      grids by restricting from finer grids.
    nn=nn/2+1
    ngrid=ngrid-1
    allocate(rho(ngrid)%a(nn,nn))
    rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
end do
nn=3
allocate(uj(nn,nn))
call slvsml(uj,rho(1)%a)               Initial solution on coarsest grid.
do j=2,ng                              Nested iteration loop.
    nn=2*nn-1
    uj_1=>uj
    allocate(uj(nn,nn))
    uj=interp(uj_1)                    Interpolate from grid j-1 to next finer grid j.
    deallocate(uj_1)
    do jcycle=1,ncycle                 V-cycle loop.
        call mg(j,uj,rho(j)%a)
    end do
end do
u=uj                                   Return solution in u.
deallocate(uj)
do j=1,ng
    deallocate(rho(j)%a)
end do
deallocate(rho)
CONTAINS
RECURSIVE SUBROUTINE mg(j,u,rhs)
USE nrtype
USE nr, ONLY : interp,relax,resid,rstrct,slvsml
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: j
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
INTEGER(I4B), PARAMETER :: NPRE=1,NPOST=1
    Recursive multigrid iteration. On input, j is the current level, u is the current value of the
    solution, and rhs is the right-hand side. On output u contains the improved solution at the
    current level.
    Parameters: NPRE and NPOST are the number of relaxation sweeps before and after the
    coarse-grid correction is computed.
INTEGER(I4B) :: jpost,jpre
REAL(DP), DIMENSION((size(u,1)+1)/2,(size(u,1)+1)/2) :: res,v
if (j == 1) then                       Bottom of V: Solve on coarsest grid.
    call slvsml(u,rhs)
else                                   On downward stoke of the V.
    do jpre=1,NPRE                     Pre-smoothing.
        call relax(u,rhs)
    end do
    res=rstrct(resid(u,rhs))           Restriction of the residual is the next r.h.s.
    v=0.0                              Zero for initial guess in next relaxation.
    call mg(j-1,v,res)                 Recursive call for the coarse grid correction.
    u=u+interp(v)                      On upward stroke of V.
    do jpost=1,NPOST                   Post-smoothing.
        call relax(u,rhs)
```

```
    end do
end if
END SUBROUTINE mg
END SUBROUTINE mglin
```

f90 The Fortran 90 version of mglin (and of mgfas below) is quite different from the Fortran 77 version, although the algorithm is identical. First, we use a recursive implementation. This makes the code much more transparent. It also makes the memory management much better: we simply define the new arrays res and v as automatic arrays of the appropriate dimension on each recursive call to a coarser level. And a third benefit is that it is trivial to change the code to increase the number of multigrid iterations done at level $j-1$ by each iteration at level j, i.e., to set the quantity γ in §19.6 to a value greater than one. (Recall that $\gamma = 1$ as chosen in mglin gives V-cycles, $\gamma = 2$ gives W-cycles.) Simply enclose the recursive call in a do-loop:

```
do i=1,merge(gamma,1,j /= 2)
   call mg(j-1,v,res)
end do
```

The merge expression ensures that there is no more than one call to the coarsest level, where the problem is solved exactly.

A second improvement in the Fortran 90 version is to make the procedures resid, interp, and rstrct functions instead of subroutines. This allows us to code the algorithm exactly as written mathematically.

TYPE ptr2d... The right-hand-side quantity ρ is supplied initially on the finest grid in the argument u. It has to be defined on the coarser grids by restriction, and then supplied as the right-hand side to mg in the nested iteration loop. This loop starts at the coarsest level and progresses up to the finest level. We thus need a data structure to store ρ on all the grid levels. A convenient way to implement this in Fortran 90 is to define a type ptr2d, a pointer to a two-dimensional array a that represents a grid. (In three dimensions, a would of course be three-dimensional.) We then declare the variable ρ as an allocatable array of type ptr2d:

```
TYPE(ptr2d), ALLOCATABLE :: rho(:)
```

Next we allocate storage for ρ on each level. The number of levels or grids, ng, is known only at run time:

```
allocate(rho(ng))
```

Then we allocate storage as needed on particular sized grids. For example,

```
allocate(rho(ngrid)%a(nn,nn))
```

allocates an nn $\times$ nn grid for rho on grid number ngrid.

The various subsidiary routines of mglin such as rstrct and interp are written to accept two-dimensional arrays as arguments. With the data structure we've employed, using these routines is simple. For example,

```
rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
```

will restrict rho from the grid ngrid+1 to the grid ngrid. The statement is even more readable if we mentally ignore the %a that is tagged onto each variable. (If

we actually did omit %a in the code, the compiler would think we meant the array of type ptr2d instead of the grid array.)

Note that while Fortran 90 does not allow you to declare an array of pointers directly, you can achieve the same effect by declaring your own type, as we have done with ptr2d in this example.

```
FUNCTION rstrct(uf)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: uf
REAL(DP), DIMENSION((size(uf,1)+1)/2,(size(uf,1)+1)/2) :: rstrct
```

Half-weighting restriction. If N_c is the coarse-grid dimension, the fine-grid solution is input in the $(2N_c-1)\times(2N_c-1)$ array uf, the coarse-grid solution is returned in the $N_c \times N_c$ array rstrct.

```
INTEGER(I4B) :: nc,nf
nf=assert_eq(size(uf,1),size(uf,2),'rstrct')
nc=(nf+1)/2
rstrct(2:nc-1,2:nc-1)=0.5_dp*uf(3:nf-2:2,3:nf-2:2)+0.125_dp*(&     Interior points.
    uf(4:nf-1:2,3:nf-2:2)+uf(2:nf-3:2,3:nf-2:2)+&
    uf(3:nf-2:2,4:nf-1:2)+uf(3:nf-2:2,2:nf-3:2))
rstrct(1:nc,1)=uf(1:nf:2,1)                                        Boundary points.
rstrct(1:nc,nc)=uf(1:nf:2,nf)
rstrct(1,1:nc)=uf(1,1:nf:2)
rstrct(nc,1:nc)=uf(nf,1:nf:2)
END FUNCTION rstrct



FUNCTION interp(uc)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: uc
REAL(DP), DIMENSION(2*size(uc,1)-1,2*size(uc,1)-1) :: interp
```

Coarse-to-fine prolongation by bilinear interpolation. If N_f is the fine-grid dimension and N_c the coarse-grid dimension, then $N_f = 2N_c - 1$. The coarse-grid solution is input as uc, the fine-grid solution is returned in interp.

```
INTEGER(I4B) :: nc,nf
nc=assert_eq(size(uc,1),size(uc,2),'interp')
nf=2*nc-1
interp(1:nf:2,1:nf:2)=uc(1:nc,1:nc)
```

Do elements that are copies.

```
interp(2:nf-1:2,1:nf:2)=0.5_dp*(interp(3:nf:2,1:nf:2)+ &
    interp(1:nf-2:2,1:nf:2))
```

Do odd-numbered columns, interpolating vertically.

```
interp(1:nf,2:nf-1:2)=0.5_dp*(interp(1:nf,3:nf:2)+interp(1:nf,1:nf-2:2))
```

Do even-numbered columns, interpolating horizontally.

```
END FUNCTION interp



SUBROUTINE slvsml(u,rhs)
USE nrtype
IMPLICIT NONE
REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
```

Solution of the model problem on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in rhs(1:3,1:3) and the solution is returned in u(1:3,1:3).

```
REAL(DP) :: h
u=0.0
h=0.5_dp
u(2,2)=-h*h*rhs(2,2)/4.0_dp
END SUBROUTINE slvsml
```

```
SUBROUTINE relax(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
```

Red-black Gauss-Seidel relaxation for model problem. The current value of the solution `u` is updated, using the right-hand-side function `rhs`. `u` and `rhs` are square arrays of the same odd dimension.

```
INTEGER(I4B) :: n
REAL(DP) :: h,h2
n=assert_eq(size(u,1),size(u,2),size(rhs,1),size(rhs,2),'relax')
h=1.0_dp/(n-1)
h2=h*h
```

First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:

```
u(2:n-1:2,2:n-1:2)=0.25_dp*(u(3:n:2,2:n-1:2)+u(1:n-2:2,2:n-1:2)+&
    u(2:n-1:2,3:n:2)+u(2:n-1:2,1:n-2:2)-h2*rhs(2:n-1:2,2:n-1:2))
u(3:n-2:2,3:n-2:2)=0.25_dp*(u(4:n-1:2,3:n-2:2)+u(2:n-3:2,3:n-2:2)+&
    u(3:n-2:2,4:n-1:2)+u(3:n-2:2,2:n-3:2)-h2*rhs(3:n-2:2,3:n-2:2))
```

Now do even-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:

```
u(3:n-2:2,2:n-1:2)=0.25_dp*(u(4:n-1:2,2:n-1:2)+u(2:n-3:2,2:n-1:2)+&
    u(3:n-2:2,3:n:2)+u(3:n-2:2,1:n-2:2)-h2*rhs(3:n-2:2,2:n-1:2))
u(2:n-1:2,3:n-2:2)=0.25_dp*(u(3:n:2,3:n-2:2)+u(1:n-2:2,3:n-2:2)+&
    u(2:n-1:2,4:n-1:2)+u(2:n-1:2,2:n-3:2)-h2*rhs(2:n-1:2,3:n-2:2))
END SUBROUTINE relax
```

See the discussion of red-black relaxation after `sor` on p. 1333.

```
FUNCTION resid(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,rhs
REAL(DP), DIMENSION(size(u,1),size(u,1)) :: resid
```

Returns *minus* the residual for the model problem. Input quantities are `u` and `rhs`, while the residual is returned in `resid`. All three quantities are square arrays with the same odd dimension.

```
INTEGER(I4B) :: n
REAL(DP) :: h,h2i
n=assert_eq((/size(u,1),size(u,2),size(rhs,1),size(rhs,2)/),'resid')
n=size(u,1)
h=1.0_dp/(n-1)
h2i=1.0_dp/(h*h)
resid(2:n-1,2:n-1)=-h2i*(u(3:n,2:n-1)+u(1:n-2,2:n-1)+u(2:n-1,3:n)+&
    u(2:n-1,1:n-2)-4.0_dp*u(2:n-1,2:n-1))+rhs(2:n-1,2:n-1)      Interior points.
resid(1:n,1)=0.0                                                Boundary points.
resid(1:n,n)=0.0
resid(1,1:n)=0.0
resid(n,1:n)=0.0
END FUNCTION resid
```

⋆ ⋆ ⋆

```
SUBROUTINE mgfas(u,maxcyc)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : interp,lop,rstrct,slvsm2
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: maxcyc
```

Full Multigrid Algorithm for FAS solution of nonlinear elliptic equation, here equation (19.6.44). On input u contains the right-hand side ρ in an $N \times N$ array, while on output it returns the solution. The dimension N is related to the number of grid levels used in the solution, ng below, by $N = 2^{**}ng+1$. maxcyc is the maximum number of V-cycles to be used at each level.

```
INTEGER(I4B) :: j,jcycle,n,ng,ngrid,nn
REAL(DP) :: res,trerr
TYPE ptr2d                                  Define a type so we can have an array of
    REAL(DP), POINTER :: a(:,:)                 pointers to arrays of grid variables.
END TYPE ptr2d
TYPE(ptr2d), ALLOCATABLE :: rho(:)
REAL(DP), DIMENSION(:,:), POINTER :: uj,uj_1
n=assert_eq(size(u,1),size(u,2),'mgfas')
ng=nint(log(n-1.0)/log(2.0))
if (n /= 2**ng+1) call nrerror('n-1 must be a power of 2 in mgfas')
allocate(rho(ng))
nn=n
ngrid=ng
allocate(rho(ngrid)%a(nn,nn))               Allocate storage for r.h.s. on grid ng,
rho(ngrid)%a=u                              and fill it with ρ from the fine grid.
do                                          Similarly allocate storage and fill r.h.s. by re-
    if (nn <= 3) exit                           striction on all coarse grids.
    nn=nn/2+1
    ngrid=ngrid-1
    allocate(rho(ngrid)%a(nn,nn))
    rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
end do
nn=3
allocate(uj(nn,nn))
call slvsm2(uj,rho(1)%a)                    Initial solution on coarsest grid.
do j=2,ng                                   Nested iteration loop.
    nn=2*nn-1
    uj_1=>uj
    allocate(uj(nn,nn))
    uj=interp(uj_1)                         Interpolate from grid j-1 to next finer grid
    deallocate(uj_1)                            j.
    do jcycle=1,maxcyc                      V-cycle loop.
        call mg(j,uj,trerr=trerr)
        res=sqrt(sum((lop(uj)-rho(j)%a)**2))/nn     Form residual ‖d_h‖.
        if (res < trerr) exit               No more V-cycles needed if residual small
    end do                                      enough.
end do
u=uj                                        Return solution in u.
deallocate(uj)
do j=1,ng
    deallocate(rho(j)%a)
end do
deallocate(rho)
CONTAINS

RECURSIVE SUBROUTINE mg(j,u,rhs,trerr)
USE nrtype
USE nr, ONLY : interp,lop,relax2,rstrct,slvsm2
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: j
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN), OPTIONAL :: rhs
REAL(DP), INTENT(OUT), OPTIONAL :: trerr
```

```
INTEGER(I4B), PARAMETER :: NPRE=1,NPOST=1
REAL(DP), PARAMETER :: ALPHA=0.33_dp
```

Recursive multigrid iteration. On input, `j` is the current level and `u` is the current value of the solution. For the first call on a given level, the right-hand side is zero, and the optional argument `rhs` is not present. Subsequent recursive calls supply a nonzero `rhs` as in equation (19.6.33). On output `u` contains the improved solution at the current level. When the first call on a given level is made, the relative truncation error τ is returned in the optional argument `trerr`.

Parameters: `NPRE` and `NPOST` are the number of relaxation sweeps before and after the coarse-grid correction is computed; `ALPHA` relates the estimated truncation error to the norm of the residual.

```
INTEGER(I4B) :: jpost,jpre
REAL(DP), DIMENSION((size(u,1)+1)/2,(size(u,1)+1)/2) :: v,ut,tau
if (j == 1) then                          Bottom of V: Solve on coarsest grid.
    call slvsm2(u,rhs+rho(j)%a)
else                                      On downward stoke of the V.
    do jpre=1,NPRE                        Pre-smoothing.
        if (present(rhs)) then
            call relax2(u,rhs+rho(j)%a)
        else
            call relax2(u,rho(j)%a)
        end if
    end do
    ut=rstrct(u)                          \mathcal{R}\widetilde{u}_h.
    v=ut                                  Make a copy in v.
    if (present(rhs)) then
        tau=lop(ut)-rstrct(lop(u)-rhs)    Form \widetilde{\tau}_h + f_H = \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}\mathcal{L}_h(\widetilde{u}_h) + f_H.
    else
        tau=lop(ut)-rstrct(lop(u))
        trerr=ALPHA*sqrt(sum(tau**2))/size(tau,1)        Estimate truncation error \tau.
    end if
    call mg(j-1,v,tau)                    Recursive call for the coarse-grid correction.
    u=u+interp(v-ut)                      \widetilde{u}_h^{\rm new} = \widetilde{u}_h + \mathcal{P}(\widetilde{u}_H - \mathcal{R}\widetilde{u}_h)
    do jpost=1,NPOST                      Post-smoothing.
        if (present(rhs)) then
            call relax2(u,rhs+rho(j)%a)
        else
            call relax2(u,rho(j)%a)
        end if
    end do
end if
END SUBROUTINE mg
END SUBROUTINE mgfas
```

f90 See the discussion after `mglin` on p. 1336 for the changes made in the Fortran 90 versions of the multigrid routines from the Fortran 77 versions.

`TYPE ptr2d...` See discussion after `mglin` on p. 1336.

`RECURSIVE SUBROUTINE mg(j,u,rhs,trerr)` Recall that `mgfas` solves the problem $\mathcal{L}u = 0$, but that nonzero right-hand sides appear during the solution. We implement this by having `rhs` be an optional argument to `mg`. On the first call at a given level `j`, the right-hand side is zero and so you just omit it from the calling sequence. On the other hand, the truncation error `trerr` is computed only on the first call at a given level, so it is also an optional argument that does get supplied on the first call:

```
call mg(j,uj,trerr=trerr)
```

The second and subsequent calls at a given level supply `rhs=tau` but omit `trerr`:

```
call mg(j-1,v,tau)
```

Note that we can omit the keyword `rhs` from this call because the variable `tau` appears in the correct order of arguments. However, in the other call above, the keyword `trerr` must be supplied because `rhs` has been omitted.

The example equation that is solved in `mgfas`, equation (19.6.44), is almost linear, and the code is set up so that ρ is supplied as part of the right-hand side instead of pulling it over to the left-hand side. The variable `rho` is visible to `mg` by host association. Note also that the function `lop` does not include `rho`, but that the statement

```
tau=lop(ut)-rstrct(lop(u))
```

is nevertheless correct, since `rho` would cancel out if it were included in `lop`. This feature is also true in the Fortran 77 code.

```
SUBROUTINE relax2(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
```

Red-black Gauss-Seidel relaxation for equation (19.6.44). The current value of the solution `u` is updated, using the right-hand-side function `rhs`. `u` and `rhs` are square arrays of the same odd dimension.

```
INTEGER(I4B) :: n
REAL(DP) :: foh2,h,h2i
REAL(DP) :: res(size(u,1),size(u,1))
n=assert_eq(size(u,1),size(u,2),size(rhs,1),size(rhs,2),'relax2')
h=1.0_dp/(n-1)
h2i=1.0_dp/(h*h)
foh2=-4.0_dp*h2i
```

First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:

```
res(2:n-1:2,2:n-1:2)=h2i*(u(3:n:2,2:n-1:2)+u(1:n-2:2,2:n-1:2)+&
    u(2:n-1:2,3:n:2)+u(2:n-1:2,1:n-2:2)-4.0_dp*u(2:n-1:2,2:n-1:2))&
    +u(2:n-1:2,2:n-1:2)**2-rhs(2:n-1:2,2:n-1:2)
u(2:n-1:2,2:n-1:2)=u(2:n-1:2,2:n-1:2)-res(2:n-1:2,2:n-1:2)/&
    (foh2+2.0_dp*u(2:n-1:2,2:n-1:2))
res(3:n-2:2,3:n-2:2)=h2i*(u(4:n-1:2,3:n-2:2)+u(2:n-3:2,3:n-2:2)+&
    u(3:n-2:2,4:n-1:2)+u(3:n-2:2,2:n-3:2)-4.0_dp*u(3:n-2:2,3:n-2:2))&
    +u(3:n-2:2,3:n-2:2)**2-rhs(3:n-2:2,3:n-2:2)
u(3:n-2:2,3:n-2:2)=u(3:n-2:2,3:n-2:2)-res(3:n-2:2,3:n-2:2)/&
    (foh2+2.0_dp*u(3:n-2:2,3:n-2:2))
```

Now do even-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:

```
res(3:n-2:2,2:n-1:2)=h2i*(u(4:n-1:2,2:n-1:2)+u(2:n-3:2,2:n-1:2)+&
    u(3:n-2:2,3:n:2)+u(3:n-2:2,1:n-2:2)-4.0_dp*u(3:n-2:2,2:n-1:2))&
    +u(3:n-2:2,2:n-1:2)**2-rhs(3:n-2:2,2:n-1:2)
u(3:n-2:2,2:n-1:2)=u(3:n-2:2,2:n-1:2)-res(3:n-2:2,2:n-1:2)/&
    (foh2+2.0_dp*u(3:n-2:2,2:n-1:2))
res(2:n-1:2,3:n-2:2)=h2i*(u(3:n:2,3:n-2:2)+u(1:n-2:2,3:n-2:2)+&
    u(2:n-1:2,4:n-1:2)+u(2:n-1:2,2:n-3:2)-4.0_dp*u(2:n-1:2,3:n-2:2))&
    +u(2:n-1:2,3:n-2:2)**2-rhs(2:n-1:2,3:n-2:2)
u(2:n-1:2,3:n-2:2)=u(2:n-1:2,3:n-2:2)-res(2:n-1:2,3:n-2:2)/&
    (foh2+2.0_dp*u(2:n-1:2,3:n-2:2))
END SUBROUTINE relax2
```

f90 See the discussion of red-black relaxation after `sor` on p. 1333.

```
SUBROUTINE slvsm2(u,rhs)
USE nrtype
IMPLICIT NONE
REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
```
Solution of equation (19.6.44) on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in `rhs(1:3,1:3)` and the solution is returned in `u(1:3,1:3)`.
```
REAL(DP) :: disc,fact,h
u=0.0
h=0.5_dp
fact=2.0_dp/h**2
disc=sqrt(fact**2+rhs(2,2))
u(2,2)=-rhs(2,2)/(fact+disc)
END SUBROUTINE slvsm2



FUNCTION lop(u)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: u
REAL(DP), DIMENSION(size(u,1),size(u,1)) :: lop
```
Given `u`, returns $\mathcal{L}_h(\tilde{u}_h)$ for equation (19.6.44). `u` and `lop` are square arrays of the same odd dimension.
```
INTEGER(I4B) :: n
REAL(DP) :: h,h2i
n=assert_eq(size(u,1),size(u,2),'lop')
h=1.0_dp/(n-1)
h2i=1.0_dp/(h*h)
lop(2:n-1,2:n-1)=h2i*(u(3:n,2:n-1)+u(1:n-2,2:n-1)+u(2:n-1,3:n)+&
    u(2:n-1,1:n-2)-4.0_dp*u(2:n-1,2:n-1))+u(2:n-1,2:n-1)**2    Interior points.
lop(1:n,1)=0.0                                                 Boundary points.
lop(1:n,n)=0.0
lop(1,1:n)=0.0
lop(n,1:n)=0.0
END FUNCTION lop
```