Chapter B16. Integration of Ordinary Differential Equations

```
SUBROUTINE rk4(y,dydx,x,h,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
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
END INTERFACE
```

Given values for the N variables `y` and their derivatives `dydx` known at `x`, use the fourth-order Runge-Kutta method to advance the solution over an interval `h` and return the incremented variables as `yout`, which need not be a distinct array from `y`. `y`, `dydx` and `yout` are all of length N. The user supplies the subroutine `derivs(x,y,dydx)`, which returns derivatives `dydx` at `x`.

```
INTEGER(I4B) :: ndum
REAL(SP) :: h6,hh,xh
REAL(SP), DIMENSION(size(y)) :: dym,dyt,yt
ndum=assert_eq(size(y),size(dydx),size(yout),'rk4')
hh=h*0.5_sp
h6=h/6.0_sp
xh=x+hh
yt=y+hh*dydx                          First step.
call derivs(xh,yt,dyt)                Second step.
yt=y+hh*dyt
call derivs(xh,yt,dym)                Third step.
yt=y+h*dym
dym=dyt+dym
call derivs(x+h,yt,dyt)               Fourth step.
yout=y+h6*(dydx+dyt+2.0_sp*dym)       Accumulate increments with proper weights.
END SUBROUTINE rk4
```

⋆ ⋆ ⋆

```
MODULE rkdumb_path                    Storage of results.
USE nrtype
REAL(SP), DIMENSION(:), ALLOCATABLE:: xx
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: y
END MODULE rkdumb_path
```

```
SUBROUTINE rkdumb(vstart,x1,x2,nstep,derivs)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : rk4
USE rkdumb_path
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: vstart
REAL(SP), INTENT(IN) :: x1,x2
INTEGER(I4B), INTENT(IN) :: nstep
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
END INTERFACE
```

Starting from N initial values `vstart` known at `x1`, use fourth-order Runge-Kutta to advance `nstep` equal increments to `x2`. The user-supplied subroutine `derivs(x,y,dydx)` evaluates derivatives. Results are stored in the module variables `xx` and `y`.

```
INTEGER(I4B) :: k
REAL(SP) :: h,x
REAL(SP), DIMENSION(size(vstart)) :: dv,v
v(:)=vstart(:)                              Load starting values.
if (allocated(xx)) deallocate(xx)           Clear out old stored variables if necessary.
if (allocated(y)) deallocate(y)
allocate(xx(nstep+1))                       Allocate storage for saved values.
allocate(y(size(vstart),nstep+1))
y(:,1)=v(:)
xx(1)=x1
x=x1
h=(x2-x1)/nstep
do k=1,nstep                                Take nstep steps.
    call derivs(x,v,dv)
    call rk4(v,dv,x,h,v,derivs)
    if (x+h == x) call nrerror('stepsize not significant in rkdumb')
    x=x+h
    xx(k+1)=x                               Store intermediate steps.
    y(:,k+1)=v(:)
end do
END SUBROUTINE rkdumb
```

f90 `MODULE rkdumb_path` This routine needs straightforward communication of arrays with the calling program. The dimension of the arrays is not known in advance, and if the routine is called a second time we need to throw away the old array information. The Fortran 90 construction for this is to declare allocatable arrays in a module, and then test them at the beginning of the routine with `if (allocated...)`.

⋆ ⋆ ⋆

```
SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : rkck
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
```

```
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
END INTERFACE
```

Fifth order Runge-Kutta step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector `y` and its derivative `dydx` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal` against which the error is scaled. `y`, `dydx`, and `yscal` are all of the same length. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is the user-supplied subroutine that computes the right-hand-side derivatives.

```
INTEGER(I4B) :: ndum
REAL(SP) :: errmax,h,htemp,xnew
REAL(SP), DIMENSION(size(y)) :: yerr,ytemp
REAL(SP), PARAMETER :: SAFETY=0.9_sp,PGROW=-0.2_sp,PSHRNK=-0.25_sp,&
    ERRCON=1.89e-4
```

The value `ERRCON` equals `(5/SAFETY)**(1/PGROW)`, see use below.

```
ndum=assert_eq(size(y),size(dydx),size(yscal),'rkqs')
h=htry                                      Set stepsize to the initial trial value.
do
    call rkck(y,dydx,x,h,ytemp,yerr,derivs)     Take a step.
    errmax=maxval(abs(yerr(:)/yscal(:)))/eps    Evaluate accuracy.
    if (errmax <= 1.0) exit                 Step succeeded.
    htemp=SAFETY*h*(errmax**PSHRNK)         Truncation error too large, reduce stepsize.
    h=sign(max(abs(htemp),0.1_sp*abs(h)),h)     No more than a factor of 10.
    xnew=x+h
    if (xnew == x) call nrerror('stepsize underflow in rkqs')
end do                                      Go back for another try.
if (errmax > ERRCON) then                   Compute size of next step.
    hnext=SAFETY*h*(errmax**PGROW)
else                                        No more than a factor of 5 increase.
    hnext=5.0_sp*h
end if
hdid=h
x=x+h
y(:)=ytemp(:)
END SUBROUTINE rkqs
```

⋆ ⋆ ⋆

```
SUBROUTINE rkck(y,dydx,x,h,yout,yerr,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout,yerr
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
END INTERFACE
```

Given values for N variables `y` and their derivatives `dydx` known at `x`, use the fifth order Cash-Karp Runge-Kutta method to advance the solution over an interval `h` and return

the incremented variables as yout. Also return an estimate of the local truncation error in yout using the embedded fourth order method. The user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.

```
INTEGER(I4B) :: ndum
REAL(SP), DIMENSION(size(y)) :: ak2,ak3,ak4,ak5,ak6,ytemp
REAL(SP), PARAMETER :: A2=0.2_sp,A3=0.3_sp,A4=0.6_sp,A5=1.0_sp,&
    A6=0.875_sp,B21=0.2_sp,B31=3.0_sp/40.0_sp,B32=9.0_sp/40.0_sp,&
    B41=0.3_sp,B42=-0.9_sp,B43=1.2_sp,B51=-11.0_sp/54.0_sp,&
    B52=2.5_sp,B53=-70.0_sp/27.0_sp,B54=35.0_sp/27.0_sp,&
    B61=1631.0_sp/55296.0_sp,B62=175.0_sp/512.0_sp,&
    B63=575.0_sp/13824.0_sp,B64=44275.0_sp/110592.0_sp,&
    B65=253.0_sp/4096.0_sp,C1=37.0_sp/378.0_sp,&
    C3=250.0_sp/621.0_sp,C4=125.0_sp/594.0_sp,&
    C6=512.0_sp/1771.0_sp,DC1=C1-2825.0_sp/27648.0_sp,&
    DC3=C3-18575.0_sp/48384.0_sp,DC4=C4-13525.0_sp/55296.0_sp,&
    DC5=-277.0_sp/14336.0_sp,DC6=C6-0.25_sp
ndum=assert_eq(size(y),size(dydx),size(yout),size(yerr),'rkck')
ytemp=y+B21*h*dydx                          First step.
call derivs(x+A2*h,ytemp,ak2)               Second step.
ytemp=y+h*(B31*dydx+B32*ak2)
call derivs(x+A3*h,ytemp,ak3)               Third step.
ytemp=y+h*(B41*dydx+B42*ak2+B43*ak3)
call derivs(x+A4*h,ytemp,ak4)               Fourth step.
ytemp=y+h*(B51*dydx+B52*ak2+B53*ak3+B54*ak4)
call derivs(x+A5*h,ytemp,ak5)               Fifth step.
ytemp=y+h*(B61*dydx+B62*ak2+B63*ak3+B64*ak4+B65*ak5)
call derivs(x+A6*h,ytemp,ak6)               Sixth step.
yout=y+h*(C1*dydx+C3*ak3+C4*ak4+C6*ak6)     Accumulate increments with proper weights.
yerr=h*(DC1*dydx+DC3*ak3+DC4*ak4+DC5*ak5+DC6*ak6)
```

Estimate error as difference between fourth and fifth order methods.

```
END SUBROUTINE rkck
```

⋆ ⋆ ⋆

```
MODULE ode_path
USE nrtype
INTEGER(I4B) :: nok,nbad,kount
LOGICAL(LGT), SAVE :: save_steps=.false.
REAL(SP) :: dxsav
REAL(SP), DIMENSION(:), POINTER :: xp
REAL(SP), DIMENSION(:,:), POINTER :: yp
END MODULE ode_path
```

On output nok and nbad are the number of good and bad (but retried and fixed) steps taken. If save_steps is set to true in the calling program, then intermediate values are stored in xp and yp at intervals greater than dxsav. kount is the total number of saved steps.

```
SUBROUTINE odeint(ystart,x1,x2,eps,h1,hmin,derivs,rkqs)
USE nrtype; USE nrutil, ONLY : nrerror,reallocate
USE ode_path
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: ystart
REAL(SP), INTENT(IN) :: x1,x2,eps,h1,hmin
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 rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
```

```
    REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
    REAL(SP), INTENT(INOUT) :: x
    REAL(SP), INTENT(IN) :: htry,eps
    REAL(SP), INTENT(OUT) :: hdid,hnext
    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
    END INTERFACE
    END SUBROUTINE rkqs
END INTERFACE
REAL(SP), PARAMETER :: TINY=1.0e-30_sp
INTEGER(I4B), PARAMETER :: MAXSTP=10000
```

Runge-Kutta driver with adaptive stepsize control. Integrate the array of starting values `ystart` from `x1` to `x2` with accuracy `eps`, storing intermediate results in the module variables in `ode_path`. `h1` should be set as a guessed first stepsize, `hmin` as the minimum allowed stepsize (can be zero). On output `ystart` is replaced by values at the end of the integration interval. `derivs` is the user-supplied subroutine for calculating the right-hand-side derivative, while `rkqs` is the name of the stepper routine to be used.

```
INTEGER(I4B) :: nstp
REAL(SP) :: h,hdid,hnext,x,xsav
REAL(SP), DIMENSION(size(ystart)) :: dydx,y,yscal
x=x1
h=sign(h1,x2-x1)
nok=0
nbad=0
kount=0
y(:)=ystart(:)
nullify(xp,yp)
```

Pointers nullified here, but memory not deallocated. If `odeint` is called multiple times, calling program should deallocate `xp` and `yp` between calls.

```
if (save_steps) then
    xsav=x-2.0_sp*dxsav
    allocate(xp(256))
    allocate(yp(size(ystart),size(xp)))
end if
do nstp=1,MAXSTP                                  Take at most MAXSTP steps.
    call derivs(x,y,dydx)
    yscal(:)=abs(y(:))+abs(h*dydx(:))+TINY
```

Scaling used to monitor accuracy. This general purpose choice can be modified if need be.

```
    if (save_steps .and. (abs(x-xsav) > abs(dxsav))) &      Store intermediate results.
        call save_a_step
    if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x       If stepsize can overshoot, decrease.
    call rkqs(y,dydx,x,h,eps,yscal,hdid,hnext,derivs)
    if (hdid == h) then
        nok=nok+1
    else
        nbad=nbad+1
    end if
    if ((x-x2)*(x2-x1) >= 0.0) then           Are we done?
        ystart(:)=y(:)
        if (save_steps) call save_a_step      Save final step.
        RETURN                                Normal exit.
    end if
    if (abs(hnext) < hmin)&
        call nrerror('stepsize smaller than minimum in odeint')
    h=hnext
end do
call nrerror('too many steps in odeint')
```

```
CONTAINS
SUBROUTINE save_a_step
kount=kount+1
if (kount > size(xp)) then
    xp=>reallocate(xp,2*size(xp))
    yp=>reallocate(yp,size(yp,1),size(xp))
end if
xp(kount)=x
yp(:,kount)=y(:)
xsav=x
END SUBROUTINE save_a_step
END SUBROUTINE odeint
```

f90 MODULE ode_path The situation here is similar to rkdumb_path, except we don't know at run time how much storage to allocate. We may need to use reallocate from nrutil to increase the storage. The solution is pointers to arrays, with a nullify to be sure the pointer status is well-defined at the beginning of the routine.

SUBROUTINE save_a_step An internal subprogram with no arguments is like a macro in C: you could imagine just copying its code wherever it is called in the parent routine.

⋆ ⋆ ⋆

```
SUBROUTINE mmid(y,dydx,xs,htot,nstep,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), INTENT(IN) :: xs,htot
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
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
END INTERFACE
```

Modified midpoint step. Dependent variable vector y and its derivative vector dydx are input at xs. Also input is htot, the total step to be taken, and nstep, the number of substeps to be used. The output is returned as yout, which need not be a distinct array from y; if it is distinct, however, then y and dydx are returned undamaged. y, dydx, and yout must all have the same length.

```
INTEGER(I4B) :: n,ndum
REAL(SP) :: h,h2,x
REAL(SP), DIMENSION(size(y)) :: ym,yn
ndum=assert_eq(size(y),size(dydx),size(yout),'mmid')
h=htot/nstep                          Stepsize this trip.
ym=y
yn=y+h*dydx                           First step.
x=xs+h
call derivs(x,yn,yout)                Will use yout for temporary storage of derivatives.
h2=2.0_sp*h
do n=2,nstep                          General step.
    call swap(ym,yn)
    yn=yn+h2*yout
```

```
    x=x+h
    call derivs(x,yn,yout)
end do
yout=0.5_sp*(ym+yn+h*yout)          Last step.
END SUBROUTINE mmid
```

★ ★ ★

```
SUBROUTINE bsstep(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,cumsum,iminloc,nrerror,&
    outerdiff,outerprod,upper_triangle
USE nr, ONLY : mmid,pzextr
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
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
END INTERFACE
INTEGER(I4B), PARAMETER :: IMAX=9, KMAXX=IMAX-1
REAL(SP), PARAMETER :: SAFE1=0.25_sp,SAFE2=0.7_sp,REDMAX=1.0e-5_sp,&
    REDMIN=0.7_sp,TINY=1.0e-30_sp,SCALMX=0.1_sp
```

Bulirsch-Stoer step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector `y` and its derivative `dydx` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is the user-supplied subroutine that computes the right-hand-side derivatives. `y`, `dydx`, and `yscal` must all have the same length. Be sure to set `htry` on successive steps to the value of `hnext` returned from the previous step, as is the case if the routine is called by `odeint`.

Parameters: `KMAXX` is the maximum row number used in the extrapolation; `IMAX` is the next row number; `SAFE1` and `SAFE2` are safety factors; `REDMAX` is the maximum factor used when a stepsize is reduced, `REDMIN` the minimum; `TINY` prevents division by zero; 1/`SCALMX` is the maximum factor by which a stepsize can be increased.

```
INTEGER(I4B) :: k,km,ndum
INTEGER(I4B), DIMENSION(IMAX) :: nseq = (/ 2,4,6,8,10,12,14,16,18 /)
INTEGER(I4B), SAVE :: kopt,kmax
REAL(SP), DIMENSION(KMAXX,KMAXX), SAVE :: alf
REAL(SP), DIMENSION(KMAXX) :: err
REAL(SP), DIMENSION(IMAX), SAVE :: a
REAL(SP), SAVE :: epsold = -1.0_sp,xnew
REAL(SP) :: eps1,errmax,fact,h,red,scale,wrkmin,xest
REAL(SP), DIMENSION(size(y)) :: yerr,ysav,yseq
LOGICAL(LGT) :: reduct
LOGICAL(LGT), SAVE :: first=.true.
ndum=assert_eq(size(y),size(dydx),size(yscal),'bsstep')
if (eps /= epsold) then                    A new tolerance, so reinitialize.
    hnext=-1.0e29_sp                       "Impossible" values.
    xnew=-1.0e29_sp
    eps1=SAFE1*eps
    a(:)=cumsum(nseq,1)
      Compute \alpha(k,q):
    where (upper_triangle(KMAXX,KMAXX)) alf=eps1** &
        (outerdiff(a(2:),a(2:))/outerprod(arth( &
```

```
        3.0_sp,2.0_sp,KMAXX),(a(2:)-a(1)+1.0_sp)))
    epsold=eps
    do kopt=2,KMAXX-1                          Determine optimal row number for con-
        if (a(kopt+1) > a(kopt)*alf(kopt-1,kopt)) exit      vergence.
    end do
    kmax=kopt
end if
h=htry
ysav(:)=y(:)                                   Save the starting values.
if (h /= hnext .or. x /= xnew) then            A new stepsize or a new integration: Re-
    first=.true.                                   establish the order window.
    kopt=kmax
end if
reduct=.false.
main_loop: do
    do k=1,kmax                                Evaluate the sequence of modified mid-
        xnew=x+h                                   point integrations.
        if (xnew == x) call nrerror('step size underflow in bsstep')
        call mmid(ysav,dydx,x,h,nseq(k),yseq,derivs)
        xest=(h/nseq(k))**2                    Squared, since error series is even.
        call pzextr(k,xest,yseq,y,yerr)        Perform extrapolation.
        if (k /= 1) then                       Compute normalized error estimate \epsilon(k).
            errmax=maxval(abs(yerr(:)/yscal(:)))
            errmax=max(TINY,errmax)/eps        Scale error relative to tolerance.
            km=k-1
            err(km)=(errmax/SAFE1)**(1.0_sp/(2*km+1))
        end if
        if (k /= 1 .and. (k >= kopt-1 .or. first)) then     In order window.
            if (errmax < 1.0) exit main_loop                Converged.
            if (k == kmax .or. k == kopt+1) then            Check for possible step-
                red=SAFE2/err(km)                               size reduction.
                exit
            else if (k == kopt) then
                if (alf(kopt-1,kopt) < err(km)) then
                    red=1.0_sp/err(km)
                    exit
                end if
            else if (kopt == kmax) then
                if (alf(km,kmax-1) < err(km)) then
                    red=alf(km,kmax-1)*SAFE2/err(km)
                    exit
                end if
            else if (alf(km,kopt) < err(km)) then
                red=alf(km,kopt-1)/err(km)
                exit
            end if
        end if
    end do
    red=max(min(red,REDMIN),REDMAX)            Reduce stepsize by at least REDMIN and
    h=h*red                                        at most REDMAX.
    reduct=.true.
end do main_loop                               Try again.
x=xnew                                         Successful step taken.
hdid=h
first=.false.
kopt=1+iminloc(a(2:km+1)*max(err(1:km),SCALMX))
  Compute optimal row for convergence and corresponding stepsize.
scale=max(err(kopt-1),SCALMX)
wrkmin=scale*a(kopt)
hnext=h/scale
if (kopt >= k .and. kopt /= kmax .and. .not. reduct) then     Check for possible or-
    fact=max(scale/alf(kopt-1,kopt),SCALMX)                       der increase, but
    if (a(kopt+1)*fact <= wrkmin) then                            not if stepsize was
        hnext=h/fact                                              just reduced.
```

```
        kopt=kopt+1
    end if
end if
END SUBROUTINE bsstep
```

f90 `a(:)=cumsum(nseq,1)` The function `cumsum` in `nrutil` with the optional argument `seed=1` gives a direct implementation of equation (16.4.6).

`where (upper_triangle(KMAXX,KMAXX))...` The `upper_triangle` function in `nrutil` returns an upper triangular logical mask. As used here, the mask is true everywhere in the upper triangle of a `KMAXX` $\times$ `KMAXX` matrix, excluding the diagonal. An optional integer argument `extra` allows additional diagonals to be set to true. With `extra=1` the upper triangle including the diagonal would be true.

`main_loop: do` Using a named do-loop provides clear structured code that required `goto`'s in the Fortran 77 version.

`kopt=1+iminloc(...)` See the discussion of `imaxloc` on p. 1017.

$\star \quad \star \quad \star$

```
SUBROUTINE pzextr(iest,xest,yest,yz,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iest
REAL(SP), INTENT(IN) :: xest
REAL(SP), DIMENSION(:), INTENT(IN) :: yest
REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
```

Use polynomial extrapolation to evaluate N functions at $x = 0$ by fitting a polynomial to a sequence of estimates with progressively smaller values $x =$ `xest`, and corresponding function vectors `yest`. This call is number `iest` in the sequence of calls. Extrapolated function values are output as `yz`, and their estimated error is output as `dy`. `yest`, `yz`, and `dy` are arrays of length N.

```
INTEGER(I4B), PARAMETER :: IEST_MAX=16
INTEGER(I4B) :: j,nv
INTEGER(I4B), SAVE :: nvold=-1
REAL(SP) :: delta,f1,f2
REAL(SP), DIMENSION(size(yz)) :: d,tmp,q
REAL(SP), DIMENSION(IEST_MAX), SAVE :: x
REAL(SP), DIMENSION(:,:), ALLOCATABLE, SAVE :: qcol
nv=assert_eq(size(yz),size(yest),size(dy),'pzextr')
if (iest > IEST_MAX) call &
    nrerror('pzextr: probable misuse, too much extrapolation')
if (nv /= nvold) then                Set up internal storage.
    if (allocated(qcol)) deallocate(qcol)
    allocate(qcol(nv,IEST_MAX))
    nvold=nv
end if
x(iest)=xest                         Save current independent variable.
dy(:)=yest(:)
yz(:)=yest(:)
if (iest == 1) then                  Store first estimate in first column.
    qcol(:,1)=yest(:)
else
    d(:)=yest(:)
    do j=1,iest-1
        delta=1.0_sp/(x(iest-j)-xest)
        f1=xest*delta
        f2=x(iest-j)*delta
        q(:)=qcol(:,j)               Propagate tableau 1 diagonal more.
```

```
        qcol(:,j)=dy(:)
        tmp(:)=d(:)-q(:)
        dy(:)=f1*tmp(:)
        d(:)=f2*tmp(:)
        yz(:)=yz(:)+dy(:)
    end do
    qcol(:,iest)=dy(:)
end if
END SUBROUTINE pzextr
```

f90 `REAL(SP), DIMENSION(:,:), ALLOCATABLE, SAVE :: qcol` The second dimension of `qcol` is known at compile time to be `IEST_MAX`, but the first dimension is known only at run time, from `size(yz)`. The language requires us to have all dimensions allocatable if any one of them is.

`if (nv /= nvold) then...` This routine generally gets called many times with `iest` cycling repeatedly through the values $1, 2, \ldots,$ up to some value less than `IEST_MAX`. The number of variables, `nv`, is fixed during the solution of the problem. The routine might be called again in solving a different problem with a new value of `nv`. This if block ensures that `qcol` is dimensioned correctly both for the first and subsequent problems, if any.

```
SUBROUTINE rzextr(iest,xest,yest,yz,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iest
REAL(SP), INTENT(IN) :: xest
REAL(SP), DIMENSION(:), INTENT(IN) :: yest
REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
    Exact substitute for pzextr, but uses diagonal rational function extrapolation instead of
    polynomial extrapolation.
INTEGER(I4B), PARAMETER :: IEST_MAX=16
INTEGER(I4B) :: k,nv
INTEGER(I4B), SAVE :: nvold=-1
REAL(SP), DIMENSION(size(yz)) :: yy,v,c,b,b1,ddy
REAL(SP), DIMENSION(:,:), ALLOCATABLE, SAVE :: d
REAL(SP), DIMENSION(IEST_MAX), SAVE :: fx,x
nv=assert_eq(size(yz),size(dy),size(yest),'rzextr')
if (iest > IEST_MAX) call &
    nrerror('rzextr: probable misuse, too much extrapolation')
if (nv /= nvold) then
    if (allocated(d)) deallocate(d)
    allocate(d(nv,IEST_MAX))
    nvold=nv
end if
x(iest)=xest                    Save current independent variable.
if (iest == 1) then
    yz=yest
    d(:,1)=yest
    dy=yest
else
    fx(2:iest)=x(iest-1:1:-1)/xest
    yy=yest                     Evaluate next diagonal in tableau.
    v=d(1:nv,1)
    c=yy
    d(1:nv,1)=yy
    do k=2,iest
        b1=fx(k)*v
        b=b1-c
        where (b /= 0.0)
```

```
            b=(c-v)/b
            ddy=c*b
            c=b1*b
        elsewhere               Care needed to avoid division by 0.
            ddy=v
        end where
        if (k /= iest) v=d(1:nv,k)
        d(1:nv,k)=ddy
        yy=yy+ddy
    end do
    dy=ddy
    yz=yy
end if
END SUBROUTINE rzextr
```

⋆ ⋆ ⋆

```
SUBROUTINE stoerm(y,d2y,xs,htot,nstep,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: y,d2y
REAL(SP), INTENT(IN) :: xs,htot
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
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
END INTERFACE
```

Stoermer's rule for integrating $y'' = f(x, y)$ for a system of n equations. On input y contains y in its first n elements and y' in its second n elements, all evaluated at xs. d2y contains the right-hand-side function f (also evaluated at xs) in its first n elements. Its second n elements are not referenced. Also input is htot, the total step to be taken, and nstep, the number of substeps to be used. The output is returned as yout, with the same storage arrangement as y. derivs is the user-supplied subroutine that calculates f.

```
INTEGER(I4B) :: neqn,neqn1,nn,nv
REAL(SP) :: h,h2,halfh,x
REAL(SP), DIMENSION(size(y)) :: ytemp
nv=assert_eq(size(y),size(d2y),size(yout),'stoerm')
neqn=nv/2                                   Number of equations.
neqn1=neqn+1
h=htot/nstep                                Stepsize this trip.
halfh=0.5_sp*h                              First step.
ytemp(neqn1:nv)=h*(y(neqn1:nv)+halfh*d2y(1:neqn))
ytemp(1:neqn)=y(1:neqn)+ytemp(neqn1:nv)
x=xs+h
call derivs(x,ytemp,yout)                   Use yout for temporary storage of deriva-
h2=h*h                                          tives.
do nn=2,nstep                               General step.
    ytemp(neqn1:nv)=ytemp(neqn1:nv)+h2*yout(1:neqn)
    ytemp(1:neqn)=ytemp(1:neqn)+ytemp(neqn1:nv)
    x=x+h
    call derivs(x,ytemp,yout)
end do
yout(neqn1:nv)=ytemp(neqn1:nv)/h+halfh*yout(1:neqn)      Last step.
yout(1:neqn)=ytemp(1:neqn)
END SUBROUTINE stoerm
```

⋆ ⋆ ⋆

```
SUBROUTINE stiff(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,diagadd,nrerror
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
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 jacobn(x,y,dfdx,dfdy)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
    END SUBROUTINE jacobn
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXTRY=40
REAL(SP), PARAMETER :: SAFETY=0.9_sp,GROW=1.5_sp,PGROW=-0.25_sp,&
    SHRNK=0.5_sp,PSHRNK=-1.0_sp/3.0_sp,ERRCON=0.1296_sp,&
    GAM=1.0_sp/2.0_sp,&
    A21=2.0_sp,A31=48.0_sp/25.0_sp,A32=6.0_sp/25.0_sp,C21=-8.0_sp,&
    C31=372.0_sp/25.0_sp,C32=12.0_sp/5.0_sp,&
    C41=-112.0_sp/125.0_sp,C42=-54.0_sp/125.0_sp,&
    C43=-2.0_sp/5.0_sp,B1=19.0_sp/9.0_sp,B2=1.0_sp/2.0_sp,&
    B3=25.0_sp/108.0_sp,B4=125.0_sp/108.0_sp,E1=17.0_sp/54.0_sp,&
    E2=7.0_sp/36.0_sp,E3=0.0_sp,E4=125.0_sp/108.0_sp,&
    C1X=1.0_sp/2.0_sp,C2X=-3.0_sp/2.0_sp,C3X=121.0_sp/50.0_sp,&
    C4X=29.0_sp/250.0_sp,A2X=1.0_sp,A3X=3.0_sp/5.0_sp
```

Fourth order Rosenbrock step for integrating stiff ODEs, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y` and its derivative `dydx` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied subroutine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied subroutine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`. `y`, `dydx`, and `yscal` must have the same length.

Parameters: `GROW` and `SHRNK` are the largest and smallest factors by which stepsize can change in one step; `ERRCON=(GROW/SAFETY)**(1/PGROW)` and handles the case when `errmax` $\simeq 0$.

```
INTEGER(I4B) :: jtry,ndum
INTEGER(I4B), DIMENSION(size(y)) :: indx
REAL(SP), DIMENSION(size(y)) :: dfdx,dytmp,err,g1,g2,g3,g4,ysav
REAL(SP), DIMENSION(size(y),size(y)) :: a,dfdy
REAL(SP) :: d,errmax,h,xsav
ndum=assert_eq(size(y),size(dydx),size(yscal),'stiff')
xsav=x                                          Save initial values.
ysav(:)=y(:)
call jacobn(xsav,ysav,dfdx,dfdy)
```

The user must supply this subroutine to return the `n` $\times$ `n` matrix `dfdy` and the vector `dfdx`.

```
h=htry                                          Set stepsize to the initial trial value.
do jtry=1,MAXTRY
```

```
    a(:,:)=-dfdy(:,:)                          Set up the matrix 1 − γhf′.
    call diagadd(a,1.0_sp/(GAM*h))
    call ludcmp(a,indx,d)                      LU decomposition of the matrix.
    g1=dydx+h*C1X*dfdx                         Set up right-hand side for g1.
    call lubksb(a,indx,g1)                     Solve for g1.
    y=ysav+A21*g1                              Compute intermediate values of y and x.
    x=xsav+A2X*h
    call derivs(x,y,dytmp)                     Compute dydx at the intermediate values.
    g2=dytmp+h*C2X*dfdx+C21*g1/h               Set up right-hand side for g2.
    call lubksb(a,indx,g2)                     Solve for g2.
    y=ysav+A31*g1+A32*g2                       Compute intermediate values of y and x.
    x=xsav+A3X*h
    call derivs(x,y,dytmp)                     Compute dydx at the intermediate values.
    g3=dytmp+h*C3X*dfdx+(C31*g1+C32*g2)/h              Set up right-hand side for g3.
    call lubksb(a,indx,g3)                             Solve for g3.
    g4=dytmp+h*C4X*dfdx+(C41*g1+C42*g2+C43*g3)/h       Set up right-hand side for g4.
    call lubksb(a,indx,g4)                             Solve for g4.
    y=ysav+B1*g1+B2*g2+B3*g3+B4*g4             Get fourth order estimate of y and error es-
    err=E1*g1+E2*g2+E3*g3+E4*g4                    timate.
    x=xsav+h
    if (x == xsav) call &
        nrerror('stepsize not significant in stiff')
    errmax=maxval(abs(err/yscal))/eps          Evaluate accuracy.
    if (errmax <= 1.0) then                    Step succeeded. Compute size of next step
        hdid=h                                     and return.
        hnext=merge(SAFETY*h*errmax**PGROW, GROW*h, &
            errmax > ERRCON)
        RETURN
    else                                       Truncation error too large, reduce stepsize.
        hnext=SAFETY*h*errmax**PSHRNK
        h=sign(max(abs(hnext),SHRNK*abs(h)),h)
    end if
end do                                         Go back and retry step.
call nrerror('exceeded MAXTRY in stiff')
END SUBROUTINE stiff
```

f90 `call diagadd(...)` See discussion of diagadd after hqr on p. 1234.

```
SUBROUTINE jacobn(x,y,dfdx,dfdy)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
    Routine for Jacobi matrix corresponding to example in equations (16.6.27).
dfdx(:)=0.0
dfdy(1,1)=-0.013_sp-1000.0_sp*y(3)
dfdy(1,2)=0.0
dfdy(1,3)=-1000.0_sp*y(1)
dfdy(2,1)=0.0
dfdy(2,2)=-2500.0_sp*y(3)
dfdy(2,3)=-2500.0_sp*y(2)
dfdy(3,1)=-0.013_sp-1000.0_sp*y(3)
dfdy(3,2)=-2500.0_sp*y(3)
dfdy(3,3)=-1000.0_sp*y(1)-2500.0_sp*y(2)
END SUBROUTINE jacobn
```

```
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
```

Routine for right-hand side of example in equations (16.6.27).

```
dydx(1)=-0.013_sp*y(1)-1000.0_sp*y(1)*y(3)
dydx(2)=-2500.0_sp*y(2)*y(3)
dydx(3)=-0.013_sp*y(1)-1000.0_sp*y(1)*y(3)-2500.0_sp*y(2)*y(3)
END SUBROUTINE derivs
```

★ ★ ★

```
SUBROUTINE simpr(y,dydx,dfdx,dfdy,xs,htot,nstep,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,diagadd
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xs,htot
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx,dfdx
REAL(SP), DIMENSION(:,:), INTENT(IN) :: dfdy
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
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
END INTERFACE
```

Performs one step of semi-implicit midpoint rule. Input are the dependent variable `y`, its derivative `dydx`, the derivative of the right-hand side with respect to x, `dfdx`, which are all vectors of length N, and the $N \times N$ Jacobian `dfdy` at `xs`. Also input are `htot`, the total step to be taken, and `nstep`, the number of substeps to be used. The output is returned as `yout`, a vector of length N. `derivs` is the user-supplied subroutine that calculates `dydx`.

```
INTEGER(I4B) :: ndum,nn
INTEGER(I4B), DIMENSION(size(y)) :: indx
REAL(SP) :: d,h,x
REAL(SP), DIMENSION(size(y)) :: del,ytemp
REAL(SP), DIMENSION(size(y),size(y)) :: a
ndum=assert_eq((/size(y),size(dydx),size(dfdx),size(dfdy,1),&
    size(dfdy,2),size(yout)/),'simpr')
h=htot/nstep                          Stepsize this trip.
a(:,:)=-h*dfdy(:,:)                   Set up the matrix 1 − hf′.
call diagadd(a,1.0_sp)
call ludcmp(a,indx,d)                 LU decomposition of the matrix.
yout=h*(dydx+h*dfdx)                  Set up right-hand side for first step. Use yout for
call lubksb(a,indx,yout)                  temporary storage.
del=yout                              First step.
ytemp=y+del
x=xs+h
call derivs(x,ytemp,yout)             Use yout for temporary storage of derivatives.
do nn=2,nstep                         General step.
    yout=h*yout-del                   Set up right-hand side for general step.
    call lubksb(a,indx,yout)
    del=del+2.0_sp*yout
    ytemp=ytemp+del
    x=x+h
    call derivs(x,ytemp,yout)
```

```
end do
yout=h*yout-del                    Set up right-hand side for last step.
call lubksb(a,indx,yout)
yout=ytemp+yout                    Take last step.
END SUBROUTINE simpr
```

f90 `call diagadd(...)` See discussion of `diagadd` after `hqr` on p. 1234.

⋆ ⋆ ⋆

```
SUBROUTINE stifbs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,cumsum,iminloc,nrerror,&
    outerdiff,outerprod,upper_triangle
USE nr, ONLY : simpr,pzextr
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(OUT) :: hdid,hnext
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 jacobn(x,y,dfdx,dfdy)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
    END SUBROUTINE jacobn
END INTERFACE
INTEGER(I4B), PARAMETER :: IMAX=8, KMAXX=IMAX-1
REAL(SP), PARAMETER :: SAFE1=0.25_sp,SAFE2=0.7_sp,REDMAX=1.0e-5_sp,&
    REDMIN=0.7_sp,TINY=1.0e-30_sp,SCALMX=0.1_sp
```

Semi-implicit extrapolation step for integrating stiff ODEs, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y` and its derivative `dydx` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied subroutine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied subroutine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`. `y`, `dydx`, and `yscal` must all have the same length. Be sure to set `htry` on successive steps to the value of `hnext` returned from the previous step, as is the case if the routine is called by `odeint`.

```
INTEGER(I4B) :: k,km,ndum
INTEGER(I4B), DIMENSION(IMAX) :: nseq = (/ 2,6,10,14,22,34,50,70 /)
```

Sequence is different from `bsstep`.

```
INTEGER(I4B), SAVE :: kopt,kmax,nvold=-1
REAL(SP), DIMENSION(KMAXX,KMAXX), SAVE :: alf
REAL(SP), DIMENSION(KMAXX) :: err
REAL(SP), DIMENSION(IMAX), SAVE :: a
REAL(SP), SAVE :: epsold = -1.0
REAL(SP) :: eps1,errmax,fact,h,red,scale,wrkmin,xest
```

```
REAL(SP), SAVE :: xnew
REAL(SP), DIMENSION(size(y)) :: dfdx,yerr,ysav,yseq
REAL(SP), DIMENSION(size(y),size(y)) :: dfdy
LOGICAL(LGT) :: reduct
LOGICAL(LGT), SAVE :: first=.true.
ndum=assert_eq(size(y),size(dydx),size(yscal),'stifbs')
if (eps /= epsold .or. nvold /= size(y)) then        Reinitialize also if number of variables has changed.
    hnext=-1.0e29_sp
    xnew=-1.0e29_sp
    eps1=SAFE1*eps
    a(:)=cumsum(nseq,1)
    where (upper_triangle(KMAXX,KMAXX)) alf=eps1** &
        (outerdiff(a(2:),a(2:))/outerprod(arth( &
        3.0_sp,2.0_sp,KMAXX),(a(2:)-a(1)+1.0_sp)))
    epsold=eps
    nvold=size(y)                                    Save number of variables.
    a(:)=cumsum(nseq,1+nvold)                        Add cost of Jacobian evaluations to work coefficients.
    do kopt=2,KMAXX-1
        if (a(kopt+1) > a(kopt)*alf(kopt-1,kopt)) exit
    end do
    kmax=kopt
end if
h=htry
ysav(:)=y(:)
call jacobn(x,y,dfdx,dfdy)                           Evaluate Jacobian.
if (h /= hnext .or. x /= xnew) then
    first=.true.
    kopt=kmax
end if
reduct=.false.
main_loop: do
    do k=1,kmax
        xnew=x+h
        if (xnew == x) call nrerror('step size underflow in stifbs')
        call simpr(ysav,dydx,dfdx,dfdy,x,h,nseq(k),yseq,derivs)
          Here is the call to the semi-implicit midpoint rule.
        xest=(h/nseq(k))**2                          The rest of the routine is identical to bsstep.
        call pzextr(k,xest,yseq,y,yerr)
        if (k /= 1) then
            errmax=maxval(abs(yerr(:)/yscal(:)))
            errmax=max(TINY,errmax)/eps
            km=k-1
            err(km)=(errmax/SAFE1)**(1.0_sp/(2*km+1))
        end if
        if (k /= 1 .and. (k >= kopt-1 .or. first)) then
            if (errmax < 1.0) exit main_loop
            if (k == kmax .or. k == kopt+1) then
                red=SAFE2/err(km)
                exit
            else if (k == kopt) then
                if (alf(kopt-1,kopt) < err(km)) then
                    red=1.0_sp/err(km)
                    exit
                end if
            else if (kopt == kmax) then
                if (alf(km,kmax-1) < err(km)) then
                    red=alf(km,kmax-1)*SAFE2/err(km)
                    exit
                end if
            else if (alf(km,kopt) < err(km)) then
                red=alf(km,kopt-1)/err(km)
                exit
            end if
        end if
```

```
    end do
    red=max(min(red,REDMIN),REDMAX)
    h=h*red
    reduct=.true.
end do main_loop
x=xnew
hdid=h
first=.false.
kopt=1+iminloc(a(2:km+1)*max(err(1:km),SCALMX))
scale=max(err(kopt-1),SCALMX)
wrkmin=scale*a(kopt)
hnext=h/scale
if (kopt >= k .and. kopt /= kmax .and. .not. reduct) then
    fact=max(scale/alf(kopt-1,kopt),SCALMX)
    if (a(kopt+1)*fact <= wrkmin) then
        hnext=h/fact
        kopt=kopt+1
    end if
end if
END SUBROUTINE stifbs
```

f90 This routine is very similar to `bsstep`, and the same remarks about Fortran 90 constructions on p. 1305 apply here.