Chapter B10. Minimization or Maximization of Functions

```
SUBROUTINE mnbrak(ax,bx,cx,fa,fb,fc,func)
USE nrtype; USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: ax,bx
REAL(SP), INTENT(OUT) :: cx,fa,fb,fc
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: GOLD=1.618034_sp,GLIMIT=100.0_sp,TINY=1.0e-20_sp
```

Given a function `func`, and given distinct initial points `ax` and `bx`, this routine searches in the downhill direction (defined by the function as evaluated at the initial points) and returns new points `ax`, `bx`, `cx` that bracket a minimum of the function. Also returned are the function values at the three points, `fa`, `fb`, and `fc`.

Parameters: `GOLD` is the default ratio by which successive intervals are magnified; `GLIMIT` is the maximum magnification allowed for a parabolic-fit step.

```
REAL(SP) :: fu,q,r,u,ulim
fa=func(ax)
fb=func(bx)
if (fb > fa) then                                    Switch roles of a and b so that we
    call swap(ax,bx)                                     can go downhill in the direction
    call swap(fa,fb)                                     from a to b.
end if
cx=bx+GOLD*(bx-ax)                                   First guess for c.
fc=func(cx)
do                                                   Do-while-loop: Keep returning here
    if (fb < fc) RETURN                                  until we bracket.
      Compute u by parabolic extrapolation from a, b, c. TINY is used to prevent any possible
      division by zero.
    r=(bx-ax)*(fb-fc)
    q=(bx-cx)*(fb-fa)
    u=bx-((bx-cx)*q-(bx-ax)*r)/(2.0_sp*sign(max(abs(q-r),TINY),q-r))
    ulim=bx+GLIMIT*(cx-bx)
      We won't go farther than this. Test various possibilities:
    if ((bx-u)*(u-cx) > 0.0) then                    Parabolic u is between b and c: try
        fu=func(u)                                       it.
        if (fu < fc) then                            Got a minimum between b and c.
            ax=bx
            fa=fb
            bx=u
            fb=fu
            RETURN
        else if (fu > fb) then                       Got a minimum between a and u.
            cx=u
            fc=fu
            RETURN
        end if
        u=cx+GOLD*(cx-bx)                        Parabolic fit was no use. Use default
        fu=func(u)                                   magnification.
    else if ((cx-u)*(u-ulim) > 0.0) then         Parabolic fit is between c and its al-
        fu=func(u)                                   lowed limit.
        if (fu < fc) then
            bx=cx
            cx=u
            u=cx+GOLD*(cx-bx)
            call shft(fb,fc,fu,func(u))
        end if
    else if ((u-ulim)*(ulim-cx) >= 0.0) then     Limit parabolic u to maximum al-
        u=ulim                                       lowed value.
        fu=func(u)
    else                                         Reject parabolic u, use default mag-
        u=cx+GOLD*(cx-bx)                            nification.
        fu=func(u)
    end if
    call shft(ax,bx,cx,u)
    call shft(fa,fb,fc,fu)                       Eliminate oldest point and continue.
end do
CONTAINS

SUBROUTINE shft(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: d
a=b
b=c
c=d
END SUBROUTINE shft
END SUBROUTINE mnbrak
```

f90 `call shft...` There are three places in `mnbrak` where we need to shift four variables around. Rather than repeat code, we make `shft` an internal subroutine, coming after a `CONTAINS` statement. It is invisible to all procedures except `mnbrak`.

⋆ ⋆ ⋆

```
FUNCTION golden(ax,bx,cx,func,tol,xmin)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: golden
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: R=0.61803399_sp,C=1.0_sp-R
```

Given a function `func`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` (such that `bx` is between `ax` and `cx`, and `func(bx)` is less than both `func(ax)` and `func(cx)`), this routine performs a golden section search for the minimum, isolating it to a fractional precision of about `tol`. The abscissa of the minimum is returned as `xmin`, and the minimum

```
    function value is returned as golden, the returned function value.
    Parameters: The golden ratios.
REAL(SP) :: f1,f2,x0,x1,x2,x3
x0=ax                                         At any given time we will keep track of
x3=cx                                             four points, x0,x1,x2,x3.
if (abs(cx-bx) > abs(bx-ax)) then             Make x0 to x1 the smaller segment,
    x1=bx
    x2=bx+C*(cx-bx)                           and fill in the new point to be tried.
else
    x2=bx
    x1=bx-C*(bx-ax)
end if
f1=func(x1)
f2=func(x2)
  The initial function evaluations. Note that we never need to evaluate the function at the
  original endpoints.
do                                            Do-while-loop: We keep returning here.
    if (abs(x3-x0) <= tol*(abs(x1)+abs(x2))) exit
    if (f2 < f1) then                         One possible outcome,
        call shft3(x0,x1,x2,R*x2+C*x3)        its housekeeping,
        call shft2(f1,f2,func(x2))            and a new function evaluation.
    else                                      The other outcome,
        call shft3(x3,x2,x1,R*x1+C*x0)
        call shft2(f2,f1,func(x1))            and its new function evaluation.
    end if
end do                                        Back to see if we are done.
if (f1 < f2) then                             We are done. Output the best of the two
    golden=f1                                     current values.
    xmin=x1
else
    golden=f2
    xmin=x2
end if
CONTAINS

SUBROUTINE shft2(a,b,c)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b
REAL(SP), INTENT(IN) :: c
a=b
b=c
END SUBROUTINE shft2

SUBROUTINE shft3(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: d
a=b
b=c
c=d
END SUBROUTINE shft3
END FUNCTION golden
```

f90 `call shft3...call shft2...` See discussion of `shft` for `mnbrak` on p. 1202.

⋆ ⋆ ⋆

```
FUNCTION brent(ax,bx,cx,func,tol,xmin)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: brent
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: CGOLD=0.3819660_sp,ZEPS=1.0e-3_sp*epsilon(ax)
```

Given a function `func`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` (such that `bx` is between `ax` and `cx`, and `func(bx)` is less than both `func(ax)` and `func(cx)`), this routine isolates the minimum to a fractional precision of about `tol` using Brent's method. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `brent`, the returned function value.

Parameters: Maximum allowed number of iterations; golden ratio; and a small number that protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.

```
INTEGER(I4B) :: iter
REAL(SP) :: a,b,d,e,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm
a=min(ax,cx)                                a and b must be in ascending order, though
b=max(ax,cx)                                    the input abscissas need not be.
v=bx                                        Initializations...
w=v
x=v
e=0.0                                       This will be the distance moved on the step
fx=func(x)                                      before last.
fv=fx
fw=fx
do iter=1,ITMAX                             Main program loop.
    xm=0.5_sp*(a+b)
    tol1=tol*abs(x)+ZEPS
    tol2=2.0_sp*tol1
    if (abs(x-xm) <= (tol2-0.5_sp*(b-a))) then          Test for done here.
        xmin=x                              Arrive here ready to exit with best values.
        brent=fx
        RETURN
    end if
    if (abs(e) > tol1) then                 Construct a trial parabolic fit.
        r=(x-w)*(fx-fv)
        q=(x-v)*(fx-fw)
        p=(x-v)*q-(x-w)*r
        q=2.0_sp*(q-r)
        if (q > 0.0) p=-p
        q=abs(q)
        etemp=e
        e=d
        if (abs(p) >= abs(0.5_sp*q*etemp) .or. &
            p <= q*(a-x) .or. p >= q*(b-x)) then
              The above conditions determine the acceptability of the parabolic fit. Here it is
              not o.k., so we take the golden section step into the larger of the two segments.
            e=merge(a-x,b-x, x >= xm )
            d=CGOLD*e
        else                                Take the parabolic step.
            d=p/q
            u=x+d
            if (u-a < tol2 .or. b-u < tol2) d=sign(tol1,xm-x)
        end if
    else                                        Take the golden section step into the larger
        e=merge(a-x,b-x, x >= xm )                  of the two segments.
        d=CGOLD*e
    end if
    u=merge(x+d,x+sign(tol1,d), abs(d) >= tol1 )
      Arrive here with d computed either from parabolic fit, or else from golden section.
    fu=func(u)
      This is the one function evaluation per iteration.
    if (fu <= fx) then                          Now we have to decide what to do with our
        if (u >= x) then                            function evaluation. Housekeeping follows:
            a=x
        else
            b=x
        end if
        call shft(v,w,x,u)
        call shft(fv,fw,fx,fu)
    else
        if (u < x) then
            a=u
        else
            b=u
        end if
        if (fu <= fw .or. w == x) then
            v=w
            fv=fw
            w=u
            fw=fu
        else if (fu <= fv .or. v == x .or. v == w) then
            v=u
            fv=fu
        end if
    end if
end do                                          Done with housekeeping. Back for another
call nrerror('brent: exceed maximum iterations')    iteration.
CONTAINS

SUBROUTINE shft(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: d
a=b
b=c
c=d
END SUBROUTINE shft
END FUNCTION brent
```

⋆ ⋆ ⋆

```
FUNCTION dbrent(ax,bx,cx,func,dfunc,tol,xmin)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: dbrent
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func

    FUNCTION dfunc(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: dfunc
    END FUNCTION dfunc
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: ZEPS=1.0e-3_sp*epsilon(ax)
```

Given a function `func` and its derivative function `dfunc`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` [such that `bx` is between `ax` and `cx`, and `func(bx)` is less than both `func(ax)` and `func(cx)`], this routine isolates the minimum to a fractional precision of about `tol` using a modification of Brent's method that uses derivatives. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `dbrent`, the returned function value.

Parameters: Maximum allowed number of iterations, and a small number that protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.

```
INTEGER(I4B) :: iter
REAL(SP) :: a,b,d,d1,d2,du,dv,dw,dx,e,fu,fv,fw,fx,olde,tol1,tol2,&
    u,u1,u2,v,w,x,xm
```

Comments following will point out only differences from the routine `brent`. Read that routine first.

```
LOGICAL :: ok1,ok2                            Will be used as flags for whether pro-
a=min(ax,cx)                                      posed steps are acceptable or not.
b=max(ax,cx)
v=bx
w=v
x=v
e=0.0
fx=func(x)
fv=fx
fw=fx
dx=dfunc(x)                                   All our housekeeping chores are dou-
dv=dx                                             bled by the necessity of moving
dw=dx                                             derivative values around as well
do iter=1,ITMAX                                   as function values.
    xm=0.5_sp*(a+b)
    tol1=tol*abs(x)+ZEPS
    tol2=2.0_sp*tol1
    if (abs(x-xm) <= (tol2-0.5_sp*(b-a))) exit
    if (abs(e) > tol1) then
        d1=2.0_sp*(b-a)                       Initialize these d's to an out-of-bracket
        d2=d1                                     value.
        if (dw /= dx) d1=(w-x)*dx/(dx-dw)     Secant method with each point.
        if (dv /= dx) d2=(v-x)*dx/(dx-dv)
```

Which of these two estimates of `d` shall we take? We will insist that they be within the bracket, and on the side pointed to by the derivative at `x`:

```
        u1=x+d1
        u2=x+d2
        ok1=((a-u1)*(u1-b) > 0.0) .and. (dx*d1 <= 0.0)
        ok2=((a-u2)*(u2-b) > 0.0) .and. (dx*d2 <= 0.0)
        olde=e                                Movement on the step before last.
        e=d
        if (ok1 .or. ok2) then                Take only an acceptable d, and if
            if (ok1 .and. ok2) then               both are acceptable, then take
                d=merge(d1,d2, abs(d1) < abs(d2))     the smallest one.
            else
                d=merge(d1,d2,ok1)
            end if
            if (abs(d) <= abs(0.5_sp*olde)) then
                u=x+d
                if (u-a < tol2 .or. b-u < tol2) &
                    d=sign(tol1,xm-x)
            else
```

```
                e=merge(a,b, dx >= 0.0)-x
                  Decide which segment by the sign of the derivative.
                d=0.5_sp*e                          Bisect, not golden section.
            end if
        else
            e=merge(a,b, dx >= 0.0)-x
            d=0.5_sp*e                              Bisect, not golden section.
        end if
    else
        e=merge(a,b, dx >= 0.0)-x
        d=0.5_sp*e                                  Bisect, not golden section.
    end if
    if (abs(d) >= tol1) then
        u=x+d
        fu=func(u)
    else
        u=x+sign(tol1,d)
        fu=func(u)                                  If the minimum step in the downhill
        if (fu > fx) exit                               direction takes us uphill, then we
    end if                                              are done.
    du=dfunc(u)                                     Now all the housekeeping, sigh.
    if (fu <= fx) then
        if (u >= x) then
            a=x
        else
            b=x
        end if
        call mov3(v,fv,dv,w,fw,dw)
        call mov3(w,fw,dw,x,fx,dx)
        call mov3(x,fx,dx,u,fu,du)
    else
        if (u < x) then
            a=u
        else
            b=u
        end if
        if (fu <= fw .or. w == x) then
            call mov3(v,fv,dv,w,fw,dw)
            call mov3(w,fw,dw,u,fu,du)
        else if (fu <= fv .or. v == x .or. v == w) then
            call mov3(v,fv,dv,u,fu,du)
        end if
    end if
end do
if (iter > ITMAX) call nrerror('dbrent: exceeded maximum iterations')
xmin=x
dbrent=fx
CONTAINS

SUBROUTINE mov3(a,b,c,d,e,f)
REAL(SP), INTENT(IN) :: d,e,f
REAL(SP), INTENT(OUT) :: a,b,c
a=d
b=e
c=f
END SUBROUTINE mov3
END FUNCTION dbrent
```

⋆ ⋆ ⋆

```
SUBROUTINE amoeba(p,y,ftol,func,iter)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,iminloc,nrerror,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: p
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=5000
REAL(SP), PARAMETER :: TINY=1.0e-10
```

Minimization of the function func in N dimensions by the downhill simplex method of Nelder and Mead. The $(N+1) \times N$ matrix p is input. Its $N+1$ rows are N-dimensional vectors that are the vertices of the starting simplex. Also input is the vector y of length $N+1$, whose components must be preinitialized to the values of func evaluated at the $N+1$ vertices (rows) of p; and ftol the fractional convergence tolerance to be achieved in the function value (n.b.!). On output, p and y will have been reset to $N+1$ new points all within ftol of a minimum function value, and iter gives the number of function evaluations taken.

Parameters: The maximum allowed number of function evaluations, and a small number.

```
INTEGER(I4B) :: ihi,ndim                      Global variables.
REAL(SP), DIMENSION(size(p,2)) :: psum
call amoeba_private
CONTAINS

SUBROUTINE amoeba_private
IMPLICIT NONE
INTEGER(I4B) :: i,ilo,inhi
REAL(SP) :: rtol,ysave,ytry,ytmp
ndim=assert_eq(size(p,2),size(p,1)-1,size(y)-1,'amoeba')
iter=0
psum(:)=sum(p(:,:),dim=1)
do                                            Iteration loop.
    ilo=iminloc(y(:))                         Determine which point is the highest (worst),
    ihi=imaxloc(y(:))                              next-highest, and lowest (best).
    ytmp=y(ihi)
    y(ihi)=y(ilo)
    inhi=imaxloc(y(:))
    y(ihi)=ytmp
    rtol=2.0_sp*abs(y(ihi)-y(ilo))/(abs(y(ihi))+abs(y(ilo))+TINY)
      Compute the fractional range from highest to lowest and return if satisfactory.
    if (rtol < ftol) then                     If returning, put best point and value in slot
        call swap(y(1),y(ilo))                     1.
        call swap(p(1,:),p(ilo,:))
        RETURN
    end if
    if (iter >= ITMAX) call nrerror('ITMAX exceeded in amoeba')
```

Begin a new iteration. First extrapolate by a factor -1 through the face of the simplex across from the high point, i.e., reflect the simplex from the high point.

```
    ytry=amotry(-1.0_sp)
    iter=iter+1
    if (ytry <= y(ilo)) then                  Gives a result better than the best point, so
        ytry=amotry(2.0_sp)                        try an additional extrapolation by a fac-
        iter=iter+1                                tor of 2.
    else if (ytry >= y(inhi)) then            The reflected point is worse than the sec-
        ysave=y(ihi)                               ond highest, so look for an intermediate
        ytry=amotry(0.5_sp)                        lower point, i.e., do a one-dimensional
        iter=iter+1                                contraction.
```

```
            if (ytry >= ysave) then
                 Can't seem to get rid of that high point. Better contract around the lowest
                 (best) point.
                p(:,:)=0.5_sp*(p(:,:)+spread(p(ilo,:),1,size(p,1)))
                do i=1,ndim+1
                    if (i /= ilo) y(i)=func(p(i,:))
                end do
                iter=iter+ndim                  Keep track of function evaluations.
                psum(:)=sum(p(:,:),dim=1)
            end if
        end if
end do                                          Go back for the test of doneness and the next
END SUBROUTINE amoeba_private                       iteration.

FUNCTION amotry(fac)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: fac
REAL(SP) :: amotry
    Extrapolates by a factor fac through the face of the simplex across from the high point,
    tries it, and replaces the high point if the new point is better.
REAL(SP) :: fac1,fac2,ytry
REAL(SP), DIMENSION(size(p,2)) :: ptry
fac1=(1.0_sp-fac)/ndim
fac2=fac1-fac
ptry(:)=psum(:)*fac1-p(ihi,:)*fac2
ytry=func(ptry)                                 Evaluate the function at the trial point.
if (ytry < y(ihi)) then                         If it's better than the highest, then replace
    y(ihi)=ytry                                     the highest.
    psum(:)=psum(:)-p(ihi,:)+ptry(:)
    p(ihi,:)=ptry(:)
end if
amotry=ytry
END FUNCTION amotry
END SUBROUTINE amoeba
```

The only action taken by the subroutine amoeba is to call the internal subroutine amoeba_private. Why this structure? The reason has to do with meeting the twin goals of data hiding (especially for "safe" scope of variables) and program readability. The situation is this: Logically, amoeba does most of the calculating, but calls an internal subroutine amotry at several different points, with several values of the parameter fac. However, fac is not the only piece of data that must be shared with amotry; the latter also needs access to several shared variables (ihi, ndim, psum) and arguments of amoeba (p, y, func).

The obvious (but not best) way of coding this would be to put the computational guts in amoeba, with amotry as the sole internal subprogram. Assuming that fac is passed as an argument to amotry (it being the parameter that is being rapidly altered), one must decide whether to pass all the other quantities to amotry (i) as additional arguments (as is done in the Fortran 77 version), or (ii) "automatically," i.e., doing nothing except using the fact that an internal subprogram has automatic access to all of its host's entities. Each of these choices has strong disadvantages. Choice (i) is inefficient (all those arguments) and also obscures the fact that fac is the primary changing argument. Choice (ii) makes the program extremely difficult to read, because it wouldn't be obvious without careful cross-comparison of the routines *which* variables in amoeba are actually global variables that are used by amotry.

Choice (ii) is also "unsafe scoping" because it gives a nontrivially complicated internal subprogram, amotry, access to all the variables in its host. A common and difficult-to-find bug is the accidental alteration of a variable that one "thought" was local, but is actually shared. (Simple variables like `i`, `j`, and `n` are the most common culprits.)

We are therefore led to reject both choice (i) and choice (ii) in favor of a structure previously described in the subsection on Scope, Visibility, and Data Hiding in §21.5. The guts of `amoeba` are put in `amoeba_private`, a *sister routine* to `amotry`. These two siblings have mutually private name spaces. However, any variables that they need to share (including the top-level arguments of `amoeba`) are declared as variables in the enclosing `amoeba` routine. The presence of these "global variables" serves as a warning flag to the reader that data are shared between routines.

An alternative attractive way of coding the above situation would be to use a module containing `amoeba` and `amotry`. Everything would be declared private except the name `amoeba`. The global variables would be at the top level, and the arguments of `amoeba` that need to be passed to `amotry` would be handled by pointers among the global variables. Unfortunately, Fortran 90 does not support pointers to functions. Sigh!

`ilo=iminloc...ihi=imaxloc...` See discussion of these functions on p. 1017.

`call swap(y(1)...call swap(p(1,:)...` Here the `swap` routine in `nrutil` is called once with a scalar argument and once with a vector argument. Inside `nrutil` scalar and vector versions have been overloaded onto the single name `swap`, hiding all the implementation details from the calling routine.

⋆ ⋆ ⋆

```
SUBROUTINE powell(p,xi,ftol,iter,fret)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : linmin
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: xi
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), INTENT(OUT) :: fret
INTERFACE
    FUNCTION func(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: TINY=1.0e-25_sp
```

Minimization of a function `func` of N variables. (`func` is not an argument, it is a fixed function name.) Input consists of an initial starting point `p`, a vector of length N; an initial $N \times N$ matrix `xi` whose columns contain the initial set of directions (usually the N unit vectors); and `ftol`, the fractional tolerance in the function value such that failure to decrease by more than this amount on one iteration signals doneness. On output, `p` is set to the best point found, `xi` is the then-current direction set, `fret` is the returned function value at `p`, and `iter` is the number of iterations taken. The routine `linmin` is used.
Parameters: Maximum allowed iterations, and a small number.

```
INTEGER(I4B) :: i,ibig,n
REAL(SP) :: del,fp,fptt,t
REAL(SP), DIMENSION(size(p)) :: pt,ptt,xit
n=assert_eq(size(p),size(xi,1),size(xi,2),'powell')
fret=func(p)
```

```
pt(:)=p(:)                                  Save the initial point.
iter=0
do
   iter=iter+1
   fp=fret
   ibig=0
   del=0.0                                  Will be the biggest function decrease.
   do i=1,n                                 Loop over all directions in the set.
      xit(:)=xi(:,i)                        Copy the direction,
      fptt=fret
      call linmin(p,xit,fret)               minimize along it,
      if (fptt-fret > del) then             and record it if it is the largest decrease so
         del=fptt-fret                         far.
         ibig=i
      end if
   end do
   if (2.0_sp*(fp-fret) <= ftol*(abs(fp)+abs(fret))+TINY) RETURN
     Termination criterion.
   if (iter == ITMAX) call &
      nrerror('powell exceeding maximum iterations')
   ptt(:)=2.0_sp*p(:)-pt(:)                 Construct the extrapolated point and the av-
   xit(:)=p(:)-pt(:)                           erage direction moved. Save the old start-
   pt(:)=p(:)                                  ing point.
   fptt=func(ptt)                           Function value at extrapolated point.
   if (fptt >= fp) cycle                    One reason not to use new direction.
   t=2.0_sp*(fp-2.0_sp*fret+fptt)*(fp-fret-del)**2-del*(fp-fptt)**2
   if (t >= 0.0) cycle                      Other reason not to use new direction.
   call linmin(p,xit,fret)                  Move to minimum of the new direction,
   xi(:,ibig)=xi(:,n)                       and save the new direction.
   xi(:,n)=xit(:)
end do                                      Back for another iteration.
END SUBROUTINE powell
```

⋆ ⋆ ⋆

```
MODULE f1dim_mod             Used for communication from linmin to f1dim.
USE nrtype
INTEGER(I4B) :: ncom
REAL(SP), DIMENSION(:), POINTER :: pcom,xicom
CONTAINS

FUNCTION f1dim(x)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: f1dim
   Used by linmin as the one-dimensional function passed to mnbrak and brent.
INTERFACE
   FUNCTION func(x)
   USE nrtype
   REAL(SP), DIMENSION(:), INTENT(IN) :: x
   REAL(SP) :: func
   END FUNCTION func
END INTERFACE
REAL(SP), DIMENSION(:), ALLOCATABLE :: xt
allocate(xt(ncom))
xt(:)=pcom(:)+x*xicom(:)
f1dim=func(xt)
deallocate(xt)
END FUNCTION f1dim
END MODULE f1dim_mod
```

```
SUBROUTINE linmin(p,xi,fret)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : mnbrak,brent
USE f1dim_mod
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), TARGET, INTENT(INOUT) :: p,xi
REAL(SP), PARAMETER :: TOL=1.0e-4_sp
```

Given an N-dimensional point `p` and an N-dimensional direction `xi`, both vectors of length N, moves and resets `p` to where the fixed-name function `func` takes on a minimum along the direction `xi` from `p`, and replaces `xi` by the actual vector displacement that `p` was moved. Also returns as `fret` the value of `func` at the returned location `p`. This is actually all accomplished by calling the routines `mnbrak` and `brent`.
Parameter: Tolerance passed to `brent`.

```
REAL(SP) :: ax,bx,fa,fb,fx,xmin,xx
ncom=assert_eq(size(p),size(xi),'linmin')
pcom=>p                           Communicate the global variables to f1dim.
xicom=>xi
ax=0.0                            Initial guess for brackets.
xx=1.0
call mnbrak(ax,xx,bx,fa,fx,fb,f1dim)
fret=brent(ax,xx,bx,f1dim,TOL,xmin)
xi=xmin*xi                        Construct the vector results to return.
p=p+xi
END SUBROUTINE linmin
```

f90 `USE f1dim_mod` At first sight this situation is like the one involving `USE fminln` in `newt` on p. 1197: We want to pass arrays `p` and `xi` from `linmin` to `f1dim` without having them be arguments of `f1dim`. If you recall the discussion in §21.5 and on p. 1197, there are two ways of effecting this: via pointers or via allocatable arrays. There is an important difference here, however. The arrays `p` and `xi` are themselves arguments of `linmin`, and so cannot be allocatable arrays in the module. If we did want to use allocatable arrays in the module, we would have to copy `p` and `xi` into them. The pointer implementation is much more elegant, since no unnecessary copying is required. The construction here is identical to the one in `fminln` and `newt`, except that `p` and `xi` are arguments instead of automatic arrays.

⋆ ⋆ ⋆

```
MODULE df1dim_mod            Used for communication from dlinmin to f1dim and df1dim.
USE nrtype
INTEGER(I4B) :: ncom
REAL(SP), DIMENSION(:), POINTER :: pcom,xicom
CONTAINS
FUNCTION f1dim(x)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: f1dim
```

Used by `dlinmin` as the one-dimensional function passed to `mnbrak`.

```
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
REAL(SP), DIMENSION(:), ALLOCATABLE :: xt
```

```
allocate(xt(ncom))
xt(:)=pcom(:)+x*xicom(:)
f1dim=func(xt)
deallocate(xt)
END FUNCTION f1dim

FUNCTION df1dim(x)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: df1dim
    Used by dlinmin as the one-dimensional function passed to dbrent.
INTERFACE
    FUNCTION dfunc(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: dfunc
    END FUNCTION dfunc
END INTERFACE
REAL(SP), DIMENSION(:), ALLOCATABLE :: xt,df
allocate(xt(ncom),df(ncom))
xt(:)=pcom(:)+x*xicom(:)
df(:)=dfunc(xt)
df1dim=dot_product(df,xicom)
deallocate(xt,df)
END FUNCTION df1dim
END MODULE df1dim_mod
```

```
SUBROUTINE dlinmin(p,xi,fret)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : mnbrak,dbrent
USE df1dim_mod
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), TARGET :: p,xi
REAL(SP), PARAMETER :: TOL=1.0e-4_sp
    Given an N-dimensional point p and an N-dimensional direction xi, both vectors of length
    N, moves and resets p to where the fixed-name function func takes on a minimum along
    the direction xi from p, and replaces xi by the actual vector displacement that p was
    moved. Also returns as fret the value of func at the returned location p. This is actually
    all accomplished by calling the routines mnbrak and dbrent. dfunc is a fixed-name user-
    supplied function that computes the gradient of func.
    Parameter: Tolerance passed to dbrent.
REAL(SP) :: ax,bx,fa,fb,fx,xmin,xx
ncom=assert_eq(size(p),size(xi),'dlinmin')
pcom=>p                            Communicate the global variables to f1dim.
xicom=>xi
ax=0.0                             Initial guess for brackets.
xx=1.0
call mnbrak(ax,xx,bx,fa,fx,fb,f1dim)
fret=dbrent(ax,xx,bx,f1dim,df1dim,TOL,xmin)
xi=xmin*xi                         Construct the vector results to return.
p=p+xi
END SUBROUTINE dlinmin
```

f90 `USE df1dim_mod` See discussion of `USE f1dim_mod` on p. 1212.

⋆ ⋆ ⋆

```
SUBROUTINE frprmn(p,ftol,iter,fret)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : linmin
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
INTERFACE
    FUNCTION func(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP) :: func
    END FUNCTION func

    FUNCTION dfunc(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP), DIMENSION(size(p)) :: dfunc
    END FUNCTION dfunc
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: EPS=1.0e-10_sp
```

Given a starting point `p` that is a vector of length N, Fletcher-Reeves-Polak-Ribiere minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence tolerance on the function value is input as `ftol`. Returned quantities are `p` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `linmin` is called to perform line minimizations.
Parameters: `ITMAX` is the maximum allowed number of iterations; `EPS` is a small number to rectify the special case of converging to exactly zero function value.

```
INTEGER(I4B) :: its
REAL(SP) :: dgg,fp,gam,gg
REAL(SP), DIMENSION(size(p)) :: g,h,xi
fp=func(p)                              Initializations.
xi=dfunc(p)
g=-xi
h=g
xi=h
do its=1,ITMAX                          Loop over iterations.
    iter=its
    call linmin(p,xi,fret)              Next statement is the normal return:
    if (2.0_sp*abs(fret-fp) <= ftol*(abs(fret)+abs(fp)+EPS)) RETURN
    fp=fret
    xi=dfunc(p)
    gg=dot_product(g,g)
!   dgg=dot_product(xi,xi)              This statement for Fletcher-Reeves.
    dgg=dot_product(xi+g,xi)            This statement for Polak-Ribiere.
    if (gg == 0.0) RETURN               Unlikely. If gradient is exactly zero then we are al-
    gam=dgg/gg                              ready done.
    g=-xi
    h=g+gam*h
    xi=h
end do
call nrerror('frprmn: maximum iterations exceeded')
END SUBROUTINE frprmn
```

⋆ ⋆ ⋆

```
SUBROUTINE dfpmin(p,gtol,iter,fret,func,dfunc)
USE nrtype; USE nrutil, ONLY : nrerror,outerprod,unit_matrix,vabs
USE nr, ONLY : lnsrch
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: gtol
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
INTERFACE
    FUNCTION func(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP) :: func
    END FUNCTION func

    FUNCTION dfunc(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP), DIMENSION(size(p)) :: dfunc
    END FUNCTION dfunc
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: STPMX=100.0_sp,EPS=epsilon(p),TOLX=4.0_sp*EPS
```

Given a starting point p that is a vector of length N, the Broyden-Fletcher-Goldfarb-Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function func, using its gradient as calculated by a routine dfunc. The convergence requirement on zeroing the gradient is input as gtol. Returned quantities are p (the location of the minimum), iter (the number of iterations that were performed), and fret (the minimum value of the function). The routine lnsrch is called to perform approximate line minimizations.

Parameters: ITMAX is the maximum allowed number of iterations; STPMX is the scaled maximum step length allowed in line searches; EPS is the machine precision; TOLX is the convergence criterion on x values.

```
INTEGER(I4B) :: its
LOGICAL :: check
REAL(SP) :: den,fac,fad,fae,fp,stpmax,sumdg,sumxi
REAL(SP), DIMENSION(size(p)) :: dg,g,hdg,pnew,xi
REAL(SP), DIMENSION(size(p),size(p)) :: hessin
fp=func(p)                                  Calculate starting function value and gradi-
g=dfunc(p)                                      ent.
call unit_matrix(hessin)                    Initialize inverse Hessian to the unit matrix.
xi=-g                                       Initial line direction.
stpmax=STPMX*max(vabs(p),real(size(p),sp))
do its=1,ITMAX                              Main loop over the iterations.
    iter=its
    call lnsrch(p,fp,g,xi,pnew,fret,stpmax,check,func)
      The new function evaluation occurs in lnsrch; save the function value in fp for the next
      line search. It is usually safe to ignore the value of check.
    fp=fret
    xi=pnew-p                               Update the line direction,
    p=pnew                                  and the current point.
    if (maxval(abs(xi)/max(abs(p),1.0_sp)) < TOLX) RETURN
      Test for convergence on Δx.
    dg=g                                    Save the old gradient,
    g=dfunc(p)                              and get the new gradient.
    den=max(fret,1.0_sp)
    if (maxval(abs(g)*max(abs(p),1.0_sp)/den) < gtol) RETURN
      Test for convergence on zero gradient.
    dg=g-dg                                 Compute difference of gradients,
    hdg=matmul(hessin,dg)                   and difference times current matrix.
    fac=dot_product(dg,xi)                  Calculate dot products for the denominators.
    fae=dot_product(dg,hdg)
    sumdg=dot_product(dg,dg)
    sumxi=dot_product(xi,xi)
    if (fac > sqrt(EPS*sumdg*sumxi)) then        Skip update if fac not sufficiently
        fac=1.0_sp/fac                              positive.
        fad=1.0_sp/fae
        dg=fac*xi-fad*hdg                   Vector that makes BFGS different from DFP.
        hessin=hessin+fac*outerprod(xi,xi)-&       The BFGS updating formula.
            fad*outerprod(hdg,hdg)+fae*outerprod(dg,dg)
    end if
    xi=-matmul(hessin,g)                    Now calculate the next direction to go,
end do                                      and go back for another iteration.
call nrerror('dfpmin: too many iterations')
END SUBROUTINE dfpmin
```

f90 `call unit_matrix(hessin)` The `unit_matrix` routine in `nrutil` does exactly what its name suggests. The routine `dfpmin` makes use of `outerprod` from `nrutil`, as well as the matrix intrinsics `matmul` and `dot_product`, to simplify and parallelize the coding.

⋆ ⋆ ⋆

```
SUBROUTINE simplx(a,m1,m2,m3,icase,izrov,iposv)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,ifirstloc,imaxloc,&
    nrerror,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: m1,m2,m3
INTEGER(I4B), INTENT(OUT) :: icase
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: izrov,iposv
REAL(SP), PARAMETER :: EPS=1.0e-6_sp
    Simplex method for linear programming. Input parameters a, m1, m2, and m3, and output
    parameters a, icase, izrov, and iposv are described above the routine in Vol. 1. Dimen-
    sions are (M+2) × (N+1) for a, M for iposv, N for izrov, with m1+m2+m3 = M.
    Parameter: EPS is the absolute precision, which should be adjusted to the scale of your
    variables.
INTEGER(I4B) :: ip,k,kh,kp,nl1,m,n
INTEGER(I4B), DIMENSION(size(a,2)) :: l1
INTEGER(I4B), DIMENSION(m2) :: l3
REAL(SP) :: bmax
LOGICAL(LGT) :: init
m=assert_eq(size(a,1)-2,size(iposv),'simplx: m')
n=assert_eq(size(a,2)-1,size(izrov),'simplx: n')
if (m /= m1+m2+m3) call nrerror('simplx: bad input constraint counts')
if (any(a(2:m+1,1) < 0.0)) call nrerror('bad input tableau in simplx')
  Constants b_i must be nonnegative.
nl1=n
l1(1:n)=arth(1,1,n)
  Initialize index list of columns admissible for exchange.
izrov(:)=l1(1:n)                            Initially make all variables right-hand.
iposv(:)=n+arth(1,1,m)
  Initial left-hand variables. m1 type constraints are represented by having their slack variable
  initially left-hand, with no artificial variable. m2 type constraints have their slack variable
  initially left-hand, with a minus sign, and their artificial variable handled implicitly during
  their first exchange. m3 type constraints have their artificial variable initially left-hand.
init=.true.
phase1: do
    if (init) then                          Initial pass only.
        if (m2+m3 == 0) exit phase1         Origin is a feasible solution. Go to phase two.
        init=.false.
        l3(1:m2)=1
          Initialize list of m2 constraints whose slack variables have never been exchanged out
          of the initial basis.
        a(m+2,1:n+1)=-sum(a(m1+2:m+1,1:n+1),dim=1)        Compute the auxiliary objec-
    end if                                                    tive function.
    if (nl1 > 0) then
        kp=l1(imaxloc(a(m+2,l1(1:nl1)+1)))        Find the maximum coefficient of the
        bmax=a(m+2,kp+1)                               auxiliary objective function.
    else
        bmax=0.0
    end if
    phase1a: do
        if (bmax <= EPS .and. a(m+2,1) < -EPS) then
              Auxiliary objective function is still negative and can't be improved, hence no
              feasible solution exists.
            icase=-1
            RETURN
        else if (bmax <= EPS .and. a(m+2,1) <= EPS) then
              Auxiliary objective function is zero and can't be improved. This signals that we
              have a feasible starting vector. Clean out the artificial variables corresponding
              to any remaining equality constraints and then eventually exit phase one.
            do ip=m1+m2+1,m
                if (iposv(ip) == ip+n) then        Found an artificial variable for an equal-
                    if (nl1 > 0) then                  ity constraint.
                        kp=l1(imaxloc(abs(a(ip+1,l1(1:nl1)+1))))
                        bmax=a(ip+1,kp+1)
                    else
                        bmax=0.0
                    end if
                    if (bmax > EPS) exit phase1a      Exchange with column correspond-
                end if                                    ing to maximum pivot ele-
            end do                                        ment in row.
            where (spread(l3(1:m2),2,n+1) == 1) &
                a(m1+2:m1+m2+1,1:n+1)=-a(m1+2:m1+m2+1,1:n+1)
                  Change sign of row for any m2 constraints still present from the initial basis.
            exit phase1                        Go to phase two.
        end if
        call simp1                             Locate a pivot element (phase one).
        if (ip == 0) then                      Maximum of auxiliary objective function is
            icase=-1                               unbounded, so no feasible solution ex-
            RETURN                                 ists.
        end if
        exit phase1a
    end do phase1a
    call simp2(m+1,n)                          Exchange a left- and a right-hand variable.
    if (iposv(ip) >= n+m1+m2+1) then           Exchanged out an artificial variable for an
        k=ifirstloc(l1(1:nl1) == kp)               equality constraint. Make sure it stays
        nl1=nl1-1                                  out by removing it from the l1 list.
        l1(k:nl1)=l1(k+1:nl1+1)
    else
        kh=iposv(ip)-m1-n
        if (kh >= 1) then                      Exchanged out an m2 type constraint.
            if (l3(kh) /= 0) then              If it's the first time, correct the pivot col-
                l3(kh)=0                           umn for the minus sign and the implicit
                a(m+2,kp+1)=a(m+2,kp+1)+1.0_sp         artificial variable.
                a(1:m+2,kp+1)=-a(1:m+2,kp+1)
            end if
        end if
    end if
    call swap(izrov(kp),iposv(ip))             Update lists of left- and right-hand variables.
end do phase1                                  If still in phase one, go back again.
phase2: do
      We have an initial feasible solution. Now optimize it.
    if (nl1 > 0) then
        kp=l1(imaxloc(a(1,l1(1:nl1)+1)))       Test the z-row for doneness.
        bmax=a(1,kp+1)
    else
        bmax=0.0
    end if
    if (bmax <= EPS) then                   Done. Solution found. Return with the good
        icase=0                                news.
        RETURN
    end if
    call simp1                              Locate a pivot element (phase two).
    if (ip == 0) then                       Objective function is unbounded. Report and
        icase=1                                return.
        RETURN
    end if
    call simp2(m,n)                         Exchange a left- and a right-hand variable,
    call swap(izrov(kp),iposv(ip))          update lists of left- and right-hand variables,
end do phase2                               and return for another iteration.
CONTAINS

SUBROUTINE simp1
   Locate a pivot element, taking degeneracy into account.
IMPLICIT NONE
INTEGER(I4B) :: i,k
REAL(SP) :: q,q0,q1,qp
ip=0
i=ifirstloc(a(2:m+1,kp+1) < -EPS)
if (i > m) RETURN                           No possible pivots. Return with message.
q1=-a(i+1,1)/a(i+1,kp+1)
ip=i
do i=ip+1,m
    if (a(i+1,kp+1) < -EPS) then
        q=-a(i+1,1)/a(i+1,kp+1)
        if (q < q1) then
            ip=i
            q1=q
        else if (q == q1) then              We have a degeneracy.
            do k=1,n
                qp=-a(ip+1,k+1)/a(ip+1,kp+1)
                q0=-a(i+1,k+1)/a(i+1,kp+1)
                if (q0 /= qp) exit
            end do
            if (q0 < qp) ip=i
        end if
    end if
end do
END SUBROUTINE simp1

SUBROUTINE simp2(i1,k1)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: i1,k1
   Matrix operations to exchange a left-hand and right-hand variable (see text).
INTEGER(I4B) :: ip1,kp1
REAL(SP) :: piv
INTEGER(I4B), DIMENSION(k1) :: icol
INTEGER(I4B), DIMENSION(i1) :: irow
INTEGER(I4B), DIMENSION(max(i1,k1)+1) :: itmp
ip1=ip+1
kp1=kp+1
piv=1.0_sp/a(ip1,kp1)
itmp(1:k1+1)=arth(1,1,k1+1)
icol=pack(itmp(1:k1+1),itmp(1:k1+1) /= kp1)
itmp(1:i1+1)=arth(1,1,i1+1)
irow=pack(itmp(1:i1+1),itmp(1:i1+1) /= ip1)
a(irow,kp1)=a(irow,kp1)*piv
a(irow,icol)=a(irow,icol)-outerprod(a(irow,kp1),a(ip1,icol))
a(ip1,icol)=-a(ip1,icol)*piv
a(ip1,kp1)=piv
END SUBROUTINE simp2
END SUBROUTINE simplx
```

f90 `main_procedure: do` The routine `simplx` makes extensive use of named do-loops to control the program flow. The various `exit` statements have the names of the do-loops attached to them so we can easily tell where control is being transferred to. We believe that it is almost never necessary to use `goto` statements: Code will always be clearer with well-constructed block structures.

`phase1a: do...end do phase1a` This is not a real do-loop: It is executed only once, as you can see from the unconditional `exit` before the `end do`. We use this construction to define a block of code that is traversed once but that has several possible exit points.

```
where (spread(l3(1:m12-m1),2,n+1) == 1) &
   a(m1+2:m12+1,1:n+1)=-a(m1+2:m12+1,1:n+1)
```

These lines are equivalent to

```
do i=m1+1,m12
   if (l3(i-m1) == 1) a(i+1,1:n+1)=-a(i+1,1:n+1)
end do
```

⋆ ⋆ ⋆

```
SUBROUTINE anneal(x,y,iorder)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,swap
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
   This algorithm finds the shortest round-trip path to N cities whose coordinates are in the
   length N arrays x, y. The length N array iorder specifies the order in which the cities are
   visited. On input, the elements of iorder may be set to any permutation of the numbers
   1...N. This routine will return the best alternative path it can find.
INTEGER(I4B), DIMENSION(6) :: n
INTEGER(I4B) :: i1,i2,j,k,nlimit,ncity,nn,nover,nsucc
REAL(SP) :: de,harvest,path,t,tfactr
LOGICAL(LGT) :: ans
ncity=assert_eq(size(x),size(y),size(iorder),'anneal')
nover=100*ncity                      Maximum number of paths tried at any temperature,
nlimit=10*ncity                      and of successful path changes before continuing.
tfactr=0.9_sp                        Annealing schedule: t is reduced by this factor on
t=0.5_sp                                each step.
path=sum(alen_v(x(iorder(1:ncity-1)),x(iorder(2:ncity)),&
    y(iorder(1:ncity-1)),y(iorder(2:ncity))))        Calculate initial path length.
i1=iorder(ncity)                                Close the loop by tying path ends to-
i2=iorder(1)                                       gether.
path=path+alen(x(i1),x(i2),y(i1),y(i2))
do j=1,100                                      Try up to 100 temperature steps.
    nsucc=0
    do k=1,nover
        do
            call ran1(harvest)
            n(1)=1+int(ncity*harvest)               Choose beginning of segment ...
            call ran1(harvest)
            n(2)=1+int((ncity-1)*harvest)           ... and end of segment.
            if (n(2) >= n(1)) n(2)=n(2)+1
            nn=1+mod((n(1)-n(2)+ncity-1),ncity)     nn is the number of cities not on
            if (nn >= 3) exit                          the segment.
        end do
        call ran1(harvest)
          Decide whether to do a reversal or a transport.
        if (harvest < 0.5_sp) then             Do a transport.
            call ran1(harvest)
            n(3)=n(2)+int(abs(nn-2)*harvest)+1
            n(3)=1+mod(n(3)-1,ncity)           Transport to a location not on the path.
            call trncst(x,y,iorder,n,de)       Calculate cost.
            call metrop(de,t,ans)              Consult the oracle.
            if (ans) then
                nsucc=nsucc+1
                path=path+de
                call trnspt(iorder,n)          Carry out the transport.
            end if
        else                                   Do a path reversal.
            call revcst(x,y,iorder,n,de)       Calculate cost.
            call metrop(de,t,ans)              Consult the oracle.
            if (ans) then
                nsucc=nsucc+1
                path=path+de
                call revers(iorder,n)          Carry out the reversal.
            end if
        end if
        if (nsucc >= nlimit) exit              Finish early if we have enough successful
    end do                                        changes.
    write(*,*)
    write(*,*) 'T =',t,' Path Length =',path
    write(*,*) 'Successful Moves: ',nsucc
    t=t*tfactr                                 Annealing schedule.
    if (nsucc == 0) RETURN                     If no success, we are done.
end do
CONTAINS

FUNCTION alen(x1,x2,y1,y2)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,y1,y2
REAL(SP) :: alen
    Computes distance between two cities.
alen=sqrt((x2-x1)**2+(y2-y1)**2)
END FUNCTION alen

FUNCTION alen_v(x1,x2,y1,y2)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,x2,y1,y2
REAL(SP), DIMENSION(size(x1)) :: alen_v
    Computes distances between pairs of cities.
alen_v=sqrt((x2-x1)**2+(y2-y1)**2)
END FUNCTION alen_v

SUBROUTINE metrop(de,t,ans)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: de,t
LOGICAL(LGT), INTENT(OUT) :: ans
    Metropolis algorithm. ans is a logical variable that issues a verdict on whether to accept a
    reconfiguration that leads to a change de in the objective function. If de<0, ans=.true.,
    while if de>0, ans is only .true. with probability exp(-de/t), where t is a temperature
    determined by the annealing schedule.
call ran1(harvest)
ans=(de < 0.0) .or. (harvest < exp(-de/t))
END SUBROUTINE metrop

SUBROUTINE revcst(x,y,iorder,n,de)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: n
REAL(SP), INTENT(OUT) :: de
```

This subroutine returns the value of the cost function for a proposed path reversal. The arrays x and y give the coordinates of these cities. iorder holds the present itinerary. The first two values n(1) and n(2) of array n give the starting and ending cities along the path segment which is to be reversed. On output, de is the cost of making the reversal. The actual reversal is not performed by this routine.

```
INTEGER(I4B) :: ncity
REAL(SP), DIMENSION(4) :: xx,yy
ncity=size(x)
n(3)=1+mod((n(1)+ncity-2),ncity)        Find the city before n(1) ...
n(4)=1+mod(n(2),ncity)                   ...and the city after n(2).
xx(1:4)=x(iorder(n(1:4)))                Find coordinates for the four cities involved.
yy(1:4)=y(iorder(n(1:4)))
de=-alen(xx(1),xx(3),yy(1),yy(3))&       Calculate cost of disconnecting the segment
    -alen(xx(2),xx(4),yy(2),yy(4))&          at both ends and reconnecting in the op-
    +alen(xx(1),xx(4),yy(1),yy(4))&          posite order.
    +alen(xx(2),xx(3),yy(2),yy(3))
END SUBROUTINE revcst

SUBROUTINE revers(iorder,n)
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
```

This routine performs a path segment reversal. iorder is an input array giving the present itinerary. The vector n has as its first four elements the first and last cities n(1), n(2) of the path segment to be reversed, and the two cities n(3) and n(4) that immediately precede and follow this segment. n(3) and n(4) are found by subroutine revcst. On output, iorder contains the segment from n(1) to n(2) in reversed order.

```
INTEGER(I4B) :: j,k,l,nn,ncity
ncity=size(iorder)
nn=(1+mod(n(2)-n(1)+ncity,ncity))/2      This many cities must be swapped to effect
do j=1,nn                                    the reversal.
    k=1+mod((n(1)+j-2),ncity)            Start at the ends of the segment and swap
    l=1+mod((n(2)-j+ncity),ncity)            pairs of cities, moving toward the cen-
    call swap(iorder(k),iorder(l))           ter.
end do
END SUBROUTINE revers

SUBROUTINE trncst(x,y,iorder,n,de)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: n
REAL(SP), INTENT(OUT) :: de
```

This subroutine returns the value of the cost function for a proposed path segment transport. Arrays x and y give the city coordinates. iorder is an array giving the present itinerary. The first three elements of array n give the starting and ending cities of the path to be transported, and the point among the remaining cities after which it is to be inserted. On output, de is the cost of the change. The actual transport is not performed by this routine.

```
INTEGER(I4B) :: ncity
REAL(SP), DIMENSION(6) :: xx,yy
ncity=size(x)
n(4)=1+mod(n(3),ncity)                   Find the city following n(3) ...
n(5)=1+mod((n(1)+ncity-2),ncity)         ...and the one preceding n(1) ...
n(6)=1+mod(n(2),ncity)                   ...and the one following n(2).
xx(1:6)=x(iorder(n(1:6)))                Determine coordinates for the six cities in-
yy(1:6)=y(iorder(n(1:6)))                    volved.
de=-alen(xx(2),xx(6),yy(2),yy(6))&       Calculate the cost of disconnecting the path
    -alen(xx(1),xx(5),yy(1),yy(5))&          segment from n(1) to n(2), opening a
    -alen(xx(3),xx(4),yy(3),yy(4))&          space between n(3) and n(4), connect-
    +alen(xx(1),xx(3),yy(1),yy(3))&          ing the segment in the space, and con-
    +alen(xx(2),xx(4),yy(2),yy(4))&          necting n(5) to n(6).
    +alen(xx(5),xx(6),yy(5),yy(6))
END SUBROUTINE trncst

SUBROUTINE trnspt(iorder,n)
IMPLICIT NONE
```

```
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
    This routine does the actual path transport, once metrop has approved. iorder is an
    input array giving the present itinerary. The array n has as its six elements the beginning
    n(1) and end n(2) of the path to be transported, the adjacent cities n(3) and n(4)
    between which the path is to be placed, and the cities n(5) and n(6) that precede and
    follow the path. n(4), n(5), and n(6) are calculated by subroutine trncst. On output,
    iorder is modified to reflect the movement of the path segment.
INTEGER(I4B) :: m1,m2,m3,nn,ncity
INTEGER(I4B), DIMENSION(size(iorder)) :: jorder
ncity=size(iorder)
m1=1+mod((n(2)-n(1)+ncity),ncity)          Find number of cities from n(1) to n(2) ...
m2=1+mod((n(5)-n(4)+ncity),ncity)          ...and the number from n(4) to n(5)
m3=1+mod((n(3)-n(6)+ncity),ncity)          ...and the number from n(6) to n(3).
jorder(1:m1)=iorder(1+mod((arth(1,1,m1)+n(1)-2),ncity))      Copy the chosen segment.
nn=m1
jorder(nn+1:nn+m2)=iorder(1+mod((arth(1,1,m2)+n(4)-2),ncity))
  Then copy the segment from n(4) to n(5).
nn=nn+m2
jorder(nn+1:nn+m3)=iorder(1+mod((arth(1,1,m3)+n(6)-2),ncity))
  Finally, the segment from n(6) to n(3).
iorder(1:ncity)=jorder(1:ncity)            Copy jorder back into iorder.
END SUBROUTINE trnspt
END SUBROUTINE anneal
```

⋆ ⋆ ⋆

```
SUBROUTINE amebsa(p,y,pb,yb,ftol,func,iter,temptr)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,iminloc,swap
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iter
REAL(SP), INTENT(INOUT) :: yb
REAL(SP), INTENT(IN) :: ftol,temptr
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y,pb
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: p
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NMAX=200
```

Minimization of the N-dimensional function func by simulated annealing combined with the downhill simplex method of Nelder and Mead. The $(N+1)\times N$ matrix p is input. Its $N+1$ rows are N-dimensional vectors that are the vertices of the starting simplex. Also input is the vector y of length $N+1$, whose components must be preinitialized to the values of func evaluated at the $N+1$ vertices (rows) of p; ftol, the fractional convergence tolerance to be achieved in the function value for an early return; iter, and temptr. The routine makes iter function evaluations at an annealing temperature temptr, then returns. You should then decrease temptr according to your annealing schedule, reset iter, and call the routine again (leaving other arguments unaltered between calls). If iter is returned with a positive value, then early convergence and return occurred. If you initialize yb to a very large value on the first call, then yb and pb (an array of length N) will subsequently return the best function value and point ever encountered (even if it is no longer a point in the simplex).

```
INTEGER(I4B) :: ihi,ndim                   Global variables.
REAL(SP) :: yhi
REAL(SP), DIMENSION(size(p,2)) :: psum
call amebsa_private
```

```
CONTAINS

SUBROUTINE amebsa_private
INTEGER(I4B) :: i,ilo,inhi
REAL(SP) :: rtol,ylo,ynhi,ysave,ytry
REAL(SP), DIMENSION(size(y)) :: yt,harvest
ndim=assert_eq(size(p,2),size(p,1)-1,size(y)-1,size(pb),'amebsa')
psum(:)=sum(p(:,:),dim=1)
do                                          Iteration loop.
    call ran1(harvest)
    yt(:)=y(:)-temptr*log(harvest)
      Whenever we "look at" a vertex, it gets a random thermal fluctuation.
    ilo=iminloc(yt(:))                      Determine which point is the highest (worst),
    ylo=yt(ilo)                                next-highest, and lowest (best).
    ihi=imaxloc(yt(:))
    yhi=yt(ihi)
    yt(ihi)=ylo
    inhi=imaxloc(yt(:))
    ynhi=yt(inhi)
    rtol=2.0_sp*abs(yhi-ylo)/(abs(yhi)+abs(ylo))
      Compute the fractional range from highest to lowest and return if satisfactory.
    if (rtol < ftol .or. iter < 0) then        If returning, put best point and value in
        call swap(y(1),y(ilo))                    slot 1.
        call swap(p(1,:),p(ilo,:))
        RETURN
    end if
      Begin a new iteration. First extrapolate by a factor -1 through the face of the simplex
      across from the high point, i.e., reflect the simplex from the high point.
    ytry=amotsa(-1.0_sp)
    iter=iter-1
    if (ytry <= ylo) then                   Gives a result better than the best point, so
        ytry=amotsa(2.0_sp)                    try an additional extrapolation by a fac-
        iter=iter-1                            tor of 2.
    else if (ytry >= ynhi) then             The reflected point is worse than the second-
        ysave=yhi                              highest, so look for an intermediate lower
        ytry=amotsa(0.5_sp)                    point, i.e., do a one-dimensional contrac-
        iter=iter-1                            tion.
        if (ytry >= ysave) then
              Can't seem to get rid of that high point. Better contract around the lowest
              (best) point.
            p(:,:)=0.5_sp*(p(:,:)+spread(p(ilo,:),1,size(p,1)))
            do i=1,ndim+1
                if (i /= ilo) y(i)=func(p(i,:))
            end do
            iter=iter-ndim                  Keep track of function evaluations.
            psum(:)=sum(p(:,:),dim=1)
        end if
    end if
end do
END SUBROUTINE amebsa_private

FUNCTION amotsa(fac)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: fac
REAL(SP) :: amotsa
   Extrapolates by a factor fac through the face of the simplex across from the high point,
   tries it, and replaces the high point if the new point is better.
REAL(SP) :: fac1,fac2,yflu,ytry,harv
REAL(SP), DIMENSION(size(p,2)) :: ptry
fac1=(1.0_sp-fac)/ndim
fac2=fac1-fac
ptry(:)=psum(:)*fac1-p(ihi,:)*fac2
ytry=func(ptry)
if (ytry <= yb) then                        Save the best-ever.
    pb(:)=ptry(:)
    yb=ytry
end if
call ran1(harv)
yflu=ytry+temptr*log(harv)
if (yflu < yhi) then
    y(ihi)=ytry
    yhi=yflu
    psum(:)=psum(:)-p(ihi,:)+ptry(:)
    p(ihi,:)=ptry(:)
end if
amotsa=yflu
END FUNCTION amotsa
END SUBROUTINE amebsa
```

We *added* a thermal fluctuation to all the current vertices, but we *subtract* it here, so as to give the simplex a thermal Brownian motion: It *likes* to accept any suggested change.

See the discussion of `amoeba` on p. 1209 for why the routine is coded this way.