Chapter B9. Root Finding and Nonlinear Sets of Equations

```
SUBROUTINE scrsho(func)
USE nrtype
IMPLICIT NONE
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ISCR=60,JSCR=21
    For interactive "dumb terminal" use. Produce a crude graph of the function func over the
    prompted-for interval x1,x2. Query for another plot until the user signals satisfaction.
    Parameters: Number of horizontal and vertical positions in display.
INTEGER(I4B) :: i,j,jz
REAL(SP) :: dx,dyj,x,x1,x2,ybig,ysml
REAL(SP), DIMENSION(ISCR) :: y
CHARACTER(1), DIMENSION(ISCR,JSCR) :: scr
CHARACTER(1) :: blank=' ',zero='-',yy='l',xx='-',ff='x'
do
    write (*,*) ' Enter x1,x2 (= to stop)'        Query for another plot; quit if x1=x2.
    read (*,*) x1,x2
    if (x1 == x2) RETURN
    scr(1,1:JSCR)=yy                              Fill vertical sides with character 'l'.
    scr(ISCR,1:JSCR)=yy
    scr(2:ISCR-1,1)=xx                            Fill top, bottom with character '-'.
    scr(2:ISCR-1,JSCR)=xx
    scr(2:ISCR-1,2:JSCR-1)=blank                  Fill interior with blanks.
    dx=(x2-x1)/(ISCR-1)
    x=x1
    do i=1,ISCR                                   Evaluate the function at equal intervals.
        y(i)=func(x)
        x=x+dx
    end do
    ysml=min(minval(y(:)),0.0_sp)                 Limits will include 0.
    ybig=max(maxval(y(:)),0.0_sp)
    if (ybig == ysml) ybig=ysml+1.0               Be sure to separate top and bottom.
    dyj=(JSCR-1)/(ybig-ysml)
    jz=1-ysml*dyj                                 Note which row corresponds to 0.
    scr(1:ISCR,jz)=zero
    do i=1,ISCR                                   Place an indicator at function height and 0.
        j=1+(y(i)-ysml)*dyj
        scr(i,j)=ff
    end do
    write (*,'(1x,1p,e10.3,1x,80a1)') ybig,(scr(i,JSCR),i=1,ISCR)
    do j=JSCR-1,2,-1                              Display.
        write (*,'(12x,80a1)') (scr(i,j),i=1,ISCR)
    end do
    write (*,'(1x,1p,e10.3,1x,80a1)') ysml,(scr(i,1),i=1,ISCR)
```

```
    write (*,'(12x,1p,e10.3,40x,e10.3)') x1,x2
end do
END SUBROUTINE scrsho
```

f90 `CHARACTER(1), DIMENSION(ISCR,JSCR) :: scr` In Fortran 90, the length of variables of type character should be declared as `CHARACTER(1)` or `CHARACTER(len=1)` (for a variable of length 1), rather than the older form `CHARACTER*1`. While the older form is still legal syntax, the newer one is more consistent with the syntax of other type declarations. (For variables of length 1, you can actually omit the length specifier entirely, and just say `CHARACTER`.)

⋆ ⋆ ⋆

```
SUBROUTINE zbrac(func,x1,x2,succes)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: x1,x2
LOGICAL(LGT), INTENT(OUT) :: succes
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NTRY=50
REAL(SP), PARAMETER :: FACTOR=1.6_sp
```

Given a function `func` and an initial guessed range `x1` to `x2`, the routine expands the range geometrically until a root is bracketed by the returned values `x1` and `x2` (in which case `succes` returns as `.true.`) or until the range becomes unacceptably large (in which case `succes` returns as `.false.`).

```
INTEGER(I4B) :: j
REAL(SP) :: f1,f2
if (x1 == x2) call nrerror('zbrac: you have to guess an initial range')
f1=func(x1)
f2=func(x2)
succes=.true.
do j=1,NTRY
    if ((f1 > 0.0 .and. f2 < 0.0) .or. &
        (f1 < 0.0 .and. f2 > 0.0)) RETURN
    if (abs(f1) < abs(f2)) then
        x1=x1+FACTOR*(x1-x2)
        f1=func(x1)
    else
        x2=x2+FACTOR*(x2-x1)
        f2=func(x2)
    end if
end do
succes=.false.
END SUBROUTINE zbrac
```

⋆ ⋆ ⋆

```
SUBROUTINE zbrak(func,x1,x2,n,xb1,xb2,nb)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), INTENT(OUT) :: nb
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), DIMENSION(:), POINTER :: xb1,xb2
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
```

Given a function `func` defined on the interval from `x1`–`x2` subdivide the interval into `n` equally spaced segments, and search for zero crossings of the function. `nb` is returned as the number of bracketing pairs `xb1(1:nb)`, `xb2(1:nb)` that are found. `xb1` and `xb2` are pointers to arrays of length `nb` that are dynamically allocated by the routine.

```
INTEGER(I4B) :: i
REAL(SP) :: dx
REAL(SP), DIMENSION(0:n) :: f,x
LOGICAL(LGT), DIMENSION(1:n) :: mask
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
    init=.false.
    nullify(xb1,xb2)
end if
if (associated(xb1)) deallocate(xb1)
if (associated(xb2)) deallocate(xb2)
dx=(x2-x1)/n                              Determine the spacing appropriate to the mesh.
x=x1+dx*arth(0,1,n+1)
do i=0,n                                  Evaluate the function at the mesh points.
    f(i)=func(x(i))
end do
mask=f(1:n)*f(0:n-1) <= 0.0               Record where the sign changes occur.
nb=count(mask)                            Number of sign changes.
allocate(xb1(nb),xb2(nb))
xb1(1:nb)=pack(x(0:n-1),mask)             Store the bounds of each bracket.
xb2(1:nb)=pack(x(1:n),mask)
END SUBROUTINE zbrak
```

f90 This routine shows how to return arrays `xb1` and `xb2` whose size is not known in advance. The coding is explained in the subsection on pointers in §21.5.

⋆ ⋆ ⋆

```
FUNCTION rtbis(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtbis
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
```

```
INTEGER(I4B), PARAMETER :: MAXIT=40
    Using bisection, find the root of a function func known to lie between x1 and x2. The
    root, returned as rtbis, will be refined until its accuracy is ±xacc.
    Parameter: MAXIT is the maximum allowed number of bisections.
INTEGER(I4B) :: j
REAL(SP) :: dx,f,fmid,xmid
fmid=func(x2)
f=func(x1)
if (f*fmid >= 0.0) call nrerror('rtbis: root must be bracketed')
if (f < 0.0) then            Orient the search so that f>0 lies at x+dx.
    rtbis=x1
    dx=x2-x1
else
    rtbis=x2
    dx=x1-x2
end if
do j=1,MAXIT                 Bisection loop.
    dx=dx*0.5_sp
    xmid=rtbis+dx
    fmid=func(xmid)
    if (fmid <= 0.0) rtbis=xmid
    if (abs(dx) < xacc .or. fmid == 0.0) RETURN
end do
call nrerror('rtbis: too many bisections')
END FUNCTION rtbis
```

★ ★ ★

```
FUNCTION rtflsp(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror,swap
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtflsp
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=30
    Using the false position method, find the root of a function func known to lie between x1
    and x2. The root, returned as rtflsp, is refined until its accuracy is ±xacc.
    Parameter: MAXIT is the maximum allowed number of iterations.
INTEGER(I4B) :: j
REAL(SP) :: del,dx,f,fh,fl,xh,xl
fl=func(x1)
fh=func(x2)                                  Be sure the interval brackets a root.
if ((fl > 0.0 .and. fh > 0.0) .or. &
    (fl < 0.0 .and. fh < 0.0)) call &
    nrerror('rtflsp: root must be bracketed between arguments')
if (fl < 0.0) then                           Identify the limits so that xl corresponds to
    xl=x1                                        the low side.
    xh=x2
else
    xl=x2
    xh=x1
    call swap(fl,fh)
end if
dx=xh-xl
```

```
do j=1,MAXIT                                    False position loop.
    rtflsp=xl+dx*fl/(fl-fh)                     Increment with respect to latest value.
    f=func(rtflsp)
    if (f < 0.0) then                           Replace appropriate limit.
        del=xl-rtflsp
        xl=rtflsp
        fl=f
    else
        del=xh-rtflsp
        xh=rtflsp
        fh=f
    end if
    dx=xh-xl
    if (abs(del) < xacc .or. f == 0.0) RETURN       Convergence.
end do
call nrerror('rtflsp exceed maximum iterations')
END FUNCTION rtflsp
```

⋆ ⋆ ⋆

```
FUNCTION rtsec(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror,swap
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtsec
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=30
    Using the secant method, find the root of a function func thought to lie between x1 and
    x2. The root, returned as rtsec, is refined until its accuracy is ±xacc.
    Parameter: MAXIT is the maximum allowed number of iterations.
INTEGER(I4B) :: j
REAL(SP) :: dx,f,fl,xl
fl=func(x1)
f=func(x2)
if (abs(fl) < abs(f)) then                      Pick the bound with the smaller function value
    rtsec=x1                                        as the most recent guess.
    xl=x2
    call swap(fl,f)
else
    xl=x1
    rtsec=x2
end if
do j=1,MAXIT                                    Secant loop.
    dx=(xl-rtsec)*f/(f-fl)                      Increment with respect to latest value.
    xl=rtsec
    fl=f
    rtsec=rtsec+dx
    f=func(rtsec)
    if (abs(dx) < xacc .or. f == 0.0) RETURN        Convergence.
end do
call nrerror('rtsec: exceed maximum iterations')
END FUNCTION rtsec
```

⋆ ⋆ ⋆

```
FUNCTION zriddr(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: zriddr
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=60
```

Using Ridders' method, return the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `zriddr`, will be refined to an approximate accuracy `xacc`.

```
REAL(SP), PARAMETER :: UNUSED=-1.11e30_sp
INTEGER(I4B) :: j
REAL(SP) :: fh,fl,fm,fnew,s,xh,xl,xm,xnew
fl=func(x1)
fh=func(x2)
if ((fl > 0.0 .and. fh < 0.0) .or. (fl < 0.0 .and. fh > 0.0)) then
    xl=x1
    xh=x2
    zriddr=UNUSED                              Any highly unlikely value, to simplify logic
    do j=1,MAXIT                                   below.
        xm=0.5_sp*(xl+xh)
        fm=func(xm)                            First of two function evaluations per it-
        s=sqrt(fm**2-fl*fh)                        eration.
        if (s == 0.0) RETURN
        xnew=xm+(xm-xl)*(sign(1.0_sp,fl-fh)*fm/s)        Updating formula.
        if (abs(xnew-zriddr) <= xacc) RETURN
        zriddr=xnew
        fnew=func(zriddr)                      Second of two function evaluations per
        if (fnew == 0.0) RETURN                    iteration.
        if (sign(fm,fnew) /= fm) then          Bookkeeping to keep the root bracketed
            xl=xm                                  on next iteration.
            fl=fm
            xh=zriddr
            fh=fnew
        else if (sign(fl,fnew) /= fl) then
            xh=zriddr
            fh=fnew
        else if (sign(fh,fnew) /= fh) then
            xl=zriddr
            fl=fnew
        else
            call nrerror('zriddr: never get here')
        end if
        if (abs(xh-xl) <= xacc) RETURN
    end do
    call nrerror('zriddr: exceeded maximum iterations')
else if (fl == 0.0) then
    zriddr=x1
else if (fh == 0.0) then
    zriddr=x2
else
    call nrerror('zriddr: root must be bracketed')
end if
END FUNCTION zriddr
```

⋆ ⋆ ⋆

```
FUNCTION zbrent(func,x1,x2,tol)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,tol
REAL(SP) :: zbrent
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 :: EPS=epsilon(x1)
    Using Brent's method, find the root of a function func known to lie between x1 and x2.
    The root, returned as zbrent, will be refined until its accuracy is tol.
    Parameters: Maximum allowed number of iterations, and machine floating-point precision.
INTEGER(I4B) :: iter
REAL(SP) :: a,b,c,d,e,fa,fb,fc,p,q,r,s,tol1,xm
a=x1
b=x2
fa=func(a)
fb=func(b)
if ((fa > 0.0 .and. fb > 0.0) .or. (fa < 0.0 .and. fb < 0.0)) &
    call nrerror('root must be bracketed for zbrent')
c=b
fc=fb
do iter=1,ITMAX
    if ((fb > 0.0 .and. fc > 0.0) .or. (fb < 0.0 .and. fc < 0.0)) then
        c=a                                     Rename a, b, c and adjust bounding in-
        fc=fa                                       terval d.
        d=b-a
        e=d
    end if
    if (abs(fc) < abs(fb)) then
        a=b
        b=c
        c=a
        fa=fb
        fb=fc
        fc=fa
    end if
    tol1=2.0_sp*EPS*abs(b)+0.5_sp*tol           Convergence check.
    xm=0.5_sp*(c-b)
    if (abs(xm) <= tol1 .or. fb == 0.0) then
        zbrent=b
        RETURN
    end if
    if (abs(e) >= tol1 .and. abs(fa) > abs(fb)) then
        s=fb/fa                                 Attempt inverse quadratic interpolation.
        if (a == c) then
            p=2.0_sp*xm*s
            q=1.0_sp-s
        else
            q=fa/fc
            r=fb/fc
            p=s*(2.0_sp*xm*q*(q-r)-(b-a)*(r-1.0_sp))
            q=(q-1.0_sp)*(r-1.0_sp)*(s-1.0_sp)
        end if
        if (p > 0.0) q=-q                       Check whether in bounds.
        p=abs(p)
        if (2.0_sp*p < min(3.0_sp*xm*q-abs(tol1*q),abs(e*q))) then
            e=d                                 Accept interpolation.
```

```
            d=p/q
        else
            d=xm                                  Interpolation failed; use bisection.
            e=d
        end if
    else                                          Bounds decreasing too slowly; use bisec-
        d=xm                                          tion.
        e=d
    end if
    a=b                                           Move last best guess to a.
    fa=fb
    b=b+merge(d,sign(tol1,xm), abs(d) > tol1 )    Evaluate new trial root.
    fb=func(b)
end do
call nrerror('zbrent: exceeded maximum iterations')
zbrent=b
END FUNCTION zbrent
```

f90 `REAL(SP), PARAMETER :: EPS=epsilon(x1)` The routine `zbrent` works best when EPS is *exactly* the machine precision. The Fortran 90 intrinsic function `epsilon` allows us to code this in a portable fashion.

```
FUNCTION rtnewt(funcd,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtnewt
INTERFACE
    SUBROUTINE funcd(x,fval,fderiv)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: fval,fderiv
    END SUBROUTINE funcd
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=20
    Using the Newton-Raphson method, find the root of a function known to lie in the interval
    [x1, x2]. The root rtnewt will be refined until its accuracy is known within ±xacc. funcd
    is a user-supplied subroutine that returns both the function value and the first derivative of
    the function.
    Parameter: MAXIT is the maximum number of iterations.
INTEGER(I4B) :: j
REAL(SP) :: df,dx,f
rtnewt=0.5_sp*(x1+x2)                             Initial guess.
do j=1,MAXIT
    call funcd(rtnewt,f,df)
    dx=f/df
    rtnewt=rtnewt-dx
    if ((x1-rtnewt)*(rtnewt-x2) < 0.0)&
        call nrerror('rtnewt: values jumped out of brackets')
    if (abs(dx) < xacc) RETURN                    Convergence.
end do
call nrerror('rtnewt exceeded maximum iterations')
END FUNCTION rtnewt
```

⋆ ⋆ ⋆

```
FUNCTION rtsafe(funcd,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtsafe
INTERFACE
    SUBROUTINE funcd(x,fval,fderiv)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: fval,fderiv
    END SUBROUTINE funcd
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=100
    Using a combination of Newton-Raphson and bisection, find the root of a function bracketed
    between x1 and x2. The root, returned as the function value rtsafe, will be refined until
    its accuracy is known within ±xacc. funcd is a user-supplied subroutine that returns both
    the function value and the first derivative of the function.
    Parameter: MAXIT is the maximum allowed number of iterations.
INTEGER(I4B) :: j
REAL(SP) :: df,dx,dxold,f,fh,fl,temp,xh,xl
call funcd(x1,fl,df)
call funcd(x2,fh,df)
if ((fl > 0.0 .and. fh > 0.0) .or. &
    (fl < 0.0 .and. fh < 0.0)) &
    call nrerror('root must be bracketed in rtsafe')
if (fl == 0.0) then
    rtsafe=x1
    RETURN
else if (fh == 0.0) then
    rtsafe=x2
    RETURN
else if (fl < 0.0) then                 Orient the search so that f(xl) < 0.
    xl=x1
    xh=x2
else
    xh=x1
    xl=x2
end if
rtsafe=0.5_sp*(x1+x2)                   Initialize the guess for root,
dxold=abs(x2-x1)                        the "stepsize before last,"
dx=dxold                                and the last step.
call funcd(rtsafe,f,df)
do j=1,MAXIT                            Loop over allowed iterations.
    if (((rtsafe-xh)*df-f)*((rtsafe-xl)*df-f) > 0.0 .or. &
        abs(2.0_sp*f) > abs(dxold*df) ) then
          Bisect if Newton out of range, or not decreasing fast enough.
        dxold=dx
        dx=0.5_sp*(xh-xl)
        rtsafe=xl+dx
        if (xl == rtsafe) RETURN        Change in root is negligible.
    else                                Newton step acceptable. Take it.
        dxold=dx
        dx=f/df
        temp=rtsafe
        rtsafe=rtsafe-dx
        if (temp == rtsafe) RETURN
    end if
    if (abs(dx) < xacc) RETURN          Convergence criterion.
    call funcd(rtsafe,f,df)             One new function evaluation per iteration.
    if (f < 0.0) then                   Maintain the bracket on the root.
        xl=rtsafe
    else
        xh=rtsafe
```

```
    end if
end do
call nrerror('rtsafe: exceeded maximum iterations')
END FUNCTION rtsafe
```

⋆ ⋆ ⋆

```
SUBROUTINE laguer(a,x,its)
USE nrtype; USE nrutil, ONLY : nrerror,poly,poly_term
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: its
COMPLEX(SPC), INTENT(INOUT) :: x
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
REAL(SP), PARAMETER :: EPS=epsilon(1.0_sp)
INTEGER(I4B), PARAMETER :: MR=8,MT=10,MAXIT=MT*MR
```

Given an array of $M+1$ complex coefficients a of the polynomial $\sum_{i=1}^{M+1} \mathtt{a}(i)x^{i-1}$, and given a complex value x, this routine improves x by Laguerre's method until it converges, within the achievable roundoff limit, to a root of the given polynomial. The number of iterations taken is returned as its.

Parameters: EPS is the estimated fractional roundoff error. We try to break (rare) limit cycles with MR different fractional values, once every MT steps, for MAXIT total allowed iterations.

```
INTEGER(I4B) :: iter,m
REAL(SP) :: abx,abp,abm,err
COMPLEX(SPC) :: dx,x1,f,g,h,sq,gp,gm,g2
COMPLEX(SPC), DIMENSION(size(a)) :: b,d
REAL(SP), DIMENSION(MR) :: frac = &
    (/ 0.5_sp,0.25_sp,0.75_sp,0.13_sp,0.38_sp,0.62_sp,0.88_sp,1.0_sp /)
```

Fractions used to break a limit cycle.

```
m=size(a)-1
do iter=1,MAXIT                              Loop over iterations up to allowed maximum.
    its=iter
    abx=abs(x)
    b(m+1:1:-1)=poly_term(a(m+1:1:-1),x)      Efficient computation of the polynomial
    d(m:1:-1)=poly_term(b(m+1:2:-1),x)            and its first two derivatives.
    f=poly(x,d(2:m))
    err=EPS*poly(abx,abs(b(1:m+1)))           Esimate of roundoff in evaluating polynomial.
    if (abs(b(1)) <= err) RETURN              We are on the root.
    g=d(1)/b(1)                               The generic case: Use Laguerre's formula.
    g2=g*g
    h=g2-2.0_sp*f/b(1)
    sq=sqrt((m-1)*(m*h-g2))
    gp=g+sq
    gm=g-sq
    abp=abs(gp)
    abm=abs(gm)
    if (abp < abm) gp=gm
    if (max(abp,abm) > 0.0) then
        dx=m/gp
    else
        dx=exp(cmplx(log(1.0_sp+abx),iter,kind=spc))
    end if
    x1=x-dx
    if (x == x1) RETURN                       Converged.
    if (mod(iter,MT) /= 0) then
        x=x1
    else                                      Every so often we take a fractional step, to
        x=x-dx*frac(iter/MT)                      break any limit cycle (itself a rare occur-
    end if                                        rence).
end do
call nrerror('laguer: too many iterations')
```

Very unusual — can occur only for complex roots. Try a different starting guess for the root.

```
END SUBROUTINE laguer
```

f90 `b(m+1:1:-1)=poly_term...f=poly(x,d(2:m))` The `poly_term` function in `nrutil` tabulates the partial sums of a polynomial, while `poly` evaluates the polynomial at `x`. In this example, we use `poly_term` on the coefficient array in reverse order, so that the value of the polynomial ends up in `b(1)` and the value of its first derivative in `d(1)`.

`dx=exp(cmplx(log(1.0_sp+abx),iter,kind=spc))` The intrinsic function `cmplx` returns a quantity of type default complex unless the `kind` argument is present. To facilitate converting our routines from single to double precision, we always include the `kind` argument explicitly so that when you redefine `spc` in `nrtype` to be double-precision complex the conversions are carried out correctly.

⋆ ⋆ ⋆

```
SUBROUTINE zroots(a,roots,polish)
USE nrtype; USE nrutil, ONLY : assert_eq,poly_term
USE nr, ONLY : laguer,indexx
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: roots
LOGICAL(LGT), INTENT(IN) :: polish
REAL(SP), PARAMETER :: EPS=1.0e-6_sp
    Given the array of M + 1 complex coefficients a of the polynomial sum_{i=1}^{M+1} a(i)x^{i-1}, this
    routine successively calls laguer and finds all M complex roots. The logical variable
    polish should be input as .true. if polishing (also by Laguerre's method) is desired,
    .false. if the roots will be subsequently polished by other means.
    Parameter: EPS is a small number.
INTEGER(I4B) :: j,its,m
INTEGER(I4B), DIMENSION(size(roots)) :: indx
COMPLEX(SPC) :: x
COMPLEX(SPC), DIMENSION(size(a)) :: ad
m=assert_eq(size(roots),size(a)-1,'zroots')
ad(:)=a(:)                                  Copy of coefficients for successive deflation.
do j=m,1,-1                                 Loop over each root to be found.
    x=cmplx(0.0_sp,kind=spc)
      Start at zero to favor convergence to smallest remaining root.
    call laguer(ad(1:j+1),x,its)            Find the root.
    if (abs(aimag(x)) <= 2.0_sp*EPS**2*abs(real(x))) &
        x=cmplx(real(x),kind=spc)
    roots(j)=x
    ad(j:1:-1)=poly_term(ad(j+1:2:-1),x)        Forward deflation.
end do
if (polish) then
    do j=1,m                                Polish the roots using the undeflated coeffi-
        call laguer(a(:),roots(j),its)          cients.
    end do
end if
call indexx(real(roots),indx)               Sort roots by their real parts.
roots=roots(indx)
END SUBROUTINE zroots
```

f90 `x=cmplx(0.0_sp,kind=spc)...x=cmplx(real(x),kind=spc)` See the discussion of why we include `kind=spc` just above. Note that while `real(x)` returns type default real if `x` is integer or real, it returns single or double precision correctly if `x` is complex.

⋆ ⋆ ⋆

```
SUBROUTINE zrhqr(a,rtr,rti)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : balanc,hqr,indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: rtr,rti
```

Find all the roots of a polynomial with real coefficients, $\sum_{i=1}^{M+1} \mathtt{a}(i)x^{i-1}$, given the array of $M+1$ coefficients a. The method is to construct an upper Hessenberg matrix whose eigenvalues are the desired roots, and then use the routines balanc and hqr. The real and imaginary parts of the M roots are returned in rtr and rti, respectively.

```
INTEGER(I4B) :: k,m
INTEGER(I4B), DIMENSION(size(rtr)) :: indx
REAL(SP), DIMENSION(size(a)-1,size(a)-1) :: hess
m=assert_eq(size(rtr),size(rti),size(a)-1,'zrhqr')
if (a(m+1) == 0.0) call &
    nrerror('zrhqr: Last value of array a must not be 0')
hess(1,:)=-a(m:1:-1)/a(m+1)          Construct the matrix.
hess(2:m,:)=0.0
do k=1,m-1
    hess(k+1,k)=1.0
end do
call balanc(hess)                    Find its eigenvalues.
call hqr(hess,rtr,rti)
call indexx(rtr,indx)                Sort roots by their real parts.
rtr=rtr(indx)
rti=rti(indx)
END SUBROUTINE zrhqr
```

⋆ ⋆ ⋆

```
SUBROUTINE qroot(p,b,c,eps)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : poldiv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: p
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: eps
INTEGER(I4B), PARAMETER :: ITMAX=20
REAL(SP), PARAMETER :: TINY=1.0e-6_sp
```

Given an array of N coefficients p of a polynomial of degree $N-1$, and trial values for the coefficients of a quadratic factor $x^2 + \mathtt{b}x + \mathtt{c}$, improve the solution until the coefficients b,c change by less than eps. The routine poldiv of §5.3 is used.

Parameters: ITMAX is the maximum number of iterations, TINY is a small number.

```
INTEGER(I4B) :: iter,n
REAL(SP) :: delb,delc,div,r,rb,rc,s,sb,sc
REAL(SP), DIMENSION(3) :: d
REAL(SP), DIMENSION(size(p)) :: q,qq,rem
n=size(p)
d(3)=1.0
do iter=1,ITMAX
    d(2)=b
    d(1)=c
    call poldiv(p,d,q,rem)
    s=rem(1)                         First division gives r,s.
    r=rem(2)
    call poldiv(q(1:n-1),d(:),qq(1:n-1),rem(1:n-1))
    sc=-rem(1)                       Second division gives partial r,s with respect
    rc=-rem(2)                           to c.
    sb=-c*rc
    rb=sc-b*rc
    div=1.0_sp/(sb*rc-sc*rb)         Solve 2x2 equation.
```

```
    delb=(r*sc-s*rc)*div
    delc=(-r*sb+s*rb)*div
    b=b+delb
    c=c+delc
    if ((abs(delb) <= eps*abs(b) .or. abs(b) < TINY) .and. &
        (abs(delc) <= eps*abs(c) .or. abs(c) < TINY)) RETURN   Coefficients converged.
end do
call nrerror('qroot: too many iterations')
END SUBROUTINE qroot
```

★ ★ ★

```
SUBROUTINE mnewt(ntrial,x,tolx,tolf,usrfun)
USE nrtype
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ntrial
REAL(SP), INTENT(IN) :: tolx,tolf
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
INTERFACE
    SUBROUTINE usrfun(x,fvec,fjac)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: fvec
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: fjac
    END SUBROUTINE usrfun
END INTERFACE
    Given an initial guess x for a root in N dimensions, take ntrial Newton-Raphson steps to
    improve the root. Stop if the root converges in either summed absolute variable increments
    tolx or summed absolute function values tolf.
INTEGER(I4B) :: i
INTEGER(I4B), DIMENSION(size(x)) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(size(x)) :: fvec,p
REAL(SP), DIMENSION(size(x),size(x)) :: fjac
do i=1,ntrial
    call usrfun(x,fvec,fjac)
      User subroutine supplies function values at x in fvec and Jacobian matrix in fjac.
    if (sum(abs(fvec)) <= tolf) RETURN        Check function convergence.
    p=-fvec                                   Right-hand side of linear equations.
    call ludcmp(fjac,indx,d)                  Solve linear equations using LU decom-
    call lubksb(fjac,indx,p)                      position.
    x=x+p                                     Update solution.
    if (sum(abs(p)) <= tolx) RETURN           Check root convergence.
end do
END SUBROUTINE mnewt
```

★ ★ ★

```
SUBROUTINE lnsrch(xold,fold,g,p,x,f,stpmax,check,func)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,vabs
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xold,g
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
REAL(SP), INTENT(IN) :: fold,stpmax
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
REAL(SP), INTENT(OUT) :: f
LOGICAL(LGT), INTENT(OUT) :: check
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP) :: func
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: ALF=1.0e-4_sp,TOLX=epsilon(x)
```

Given an N-dimensional point xold, the value of the function and gradient there, fold and g, and a direction p, finds a new point x along the direction p from xold where the function func has decreased "sufficiently." xold, g, p, and x are all arrays of length N. The new function value is returned in f. stpmax is an input quantity that limits the length of the steps so that you do not try to evaluate the function in regions where it is undefined or subject to overflow. p is usually the Newton direction. The output quantity check is false on a normal exit. It is true when x is too close to xold. In a minimization algorithm, this usually signals convergence and can be ignored. However, in a zero-finding algorithm the calling program should check whether the convergence is spurious.

Parameters: ALF ensures sufficient decrease in function value; TOLX is the convergence criterion on Δx.

```
INTEGER(I4B) :: ndum
REAL(SP) :: a,alam,alam2,alamin,b,disc,f2,pabs,rhs1,rhs2,slope,tmplam
ndum=assert_eq(size(g),size(p),size(x),size(xold),'lnsrch')
check=.false.
pabs=vabs(p(:))
if (pabs > stpmax) p(:)=p(:)*stpmax/pabs          Scale if attempted step is too big.
slope=dot_product(g,p)
if (slope >= 0.0) call nrerror('roundoff problem in lnsrch')
alamin=TOLX/maxval(abs(p(:))/max(abs(xold(:)),1.0_sp))      Compute λmin.
alam=1.0                                          Always try full Newton step first.
do                                                Start of iteration loop.
    x(:)=xold(:)+alam*p(:)
    f=func(x)
    if (alam < alamin) then                       Convergence on Δx. For zero find-
        x(:)=xold(:)                                  ing, the calling program should
        check=.true.                                  verify the convergence.
        RETURN
    else if (f <= fold+ALF*alam*slope) then       Sufficient function decrease.
        RETURN
    else                                          Backtrack.
        if (alam == 1.0) then                     First time.
            tmplam=-slope/(2.0_sp*(f-fold-slope))
        else                                      Subsequent backtracks.
            rhs1=f-fold-alam*slope
            rhs2=f2-fold-alam2*slope
            a=(rhs1/alam**2-rhs2/alam2**2)/(alam-alam2)
            b=(-alam2*rhs1/alam**2+alam*rhs2/alam2**2)/&
                (alam-alam2)
            if (a == 0.0) then
                tmplam=-slope/(2.0_sp*b)
            else
                disc=b*b-3.0_sp*a*slope
                if (disc < 0.0) then
                    tmplam=0.5_sp*alam
                else if (b <= 0.0) then
```

```
                tmplam=(-b+sqrt(disc))/(3.0_sp*a)
            else
                tmplam=-slope/(b+sqrt(disc))
            end if
        end if
        if (tmplam > 0.5_sp*alam) tmplam=0.5_sp*alam    λ ≤ 0.5λ1.
    end if
end if
alam2=alam
f2=f
alam=max(tmplam,0.1_sp*alam)                    λ ≥ 0.1λ1.
end do                                          Try again.
END SUBROUTINE lnsrch



SUBROUTINE newt(x,check)
USE nrtype; USE nrutil, ONLY : nrerror,vabs
USE nr, ONLY : fdjac,lnsrch,lubksb,ludcmp
USE fminln                                  Communicates with fmin.
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
LOGICAL(LGT), INTENT(OUT) :: check
INTEGER(I4B), PARAMETER :: MAXITS=200
REAL(SP), PARAMETER :: TOLF=1.0e-4_sp,TOLMIN=1.0e-6_sp,TOLX=epsilon(x),&
    STPMX=100.0
    Given an initial guess x for a root in N dimensions, find the root by a globally convergent Newton's method. The length N vector of functions to be zeroed, called fvec in the routine below, is returned by a user-supplied routine that must be called funcv and have the declaration FUNCTION funcv(x). The output quantity check is false on a normal return and true if the routine has converged to a local minimum of the function fmin defined below. In this case try restarting from a different initial guess.
    Parameters: MAXITS is the maximum number of iterations; TOLF sets the convergence criterion on function values; TOLMIN sets the criterion for deciding whether spurious convergence to a minimum of fmin has occurred; TOLX is the convergence criterion on δx; STPMX is the scaled maximum step length allowed in line searches.
INTEGER(I4B) :: its
INTEGER(I4B), DIMENSION(size(x)) :: indx
REAL(SP) :: d,f,fold,stpmax
REAL(SP), DIMENSION(size(x)) :: g,p,xold
REAL(SP), DIMENSION(size(x)), TARGET :: fvec
REAL(SP), DIMENSION(size(x),size(x)) :: fjac
fmin_fvecp=>fvec
f=fmin(x)                                   fvec is also computed by this call.
if (maxval(abs(fvec(:))) < 0.01_sp*TOLF) then       Test for initial guess being a root.
    check=.false.                                       Use more stringent test than
    RETURN                                              simply TOLF.
end if
stpmax=STPMX*max(vabs(x(:)),real(size(x),sp))       Calculate stpmax for line searches.
do its=1,MAXITS                             Start of iteration loop.
    call fdjac(x,fvec,fjac)
      If analytic Jacobian is available, you can replace the routine fdjac below with your own routine.
    g(:)=matmul(fvec(:),fjac(:,:))          Compute ∇f for the line search.
    xold(:)=x(:)                            Store x,
    fold=f                                  and f.
    p(:)=-fvec(:)                           Right-hand side for linear equations.
    call ludcmp(fjac,indx,d)                Solve linear equations by LU decomposition.
    call lubksb(fjac,indx,p)
    call lnsrch(xold,fold,g,p,x,f,stpmax,check,fmin)
      lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.
    if (maxval(abs(fvec(:))) < TOLF) then       Test for convergence on function val-
        check=.false.                               ues.
        RETURN
```

```
    end if
    if (check) then                          Check for gradient of f zero, i.e., spurious
        check=(maxval(abs(g(:))*max(abs(x(:)),1.0_sp) / &       convergence.
            max(f,0.5_sp*size(x))) < TOLMIN)
        RETURN                               Test for convergence on δx.
    end if
    if (maxval(abs(x(:)-xold(:))/max(abs(x(:)),1.0_sp)) < TOLX) &
        RETURN
end do
call nrerror('MAXITS exceeded in newt')
END SUBROUTINE newt
```

f90 USE fminln Here we have an example of how to pass an array fvec to a function fmin without making it an argument of fmin. In the language of §21.5, we are using Method 2: We define a pointer fmin_fvecp in the module fminln:

```
REAL(SP), DIMENSION(:), POINTER :: fmin_fvecp
```

fvec itself is declared as an automatic array of the appropriate size in newt:

```
REAL(SP), DIMENSION(size(x)), TARGET :: fvec
```

On entry into newt, the pointer is associated:

```
fmin_fvecp=>fvec
```

The pointer is then used in fmin as a synonym for fvec. If you are sufficiently paranoid, you can test whether fmin_fvecp has in fact been associated on entry into fmin. Heeding our admonition always to deallocate memory when it no longer is needed, you may ask where the deallocation takes place in this example. Answer: On exit from newt, the automatic array fvec is automatically freed.

The Method 1 way of setting up this task is to declare an allocatable array in the module:

```
REAL(SP), DIMENSION(:), ALLOCATABLE :: fvec
```

On entry into newt we allocate it appropriately:

```
allocate(fvec,size(x))
```

and it can now be used in both newt and fmin. Of course, we must remember to deallocate explicitly fvec on exit from newt. If we forget, all kinds of bad things would happen on a second call to newt. The status of fvec on the first return from newt becomes undefined. The status cannot be tested with if(allocated(...)), and fvec may not be referenced in any way. If we tried to guard against this by adding the SAVE attribute to the declaration of fvec, then we would generate an error from trying to allocate an already-allocated array.

```
SUBROUTINE fdjac(x,fvec,df)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: fvec
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: df
INTERFACE
```

```
    FUNCTION funcv(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funcv
    END FUNCTION funcv
END INTERFACE
REAL(SP), PARAMETER :: EPS=1.0e-4_sp
```

Computes forward-difference approximation to Jacobian. On input, x is the point at which the Jacobian is to be evaluated, and fvec is the vector of function values at the point, both arrays of length N. df is the $N \times N$ output Jacobian. FUNCTION funcv(x) is a fixed-name, user-supplied routine that returns the vector of functions at x.

Parameter: EPS is the approximate square root of the machine precision.

```
INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(x)) :: xsav,xph,h
n=assert_eq(size(x),size(fvec),size(df,1),size(df,2),'fdjac')
xsav=x
h=EPS*abs(xsav)
where (h == 0.0) h=EPS
xph=xsav+h                                  Trick to reduce finite precision error.
h=xph-xsav
do j=1,n
    x(j)=xph(j)
    df(:,j)=(funcv(x)-fvec(:))/h(j)         Forward difference formula.
    x(j)=xsav(j)
end do
END SUBROUTINE fdjac
```

```
MODULE fminln
USE nrtype; USE nrutil, ONLY : nrerror
REAL(SP), DIMENSION(:), POINTER :: fmin_fvecp
CONTAINS

FUNCTION fmin(x)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP) :: fmin
```

Returns $f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F}$ at x. FUNCTION funcv(x) is a fixed-name, user-supplied routine that returns the vector of functions at x. The pointer fmin_vecp communicates the function values back to newt.

```
INTERFACE
    FUNCTION funcv(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funcv
    END FUNCTION funcv
END INTERFACE
if (.not. associated(fmin_fvecp)) call &
    nrerror('fmin: problem with pointer for returned values')
fmin_fvecp=funcv(x)
fmin=0.5_sp*dot_product(fmin_fvecp,fmin_fvecp)
END FUNCTION fmin
END MODULE fminln
```

⋆ ⋆ ⋆

```
SUBROUTINE broydn(x,check)
USE nrtype; USE nrutil, ONLY : get_diag,lower_triangle,nrerror,&
    outerprod,put_diag,unit_matrix,vabs
USE nr, ONLY : fdjac,lnsrch,qrdcmp,qrupdt,rsolv
USE fminln                                      Communicates with fmin.
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
LOGICAL(LGT), INTENT(OUT) :: check
INTEGER(I4B), PARAMETER :: MAXITS=200
REAL(SP), PARAMETER :: EPS=epsilon(x),TOLF=1.0e-4_sp,TOLMIN=1.0e-6_sp,&
    TOLX=EPS,STPMX=100.0
```

Given an initial guess x for a root in N dimensions, find the root by Broyden's method embedded in a globally convergent strategy. The length N vector of functions to be zeroed, called fvec in the routine below, is returned by a user-supplied routine that *must* be called funcv and have the declaration FUNCTION funcv(x). The subroutine fdjac and the function fmin from newt are used. The output quantity check is false on a normal return and true if the routine has converged to a local minimum of the function fmin or if Broyden's method can make no further progress. In this case try restarting from a different initial guess.

Parameters: MAXITS is the maximum number of iterations; EPS is the machine precision; TOLF sets the convergence criterion on function values; TOLMIN sets the criterion for deciding whether spurious convergence to a minimum of fmin has occurred; TOLX is the convergence criterion on $\delta\mathbf{x}$; STPMX is the scaled maximum step length allowed in line searches.

```
INTEGER(I4B) :: i,its,k,n
REAL(SP) :: f,fold,stpmax
REAL(SP), DIMENSION(size(x)), TARGET :: fvec
REAL(SP), DIMENSION(size(x)) :: c,d,fvcold,g,p,s,t,w,xold
REAL(SP), DIMENSION(size(x),size(x)) :: qt,r
LOGICAL :: restrt,sing
fmin_fvecp=>fvec
n=size(x)
f=fmin(x)                                       fvec is also computed by this call.
if (maxval(abs(fvec(:))) < 0.01_sp*TOLF) then       Test for initial guess being a root.
    check=.false.                                       Use more stringent test than
    RETURN                                              simply TOLF.
end if
stpmax=STPMX*max(vabs(x(:)),real(n,sp))         Calculate stpmax for line searches.
restrt=.true.                                   Ensure initial Jacobian gets computed.
do its=1,MAXITS                                 Start of iteration loop.
    if (restrt) then
        call fdjac(x,fvec,r)                    Initialize or reinitialize Jacobian in r.
        call qrdcmp(r,c,d,sing)                 QR decomposition of Jacobian.
        if (sing) call nrerror('singular Jacobian in broydn')
        call unit_matrix(qt)                    Form Q^T explicitly.
        do k=1,n-1
            if (c(k) /= 0.0) then
                qt(k:n,:)=qt(k:n,:)-outerprod(r(k:n,k),&
                    matmul(r(k:n,k),qt(k:n,:)))/c(k)
            end if
        end do
        where (lower_triangle(n,n)) r(:,:)=0.0
        call put_diag(d(:),r(:,:))              Form R explicitly.
    else                                        Carry out Broyden update.
        s(:)=x(:)-xold(:)                       s = δx.
        do i=1,n                                t = R · s.
            t(i)=dot_product(r(i,i:n),s(i:n))
        end do
        w(:)=fvec(:)-fvcold(:)-matmul(t(:),qt(:,:))     w = δF − B · s.
        where (abs(w(:)) < EPS*(abs(fvec(:))+abs(fvcold(:)))) &
            w(:)=0.0                            Don't update with noisy components of
        if (any(w(:) /= 0.0)) then                  w.
            t(:)=matmul(qt(:,:),w(:))           t = Q^T · w.
            s(:)=s(:)/dot_product(s,s)          Store s/(s · s) in s.
```

```
            call qrupdt(r,qt,t,s)              Update R and Q^T.
            d(:)=get_diag(r(:,:))              Diagonal of R stored in d.
            if (any(d(:) == 0.0)) &
                call nrerror('r singular in broydn')
        end if
    end if
    p(:)=-matmul(qt(:,:),fvec(:))              r.h.s. for linear equations is -Q^T . F.
    do i=1,n                                   Compute ∇f ≈ (Q . R)^T . F for the line
        g(i)=-dot_product(r(1:i,i),p(1:i))         search.
    end do
    xold(:)=x(:)                               Store x, F, and f.
    fvcold(:)=fvec(:)
    fold=f
    call rsolv(r,d,p)                          Solve linear equations.
    call lnsrch(xold,fold,g,p,x,f,stpmax,check,fmin)
      lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.
    if (maxval(abs(fvec(:))) < TOLF) then          Test for convergence on function val-
        check=.false.                                  ues.
        RETURN
    end if
    if (check) then                            True if line search failed to find a new
        if (restrt .or. maxval(abs(g(:))*max(abs(x(:)), &              x.
            1.0_sp)/max(f,0.5_sp*n)) < TOLMIN) RETURN
              If restrt is true we have failure: We have already tried reinitializing the Jacobian. The other test is for gradient of f zero, i.e., spurious convergence.
        restrt=.true.                          Try reinitializing the Jacobian.
    else                                       Successful step; will use Broyden update
        restrt=.false.                             for next step.
        if (maxval((abs(x(:)-xold(:)))/max(abs(x(:)), &
            1.0_sp)) < TOLX) RETURN            Test for convergence on δx.
    end if
end do
call nrerror('MAXITS exceeded in broydn')
END SUBROUTINE broydn
```

f90 `USE fminln` See discussion for `newt` on p. 1197.

`qt(k:n,:)=...outerprod...matmul` Another example of the coding of equation (22.1.6).

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

`call put_diag(d(:),r(:,:))` This subroutine in `nrutil` sets the diagonal values of the matrix `r` to the values of the vector `d`. It is overloaded so that `d` could be a scalar, in which case the scalar value would be broadcast onto the diagonal of `r`.