

9.7 Globally Convergent Methods for Nonlinear Systems of Equations

We have seen that Newton's method for solving nonlinear equations has an unfortunate tendency to wander off into the wild blue yonder if the initial guess is not sufficiently close to the root. A *global* method is one that converges to a solution from almost any starting point. In this section we will develop an algorithm that combines the rapid local convergence of Newton's method with a globally convergent strategy that will guarantee some progress towards the solution at each iteration. The algorithm is closely related to the quasi-Newton method of minimization which we will describe in §10.7.

Recall our discussion of §9.6: the Newton step for the set of equations

$$\mathbf{F}(\mathbf{x}) = 0 \quad (9.7.1)$$

is

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta \mathbf{x} \quad (9.7.2)$$

where

$$\delta \mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F} \quad (9.7.3)$$

Here $\mathbf{J}$ is the Jacobian matrix. How do we decide whether to accept the Newton step $\delta \mathbf{x}$? A reasonable strategy is to require that the step decrease $|\mathbf{F}|^2 = \mathbf{F} \cdot \mathbf{F}$. This is the same requirement we would impose if we were trying to minimize

$$f = \frac{1}{2} \mathbf{F} \cdot \mathbf{F} \quad (9.7.4)$$

(The $\frac{1}{2}$ is for later convenience.) Every solution to (9.7.1) minimizes (9.7.4), but there may be local minima of (9.7.4) that are not solutions to (9.7.1). Thus, as already mentioned, simply applying one of our minimum finding algorithms from Chapter 10 to (9.7.4) is *not* a good idea.

To develop a better strategy, note that the Newton step (9.7.3) is a *descent direction* for f :

$$\nabla f \cdot \delta \mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0 \quad (9.7.5)$$

Thus our strategy is quite simple: We always first try the full Newton step, because once we are close enough to the solution we will get quadratic convergence. However, we check at each iteration that the proposed step reduces f . If not, we *backtrack* along the Newton direction until we have an acceptable step. Because the Newton step is a descent direction for f , we are guaranteed to find an acceptable step by backtracking. We will discuss the backtracking algorithm in more detail below.

Note that this method essentially minimizes f by taking Newton steps designed to bring $\mathbf{F}$ to zero. This is *not* equivalent to minimizing f directly by taking Newton steps designed to bring ∇f to zero. While the method can still occasionally fail by landing on a local minimum of f , this is quite rare in practice. The routine `newt` below will warn you if this happens. The remedy is to try a new starting point.

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

Line Searches and Backtracking

When we are not close enough to the minimum of f , taking the full Newton step $\mathbf{p} = \delta\mathbf{x}$ need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that *initially* f decreases as we move in the Newton direction. So the goal is to move to a new point $\mathbf{x}_{\text{new}}$ along the *direction* of the Newton step $\mathbf{p}$, but not necessarily all the way:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \lambda\mathbf{p}, \quad 0 < \lambda \leq 1 \quad (9.7.6)$$

The aim is to find λ so that $f(\mathbf{x}_{\text{old}} + \lambda\mathbf{p})$ has decreased sufficiently. Until the early 1970s, standard practice was to choose λ so that $\mathbf{x}_{\text{new}}$ exactly minimizes f in the direction $\mathbf{p}$. However, we now know that it is extremely wasteful of function evaluations to do so. A better strategy is as follows: Since $\mathbf{p}$ is always the Newton direction in our algorithms, we first try $\lambda = 1$, the full Newton step. This will lead to quadratic convergence when $\mathbf{x}$ is sufficiently close to the solution. However, if $f(\mathbf{x}_{\text{new}})$ does not meet our acceptance criteria, we *backtrack* along the Newton direction, trying a smaller value of λ , until we find a suitable point. Since the Newton direction is a descent direction, we are guaranteed to decrease f for sufficiently small λ .

What should the criterion for accepting a step be? It is *not* sufficient to require merely that $f(\mathbf{x}_{\text{new}}) < f(\mathbf{x}_{\text{old}})$. This criterion can fail to converge to a minimum of f in one of two ways. First, it is possible to construct a sequence of steps satisfying this criterion with f decreasing too slowly relative to the step lengths. Second, one can have a sequence where the step lengths are too small relative to the initial rate of decrease of f . (For examples of such sequences, see [1], p. 117.)

A simple way to fix the first problem is to require the *average* rate of decrease of f to be at least some fraction α of the *initial* rate of decrease $\nabla f \cdot \mathbf{p}$:

$$f(\mathbf{x}_{\text{new}}) \leq f(\mathbf{x}_{\text{old}}) + \alpha \nabla f \cdot (\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{old}}) \quad (9.7.7)$$

Here the parameter α satisfies $0 < \alpha < 1$. We can get away with quite small values of α ; $\alpha = 10^{-4}$ is a good choice.

The second problem can be fixed by requiring the rate of decrease of f at $\mathbf{x}_{\text{new}}$ to be greater than some fraction β of the rate of decrease of f at $\mathbf{x}_{\text{old}}$. In practice, we will not need to impose this second constraint because our backtracking algorithm will have a built-in cutoff to avoid taking steps that are too small.

Here is the strategy for a practical backtracking routine: Define

$$g(\lambda) \equiv f(\mathbf{x}_{\text{old}} + \lambda\mathbf{p}) \quad (9.7.8)$$

so that

$$g'(\lambda) = \nabla f \cdot \mathbf{p} \quad (9.7.9)$$

If we need to backtrack, then we model g with the most current information we have and choose λ to minimize the model. We start with $g(0)$ and $g'(0)$ available. The first step is always the Newton step, $\lambda = 1$. If this step is not acceptable, we have available $g(1)$ as well. We can therefore model $g(\lambda)$ as a quadratic:

$$g(\lambda) \approx [g(1) - g(0) - g'(0)]\lambda^2 + g'(0)\lambda + g(0) \quad (9.7.10)$$

Taking the derivative of this quadratic, we find that it is a minimum when

$$\lambda = -\frac{g'(0)}{2[g(1) - g(0) - g'(0)]} \quad (9.7.11)$$

Since the Newton step failed, we can show that $\lambda \lesssim \frac{1}{2}$ for small α . We need to guard against too small a value of λ , however. We set $\lambda_{\text{min}} = 0.1$.

On second and subsequent backtracks, we model g as a cubic in λ , using the previous value $g(\lambda_1)$ and the second most recent value $g(\lambda_2)$:

$$g(\lambda) = a\lambda^3 + b\lambda^2 + g'(0)\lambda + g(0) \quad (9.7.12)$$

Requiring this expression to give the correct values of g at λ_1 and λ_2 gives two equations that can be solved for the coefficients a and b :

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} 1/\lambda_1^2 & -1/\lambda_2^2 \\ -\lambda_2/\lambda_1^2 & \lambda_1/\lambda_2^2 \end{bmatrix} \cdot \begin{bmatrix} g(\lambda_1) - g'(0)\lambda_1 - g(0) \\ g(\lambda_2) - g'(0)\lambda_2 - g(0) \end{bmatrix} \quad (9.7.13)$$

The minimum of the cubic (9.7.12) is at

$$\lambda = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a} \quad (9.7.14)$$

We enforce that λ lie between $\lambda_{\max} = 0.5\lambda_1$ and $\lambda_{\min} = 0.1\lambda_1$.

The routine has two additional features, a minimum step length `alamin` and a maximum step length `stpmx`. `lnsrch` will also be used in the quasi-Newton minimization routine `dfpmin` in the next section.

```
SUBROUTINE lnsrch(n,xold,fold,g,p,x,f,stpmx,check,func)
INTEGER n
LOGICAL check
REAL f,fold,stpmx,g(n),p(n),x(n),xold(n),func,ALF,TOLX
PARAMETER (ALF=1.e-4,TOLX=1.e-7)
EXTERNAL func
```

C USES func

Given an n -dimensional point `xold(1:n)`, the value of the function and gradient there, `fold` and `g(1:n)`, and a direction `p(1:n)`, finds a new point `x(1:n)` along the direction `p` from `xold` where the function `func` has decreased "sufficiently." The new function value is returned in `f`. `stpmx` 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 i
REAL a,alam,alam2,alamin,b,disc,f2,rhs1,rhs2,slope,
* sum,temp,test,tmplam
check=.false.
```

```
sum=0.
```

```
do 11 i=1,n
```

```
sum=sum+p(i)*p(i)
```

```
enddo 11
```

```
sum=sqrt(sum)
```

```
if(sum.gt.stpmx)then
```

```
do 12 i=1,n
```

```
p(i)=p(i)*stpmx/sum
```

```
enddo 12
```

```
endif
```

```
slope=0.
```

```
do 13 i=1,n
```

```
slope=slope+g(i)*p(i)
```

```
enddo 13
```

```
if(slope.ge.0.) pause 'roundoff problem in lnsrch'
```

```
test=0.
```

```
do 14 i=1,n
```

```
temp=abs(p(i))/max(abs(xold(i)),1.)
```

```
if(temp.gt.test)test=temp
```

```
enddo 14
```

```
alamin=TOLX/test
```

```
alam=1.
```

```
1 continue
```

```
do 15 i=1,n
```

```
x(i)=xold(i)+alam*p(i)
```

```
enddo 15
```

Scale if attempted step is too big.

Compute $\lambda_{\min}$.

Always try full Newton step first.
Start of iteration loop.

```

f=func(x)
if(alam.lt.amin)then
    do 16 i=1,n
        x(i)=xold(i)
    enddo 16
    check=.true.
    return
else if(f.le.fold+ALF*alam*slope)then
    return
else
    if(alam.eq.1.)then
        tmlam=-slope/(2.*(f-fold-slope))
    else
        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.eq.0.)then
            tmlam=-slope/(2.*b)
        else
            disc=b*b-3.*a*slope
            if(disc.lt.0.)then
                tmlam=.5*alam
            else if(b.le.0.)then
                tmlam=(-b+sqrt(disc))/(3.*a)
            else
                tmlam=-slope/(b+sqrt(disc))
            endif
        endif
        if(tmlam.gt..5*alam)tmlam=.5*alam
    endif
endif
alam2=alam
f2=f
alam=max(tmlam,.1*alam)
goto 1
END

```

Convergence on Δx . For zero finding, the calling program should verify the convergence.

Sufficient function decrease.

Backtrack.

First time.

Subsequent backtracks.

$\lambda \leq 0.5\lambda_1$.

$\lambda \geq 0.1\lambda_1$. Try again.

Here now is the globally convergent Newton routine `newt` that uses `lnsrch`. A feature of `newt` is that you need not supply the Jacobian matrix analytically; the routine will attempt to compute the necessary partial derivatives of **F** by finite differences in the routine `fdjac`. This routine uses some of the techniques described in §5.7 for computing numerical derivatives. Of course, you can always replace `fdjac` with a routine that calculates the Jacobian analytically if this is easy for you to do.

```

SUBROUTINE newt(x,n,check)
INTEGER n,nn,NP,MAXITS
LOGICAL check
REAL x(n),fvec,TOLF,TOLMIN,TOLX,STPMX
PARAMETER (NP=40,MAXITS=200,TOLF=1.e-4,TOLMIN=1.e-6,TOLX=1.e-7,
    STPMX=100.)
COMMON /newtv/ fvec(NP),nn
SAVE /newtv/
C USES fdjac,fmin,lnsrch,lubksb,ludcmp

```

Given an initial guess $x(1:n)$ for a root in n dimensions, find the root by a globally convergent Newton's method. The vector of functions to be zeroed, called `fvec(1:n)` in the routine below, is returned by a user-supplied subroutine that *must* be called `funcv` and have the declaration `subroutine funcv(n,x,fvec)`. 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: `NP` is the maximum expected value of `n`; `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 i,its,j,indx(NP)
REAL d,den,f,fold,stpmx,sum,temp,test,fjac(NP,NP),
*   g(NP),p(NP),xold(NP),fmin
EXTERNAL fmin
nn=n
f=fmin(x)                                The vector fvec is also computed by this call.
test=0.                                  Test for initial guess being a root. Use more stringent test than simply TOLF.
do 11 i=1,n
    if(abs(fvec(i)).gt.test)test=abs(fvec(i))
enddo 11
if(test.lt..01*TOLF)then
    check=.false.
    return
endif
sum=0.                                  Calculate stpmx for line searches.
do 12 i=1,n
    sum=sum+x(i)**2
enddo 12
stpmx=STPMX*max(sqrt(sum),float(n))
do 21 its=1,MAXITS                        Start of iteration loop.
    call fdjac(n,x,fvec,NP,fjac)
    If analytic Jacobian is available, you can replace the routine fdjac below with your own routine.
    do 14 i=1,n                            Compute  $\nabla f$  for the line search.
        sum=0.
        do 13 j=1,n
            sum=sum+fjac(j,i)*fvec(j)
        enddo 13
        g(i)=sum
    enddo 14
    do 15 i=1,n                            Store x,
        xold(i)=x(i)
    enddo 15
    fold=f                                and f.
    do 16 i=1,n                            Right-hand side for linear equations.
        p(i)=-fvec(i)
    enddo 16
    call ludcmp(fjac,n,NP,indx,d)          Solve linear equations by LU decomposition.
    call lubksb(fjac,n,NP,indx,p)
    call lnsrch(n,xold,fold,g,p,x,f,stpmx,check,fmin)
    lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.
    test=0.                                Test for convergence on function values.
    do 17 i=1,n
        if(abs(fvec(i)).gt.test)test=abs(fvec(i))
    enddo 17
    if(test.lt.TOLF)then
        check=.false.
        return
    endif
    if(check)then                          Check for gradient of f zero, i.e., spurious convergence.
        test=0.
        den=max(f,.5*n)
        do 18 i=1,n
            temp=abs(g(i))*max(abs(x(i)),1.)/den
            if(temp.gt.test)test=temp
        enddo 18
        if(test.lt.TOLMIN)then
            check=.true.
        else
            check=.false.
        endif
    endif
    return

```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
 Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

endif
test=0.                                Test for convergence on  $\delta x$ .
do 19 i=1,n
    temp=(abs(x(i)-xold(i)))/max(abs(x(i)),1.)
    if(temp.gt.test)test=temp
enddo 19
if(test.lt.TOLX)return
enddo 21
pause 'MAXITS exceeded in newt'
END

```

```

SUBROUTINE fdjac(n,x,fvec,np,df)
INTEGER n,np,NMAX
REAL df(np,np),fvec(n),x(n),EPS
PARAMETER (NMAX=40,EPS=1.e-4)

```

C *USES funcv*

Computes forward-difference approximation to Jacobian. On input, $x(1:n)$ is the point at which the Jacobian is to be evaluated, $fvec(1:n)$ is the vector of function values at the point, and np is the physical dimension of the Jacobian array $df(1:n,1:n)$ which is output. subroutine $funcv(n,x,f)$ is a fixed-name, user-supplied routine that returns the vector of functions at x .

Parameters: $NMAX$ is the maximum value of n ; EPS is the approximate square root of the machine precision.

```

INTEGER i,j
REAL h,temp,f(NMAX)
do 12 j=1,n
    temp=x(j)
    h=EPS*abs(temp)
    if(h.eq.0.)h=EPS
    x(j)=temp+h                                Trick to reduce finite precision error.
    h=x(j)-temp
    call funcv(n,x,f)
    x(j)=temp
do 11 i=1,n
    df(i,j)=(f(i)-fvec(i))/h                    Forward difference formula.
enddo 11
enddo 12
return
END

```

```

FUNCTION fmin(x)
INTEGER n,NP
REAL fmin,x(*),fvec
PARAMETER (NP=40)
COMMON /newtv/ fvec(NP),n
SAVE /newtv/

```

C *USES funcv*

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

```

INTEGER i
REAL sum
call funcv(n,x,fvec)
sum=0.
do 11 i=1,n
    sum=sum+fvec(i)**2
enddo 11
fmin=0.5*sum
return
END

```

The routine `newt` assumes that typical values of all components of $\mathbf{x}$ and of $\mathbf{F}$ are of order unity, and it can fail if this assumption is badly violated. You should rescale the variables by their typical values before invoking `newt` if this problem occurs.

Multidimensional Secant Methods: Broyden's Method

Newton's method as implemented above is quite powerful, but it still has several disadvantages. One drawback is that the Jacobian matrix is needed. In many problems analytic derivatives are unavailable. If function evaluation is expensive, then the cost of finite-difference determination of the Jacobian can be prohibitive.

Just as the quasi-Newton methods to be discussed in §10.7 provide cheap approximations for the Hessian matrix in minimization algorithms, there are quasi-Newton methods that provide cheap approximations to the Jacobian for zero finding. These methods are often called *secant methods*, since they reduce to the secant method (§9.2) in one dimension (see, e.g., [1]). The best of these methods still seems to be the first one introduced, *Broyden's method* [2].

Let us denote the approximate Jacobian by $\mathbf{B}$. Then the i th quasi-Newton step $\delta \mathbf{x}_i$ is the solution of

$$\mathbf{B}_i \cdot \delta \mathbf{x}_i = -\mathbf{F}_i \quad (9.7.15)$$

where $\delta \mathbf{x}_i = \mathbf{x}_{i+1} - \mathbf{x}_i$ (cf. equation 9.7.3). The quasi-Newton or secant condition is that $\mathbf{B}_{i+1}$ satisfy

$$\mathbf{B}_{i+1} \cdot \delta \mathbf{x}_i = \delta \mathbf{F}_i \quad (9.7.16)$$

where $\delta \mathbf{F}_i = \mathbf{F}_{i+1} - \mathbf{F}_i$. This is the generalization of the one-dimensional secant approximation to the derivative, $\delta F / \delta x$. However, equation (9.7.16) does not determine $\mathbf{B}_{i+1}$ uniquely in more than one dimension.

Many different auxiliary conditions to pin down $\mathbf{B}_{i+1}$ have been explored, but the best-performing algorithm in practice results from Broyden's formula. This formula is based on the idea of getting $\mathbf{B}_{i+1}$ by making the least change to $\mathbf{B}_i$ consistent with the secant equation (9.7.16). Broyden showed that the resulting formula is

$$\mathbf{B}_{i+1} = \mathbf{B}_i + \frac{(\delta \mathbf{F}_i - \mathbf{B}_i \cdot \delta \mathbf{x}_i) \otimes \delta \mathbf{x}_i}{\delta \mathbf{x}_i \cdot \delta \mathbf{x}_i} \quad (9.7.17)$$

You can easily check that $\mathbf{B}_{i+1}$ satisfies (9.7.16).

Early implementations of Broyden's method used the Sherman-Morrison formula, equation (2.7.2), to invert equation (9.7.17) analytically,

$$\mathbf{B}_{i+1}^{-1} = \mathbf{B}_i^{-1} + \frac{(\delta \mathbf{x}_i - \mathbf{B}_i^{-1} \cdot \delta \mathbf{F}_i) \otimes \delta \mathbf{x}_i \cdot \mathbf{B}_i^{-1}}{\delta \mathbf{x}_i \cdot \mathbf{B}_i^{-1} \cdot \delta \mathbf{F}_i} \quad (9.7.18)$$

Then instead of solving equation (9.7.3) by e.g., *LU* decomposition, one determined

$$\delta \mathbf{x}_i = -\mathbf{B}_i^{-1} \cdot \mathbf{F}_i \quad (9.7.19)$$

by matrix multiplication in $O(N^2)$ operations. The disadvantage of this method is that it cannot easily be embedded in a globally convergent strategy, for which the gradient of equation (9.7.4) requires $\mathbf{B}$, not $\mathbf{B}^{-1}$,

$$\nabla(\tfrac{1}{2} \mathbf{F} \cdot \mathbf{F}) \simeq \mathbf{B}^T \cdot \mathbf{F} \quad (9.7.20)$$

Accordingly, we implement the update formula in the form (9.7.17).

However, we can still preserve the $O(N^2)$ solution of (9.7.3) by using *QR* decomposition (§2.10) instead of *LU* decomposition. The reason is that because of the special form of equation (9.7.17), the *QR* decomposition of $\mathbf{B}_i$ can be updated into the *QR* decomposition of $\mathbf{B}_{i+1}$ in $O(N^2)$ operations (§2.10). All we need is an initial approximation $\mathbf{B}_0$ to start the ball rolling. It is often acceptable to start simply with the identity matrix, and then allow $O(N)$ updates to produce a reasonable approximation to the Jacobian. We prefer to spend the first N function evaluations on a finite-difference approximation to initialize $\mathbf{B}$ via a call to `fdjac`.

Since $\mathbf{B}$ is not the exact Jacobian, we are not guaranteed that $\delta \mathbf{x}$ is a descent direction for $f = \tfrac{1}{2} \mathbf{F} \cdot \mathbf{F}$ (cf. equation 9.7.5). Thus the line search algorithm can fail to return a suitable step if $\mathbf{B}$ wanders far from the true Jacobian. In this case, we reinitialize $\mathbf{B}$ by another call to `fdjac`.

Like the secant method in one dimension, Broyden's method converges superlinearly once you get close enough to the root. Embedded in a global strategy, it is almost as robust

as Newton's method, and often needs far fewer function evaluations to determine a zero. Note that the final value of **B** is *not* always close to the true Jacobian at the root, even when the method converges.

The routine `broydn` given below is very similar to `newt` in organization. The principal differences are the use of *QR* decomposition instead of *LU*, and the updating formula instead of directly determining the Jacobian. The remarks at the end of `newt` about scaling the variables apply equally to `broydn`.

```

SUBROUTINE broydn(x,n,check)
  INTEGER n,nn,NP,MAXITS
  REAL x(n),fvec,EPS,TOLF,TOLMIN,TOLX,STPMX
  LOGICAL check
  PARAMETER (NP=40,MAXITS=200,EPS=1.e-7,TOLF=1.e-4,TOLMIN=1.e-6,
*           TOLX=EPS,STPMX=100.)
  COMMON /newtv/ fvec(NP),nn           Communicates with fmin.
  SAVE /newtv/
C  USES fdjac,fmin,lnsrch,qrdcmp,qrupdt,rsolv
  Given an initial guess x(1:n) for a root in n dimensions, find the root by Broyden's method
  embedded in a globally convergent strategy. The vector of functions to be zeroed, called
  fvec(1:n) in the routine below, is returned by a user-supplied subroutine that must be
  called funcv and have the declaration subroutine funcv(n,x,fvec). 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: NP is the maximum expected value of n; MAXITS is the maximum number of
  iterations; EPS is close to 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  $\delta x$ ; STPMX is the
  scaled maximum step length allowed in line searches.
  INTEGER i,its,j,k
  REAL den,f,fold,stpmax,sum,temp,test,c(NP),d(NP),fvcold(NP),
*       g(NP),p(NP),qt(NP,NP),r(NP,NP),s(NP),t(NP),w(NP),
*       xold(NP),fmin
  LOGICAL restrt,sing,skip
  EXTERNAL fmin
  nn=n
  f=fmin(x)           The vector fvec is also computed by this call.
  test=0.             Test for initial guess being a root. Use more strin-
                      gent test than simply TOLF.
  do 11 i=1,n
    if(abs(fvec(i)).gt.test)test=abs(fvec(i))
  enddo 11
  if(test.lt..01*TOLF)then
    check=.false.
    return
  endif
  sum=0.              Calculate stpmax for line searches.
  do 12 i=1,n
    sum=sum+x(i)**2
  enddo 12
  stpmax=STPMX*max(sqrt(sum),float(n))
  restrt=.true.       Ensure initial Jacobian gets computed.
  do 42 its=1,MAXITS   Start of iteration loop.
    if(restrt)then
      call fdjac(n,x,fvec,NP,r)      Initialize or reinitialize Jacobian in r.
      call qrdcmp(r,n,NP,c,d,sing)   QR decomposition of Jacobian.
      if(sing) pause 'singular Jacobian in broydn'
      do 14 i=1,n                     Form  $Q^T$  explicitly.
        do 13 j=1,n
          qt(i,j)=0.
        enddo 13
        qt(i,i)=1.
      enddo 14

```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
 Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

do 18 k=1,n-1
  if(c(k).ne.0.)then
    do 17 j=1,n
      sum=0.
      do 15 i=k,n
        sum=sum+r(i,k)*qt(i,j)
      enddo 15
      sum=sum/c(k)
      do 16 i=k,n
        qt(i,j)=qt(i,j)-sum*r(i,k)
      enddo 16
    enddo 17
  endif
enddo 18
do 21 i=1,n
  r(i,i)=d(i)
  do 19 j=1,i-1
    r(i,j)=0.
  enddo 19
enddo 21
else
  Carry out Broyden update.
  s =  $\delta x$ .
  do 22 i=1,n
    s(i)=x(i)-xold(i)
  enddo 22
  do 24 i=1,n
    sum=0.
    do 23 j=i,n
      sum=sum+r(i,j)*s(j)
    enddo 23
    t(i)=sum
  enddo 24
  skip=.true.
  do 26 i=1,n
    sum=0.
    do 25 j=1,n
      sum=sum+qt(j,i)*t(j)
    enddo 25
    w(i)=fvec(i)-fvcold(i)-sum
    if(abs(w(i)).ge.EPS*(abs(fvec(i))+abs(fvcold(i))))then
      Don't update with noisy components of w.
      skip=.false.
    else
      w(i)=0.
    endif
  enddo 26
  if(.not.skip)then
    do 28 i=1,n
      sum=0.
      do 27 j=1,n
        sum=sum+qt(i,j)*w(j)
      enddo 27
      t(i)=sum
    enddo 28
    den=0.
    do 29 i=1,n
      den=den+s(i)**2
    enddo 29
    do 31 i=1,n
      s(i)=s(i)/den
    enddo 31
    call qrupdt(r,qt,n,NP,t,s) Update  $\mathbf{R}$  and  $\mathbf{Q}^T$ .
    do 32 i=1,n
      if(r(i,i).eq.0.) pause 'r singular in broydn'
      d(i)=r(i,i)
    enddo 32
    Diagonal of  $\mathbf{R}$  stored in d.
  endif
endif

```

Form $\mathbf{R}$ explicitly.

Carry out Broyden update.

 $s = \delta x$. $t = \mathbf{R} \cdot s$. $w = \delta F - \mathbf{B} \cdot s$. $t = \mathbf{Q}^T \cdot w$.Store $s/(s \cdot s)$ in s .Update $\mathbf{R}$ and $\mathbf{Q}^T$.Diagonal of $\mathbf{R}$ stored in d.

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
 Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```

        enddo 32
    endif
endif
do 34 i=1,n
    sum=0.
    do 33 j=1,n
        sum=sum+qt(i,j)*fvec(j)
    enddo 33
    p(i)=-sum
enddo 34
do 36 i=n,1,-1
    sum=0.
    do 35 j=1,i
        sum=sum-r(j,i)*p(j)
    enddo 35
    g(i)=sum
enddo 36
do 37 i=1,n
    xold(i)=x(i)
    fvcold(i)=fvec(i)
enddo 37
fold=f
call rsolv(r,n,NP,d,p)
call lnsrch(n,xold,fold,g,p,x,f,stpmax,check,fmin)
test=0.
do 38 i=1,n
    if(abs(fvec(i)).gt.test)test=abs(fvec(i))
enddo 38
if(test.lt.TOLF)then
    check=.false.
    return
endif
if(check)then
    if(restrt)then
        return
    else
        test=0.
        den=max(f,.5*n)
        do 39 i=1,n
            temp=abs(g(i))*max(abs(x(i)),1.)/den
            if(temp.gt.test)test=temp
        enddo 39
        if(test.lt.TOLMIN)then
            return
        else
            restrt=.true.
        endif
    endif
endif
else
    restrt=.false.
    test=0.
    do 41 i=1,n
        temp=(abs(x(i)-xold(i)))/max(abs(x(i)),1.)
        if(temp.gt.test)test=temp
    enddo 41
    if(test.lt.TOLX)return
endif
enddo 42
pause 'MAXITS exceeded in broydn'
END

```

Right-hand side for linear equations is $-Q^T \cdot F$.

Compute $\nabla f \approx (Q \cdot R)^T \cdot F$ for the line search.

Store x and F .

Store f .

Solve linear equations.

lnsrch returns new x and f . It also calculates $fvec$ at the new x when it calls $fmin$.

Test for convergence on function values.

True if line search failed to find a new x .

Failure; already tried reinitializing the Jacobian.

Check for gradient of f zero, i.e., spurious convergence.

Try reinitializing the Jacobian.

Successful step; will use Broyden update for next step.

Test for convergence on δx .

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
 Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

More Advanced Implementations

One of the principal ways that the methods described so far can fail is if $\mathbf{J}$ (in Newton's method) or $\mathbf{B}$ in (Broyden's method) becomes singular or nearly singular, so that $\delta\mathbf{x}$ cannot be determined. If you are lucky, this situation will not occur very often in practice. Methods developed so far to deal with this problem involve monitoring the condition number of $\mathbf{J}$ and perturbing $\mathbf{J}$ if singularity or near singularity is detected. This is most easily implemented if the QR decomposition is used instead of LU in Newton's method (see [1] for details). Our personal experience is that, while such an algorithm can solve problems where $\mathbf{J}$ is exactly singular and the standard Newton's method fails, it is occasionally less robust on other problems where LU decomposition succeeds. Clearly implementation details involving roundoff, underflow, etc., are important here and the last word is yet to be written.

Our global strategies both for minimization and zero finding have been based on line searches. Other global algorithms, such as the *hook step* and *dogleg step* methods, are based instead on the *model-trust region* approach, which is related to the Levenberg-Marquardt algorithm for nonlinear least-squares (§15.5). While somewhat more complicated than line searches, these methods have a reputation for robustness even when starting far from the desired zero or minimum [1].

CITED REFERENCES AND FURTHER READING:

- Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]
 Broyden, C.G. 1965, *Mathematics of Computation*, vol. 19, pp. 577–593. [2]

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
 Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).