such methods can still occasionally fail by coming to rest on a local minimum of F, they often succeed where a direct attack via Newton's method alone fails. The next section deals with these methods.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 14. [1]

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press).

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press).

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

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

is

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta\mathbf{x} \tag{9.7.2}$$

where

$$\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F} \tag{9.7.3}$$

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

$$f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F} \tag{9.7.4}$$

(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:

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

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.

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:

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

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 $\nabla f \cdot \mathbf{p}$:

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

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

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

so that

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

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:

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

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

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

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 $\lambda_{\min} = 0.1$.

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 $g(\lambda_2)$:

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

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:

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

The minimum of the cubic (9.7.12) is at

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

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 `stpmax`. `lnsrch` will also be used in the quasi-Newton minimization routine `dfpmin` in the next section.

```
#include <math.h>
#include "nrutil.h"
#define ALF 1.0e-4                        Ensures sufficient decrease in function value.
#define TOLX 1.0e-7                       Convergence criterion on Δx.

void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
    float *f, float stpmax, int *check, float (*func)(float []))
```
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`. `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 (0) on a normal exit. It is true (1) 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. Some “difficult” problems may require double precision in this routine.
```
{
    int i;
    float a,alam,alam2,alamin,b,disc,f2,rhs1,rhs2,slope,sum,temp,
        test,tmplam;

    *check=0;
    for (sum=0.0,i=1;i<=n;i++) sum += p[i]*p[i];
    sum=sqrt(sum);
    if (sum > stpmax)
        for (i=1;i<=n;i++) p[i] *= stpmax/sum;    Scale if attempted step is too big.
    for (slope=0.0,i=1;i<=n;i++)
        slope += g[i]*p[i];
    if (slope >= 0.0) nrerror("Roundoff problem in lnsrch.");
    test=0.0;                                     Compute λmin.
    for (i=1;i<=n;i++) {
```

```
        temp=fabs(p[i])/FMAX(fabs(xold[i]),1.0);
        if (temp > test) test=temp;
    }
    alamin=TOLX/test;
    alam=1.0;                                   Always try full Newton step first.
    for (;;) {                                  Start of iteration loop.
        for (i=1;i<=n;i++) x[i]=xold[i]+alam*p[i];
        *f=(*func)(x);
        if (alam < alamin) {                    Convergence on Δx. For zero find-
            for (i=1;i<=n;i++) x[i]=xold[i];       ing, the calling program should
            *check=1;                              verify the convergence.
            return;
        } else if (*f <= fold+ALF*alam*slope) return;      Sufficient function decrease.
        else {                                              Backtrack.
            if (alam == 1.0)
                tmplam = -slope/(2.0*(*f-fold-slope));      First time.
            else {                                          Subsequent backtracks.
                rhs1 = *f-fold-alam*slope;
                rhs2=f2-fold-alam2*slope;
                a=(rhs1/(alam*alam)-rhs2/(alam2*alam2))/(alam-alam2);
                b=(-alam2*rhs1/(alam*alam)+alam*rhs2/(alam2*alam2))/(alam-alam2);
                if (a == 0.0) tmplam = -slope/(2.0*b);
                else {
                    disc=b*b-3.0*a*slope;
                    if (disc < 0.0) tmplam=0.5*alam;
                    else if (b <= 0.0) tmplam=(-b+sqrt(disc))/(3.0*a);
                    else tmplam=-slope/(b+sqrt(disc));
                }
                if (tmplam > 0.5*alam)
                    tmplam=0.5*alam;                λ ≤ 0.5λ1.
            }
        }
        alam2=alam;
        f2 = *f;
        alam=FMAX(tmplam,0.1*alam);                 λ ≥ 0.1λ1.
    }                                               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.

```
#include <math.h>
#include "nrutil.h"
#define MAXITS 200
#define TOLF 1.0e-4
#define TOLMIN 1.0e-6
#define TOLX 1.0e-7
#define STPMX 100.0
```

Here `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 $\delta\mathbf{x}$; `STPMX` is the scaled maximum step length allowed in line searches.

```
int nn;                                         Global variables to communicate with fmin.
float *fvec;
void (*nrfuncv)(int n, float v[], float f[]);
#define FREERETURN {free_vector(fvec,1,n);free_vector(xold,1,n);\
    free_vector(p,1,n);free_vector(g,1,n);free_matrix(fjac,1,n,1,n);\
    free_ivector(indx,1,n);return;}
```

```
void newt(float x[], int n, int *check,
    void (*vecfunc)(int, float [], float []))
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 the user-supplied routine vecfunc(n,x,fvec). The output quantity
check is false (0) on a normal return and true (1) 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.
{
    void fdjac(int n, float x[], float fvec[], float **df,
        void (*vecfunc)(int, float [], float []));
    float fmin(float x[]);
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
         float *f, float stpmax, int *check, float (*func)(float []));
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,its,j,*indx;
    float d,den,f,fold,stpmax,sum,temp,test,**fjac,*g,*p,*xold;

    indx=ivector(1,n);
    fjac=matrix(1,n,1,n);
    g=vector(1,n);
    p=vector(1,n);
    xold=vector(1,n);
    fvec=vector(1,n);                       Define global variables.
    nn=n;
    nrfuncv=vecfunc;
    f=fmin(x);                              fvec is also computed by this call.
    test=0.0;                               Test for initial guess being a root. Use
    for (i=1;i<=n;i++)                          more stringent test than simply TOLF.
        if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
    if (test < 0.01*TOLF) {
        *check=0;
        FREERETURN
    }
    for (sum=0.0,i=1;i<=n;i++) sum += SQR(x[i]);      Calculate stpmax for line searches.
    stpmax=STPMX*FMAX(sqrt(sum),(float)n);
    for (its=1;its<=MAXITS;its++) {         Start of iteration loop.
        fdjac(n,x,fvec,fjac,vecfunc);
        If analytic Jacobian is available, you can replace the routine fdjac below with your
        own routine.
        for (i=1;i<=n;i++) {                Compute ∇f for the line search.
            for (sum=0.0,j=1;j<=n;j++) sum += fjac[j][i]*fvec[j];
            g[i]=sum;
        }
        for (i=1;i<=n;i++) xold[i]=x[i];    Store x,
        fold=f;                             and f.
        for (i=1;i<=n;i++) p[i] = -fvec[i]; Right-hand side for linear equations.
        ludcmp(fjac,n,indx,&d);             Solve linear equations by LU decompo-
        lubksb(fjac,n,indx,p);                  sition.
        lnsrch(n,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.
        test=0.0;                           Test for convergence on function val-
        for (i=1;i<=n;i++)                      ues.
            if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
        if (test < TOLF) {
            *check=0;
            FREERETURN
        }
        if (*check) {                       Check for gradient of f zero, i.e., spuri-
            test=0.0;                           ous convergence.
            den=FMAX(f,0.5*n);
            for (i=1;i<=n;i++) {
                temp=fabs(g[i])*FMAX(fabs(x[i]),1.0)/den;
```

```
                if (temp > test) test=temp;
            }
            *check=(test < TOLMIN ? 1 : 0);
            FREERETURN
        }
        test=0.0;                                   Test for convergence on δx.
        for (i=1;i<=n;i++) {
            temp=(fabs(x[i]-xold[i]))/FMAX(fabs(x[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX) FREERETURN
    }
    nrerror("MAXITS exceeded in newt");
}
```

```
#include <math.h>
#include "nrutil.h"
#define EPS 1.0e-4                  Approximate square root of the machine precision.

void fdjac(int n, float x[], float fvec[], float **df,
    void (*vecfunc)(int, float [], float []))
```

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 `vecfunc(n,x,f)` is a user-supplied routine that returns the vector of functions at `x`. On output, `df[1..n][1..n]` is the Jacobian array.

```
{
    int i,j;
    float h,temp,*f;

    f=vector(1,n);
    for (j=1;j<=n;j++) {
        temp=x[j];
        h=EPS*fabs(temp);
        if (h == 0.0) h=EPS;
        x[j]=temp+h;                Trick to reduce finite precision error.
        h=x[j]-temp;
        (*vecfunc)(n,x,f);
        x[j]=temp;
        for (i=1;i<=n;i++) df[i][j]=(f[i]-fvec[i])/h;       Forward difference for-
    }                                                            mula.
    free_vector(f,1,n);
}
```

```
#include "nrutil.h"

extern int nn;
extern float *fvec;
extern void (*nrfuncv)(int n, float v[], float f[]);

float fmin(float x[])
```

Returns $f = \frac{1}{2}\mathbf{F}\cdot\mathbf{F}$ at `x`. The global pointer `*nrfuncv` points to a routine that returns the vector of functions at `x`. It is set to point to a user-supplied routine in the calling program. Global variables also communicate the function values back to the calling program.

```
{
    int i;
    float sum;

    (*nrfuncv)(nn,x,fvec);
    for (sum=0.0,i=1;i<=nn;i++) sum += SQR(fvec[i]);
    return 0.5*sum;
}
```

The routine `newt` assumes that typical values of all components of **x** and of **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 **B**. Then the ith quasi-Newton step $\delta\mathbf{x}_i$ is the solution of

$$\mathbf{B}_i \cdot \delta\mathbf{x}_i = -\mathbf{F}_i \tag{9.7.15}$$

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

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

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

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

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,

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

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

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

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 **B**, not $\mathbf{B}^{-1}$,

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

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 **B** via a call to `fdjac`.

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

```
#include <math.h>
#include "nrutil.h"
#define MAXITS 200
#define EPS 1.0e-7
#define TOLF 1.0e-4
#define TOLX EPS
#define STPMX 100.0
#define TOLMIN 1.0e-6
```
Here `MAXITS` is the maximum number of iterations; `EPS` is a number close to the machine precision; `TOLF` is the convergence criterion on function values; `TOLX` is the convergence criterion on $\delta\mathbf{x}$; `STPMX` is the scaled maximum step length allowed in line searches; `TOLMIN` is used to decide whether spurious convergence to a minimum of `fmin` has occurred.
```
#define FREERETURN {free_vector(fvec,1,n);free_vector(xold,1,n);\
    free_vector(w,1,n);free_vector(t,1,n);free_vector(s,1,n);\
    free_matrix(r,1,n,1,n);free_matrix(qt,1,n,1,n);free_vector(p,1,n);\
    free_vector(g,1,n);free_vector(fvcold,1,n);free_vector(d,1,n);\
    free_vector(c,1,n);return;}

int nn;                                  Global variables to communicate with fmin.
float *fvec;
void (*nrfuncv)(int n, float v[], float f[]);

void broydn(float x[], int n, int *check,
    void (*vecfunc)(int, float [], float []))
```
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 the user-supplied routine `vecfunc(n,x,fvec)`. The routine `fdjac` and the function `fmin` from `newt` are used. The output quantity `check` is false (0) on a normal return and true (1) 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.
```
{
    void fdjac(int n, float x[], float fvec[], float **df,
        void (*vecfunc)(int, float [], float []));
    float fmin(float x[]);
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
         float *f, float stpmax, int *check, float (*func)(float []));
    void qrdcmp(float **a, int n, float *c, float *d, int *sing);
    void qrupdt(float **r, float **qt, int n, float u[], float v[]);
    void rsolv(float **a, int n, float d[], float b[]);
    int i,its,j,k,restrt,sing,skip;
    float den,f,fold,stpmax,sum,temp,test,*c,*d,*fvcold;
    float *g,*p,**qt,**r,*s,*t,*w,*xold;

    c=vector(1,n);
    d=vector(1,n);
    fvcold=vector(1,n);
    g=vector(1,n);
    p=vector(1,n);
```

```
qt=matrix(1,n,1,n);
r=matrix(1,n,1,n);
s=vector(1,n);
t=vector(1,n);
w=vector(1,n);
xold=vector(1,n);
fvec=vector(1,n);                       Define global variables.
nn=n;
nrfuncv=vecfunc;
f=fmin(x);                              The vector fvec is also computed by this
test=0.0;                                   call.
for (i=1;i<=n;i++)                      Test for initial guess being a root. Use more
    if (fabs(fvec[i]) > test)test=fabs(fvec[i]);        stringent test than sim-
if (test < 0.01*TOLF) {                                 ply TOLF.
    *check=0;
    FREERETURN
}
for (sum=0.0,i=1;i<=n;i++) sum += SQR(x[i]);      Calculate stpmax for line searches.
stpmax=STPMX*FMAX(sqrt(sum),(float)n);
restrt=1;                               Ensure initial Jacobian gets computed.
for (its=1;its<=MAXITS;its++) {         Start of iteration loop.
    if (restrt) {
        fdjac(n,x,fvec,r,vecfunc);      Initialize or reinitialize Jacobian in r.
        qrdcmp(r,n,c,d,&sing);          QR decomposition of Jacobian.
        if (sing) nrerror("singular Jacobian in broydn");
        for (i=1;i<=n;i++) {                      Form Q^T explicitly.
            for (j=1;j<=n;j++) qt[i][j]=0.0;
            qt[i][i]=1.0;
        }
        for (k=1;k<n;k++) {
            if (c[k]) {
                for (j=1;j<=n;j++) {
                    sum=0.0;
                    for (i=k;i<=n;i++)
                        sum += r[i][k]*qt[i][j];
                    sum /= c[k];
                    for (i=k;i<=n;i++)
                        qt[i][j] -= sum*r[i][k];
                }
            }
        }
        for (i=1;i<=n;i++) {                      Form R explicitly.
            r[i][i]=d[i];
            for (j=1;j<i;j++) r[i][j]=0.0;
        }
    } else {                                      Carry out Broyden update.
        for (i=1;i<=n;i++) s[i]=x[i]-xold[i];     s = δx.
        for (i=1;i<=n;i++) {                      t = R · s.
            for (sum=0.0,j=i;j<=n;j++) sum += r[i][j]*s[j];
            t[i]=sum;
        }
        skip=1;
        for (i=1;i<=n;i++) {                      w = δF − B · s.
            for (sum=0.0,j=1;j<=n;j++) sum += qt[j][i]*t[j];
            w[i]=fvec[i]-fvcold[i]-sum;
            if (fabs(w[i]) >= EPS*(fabs(fvec[i])+fabs(fvcold[i]))) skip=0;
            Don't update with noisy components of w.
            else w[i]=0.0;
        }
        if (!skip) {
            for (i=1;i<=n;i++) {                  t = Q^T · w.
                for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*w[j];
                t[i]=sum;
            }
```

```
            for (den=0.0,i=1;i<=n;i++) den += SQR(s[i]);
            for (i=1;i<=n;i++) s[i] /= den;        Store s/(s·s) in s.
            qrupdt(r,qt,n,t,s);                    Update R and Q^T.
            for (i=1;i<=n;i++) {
                if (r[i][i] == 0.0) nrerror("r singular in broydn");
                d[i]=r[i][i];                      Diagonal of R stored in d.
            }
        }
    }
    for (i=1;i<=n;i++) {              Compute ∇f ≈ (Q·R)^T·F for the line search.
        for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*fvec[j];
        g[i]=sum;
    }
    for (i=n;i>=1;i--) {
        for (sum=0.0,j=1;j<=i;j++) sum += r[j][i]*g[j];
        g[i]=sum;
    }
    for (i=1;i<=n;i++) {              Store x and F.
        xold[i]=x[i];
        fvcold[i]=fvec[i];
    }
    fold=f;                           Store f.
    for (i=1;i<=n;i++) {              Right-hand side for linear equations is -Q^T·F.
        for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*fvec[j];
        p[i] = -sum;
    }
    rsolv(r,n,d,p);                      Solve linear equations.
    lnsrch(n,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.
    test=0.0;                            Test for convergence on function values.
    for (i=1;i<=n;i++)
        if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
    if (test < TOLF) {
        *check=0;
        FREERETURN
    }
    if (*check) {                        True if line search failed to find a new x.
        if (restrt) FREERETURN           Failure; already tried reinitializing the Jaco-
        else {                               bian.
            test=0.0;                    Check for gradient of f zero, i.e., spurious
            den=FMAX(f,0.5*n);               convergence.
            for (i=1;i<=n;i++) {
                temp=fabs(g[i])*FMAX(fabs(x[i]),1.0)/den;
                if (temp > test) test=temp;
            }
            if (test < TOLMIN) FREERETURN
            else restrt=1;               Try reinitializing the Jacobian.
        }
    } else {                             Successful step; will use Broyden update for
        restrt=0;                            next step.
        test=0.0;                        Test for convergence on δx.
        for (i=1;i<=n;i++) {
            temp=(fabs(x[i]-xold[i]))/FMAX(fabs(x[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX) FREERETURN
    }
  }
  nrerror("MAXITS exceeded in broydn");
  FREERETURN
}
```

More Advanced Implementations

One of the principal ways that the methods described so far can fail is if **J** (in Newton's method) or **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 **J** and perturbing **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 **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]