

Chapter 9. Root Finding and Nonlinear Sets of Equations

9.0 Introduction

We now consider that most basic of tasks, solving equations numerically. While most equations are born with both a right-hand side and a left-hand side, one traditionally moves all terms to the left, leaving

$$f(x) = 0 \tag{9.0.1}$$

whose solution or solutions are desired. When there is only one independent variable, the problem is *one-dimensional*, namely to find the root or roots of a function.

With more than one independent variable, more than one equation can be satisfied simultaneously. You likely once learned the *implicit function theorem* which (in this context) gives us the hope of satisfying N equations in N unknowns simultaneously. Note that we have only hope, not certainty. A nonlinear set of equations may have no (real) solutions at all. Contrariwise, it may have more than one solution. The implicit function theorem tells us that “generically” the solutions will be distinct, pointlike, and separated from each other. If, however, life is so unkind as to present you with a nongeneric, i.e., degenerate, case, then you can get a continuous family of solutions. In vector notation, we want to find one or more N -dimensional solution vectors \mathbf{x} such that

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{9.0.2}$$

where \mathbf{f} is the N -dimensional vector-valued function whose components are the individual equations to be satisfied simultaneously.

Don’t be fooled by the apparent notational similarity of equations (9.0.2) and (9.0.1). Simultaneous solution of equations in N dimensions is *much* more difficult than finding roots in the one-dimensional case. The principal difference between one and many dimensions is that, in one dimension, it is possible to bracket or “trap” a root between bracketing values, and then hunt it down like a rabbit. In multidimensions, you can never be sure that the root is there at all until you have found it.

Except in linear problems, root finding invariably proceeds by iteration, and this is equally true in one or in many dimensions. Starting from some approximate trial solution, a useful algorithm will improve the solution until some predetermined convergence criterion is satisfied. For smoothly varying functions, good algorithms

will always converge, *provided* that the initial guess is good enough. Indeed one can even determine in advance the rate of convergence of most algorithms.

It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hamming's motto, "the purpose of computing is insight, not numbers," is particularly apt in the area of finding roots. You should repeat this motto aloud whenever your program converges, with ten-digit accuracy, to the wrong root of a problem, or whenever it fails to converge because there is actually *no* root, or because there is a root but your initial estimate was not sufficiently close to it.

"This talk of insight is all very well, but what do I actually do?" For one-dimensional root finding, it is possible to give some straightforward answers: You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next, you should always bracket a root, that is, know that the function changes sign in an identified interval, before trying to converge to the root's value.

Finally (this is advice with which some daring souls might disagree, but we give it nonetheless) never let your iteration method get outside of the best bracketing bounds obtained at any stage. We will see below that some pedagogically important algorithms, such as *secant method* or *Newton-Raphson*, can violate this last constraint, and are thus not recommended unless certain fixups are implemented.

Multiple roots, or very close roots, are a real problem, especially if the multiplicity is an even number. In that case, there may be no readily apparent sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-liners: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 10 to determine whether a tantalizing dip in the function really does cross zero or not. (You can easily modify the simple golden section routine of §10.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a *double* root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent's algorithm in §9.3 is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function's derivative. Ridders' method (§9.2) is concise, and a close competitor.
- When you can compute the function's derivative, the routine `rtsafe` in §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.
- Roots of polynomials are a special case. Laguerre's method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned!
- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works *very* well if you can supply a

good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Avoiding implementations for specific computers, this book must generally steer clear of interactive or graphics-related routines. We make an exception right now. The following routine, which produces a crude function plot with interactively scaled axes, can save you a lot of grief as you enter the world of root finding.

```
#include <stdio.h>
#define ISCR 60          Number of horizontal and vertical positions in display.
#define JSCR 21
#define BLANK ' '
#define ZERO '-'
#define YY 'l'
#define XX '-'
#define FF 'x'

void scrsho(float (*fx)(float))
For interactive CRT terminal use. Produce a crude graph of the function fx over the prompted-
for interval x1,x2. Query for another plot until the user signals satisfaction.
{
    int jz,j,i;
    float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
    char scr[ISCR+1][JSCR+1];

    for (;;) {
        printf("\nEnter x1 x2 (x1=x2 to stop):\n");      Query for another plot, quit
        scanf("%f %f",&x1,&x2);                          if x1=x2.
        if (x1 == x2) break;
        for (j=1;j<=JSCR;j++)                            Fill vertical sides with character 'l'.
            scr[1][j]=scr[ISCR][j]=YY;
        for (i=2;i<=(ISCR-1);i++) {
            scr[i][1]=scr[i][JSCR]=XX;                    Fill top, bottom with character '-'.
            for (j=2;j<=(JSCR-1);j++)                    Fill interior with blanks.
                scr[i][j]=BLANK;
        }
        dx=(x2-x1)/(ISCR-1);
        x=x1;
        ysml=ybig=0.0;                                     Limits will include 0.
        for (i=1;i<=ISCR;i++) {                           Evaluate the function at equal intervals.
            y[i]=(*fx)(x);                                  Find the largest and smallest val-
            if (y[i] < ysml) ysml=y[i];                     ues.
            if (y[i] > ybig) ybig=y[i];
            x += dx;
        }
        if (ybig == ysml) ybig=ysml+1.0;                   Be sure to separate top and bottom.
        dyj=(JSCR-1)/(ybig-ysml);
        jz=1-(int) (ysml*dyj);
        for (i=1;i<=ISCR;i++) {
            scr[i][jz]=ZERO;
            j=1+(int) ((y[i]-ysml)*dyj);
            scr[i][j]=FF;
        }
        printf(" %10.3f ",ybig);
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][JSCR]);
        printf("\n");
        for (j=(JSCR-1);j>=2;j--) {                        Display.
            printf("%12s"," ");
            for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
            printf("\n");
        }
        printf(" %10.3f ",ysml);
    }
}
```

```

    for (i=1;i<=ISCR;i++) printf("%c",scr[i][1]);
    printf("\n");
    printf("%8s %10.3f %44s %10.3f\n", " ",x1, " ",x2);
}
}

```

CITED REFERENCES AND FURTHER READING:

- Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 5.
- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapters 2, 7, and 14.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 8.
- Householder, A.S. 1970, *The Numerical Treatment of a Single Nonlinear Equation* (New York: McGraw-Hill).

9.1 Bracketing and Bisection

We will say that a root is *bracketed* in the interval (a, b) if $f(a)$ and $f(b)$ have opposite signs. If the function is continuous, then at least one root must lie in that interval (the *intermediate value theorem*). If the function is discontinuous, but bounded, then instead of a root there might be a step discontinuity which crosses zero (see Figure 9.1.1). For numerical purposes, that might as well be a root, since the behavior is indistinguishable from the case of a continuous function whose zero crossing occurs in between two “adjacent” floating-point numbers in a machine’s finite-precision representation. Only for functions with singularities is there the possibility that a bracketed root is not really there, as for example

$$f(x) = \frac{1}{x - c} \quad (9.1.1)$$

Some root-finding algorithms (e.g., bisection in this section) will readily converge to c in (9.1.1). Luckily there is not much possibility of your mistaking c , or any number x close to it, for a root, since mere evaluation of $|f(x)|$ will give a very large, rather than a very small, result.

If you are given a function in a black box, there is no sure way of bracketing its roots, or of even determining that it has roots. If you like pathological examples, think about the problem of locating the two real roots of equation (3.0.1), which dips below zero only in the ridiculously small interval of about $x = \pi \pm 10^{-667}$.

In the next chapter we will deal with the related problem of bracketing a function’s minimum. There it is possible to give a procedure that always succeeds; in essence, “Go downhill, taking steps of increasing size, until your function starts back uphill.” There is no analogous procedure for roots. The procedure “go downhill until your function changes sign,” can be foiled by a function that has a simple extremum. Nevertheless, if you are prepared to deal with a “failure” outcome, this procedure is often a good first start; success is usual if your function has opposite signs in the limit $x \rightarrow \pm\infty$.

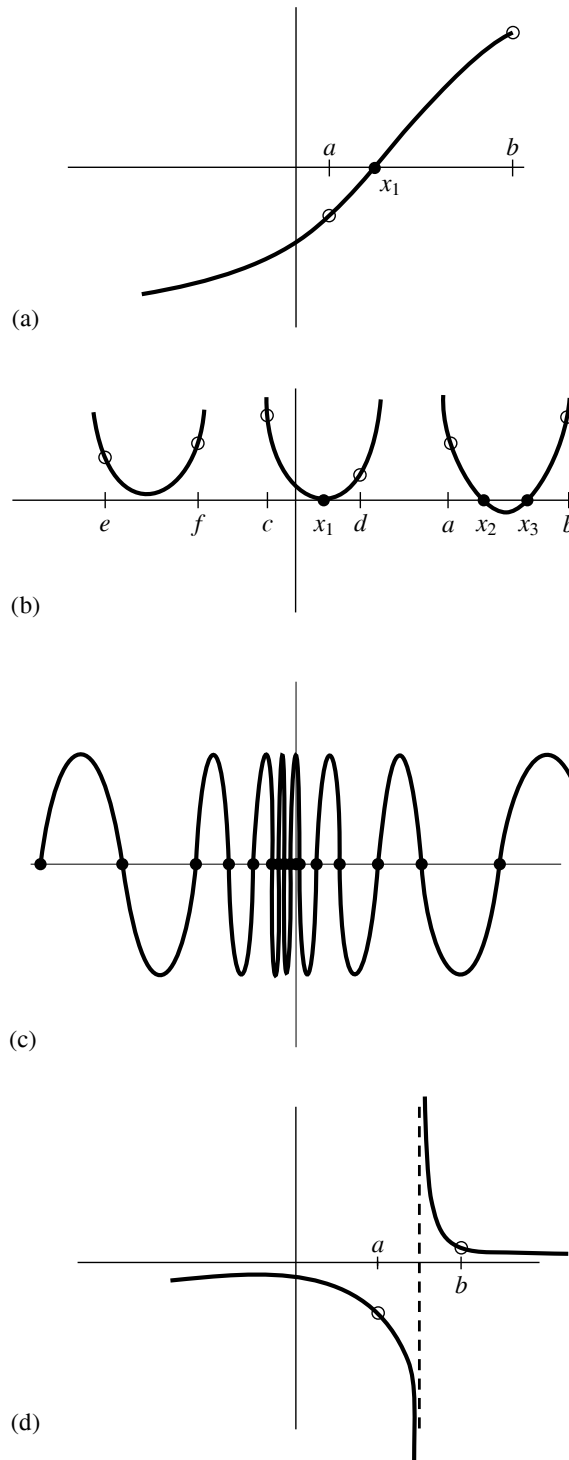


Figure 9.1.1. Some situations encountered while root finding: (a) shows an isolated root x_1 bracketed by two points a and b at which the function has opposite signs; (b) illustrates that there is not necessarily a sign change in the function near a double root (in fact, there is not necessarily a root!); (c) is a pathological function with many roots; in (d) the function has opposite signs at points a and b , but the points bracket a singularity, not a root.

```

#include <math.h>
#define FACTOR 1.6
#define NTRY 50

int zbrac(float (*func)(float), float *x1, float *x2)
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 zbrac
returns 1) or until the range becomes unacceptably large (in which case zbrac returns 0).
{
    void nrerror(char error_text[]);
    int j;
    float f1,f2;

    if (*x1 == *x2) nrerror("Bad initial range in zbrac");
    f1=(*func)(*x1);
    f2=(*func)(*x2);
    for (j=1;j<=NTRY;j++) {
        if (f1*f2 < 0.0) return 1;
        if (fabs(f1) < fabs(f2))
            f1=(*func)(*x1 += FACTOR*(x1-x2));
        else
            f2=(*func)(*x2 += FACTOR*(x2-x1));
    }
    return 0;
}

```

Alternatively, you might want to “look inward” on an initial interval, rather than “look outward” from it, asking if there are any roots of the function $f(x)$ in the interval from x_1 to x_2 when a search is carried out by subdivision into n equal intervals. The following function calculates brackets for up to nb distinct intervals which each contain one or more roots.

```

void zbrak(float (*fx)(float), float x1, float x2, int n, float xb1[],
    float xb2[], int *nb)
Given a function fx 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 input as the maximum num-
ber of roots sought, and is reset to the number of bracketing pairs xb1[1..nb], xb2[1..nb]
that are found.
{
    int nbb,i;
    float x,fp,fc,dx;

    nbb=0;
    dx=(x2-x1)/n;
    fp=(*fx)(x=x1);
    for (i=1;i<=n;i++) {
        fc=(*fx)(x += dx);
        if (fc*fp <= 0.0) {
            xb1[++nbb]=x-dx;
            xb2[nbb]=x;
            if (*nb == nbb) return;
        }
        fp=fc;
    }
    *nb = nbb;
}

```

Determine the spacing appropriate to the mesh.

Loop over all intervals

If a sign change occurs then record values for the bounds.

Bisection Method

Once we know that an interval contains a root, several classical procedures are available to refine it. These proceed with varying degrees of speed and sureness towards the answer. Unfortunately, the methods that are guaranteed to converge plod along most slowly, while those that rush to the solution in the best cases can also dash rapidly to infinity without warning if measures are not taken to avoid such behavior.

The *bisection method* is one that cannot fail. It is thus not to be sneered at as a method for otherwise badly behaved problems. The idea is simple. Over some interval the function is known to pass through zero because it changes sign. Evaluate the function at the interval's midpoint and examine its sign. Use the midpoint to replace whichever limit has the same sign. After each iteration the bounds containing the root decrease by a factor of two. If after n iterations the root is known to be within an interval of size ϵ_n , then after the next iteration it will be bracketed within an interval of size

$$\epsilon_{n+1} = \epsilon_n/2 \quad (9.1.2)$$

neither more nor less. Thus, we know in advance the number of iterations required to achieve a given tolerance in the solution,

$$n = \log_2 \frac{\epsilon_0}{\epsilon} \quad (9.1.3)$$

where ϵ_0 is the size of the initially bracketing interval, ϵ is the desired ending tolerance.

Bisection *must* succeed. If the interval happens to contain two or more roots, bisection will find one of them. If the interval contains no roots and merely straddles a singularity, it will converge on the singularity.

When a method converges as a factor (less than 1) times the previous uncertainty to the first power (as is the case for bisection), it is said to converge *linearly*. Methods that converge as a higher power,

$$\epsilon_{n+1} = \text{constant} \times (\epsilon_n)^m \quad m > 1 \quad (9.1.4)$$

are said to converge *superlinearly*. In other contexts “linear” convergence would be termed “exponential,” or “geometrical.” That is not too bad at all: Linear convergence means that successive significant figures are won linearly with computational effort.

It remains to discuss practical criteria for convergence. It is crucial to keep in mind that computers use a fixed number of binary digits to represent floating-point numbers. While your function might analytically pass through zero, it is possible that its computed value is never zero, for any floating-point argument. One must decide what accuracy on the root is attainable: Convergence to within 10^{-6} in absolute value is reasonable when the root lies near 1, but certainly unachievable if the root lies near 10^{26} . One might thus think to specify convergence by a relative (fractional) criterion, but this becomes unworkable for roots near zero. To be most general, the routines below will require you to specify an absolute tolerance, such that iterations continue until the interval becomes smaller than this tolerance in absolute units. Usually you may wish to take the tolerance to be $\epsilon(|x_1| + |x_2|)/2$ where ϵ is the machine precision and x_1 and x_2 are the initial brackets. When the root lies near zero you ought to consider carefully what reasonable tolerance means for your function. The following routine quits after 40 bisections in any event, with $2^{-40} \approx 10^{-12}$.

```

#include <math.h>
#define JMAX 40                Maximum allowed number of bisections.

float rtbis(float (*func)(float), float x1, float x2, float xacc)
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  $\pm xacc$ .
{
    void nrerror(char error_text[]);
    int j;
    float dx,f,fmid,xmid,rtb;

    f=(*func)(x1);
    fmid=(*func)(x2);
    if (f*fmid >= 0.0) nrerror("Root must be bracketed for bisection in rtbis");
    rtb = f < 0.0 ? (dx=x2-x1,x1) : (dx=x1-x2,x2);    Orient the search so that f>0
    for (j=1;j<=JMAX;j++) {                            lies at x+dx.
        fmid=(*func)(xmid=rtb+(dx *= 0.5));            Bisection loop.
        if (fmid <= 0.0) rtb=xmid;
        if (fabs(dx) < xacc || fmid == 0.0) return rtb;
    }
    nrerror("Too many bisections in rtbis");
    return 0.0;                                         Never get here.
}

```

9.2 Secant Method, False Position Method, and Ridders' Method

For functions that are smooth near a root, the methods known respectively as *false position* (or *regula falsi*) and *secant method* generally converge faster than bisection. In both of these methods the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis. After each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root.

The *only* difference between the methods is that secant retains the most recent of the prior estimates (Figure 9.2.1; this requires an arbitrary choice on the first iteration), while false position retains that prior estimate for which the function value has opposite sign from the function value at the current best estimate of the root, so that the two points continue to bracket the root (Figure 9.2.2). Mathematically, the secant method converges more rapidly near a root of a sufficiently continuous function. Its order of convergence can be shown to be the “golden ratio” 1.618..., so that

$$\lim_{k \rightarrow \infty} |\epsilon_{k+1}| \approx \text{const} \times |\epsilon_k|^{1.618} \quad (9.2.1)$$

The secant method has, however, the disadvantage that the root does not necessarily remain bracketed. For functions that are *not* sufficiently continuous, the algorithm can therefore not be guaranteed to converge: Local behavior might send it off towards infinity.

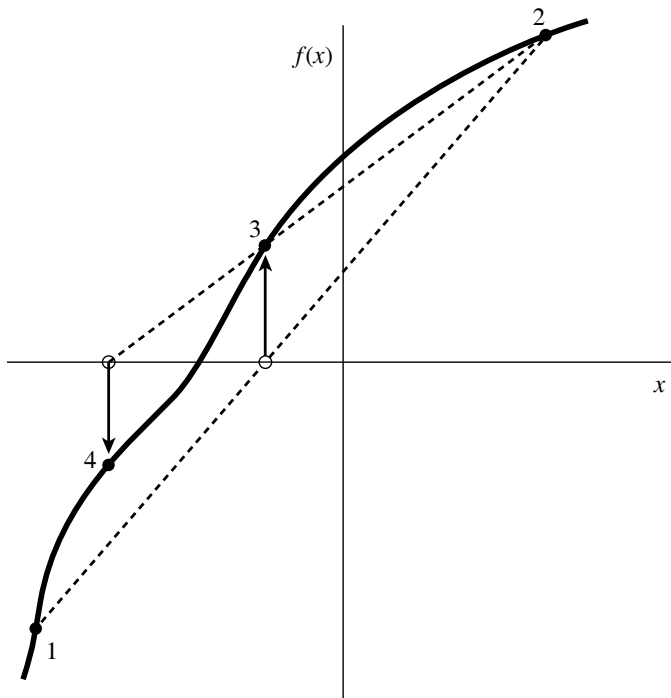


Figure 9.2.1. Secant method. Extrapolation or interpolation lines (dashed) are drawn through the two most recently evaluated points, whether or not they bracket the function. The points are numbered in the order that they are used.

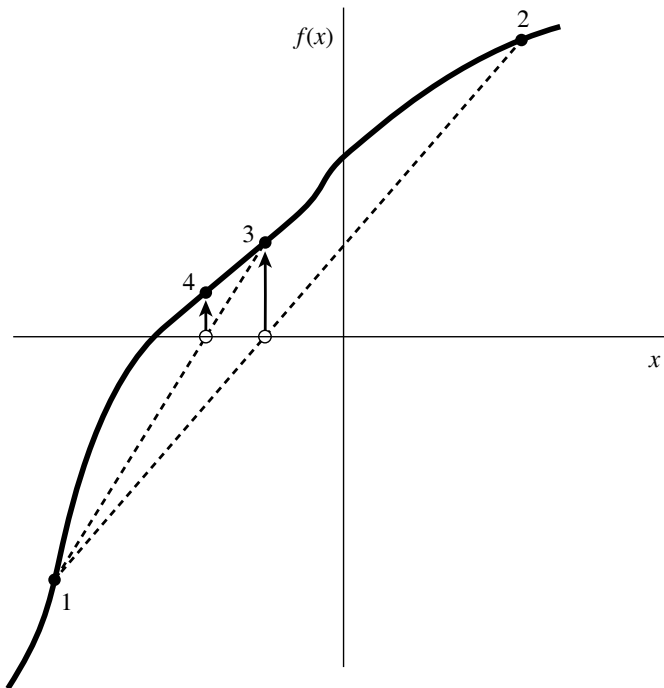


Figure 9.2.2. False position method. Interpolation lines (dashed) are drawn through the most recent points *that bracket the root*. In this example, point 1 thus remains “active” for many steps. False position converges less rapidly than the secant method, but it is more certain.

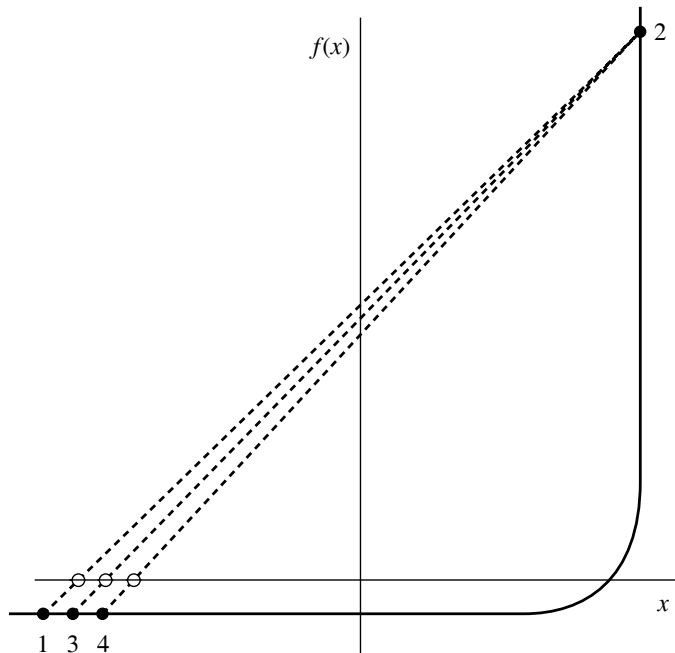


Figure 9.2.3. Example where both the secant and false position methods will take many iterations to arrive at the true root. This function would be difficult for many other root-finding methods.

False position, since it sometimes keeps an older rather than newer function evaluation, has a lower order of convergence. Since the newer function value will *sometimes* be kept, the method is often superlinear, but estimation of its exact order is not so easy.

Here are sample implementations of these two related methods. While these methods are standard textbook fare, *Ridders' method*, described below, or *Brent's method*, in the next section, are almost always better choices. Figure 9.2.3 shows the behavior of secant and false-position methods in a difficult situation.

```
#include <math.h>
#define MAXIT 30                                Set to the maximum allowed number of iterations.

float rtflsp(float (*func)(float), float x1, float x2, float xacc)
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  $\pm xacc$ .
{
    void nrerror(char error_text[]);
    int j;
    float fl,fh,xl,xh,swap,dx,del,f,rtf;

    fl=(*func)(x1);
    fh=(*func)(x2);
    if (fl*fh > 0.0) nrerror("Root must be bracketed in rtflsp");
    if (fl < 0.0) {
        xl=x1;
        xh=x2;
    } else {
        xl=x2;
        xh=x1;
        swap=fl;
    }
    Be sure the interval brackets a root.
    Identify the limits so that xl corresponds to the low
    side.
```

```

        fl=fh;
        fh=swap;
    }
    dx=xh-xl;
    for (j=1;j<=MAXIT;j++) {
        rtf=xl+dx*f1/(f1-fh);
        f=(*func)(rtf);
        if (f < 0.0) {
            del=xl-rtf;
            xl=rtf;
            fl=f;
        } else {
            del=xh-rtf;
            xh=rtf;
            fh=f;
        }
        dx=xh-xl;
        if (fabs(del) < xacc || f == 0.0) return rtf;
    }
    nrerror("Maximum number of iterations exceeded in rtf1sp");
    return 0.0;
}

```

False position loop.
Increment with respect to latest value.
Replace appropriate limit.
Convergence.
Never get here.

```

#include <math.h>
#define MAXIT 30

float rtsec(float (*func)(float), float x1, float x2, float xacc)
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  $\pm xacc$ .
{
    void nrerror(char error_text[]);
    int j;
    float fl,f,dx,swap,xl,rts;

    fl=(*func)(x1);
    f=(*func)(x2);
    if (fabs(fl) < fabs(f)) {
        rts=x1;
        xl=x2;
        swap=fl;
        fl=f;
        f=swap;
    } else {
        xl=x1;
        rts=x2;
    }
    for (j=1;j<=MAXIT;j++) {
        dx=(xl-rts)*f/(f-fl);
        xl=rts;
        fl=f;
        rts += dx;
        f=(*func)(rts);
        if (fabs(dx) < xacc || f == 0.0) return rts;
    }
    nrerror("Maximum number of iterations exceeded in rtsec");
    return 0.0;
}

```

Maximum allowed number of iterations.
Pick the bound with the smaller function value as the most recent guess.
Secant loop.
Increment with respect to latest value.
Convergence.
Never get here.

Ridders' Method

A powerful variant on false position is due to Ridders [1]. When a root is bracketed between x_1 and x_2 , Ridders' method first evaluates the function at the midpoint $x_3 = (x_1 + x_2)/2$. It then factors out that unique exponential function which turns the residual function into a straight line. Specifically, it solves for a factor e^Q that gives

$$f(x_1) - 2f(x_3)e^Q + f(x_2)e^{2Q} = 0 \quad (9.2.2)$$

This is a quadratic equation in e^Q , which can be solved to give

$$e^Q = \frac{f(x_3) + \text{sign}[f(x_2)]\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}{f(x_2)} \quad (9.2.3)$$

Now the false position method is applied, not to the values $f(x_1), f(x_3), f(x_2)$, but to the values $f(x_1), f(x_3)e^Q, f(x_2)e^{2Q}$, yielding a new guess for the root, x_4 . The overall updating formula (incorporating the solution 9.2.3) is

$$x_4 = x_3 + (x_3 - x_1) \frac{\text{sign}[f(x_1) - f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}} \quad (9.2.4)$$

Equation (9.2.4) has some very nice properties. First, x_4 is guaranteed to lie in the interval (x_1, x_2) , so the method never jumps out of its brackets. Second, the convergence of successive applications of equation (9.2.4) is *quadratic*, that is, $m = 2$ in equation (9.1.4). Since each application of (9.2.4) requires two function evaluations, the actual order of the method is $\sqrt{2}$, not 2; but this is still quite respectably superlinear: the number of significant digits in the answer approximately *doubles* with each two function evaluations. Third, taking out the function's "bend" via exponential (that is, ratio) factors, rather than via a polynomial technique (e.g., fitting a parabola), turns out to give an extraordinarily robust algorithm. In both reliability and speed, Ridders' method is generally competitive with the more highly developed and better established (but more complicated) method of Van Wijngaarden, Dekker, and Brent, which we next discuss.

```
#include <math.h>
#include "nrutil.h"
#define MAXIT 60
#define UNUSED (-1.11e30)
```

```
float zriddr(float (*func)(float), float x1, float x2, float xacc)
```

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`.

```
{
    int j;
    float ans, fh, fl, fm, fnew, s, xh, xl, xm, xnew;

    fl = (*func)(x1);
    fh = (*func)(x2);
    if ((fl > 0.0 && fh < 0.0) || (fl < 0.0 && fh > 0.0)) {
        xl = x1;
        xh = x2;
        ans = UNUSED;

```

Any highly unlikely value, to simplify logic below.

```

for (j=1;j<=MAXIT;j++) {
    xm=0.5*(xl+xh);
    fm=(*func)(xm);
    s=sqrt(fm*fm-fl*fh);
    if (s == 0.0) return ans;
    xnew=xm+(xm-xl)*((fl >= fh ? 1.0 : -1.0)*fm/s);
    if (fabs(xnew-ans) <= xacc) return ans;
    ans=xnew;
    fnew=(*func)(ans);
    if (fnew == 0.0) return ans;
    if (SIGN(fm,fnew) != fm) {
        xl=xm;
        fl=fm;
        xh=ans;
        fh=fnew;
    } else if (SIGN(fl,fnew) != fl) {
        xh=ans;
        fh=fnew;
    } else if (SIGN(fh,fnew) != fh) {
        xl=ans;
        fl=fnew;
    } else nrerror("never get here.");
    if (fabs(xh-xl) <= xacc) return ans;
}
nrerror("zridr exceed maximum iterations");
}
else {
    if (fl == 0.0) return x1;
    if (fh == 0.0) return x2;
    nrerror("root must be bracketed in zridr.");
}
}
return 0.0;
}

```

First of two function evaluations per iteration.

Updating formula.

Second of two function evaluations per iteration.

Bookkeeping to keep the root bracketed on next iteration.

Never get here.

CITED REFERENCES AND FURTHER READING:

- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.3.
- Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press), Chapter 12.
- Ridders, C.J.F. 1979, *IEEE Transactions on Circuits and Systems*, vol. CAS-26, pp. 979–980. [1]

9.3 Van Wijngaarden–Dekker–Brent Method

While secant and false position formally converge faster than bisection, one finds in practice pathological functions for which bisection converges more rapidly. These can be choppy, discontinuous functions, or even smooth functions if the second derivative changes sharply near the root. Bisection always halves the interval, while secant and false position can sometimes spend many cycles slowly pulling distant bounds closer to a root. Ridders' method does a much better job, but it too can sometimes be fooled. Is there a way to combine superlinear convergence with the sureness of bisection?

Yes. We can keep track of whether a supposedly superlinear method is actually converging the way it is supposed to, and, if it is not, we can intersperse bisection steps so as to guarantee *at least* linear convergence. This kind of super-strategy requires attention to bookkeeping detail, and also careful consideration of how roundoff errors can affect the guiding strategy. Also, we must be able to determine reliably when convergence has been achieved.

An excellent algorithm that pays close attention to these matters was developed in the 1960s by van Wijngaarden, Dekker, and others at the Mathematical Center in Amsterdam, and later improved by Brent [1]. For brevity, we refer to the final form of the algorithm as *Brent's method*. The method is *guaranteed* (by Brent) to converge, so long as the function can be evaluated within the initial interval known to contain a root.

Brent's method combines root bracketing, bisection, and *inverse quadratic interpolation* to converge from the neighborhood of a zero crossing. While the false position and secant methods assume approximately linear behavior between two prior root estimates, inverse quadratic interpolation uses three prior points to fit an inverse quadratic function (x as a quadratic function of y) whose value at $y = 0$ is taken as the next estimate of the root x . Of course one must have contingency plans for what to do if the root falls outside of the brackets. Brent's method takes care of all that. If the three point pairs are $[a, f(a)]$, $[b, f(b)]$, $[c, f(c)]$ then the interpolation formula (cf. equation 3.1.1) is

$$x = \frac{[y - f(a)][y - f(b)]c}{[f(c) - f(a)][f(c) - f(b)]} + \frac{[y - f(b)][y - f(c)]a}{[f(a) - f(b)][f(a) - f(c)]} + \frac{[y - f(c)][y - f(a)]b}{[f(b) - f(c)][f(b) - f(a)]} \quad (9.3.1)$$

Setting y to zero gives a result for the next root estimate, which can be written as

$$x = b + P/Q \quad (9.3.2)$$

where, in terms of

$$R \equiv f(b)/f(c), \quad S \equiv f(b)/f(a), \quad T \equiv f(a)/f(c) \quad (9.3.3)$$

we have

$$P = S[T(R - T)(c - b) - (1 - R)(b - a)] \quad (9.3.4)$$

$$Q = (T - 1)(R - 1)(S - 1) \quad (9.3.5)$$

In practice b is the current best estimate of the root and P/Q ought to be a “small” correction. Quadratic methods work well only when the function behaves smoothly; they run the serious risk of giving very bad estimates of the next root or causing machine failure by an inappropriate division by a very small number ($Q \approx 0$). Brent's method guards against this problem by maintaining brackets on the root and checking where the interpolation would land before carrying out the division. When the correction P/Q would not land within the bounds, or when the bounds are not collapsing rapidly enough, the algorithm takes a bisection step. Thus,

Brent's method combines the sureness of bisection with the speed of a higher-order method when appropriate. We recommend it as the method of choice for general one-dimensional root finding where a function's values only (and not its derivative or functional form) are available.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100           Maximum allowed number of iterations.
#define EPS 3.0e-8         Machine floating-point precision.

float zbrent(float (*func)(float), float x1, float x2, float tol)
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.
{
    int iter;
    float a=x1,b=x2,c=x2,d,e,min1,min2;
    float fa=(*func)(a),fb=(*func)(b),fc,p,q,r,s,tol1,xm;

    if ((fa > 0.0 && fb > 0.0) || (fa < 0.0 && fb < 0.0))
        nrerror("Root must be bracketed in zbrent");
    fc=fb;
    for (iter=1;iter<=ITMAX;iter++) {
        if ((fb > 0.0 && fc > 0.0) || (fb < 0.0 && fc < 0.0)) {
            c=a;                                Rename a, b, c and adjust bounding interval
            fc=fa;                                d.
            e=d=b-a;
        }
        if (fabs(fc) < fabs(fb)) {
            a=b;
            b=c;
            c=a;
            fa=fb;
            fb=fc;
            fc=fa;
        }
        tol1=2.0*EPS*fabs(b)+0.5*tol;           Convergence check.
        xm=0.5*(c-b);
        if (fabs(xm) <= tol1 || fb == 0.0) return b;
        if (fabs(e) >= tol1 && fabs(fa) > fabs(fb)) {
            s=fb/fa;                             Attempt inverse quadratic interpolation.
            if (a == c) {
                p=2.0*xm*s;
                q=1.0-s;
            } else {
                q=fa/fc;
                r=fb/fc;
                p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0));
                q=(q-1.0)*(r-1.0)*(s-1.0);
            }
            if (p > 0.0) q = -q;                   Check whether in bounds.
            p=fabs(p);
            min1=3.0*xm*q-fabs(tol1*q);
            min2=fabs(e*q);
            if (2.0*p < (min1 < min2 ? min1 : min2)) {
                e=d;                             Accept interpolation.
                d=p/q;
            } else {
                d=xm;                             Interpolation failed, use bisection.
                e=d;
            }
        } else {
            d=xm;                                 Bounds decreasing too slowly, use bisection.
            e=d;
        }
    }
}
```

```

    }
    a=b;                                Move last best guess to a.
    fa=fb;
    if (fabs(d) > tol1)                  Evaluate new trial root.
        b += d;
    else
        b += SIGN(tol1,xm);
    fb=(*func)(b);
}
nrerror("Maximum number of iterations exceeded in zbrent");
return 0.0;                            Never get here.
}

```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 3, 4. [1]
 Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §7.2.

9.4 Newton-Raphson Method Using Derivative

Perhaps the most celebrated of all one-dimensional root-finding routines is *Newton's method*, also called the *Newton-Raphson method*. This method is distinguished from the methods of previous sections by the fact that it requires the evaluation of both the function $f(x)$, and the derivative $f'(x)$, at arbitrary points x . The Newton-Raphson formula consists geometrically of extending the tangent line at a current point x_i until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero-crossing (see Figure 9.4.1). Algebraically, the method derives from the familiar Taylor series expansion of a function in the neighborhood of a point,

$$f(x + \delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \dots \quad (9.4.1)$$

For small enough values of δ , and for well-behaved functions, the terms beyond linear are unimportant, hence $f(x + \delta) = 0$ implies

$$\delta = -\frac{f(x)}{f'(x)}. \quad (9.4.2)$$

Newton-Raphson is not restricted to one dimension. The method readily generalizes to multiple dimensions, as we shall see in §9.6 and §9.7, below.

Far from a root, where the higher-order terms in the series *are* important, the Newton-Raphson formula can give grossly inaccurate, meaningless corrections. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. This can be death to the method (see Figure 9.4.2). If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson sends its solution off to limbo, with vanishingly small hope of recovery.

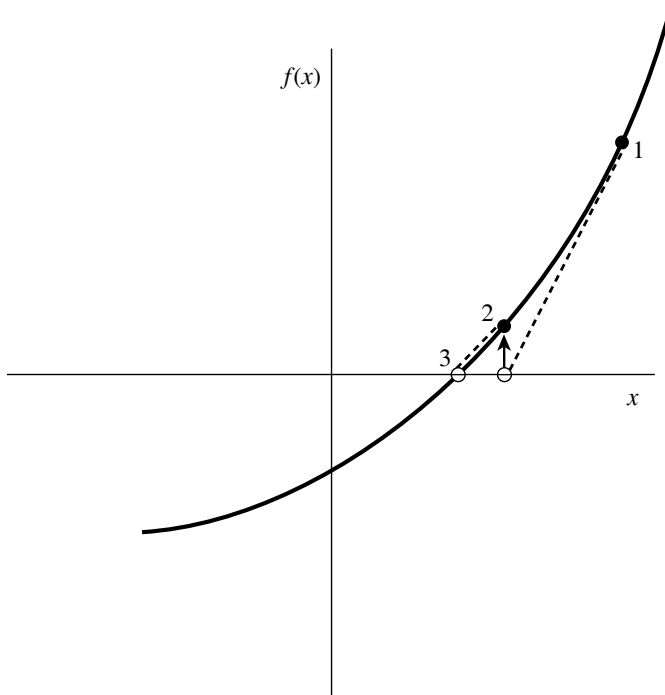


Figure 9.4.1. Newton's method extrapolates the local derivative to find the next estimate of the root. In this example it works well and converges quadratically.

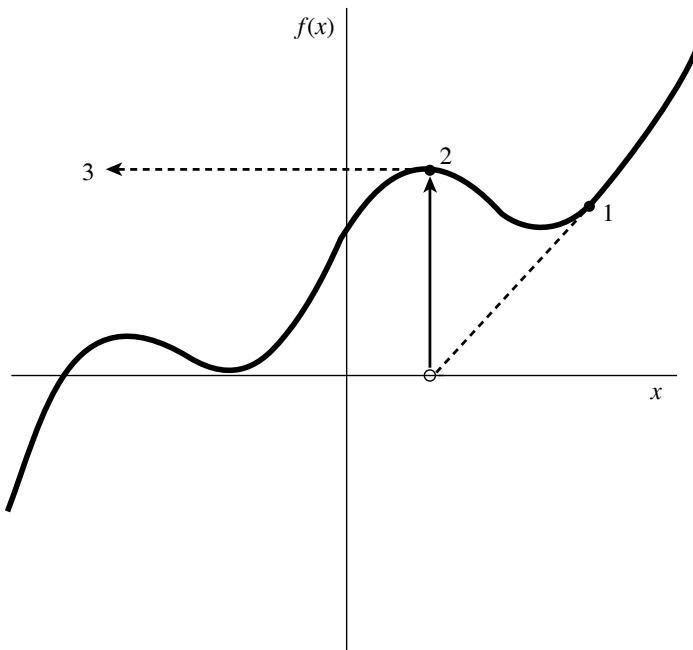


Figure 9.4.2. Unfortunate case where Newton's method encounters a local extremum and shoots off to outer space. Here bracketing bounds, as in `rtsafe`, would save the day.

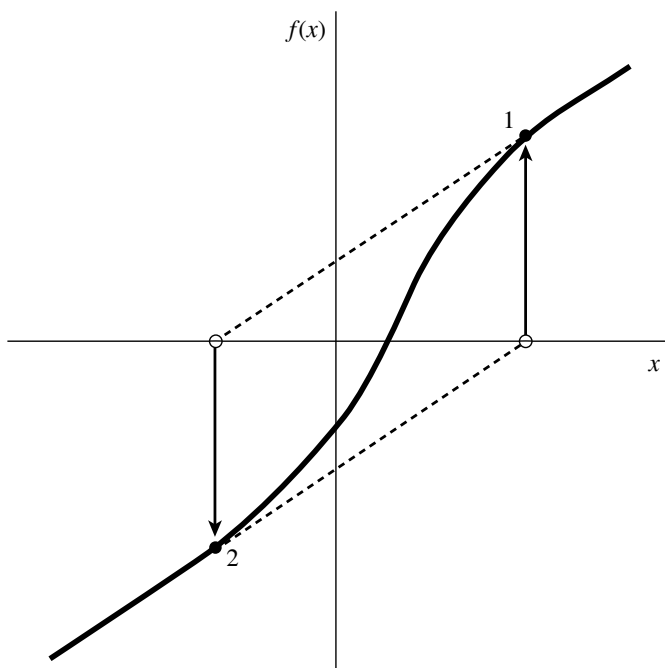


Figure 9.4.3. Unfortunate case where Newton's method enters a nonconvergent cycle. This behavior is often encountered when the function f is obtained, in whole or in part, by table interpolation. With a better initial guess, the method would have succeeded.

Like most powerful tools, Newton-Raphson can be destructively used in inappropriate circumstances. Figure 9.4.3 demonstrates another possible pathology.

Why do we call Newton-Raphson powerful? The answer lies in its rate of convergence: Within a small distance ϵ of x the function and its derivative are approximately:

$$\begin{aligned} f(x + \epsilon) &= f(x) + \epsilon f'(x) + \epsilon^2 \frac{f''(x)}{2} + \cdots, \\ f'(x + \epsilon) &= f'(x) + \epsilon f''(x) + \cdots \end{aligned} \quad (9.4.3)$$

By the Newton-Raphson formula,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \quad (9.4.4)$$

so that

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)}. \quad (9.4.5)$$

When a trial solution x_i differs from the true root by ϵ_i , we can use (9.4.3) to express $f(x_i)$, $f'(x_i)$ in (9.4.4) in terms of ϵ_i and derivatives at the root itself. The result is a recurrence relation for the deviations of the trial solutions

$$\epsilon_{i+1} = -\epsilon_i^2 \frac{f''(x)}{2f'(x)}. \quad (9.4.6)$$

Equation (9.4.6) says that Newton-Raphson converges *quadratically* (cf. equation 9.2.3). Near a root, the number of significant digits approximately *doubles* with each step. This very strong convergence property makes Newton-Raphson the method of choice for any function whose derivative can be evaluated efficiently, and whose derivative is continuous and nonzero in the neighborhood of a root.

Even where Newton-Raphson is rejected for the early stages of convergence (because of its poor global convergence properties), it is very common to “polish up” a root with one or two steps of Newton-Raphson, which can multiply by two or four its number of significant figures!

For an efficient realization of Newton-Raphson the user provides a routine that evaluates both $f(x)$ and its first derivative $f'(x)$ at the point x . The Newton-Raphson formula can also be applied using a numerical difference to approximate the true local derivative,

$$f'(x) \approx \frac{f(x+dx) - f(x)}{dx}. \quad (9.4.7)$$

This is not, however, a recommended procedure for the following reasons: (i) You are doing two function evaluations per step, so *at best* the superlinear order of convergence will be only $\sqrt{2}$. (ii) If you take dx too small you will be wiped out by roundoff, while if you take it too large your order of convergence will be only linear, no better than using the *initial* evaluation $f'(x_0)$ for all subsequent steps. Therefore, Newton-Raphson with numerical derivatives is (in one dimension) always dominated by the secant method of §9.2. (In multidimensions, where there is a paucity of available methods, Newton-Raphson with numerical derivatives must be taken more seriously. See §§9.6–9.7.)

The following function calls a user supplied function `funcd(x,fn,df)` which supplies the function value as `fn` and the derivative as `df`. We have included input bounds on the root simply to be consistent with previous root-finding routines: Newton does not adjust bounds, and works only on local information at the point x . The bounds are used only to pick the midpoint as the first guess, and to reject the solution if it wanders outside of the bounds.

```
#include <math.h>
#define JMAX 20                                Set to maximum number of iterations.

float rtnewt(void (*funcd)(float, float *, float *), float x1, float x2,
             float xacc)
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 routine that returns both the function value and the first derivative of the
function at the point x.
{
    void nrrerror(char error_text[]);
    int j;
    float df,dx,f,rtn;

    rtn=0.5*(x1+x2);                            Initial guess.
    for (j=1;j<=JMAX;j++) {
        (*funcd)(rtn,&f,&df);
        dx=f/df;
        rtn -= dx;
        if ((x1-rtn)*(rtn-x2) < 0.0)
            nrrerror("Jumped out of brackets in rtnewt");
```

```

        if (fabs(dx) < xacc) return rtn;          Convergence.
    }
    nrerror("Maximum number of iterations exceeded in rtnewt");
    return 0.0;                                  Never get here.
}

```

While Newton-Raphson's global convergence properties are poor, it is fairly easy to design a fail-safe routine that utilizes a combination of bisection and Newton-Raphson. The hybrid algorithm takes a bisection step whenever Newton-Raphson would take the solution out of bounds, or whenever Newton-Raphson is not reducing the size of the brackets rapidly enough.

```

#include <math.h>
#define MAXIT 100                                Maximum allowed number of iterations.

float rtsafe(void (*funcd)(float, float *, float *), float x1, float x2,
             float xacc)
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  $\pm xacc$ . funcd is a user-supplied routine that returns both the
function value and the first derivative of the function.
{
    void nrerror(char error_text[]);
    int j;
    float df,dx,dxold,f,fh,fl;
    float temp,xh,xl,rts;

    (*funcd)(x1,&fl,&df);
    (*funcd)(x2,&fh,&df);
    if ((fl > 0.0 && fh > 0.0) || (fl < 0.0 && fh < 0.0))
        nrerror("Root must be bracketed in rtsafe");
    if (fl == 0.0) return x1;
    if (fh == 0.0) return x2;
    if (fl < 0.0) {                                Orient the search so that  $f(x_1) < 0$ .
        xl=x1;
        xh=x2;
    } else {
        xh=x1;
        xl=x2;
    }
    rts=0.5*(x1+x2);                                Initialize the guess for root,
    dxold=fabs(x2-x1);                               the "stepsize before last,"
    dx=dxold;                                         and the last step.
    (*funcd)(rts,&f,&df);
    for (j=1;j<=MAXIT;j++) {                          Loop over allowed iterations.
        if (((rts-xh)*df-f)*((rts-xl)*df-f) > 0.0)    Bisect if Newton out of range,
            || (fabs(2.0*f) > fabs(dxold*df)) {        or not decreasing fast enough.
                dxold=dx;
                dx=0.5*(xh-xl);
                rts=xl+dx;
                if (xl == rts) return rts;             Change in root is negligible.
            } else {                                    Newton step acceptable. Take it.
                dxold=dx;
                dx=f/df;
                temp=rts;
                rts -= dx;
                if (temp == rts) return rts;
            }
        }
        if (fabs(dx) < xacc) return rts;              Convergence criterion.
        (*funcd)(rts,&f,&df);
        The one new function evaluation per iteration.
    }
}

```

```

    if (f < 0.0)                                Maintain the bracket on the root.
        xl=rts;
    else
        xh=rts;
}
nrerror("Maximum number of iterations exceeded in rtsafe");
return 0.0;                                    Never get here.
}

```

For many functions the derivative $f'(x)$ often converges to machine accuracy before the function $f(x)$ itself does. When that is the case one need not subsequently update $f'(x)$. This shortcut is recommended only when you confidently understand the generic behavior of your function, but it speeds computations when the derivative calculation is laborious. (Formally this makes the convergence only linear, but if the derivative isn't changing anyway, you can do no better.)

Newton-Raphson and Fractals

An interesting sidelight to our repeated warnings about Newton-Raphson's unpredictable global convergence properties — its very rapid local convergence notwithstanding — is to investigate, for some particular equation, the set of starting values from which the method does, or doesn't converge to a root.

Consider the simple equation

$$z^3 - 1 = 0 \quad (9.4.8)$$

whose single real root is $z = 1$, but which also has complex roots at the other two cube roots of unity, $\exp(\pm 2\pi i/3)$. Newton's method gives the iteration

$$z_{j+1} = z_j - \frac{z_j^3 - 1}{3z_j^2} \quad (9.4.9)$$

Up to now, we have applied an iteration like equation (9.4.9) only for real starting values z_0 , but in fact all of the equations in this section also apply in the complex plane. We can therefore map out the complex plane into regions from which a starting value z_0 , iterated in equation (9.4.9), will, or won't, converge to $z = 1$. Naively, we might expect to find a "basin of convergence" somehow surrounding the root $z = 1$. We surely do not expect the basin of convergence to fill the whole plane, because the plane must also contain regions that converge to each of the two complex roots. In fact, by symmetry, the three regions must have identical shapes. Perhaps they will be three symmetric 120° wedges, with one root centered in each?

Now take a look at Figure 9.4.4, which shows the result of a numerical exploration. The basin of convergence does indeed cover $1/3$ the area of the complex plane, but its boundary is highly irregular — in fact, *fractal*. (A fractal, so called, has self-similar structure that repeats on all scales of magnification.) How does this fractal emerge from something as simple as Newton's method, and an equation as simple as (9.4.8)? The answer is already implicit in Figure 9.4.2, which showed how, on the real line, a local extremum causes Newton's method to shoot off to infinity. Suppose one is *slightly* removed from such a point. Then one might be shot off not to infinity, but — by luck — right into the basin of convergence of the desired

Figure 9.4.4. The complex z plane with real and imaginary components in the range $(-2, 2)$. The black region is the set of points from which Newton's method converges to the root $z = 1$ of the equation $z^3 - 1 = 0$. Its shape is fractal.

root. But that means that in the neighborhood of an extremum there must be a tiny, perhaps distorted, copy of the basin of convergence — a kind of “one-bounce away” copy. Similar logic shows that there can be “two-bounce” copies, “three-bounce” copies, and so on. A fractal thus emerges.

Notice that, for equation (9.4.8), almost the whole real axis is in the domain of convergence for the root $z = 1$. We say “almost” because of the peculiar discrete points on the negative real axis whose convergence is indeterminate (see figure). What happens if you start Newton's method from one of these points? (Try it.)

CITED REFERENCES AND FURTHER READING:

- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 2.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.4.
- Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press).
- Mandelbrot, B.B. 1983, *The Fractal Geometry of Nature* (San Francisco: W.H. Freeman).
- Peitgen, H.-O., and Saupe, D. (eds.) 1988, *The Science of Fractal Images* (New York: Springer-Verlag).

9.5 Roots of Polynomials

Here we present a few methods for finding roots of polynomials. These will serve for most practical problems involving polynomials of low-to-moderate degree or for well-conditioned polynomials of higher degree. Not as well appreciated as it ought to be is the fact that some polynomials are exceedingly ill-conditioned. The tiniest changes in a polynomial's coefficients can, in the worst case, send its roots sprawling all over the complex plane. (An infamous example due to Wilkinson is detailed by Acton [1].)

Recall that a polynomial of degree n will have n roots. The roots can be real or complex, and they might not be distinct. If the coefficients of the polynomial are real, then complex roots will occur in pairs that are conjugate, i.e., if $x_1 = a + bi$ is a root then $x_2 = a - bi$ will also be a root. When the coefficients are complex, the complex roots need not be related.

Multiple roots, or closely spaced roots, produce the most difficulty for numerical algorithms (see Figure 9.5.1). For example, $P(x) = (x - a)^2$ has a double real root at $x = a$. However, we cannot bracket the root by the usual technique of identifying neighborhoods where the function changes sign, nor will slope-following methods such as Newton-Raphson work well, because both the function and its derivative vanish at a multiple root. Newton-Raphson *may* work, but slowly, since large roundoff errors can occur. When a root is known in advance to be multiple, then special methods of attack are readily devised. Problems arise when (as is generally the case) we do not know in advance what pathology a root will display.

Deflation of Polynomials

When seeking several or all roots of a polynomial, the total effort can be significantly reduced by the use of *deflation*. As each root r is found, the polynomial is factored into a product involving the root and a reduced polynomial of degree one less than the original, i.e., $P(x) = (x - r)Q(x)$. Since the roots of Q are exactly the remaining roots of P , the effort of finding additional roots decreases, because we work with polynomials of lower and lower degree as we find successive roots. Even more important, with deflation we can avoid the blunder of having our iterative method converge twice to the same (nonmultiple) root instead of separately to two different roots.

Deflation, which amounts to synthetic division, is a simple operation that acts on the array of polynomial coefficients. The concise code for synthetic division by a monomial factor was given in §5.3 above. You can deflate complex roots either by converting that code to complex data type, or else — in the case of a polynomial with real coefficients but possibly complex roots — by deflating by a quadratic factor,

$$[x - (a + ib)][x - (a - ib)] = x^2 - 2ax + (a^2 + b^2) \quad (9.5.1)$$

The routine `poldiv` in §5.3 can be used to divide the polynomial by this factor.

Deflation must, however, be utilized with care. Because each new root is known with only finite accuracy, errors creep into the determination of the coefficients of the successively deflated polynomial. Consequently, the roots can become more and more inaccurate. It matters a lot whether the inaccuracy creeps in stably (plus or

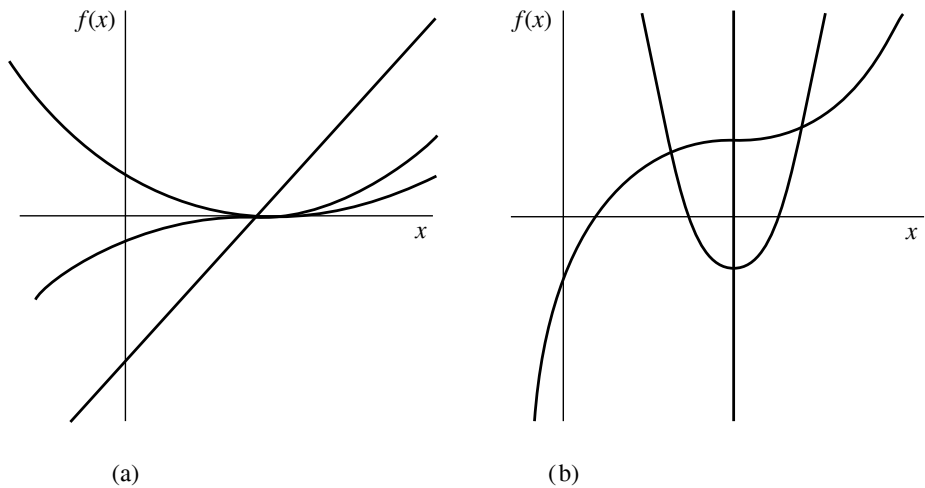


Figure 9.5.1. (a) Linear, quadratic, and cubic behavior at the roots of polynomials. Only under high magnification (b) does it become apparent that the cubic has one, not three, roots, and that the quadratic has two roots rather than none.

minus a few multiples of the machine precision at each stage) or unstably (erosion of successive significant figures until the results become meaningless). Which behavior occurs depends on just how the root is divided out. *Forward deflation*, where the new polynomial coefficients are computed in the order from the highest power of x down to the constant term, was illustrated in §5.3. This turns out to be stable if the root of smallest absolute value is divided out at each stage. Alternatively, one can do *backward deflation*, where new coefficients are computed in order from the constant term up to the coefficient of the highest power of x . This is stable if the remaining root of *largest* absolute value is divided out at each stage.

A polynomial whose coefficients are interchanged “end-to-end,” so that the constant becomes the highest coefficient, etc., has its roots mapped into their reciprocals. (Proof: Divide the whole polynomial by its highest power x^n and rewrite it as a polynomial in $1/x$.) The algorithm for backward deflation is therefore virtually identical to that of forward deflation, except that the original coefficients are taken in reverse order and the reciprocal of the deflating root is used. Since we will use forward deflation below, we leave to you the exercise of writing a concise coding for backward deflation (as in §5.3). For more on the stability of deflation, consult [2].

To minimize the impact of increasing errors (even stable ones) when using deflation, it is advisable to treat roots of the successively deflated polynomials as only *tentative* roots of the original polynomial. One then *polishes* these tentative roots by taking them as initial guesses that are to be re-solved for, using the *nondeflated* original polynomial P . Again you must beware lest two deflated roots are inaccurate enough that, under polishing, they both converge to the same undeflated root; in that case you gain a spurious root-multiplicity and lose a distinct root. This is detectable, since you can compare each polished root for equality to previous ones from distinct tentative roots. When it happens, you are advised to deflate the polynomial just once (and for this root only), then again polish the tentative root, or to use Maehly’s procedure (see equation 9.5.29 below).

Below we say more about techniques for polishing real and complex-conjugate

tentative roots. First, let's get back to overall strategy.

There are two schools of thought about how to proceed when faced with a polynomial of real coefficients. One school says to go after the easiest quarry, the real, distinct roots, by the same kinds of methods that we have discussed in previous sections for general functions, i.e., trial-and-error bracketing followed by a safe Newton-Raphson as in `rtsafe`. Sometimes you are *only* interested in real roots, in which case the strategy is complete. Otherwise, you then go after quadratic factors of the form (9.5.1) by any of a variety of methods. One such is Bairstow's method, which we will discuss below in the context of root polishing. Another is Muller's method, which we here briefly discuss.

Muller's Method

Muller's method generalizes the secant method, but uses quadratic interpolation among three points instead of linear interpolation between two. Solving for the zeros of the quadratic allows the method to find complex pairs of roots. Given *three* previous guesses for the root x_{i-2} , x_{i-1} , x_i , and the values of the polynomial $P(x)$ at those points, the next approximation x_{i+1} is produced by the following formulas,

$$\begin{aligned} q &\equiv \frac{x_i - x_{i-1}}{x_{i-1} - x_{i-2}} \\ A &\equiv qP(x_i) - q(1+q)P(x_{i-1}) + q^2P(x_{i-2}) \\ B &\equiv (2q+1)P(x_i) - (1+q)^2P(x_{i-1}) + q^2P(x_{i-2}) \\ C &\equiv (1+q)P(x_i) \end{aligned} \tag{9.5.2}$$

followed by

$$x_{i+1} = x_i - (x_i - x_{i-1}) \left[\frac{2C}{B \pm \sqrt{B^2 - 4AC}} \right] \tag{9.5.3}$$

where the sign in the denominator is chosen to make its absolute value or modulus as large as possible. You can start the iterations with any three values of x that you like, e.g., three equally spaced values on the real axis. Note that you must allow for the possibility of a complex denominator, and subsequent complex arithmetic, in implementing the method.

Muller's method is sometimes also used for finding complex zeros of analytic functions (not just polynomials) in the complex plane, for example in the IMSL routine `ZANLY` [3].

Laguerre's Method

The second school regarding overall strategy happens to be the one to which we belong. That school advises you to use one of a very small number of methods that will converge (though with greater or lesser efficiency) to all types of roots: real, complex, single, or multiple. Use such a method to get tentative values for all n roots of your n th degree polynomial. Then go back and polish them as you desire.

Laguerre's method is by far the most straightforward of these general, complex methods. It does require complex arithmetic, even while converging to real roots; however, for polynomials with all real roots, it is guaranteed to converge to a root from any starting point. For polynomials with some complex roots, little is theoretically proved about the method's convergence. Much empirical experience, however, suggests that nonconvergence is extremely unusual, and, further, can almost always be fixed by a simple scheme to break a nonconverging limit cycle. (This is implemented in our routine, below.) An example of a polynomial that requires this cycle-breaking scheme is one of high degree ($\gtrsim 20$), with all its roots just outside of the complex unit circle, approximately equally spaced around it. When the method converges on a simple complex zero, it is known that its convergence is third order.

In some instances the complex arithmetic in the Laguerre method is no disadvantage, since the polynomial itself may have complex coefficients.

To motivate (although not rigorously derive) the Laguerre formulas we can note the following relations between the polynomial and its roots and derivatives

$$P_n(x) = (x - x_1)(x - x_2) \dots (x - x_n) \quad (9.5.4)$$

$$\ln |P_n(x)| = \ln |x - x_1| + \ln |x - x_2| + \dots + \ln |x - x_n| \quad (9.5.5)$$

$$\frac{d \ln |P_n(x)|}{dx} = +\frac{1}{x - x_1} + \frac{1}{x - x_2} + \dots + \frac{1}{x - x_n} = \frac{P'_n}{P_n} \equiv G \quad (9.5.6)$$

$$\begin{aligned} -\frac{d^2 \ln |P_n(x)|}{dx^2} &= +\frac{1}{(x - x_1)^2} + \frac{1}{(x - x_2)^2} + \dots + \frac{1}{(x - x_n)^2} \\ &= \left[\frac{P'_n}{P_n} \right]^2 - \frac{P''_n}{P_n} \equiv H \end{aligned} \quad (9.5.7)$$

Starting from these relations, the Laguerre formulas make what Acton [1] nicely calls “a rather drastic set of assumptions”: The root x_1 that we seek is assumed to be located some distance a from our current guess x , while *all other roots* are assumed to be located at a distance b

$$x - x_1 = a \quad ; \quad x - x_i = b \quad i = 2, 3, \dots, n \quad (9.5.8)$$

Then we can express (9.5.6), (9.5.7) as

$$\frac{1}{a} + \frac{n-1}{b} = G \quad (9.5.9)$$

$$\frac{1}{a^2} + \frac{n-1}{b^2} = H \quad (9.5.10)$$

which yields as the solution for a

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}} \quad (9.5.11)$$

where the sign should be taken to yield the largest magnitude for the denominator. Since the factor inside the square root can be negative, a can be complex. (A more rigorous justification of equation 9.5.11 is in [4].)

The method operates iteratively: For a trial value x , a is calculated by equation (9.5.11). Then $x - a$ becomes the next trial value. This continues until a is sufficiently small.

The following routine implements the Laguerre method to find one root of a given polynomial of degree m , whose coefficients can be complex. As usual, the first coefficient $a[0]$ is the constant term, while $a[m]$ is the coefficient of the highest power of x . The routine implements a simplified version of an elegant stopping criterion due to Adams [5], which neatly balances the desire to achieve full machine accuracy, on the one hand, with the danger of iterating forever in the presence of roundoff error, on the other.

```
#include <math.h>
#include "complex.h"
#include "nrutil.h"
#define EPSS 1.0e-7
#define MR 8
#define MT 10
#define MAXIT (MT*MR)
Here EPSS 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.
```

```
void laguer(fcomplex a[], int m, fcomplex *x, int *its)
```

Given the degree m and the $m+1$ complex coefficients $a[0..m]$ of the polynomial $\sum_{i=0}^m a[i]x^i$, 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 .

```
{
    int iter,j;
    float abx,abp,abm,err;
    fcomplex dx,x1,b,d,f,g,h,sq,gp,gm,g2;
    static float frac[MR+1] = {0.0,0.5,0.25,0.75,0.13,0.38,0.62,0.88,1.0};
    Fractions used to break a limit cycle.

    for (iter=1;iter<=MAXIT;iter++) {          Loop over iterations up to allowed maximum.
        *its=iter;
        b=a[m];
        err=Cabs(b);
        d=f=Complex(0.0,0.0);
        abx=Cabs(*x);
        for (j=m-1;j>=0;j--) {                  Efficient computation of the polynomial and
            f=Cadd(Cmul(*x,f),d);                its first two derivatives. f stores  $P''/2$ .
            d=Cadd(Cmul(*x,d),b);
            b=Cadd(Cmul(*x,b),a[j]);
            err=Cabs(b)+abx*err;
        }
        err *= EPSS;
        Estimate of roundoff error in evaluating polynomial.
        if (Cabs(b) <= err) return;              We are on the root.
        g=Cdiv(d,b);                             The generic case: use Laguerre's formula.
        g2=Cmul(g,g);
        h=Csub(g2,RCmul(2.0,Cdiv(f,b)));
        sq=Csqrt(RCmul((float) (m-1),Csub(RCmul((float) m,h),g2)));
        gp=Cadd(g,sq);
        gm=Csub(g,sq);
        abp=Cabs(gp);
        abm=Cabs(gm);
        if (abp < abm) gp=gm;
        dx=((FMAX(abp,abm) > 0.0 ? Cdiv(Complex((float) m,0.0),gp)
            : RCmul(1+abx,Complex(cos((float)iter),sin((float)iter)))));
        x1=Csub(*x,dx);
    }
```

```

    if (x->r == x1.r && x->i == x1.i) return;          Converged.
    if (iter % MT) *x=x1;
    else *x=Csub(*x,RCmul(frac[iter/MT],dx));
    Every so often we take a fractional step, to break any limit cycle (itself a rare occurrence).
}
nrerror("too many iterations in laguer");
Very unusual — can occur only for complex roots. Try a different starting guess for the root.
return;
}

```

Here is a driver routine that calls `laguer` in succession for each root, performs the deflation, optionally polishes the roots by the same Laguerre method — if you are not going to polish in some other way — and finally sorts the roots by their real parts. (We will use this routine in Chapter 13.)

```

#include <math.h>
#include "complex.h"
#define EPS 2.0e-6
#define MAXM 100
A small number, and maximum anticipated value of m.

void zroots(fcomplex a[], int m, fcomplex roots[], int polish)
Given the degree m and the m+1 complex coefficients a[0..m] of the polynomial  $\sum_{i=0}^m a(i)x^i$ ,
this routine successively calls laguer and finds all m complex roots in roots[1..m]. The
boolean variable polish should be input as true (1) if polishing (also by Laguerre's method)
is desired, false (0) if the roots will be subsequently polished by other means.
{
    void laguer(fcomplex a[], int m, fcomplex *x, int *its);
    int i, its, j, jj;
    fcomplex x, b, c, ad[MAXM];

    for (j=0; j<=m; j++) ad[j]=a[j];          Copy of coefficients for successive deflation.
    for (j=m; j>=1; j--) {                    Loop over each root to be found.
        x=Complex(0.0,0.0);                  Start at zero to favor convergence to small-
        laguer(ad, j, &x, &its);              est remaining root, and find the root.
        if (fabs(x.i) <= 2.0*EPS*fabs(x.r)) x.i=0.0;
        roots[j]=x;
        b=ad[j];                              Forward deflation.
        for (jj=j-1; jj>=0; jj--) {
            c=ad[jj];
            ad[jj]=b;
            b=Cadd(Cmul(x,b),c);
        }
    }
    if (polish)
        for (j=1; j<=m; j++)                Polish the roots using the undeflated coefficients.
            laguer(a, m, &roots[j], &its);
    for (j=2; j<=m; j++) {                    Sort roots by their real parts by straight insertion.
        x=roots[j];
        for (i=j-1; i>=1; i--) {
            if (roots[i].r <= x.r) break;
            roots[i+1]=roots[i];
        }
        roots[i+1]=x;
    }
}

```

Eigenvalue Methods

The eigenvalues of a matrix \mathbf{A} are the roots of the “characteristic polynomial” $P(x) = \det[\mathbf{A} - x\mathbf{I}]$. However, as we will see in Chapter 11, root-finding is not generally an efficient way to find eigenvalues. Turning matters around, we can use the more efficient eigenvalue methods that are discussed in Chapter 11 to find the roots of arbitrary polynomials. You can easily verify (see, e.g., [6]) that the characteristic polynomial of the special $m \times m$ companion matrix

$$\mathbf{A} = \begin{pmatrix} -\frac{a_{m-1}}{a_m} & -\frac{a_{m-2}}{a_m} & \cdots & -\frac{a_1}{a_m} & -\frac{a_0}{a_m} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \quad (9.5.12)$$

is equivalent to the general polynomial

$$P(x) = \sum_{i=0}^m a_i x^i \quad (9.5.13)$$

If the coefficients a_i are real, rather than complex, then the eigenvalues of \mathbf{A} can be found using the routines `balanc` and `hqr` in §§11.5–11.6 (see discussion there). This method, implemented in the routine `zrhqr` following, is typically about a factor 2 slower than `zroots` (above). However, for some classes of polynomials, it is a more robust technique, largely because of the fairly sophisticated convergence methods embodied in `hqr`. If your polynomial has real coefficients, and you are having trouble with `zroots`, then `zrhqr` is a recommended alternative.

```
#include "nrutil.h"
#define MAXM 50

void zrhqr(float a[], int m, float rtr[], float rti[])
Find all the roots of a polynomial with real coefficients,  $\sum_{i=0}^m a(i)x^i$ , given the degree m
and the coefficients a[0..m]. 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 roots are returned in rtr[1..m] and rti[1..m], respectively.
{
    void balanc(float **a, int n);
    void hqr(float **a, int n, float wr[], float wi[]);
    int j,k;
    float **hess,xr,xi;

    hess=matrix(1,MAXM,1,MAXM);
    if (m > MAXM || a[m] == 0.0) nrerror("bad args in zrhqr");
    for (k=1;k<=m;k++) { Construct the matrix.
        hess[1][k] = -a[m-k]/a[m];
        for (j=2;j<=m;j++) hess[j][k]=0.0;
        if (k != m) hess[k+1][k]=1.0;
    }
    balanc(hess,m); Find its eigenvalues.
    hqr(hess,m,rtr,rti);
    for (j=2;j<=m;j++) { Sort roots by their real parts by straight insertion.
        xr=rtr[j];
        xi=rti[j];
```

```

    for (k=j-1;k>=1;k--) {
        if (rtr[k] <= xr) break;
        rtr[k+1]=rtr[k];
        rti[k+1]=rti[k];
    }
    rtr[k+1]=xr;
    rti[k+1]=xi;
}
free_matrix(hess,1,MAXM,1,MAXM);
}

```

Other Sure-Fire Techniques

The *Jenkins-Traub method* has become practically a standard in black-box polynomial root-finders, e.g., in the IMSL library [3]. The method is too complicated to discuss here, but is detailed, with references to the primary literature, in [4].

The *Lehmer-Schur algorithm* is one of a class of methods that isolate roots in the complex plane by generalizing the notion of one-dimensional bracketing. It is possible to determine efficiently whether there are any polynomial roots within a circle of given center and radius. From then on it is a matter of bookkeeping to hunt down all the roots by a series of decisions regarding where to place new trial circles. Consult [1] for an introduction.

Techniques for Root-Polishing

Newton-Raphson works very well for real roots once the neighborhood of a root has been identified. The polynomial and its derivative can be efficiently simultaneously evaluated as in §5.3. For a polynomial of degree n with coefficients $c[0] \dots c[n]$, the following segment of code embodies one cycle of Newton-Raphson:

```

p=c[n]*x+c[n-1];
p1=c[n];
for(i=n-2;i>=0;i--) {
    p1=p+p1*x;
    p=c[i]+p*x;
}
if (p1 == 0.0) nrerror("derivative should not vanish");
x -= p/p1;

```

Once all real roots of a polynomial have been polished, one must polish the complex roots, either directly, or by looking for quadratic factors.

Direct polishing by Newton-Raphson is straightforward for complex roots if the above code is converted to complex data types. With real polynomial coefficients, note that your starting guess (tentative root) *must* be off the real axis, otherwise you will never get off that axis — and may get shot off to infinity by a minimum or maximum of the polynomial.

For real polynomials, the alternative means of polishing complex roots (or, for that matter, double real roots) is *Bairstow's method*, which seeks quadratic factors. The advantage

of going after quadratic factors is that it avoids all complex arithmetic. Bairstow's method seeks a quadratic factor that embodies the two roots $x = a \pm ib$, namely

$$x^2 - 2ax + (a^2 + b^2) \equiv x^2 + Bx + C \quad (9.5.14)$$

In general if we divide a polynomial by a quadratic factor, there will be a linear remainder

$$P(x) = (x^2 + Bx + C)Q(x) + Rx + S. \quad (9.5.15)$$

Given B and C , R and S can be readily found, by polynomial division (§5.3). We can consider R and S to be adjustable functions of B and C , and they will be zero if the quadratic factor is a divisor of $P(x)$.

In the neighborhood of a root a first-order Taylor series expansion approximates the variation of R, S with respect to small changes in B, C

$$R(B + \delta B, C + \delta C) \approx R(B, C) + \frac{\partial R}{\partial B} \delta B + \frac{\partial R}{\partial C} \delta C \quad (9.5.16)$$

$$S(B + \delta B, C + \delta C) \approx S(B, C) + \frac{\partial S}{\partial B} \delta B + \frac{\partial S}{\partial C} \delta C \quad (9.5.17)$$

To evaluate the partial derivatives, consider the derivative of (9.5.15) with respect to C . Since $P(x)$ is a fixed polynomial, it is independent of C , hence

$$0 = (x^2 + Bx + C) \frac{\partial Q}{\partial C} + Q(x) + \frac{\partial R}{\partial C} x + \frac{\partial S}{\partial C} \quad (9.5.18)$$

which can be rewritten as

$$-Q(x) = (x^2 + Bx + C) \frac{\partial Q}{\partial C} + \frac{\partial R}{\partial C} x + \frac{\partial S}{\partial C} \quad (9.5.19)$$

Similarly, $P(x)$ is independent of B , so differentiating (9.5.15) with respect to B gives

$$-xQ(x) = (x^2 + Bx + C) \frac{\partial Q}{\partial B} + \frac{\partial R}{\partial B} x + \frac{\partial S}{\partial B} \quad (9.5.20)$$

Now note that equation (9.5.19) matches equation (9.5.15) in form. Thus if we perform a second synthetic division of $P(x)$, i.e., a division of $Q(x)$, yielding a remainder $R_1 x + S_1$, then

$$\frac{\partial R}{\partial C} = -R_1 \quad \frac{\partial S}{\partial C} = -S_1 \quad (9.5.21)$$

To get the remaining partial derivatives, evaluate equation (9.5.20) at the two roots of the quadratic, x_+ and x_- . Since

$$Q(x_{\pm}) = R_1 x_{\pm} + S_1 \quad (9.5.22)$$

we get

$$\frac{\partial R}{\partial B} x_+ + \frac{\partial S}{\partial B} = -x_+(R_1 x_+ + S_1) \quad (9.5.23)$$

$$\frac{\partial R}{\partial B} x_- + \frac{\partial S}{\partial B} = -x_-(R_1 x_- + S_1) \quad (9.5.24)$$

Solve these two equations for the partial derivatives, using

$$x_+ + x_- = -B \quad x_+ x_- = C \quad (9.5.25)$$

and find

$$\frac{\partial R}{\partial B} = BR_1 - S_1 \quad \frac{\partial S}{\partial B} = CR_1 \quad (9.5.26)$$

Bairstow's method now consists of using Newton-Raphson in two dimensions (which is actually the subject of the *next* section) to find a simultaneous zero of R and S . Synthetic division is used twice per cycle to evaluate R, S and their partial derivatives with respect to B, C . Like one-dimensional Newton-Raphson, the method works well in the vicinity of a root pair (real or complex), but it can fail miserably when started at a random point. We therefore recommend it only in the context of polishing tentative complex roots.

```

#include <math.h>
#include "nrutil.h"
#define ITMAX 20
#define TINY 1.0e-6

void qroot(float p[], int n, float *b, float *c, float eps)
Given n+1 coefficients p[0..n] of a polynomial of degree n, and trial values for the coefficients
of a quadratic factor x*x+b*x+c, improve the solution until the coefficients b,c change by less
than eps. The routine poldiv §5.3 is used.
{
    void poldiv(float u[], int n, float v[], int nv, float q[], float r[]);
    int iter;
    float sc,sb,s,rc,rb,r,dv,delc,delb;
    float *q,*qq,*rem;
    float d[3];

    q=vector(0,n);
    qq=vector(0,n);
    rem=vector(0,n);
    d[2]=1.0;
    for (iter=1;iter<=ITMAX;iter++) {
        d[1]=(*b);
        d[0]=(*c);
        poldiv(p,n,d,2,q,rem);
        s=rem[0];
        r=rem[1];
        poldiv(q,(n-1),d,2,qq,rem);
        sb = -(*c)*(rc = -rem[1]);
        rb = -(*b)*rc+(sc = -rem[0]);
        dv=1.0/(sb*rc-sc*rb);
        delb=(r*sc-s*rc)*dv;
        delc=(-r*sb+s*rb)*dv;
        *b += (delb=(r*sc-s*rc)*dv);
        *c += (delc=(-r*sb+s*rb)*dv);
        if ((fabs(delb) <= eps*fabs(*b) || fabs(*b) < TINY)
            && (fabs(delc) <= eps*fabs(*c) || fabs(*c) < TINY)) {
            free_vector(rem,0,n);
            free_vector(qq,0,n);
            free_vector(q,0,n);
            return;
        }
    }
    nrerror("Too many iterations in routine qroot");
}

```

At most ITMAX iterations.

First division r,s.

Second division partial r,s with respect to c.

Solve 2x2 equation.

Coefficients converged.

We have already remarked on the annoyance of having two tentative roots collapse to one value under polishing. You are left not knowing whether your polishing procedure has lost a root, or whether there *is* actually a double root, which was split only by roundoff errors in your previous deflation. One solution is deflate-and-repolish; but deflation is what we are trying to avoid at the polishing stage. An alternative is *Maehly's procedure*. Maehly pointed out that the derivative of the reduced polynomial

$$P_j(x) \equiv \frac{P(x)}{(x-x_1) \cdots (x-x_j)} \quad (9.5.27)$$

can be written as

$$P'_j(x) = \frac{P'(x)}{(x-x_1) \cdots (x-x_j)} - \frac{P(x)}{(x-x_1) \cdots (x-x_j)} \sum_{i=1}^j (x-x_i)^{-1} \quad (9.5.28)$$

Hence one step of Newton-Raphson, taking a guess x_k into a new guess x_{k+1} , can be written as

$$x_{k+1} = x_k - \frac{P(x_k)}{P'(x_k) - P(x_k) \sum_{i=1}^j (x_k - x_i)^{-1}} \quad (9.5.29)$$

This equation, if used with i ranging over the roots already polished, will prevent a tentative root from spuriously hopping to another one's true root. It is an example of so-called *zero suppression* as an alternative to true deflation.

Muller's method, which was described above, can also be useful at the polishing stage.

CITED REFERENCES AND FURTHER READING:

- Acton, F.S. 1970, *Numerical Methods That Work*, 1990, corrected edition (Washington: Mathematical Association of America), Chapter 7. [1]
 Peters G., and Wilkinson, J.H. 1971, *Journal of the Institute of Mathematics and its Applications*, vol. 8, pp. 16–35. [2]
IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [3]
 Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.9–8.13. [4]
 Adams, D.A. 1967, *Communications of the ACM*, vol. 10, pp. 655–658. [5]
 Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §4.4.3. [6]
 Henrici, P. 1974, *Applied and Computational Complex Analysis*, vol. 1 (New York: Wiley).
 Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§5.5–5.9.

9.6 Newton-Raphson Method for Nonlinear Systems of Equations

We make an extreme, but wholly defensible, statement: There are *no* good, general methods for solving systems of more than one nonlinear equation. Furthermore, it is not hard to see why (very likely) there *never will be* any good, general methods: Consider the case of two dimensions, where we want to solve simultaneously

$$\begin{aligned} f(x, y) &= 0 \\ g(x, y) &= 0 \end{aligned} \quad (9.6.1)$$

The functions f and g are two arbitrary functions, each of which has zero contour lines that divide the (x, y) plane into regions where their respective function is positive or negative. These zero contour boundaries are of interest to us. The solutions that we seek are those points (if any) that are common to the zero contours of f and g (see Figure 9.6.1). Unfortunately, the functions f and g have, in general, no relation to each other at all! There is nothing special about a common point from either f 's point of view, or from g 's. In order to find all common points, which are

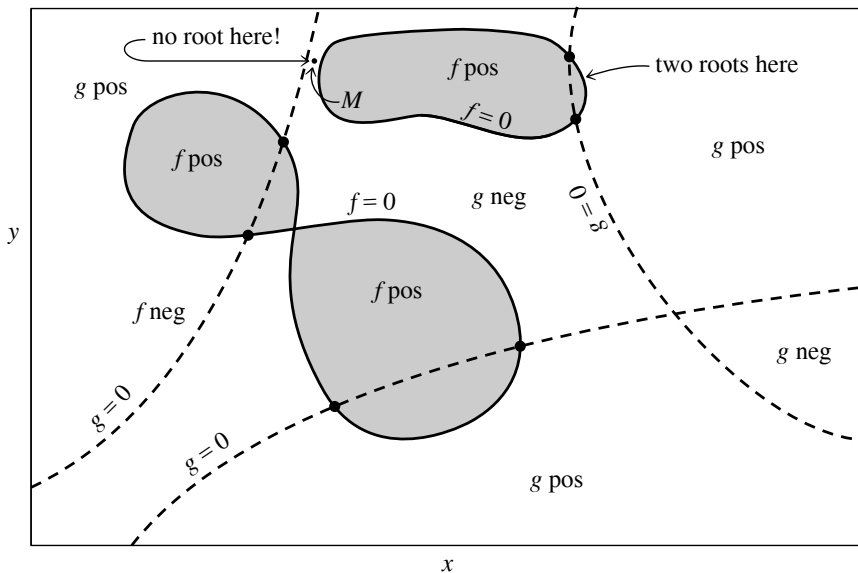


Figure 9.6.1. Solution of two nonlinear equations in two unknowns. Solid curves refer to $f(x, y)$, dashed curves to $g(x, y)$. Each equation divides the (x, y) plane into positive and negative regions, bounded by zero curves. The desired solutions are the intersections of these unrelated zero curves. The number of solutions is *a priori* unknown.

the solutions of our nonlinear equations, we will (in general) have to do neither more nor less than map out the full zero contours of both functions. Note further that the zero contours will (in general) consist of an unknown number of disjoint closed curves. How can we ever hope to know when we have found all such disjoint pieces?

For problems in more than two dimensions, we need to find points mutually common to N unrelated zero-contour hypersurfaces, each of dimension $N - 1$. You see that root finding becomes virtually impossible without insight! You will almost always have to use additional information, specific to your particular problem, to answer such basic questions as, “Do I expect a unique solution?” and “Approximately where?” Acton [1] has a good discussion of some of the particular strategies that can be tried.

In this section we will discuss the simplest multidimensional root finding method, Newton-Raphson. This method gives you a very efficient means of converging to a root, if you have a sufficiently good initial guess. It can also spectacularly fail to converge, indicating (though not proving) that your putative root does not exist nearby. In §9.7 we discuss more sophisticated implementations of the Newton-Raphson method, which try to improve on Newton-Raphson’s poor global convergence. A multidimensional generalization of the secant method, called Broyden’s method, is also discussed in §9.7.

A typical problem gives N functional relations to be zeroed, involving variables $x_i, i = 1, 2, \dots, N$:

$$F_i(x_1, x_2, \dots, x_N) = 0 \quad i = 1, 2, \dots, N. \quad (9.6.2)$$

We let \mathbf{x} denote the entire vector of values x_i and \mathbf{F} denote the entire vector of functions F_i . In the neighborhood of \mathbf{x} , each of the functions F_i can be expanded

in Taylor series

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^N \frac{\partial F_i}{\partial x_j} \delta x_j + O(\delta\mathbf{x}^2). \quad (9.6.3)$$

The matrix of partial derivatives appearing in equation (9.6.3) is the *Jacobian* matrix \mathbf{J} :

$$J_{ij} \equiv \frac{\partial F_i}{\partial x_j}. \quad (9.6.4)$$

In matrix notation equation (9.6.3) is

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2). \quad (9.6.5)$$

By neglecting terms of order $\delta\mathbf{x}^2$ and higher and by setting $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$, we obtain a set of linear equations for the corrections $\delta\mathbf{x}$ that move each function closer to zero simultaneously, namely

$$\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{F}. \quad (9.6.6)$$

Matrix equation (9.6.6) can be solved by *LU* decomposition as described in §2.3. The corrections are then added to the solution vector,

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta\mathbf{x} \quad (9.6.7)$$

and the process is iterated to convergence. In general it is a good idea to check the degree to which both functions and variables have converged. Once either reaches machine accuracy, the other won't change.

The following routine `mnewt` performs `ntrial` iterations starting from an initial guess at the solution vector `x[1..n]`. Iteration stops if either the sum of the magnitudes of the functions F_i is less than some tolerance `tolf`, or the sum of the absolute values of the corrections to δx_i is less than some tolerance `tolx`. `mnewt` calls a user supplied function `usrfun` which must provide the function values \mathbf{F} and the Jacobian matrix \mathbf{J} . If \mathbf{J} is difficult to compute analytically, you can try having `usrfun` call the routine `fdjac` of §9.7 to compute the partial derivatives by finite differences. You should not make `ntrial` too big; rather inspect to see what is happening before continuing for some further iterations.

```
#include <math.h>
#include "nrutil.h"

void usrfun(float *x,int n,float *fvec,float **fjac);
#define FREERETURN {free_matrix(fjac,1,n,1,n);free_vector(fvec,1,n);\
    free_vector(p,1,n);free_ivector(indx,1,n);return;}

void mnewt(int ntrial, float x[], int n, float tolx, float tolf)
Given an initial guess x[1..n] 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.
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
```

```

int k,i,*indx;
float errx,errf,d,*fvec,**fjac,*p;

indx=ivector(1,n);
p=vector(1,n);
fvec=vector(1,n);
fjac=matrix(1,n,1,n);
for (k=1;k<=ntrial;k++) {
    usrfun(x,n,fvec,fjac);           User function supplies function values at x in
    errf=0.0;                       fvec and Jacobian matrix in fjac.
    for (i=1;i<=n;i++) errf += fabs(fvec[i]);   Check function convergence.
    if (errf <= tolf) FREERETURN
    for (i=1;i<=n;i++) p[i] = -fvec[i];       Right-hand side of linear equations.
    ludcmp(fjac,n,indx,&d);                 Solve linear equations using LU decomposition.
    lubksb(fjac,n,indx,p);
    errx=0.0;                               Check root convergence.
    for (i=1;i<=n;i++) {                   Update solution.
        errx += fabs(p[i]);
        x[i] += p[i];
    }
    if (errx <= tolx) FREERETURN
}
FREERETURN
}

```

Newton's Method versus Minimization

In the next chapter, we will find that there *are* efficient general techniques for finding a minimum of a function of many variables. Why is that task (relatively) easy, while multidimensional root finding is often quite hard? Isn't minimization equivalent to finding a zero of an N -dimensional gradient vector, not so different from zeroing an N -dimensional function? No! The components of a gradient vector are not independent, arbitrary functions. Rather, they obey so-called integrability conditions that are highly restrictive. Put crudely, you can always find a minimum by sliding downhill on a single surface. The test of "downhillness" is thus one-dimensional. There is no analogous conceptual procedure for finding a multidimensional root, where "downhill" must mean simultaneously downhill in N separate function spaces, thus allowing a multitude of trade-offs, as to how much progress in one dimension is worth compared with progress in another.

It might occur to you to carry out multidimensional root finding by collapsing all these dimensions into one: Add up the sums of squares of the individual functions F_i to get a master function F which (i) is positive definite, and (ii) has a global minimum of zero exactly at all solutions of the original set of nonlinear equations. Unfortunately, as you will see in the next chapter, the efficient algorithms for finding minima come to rest on global and local minima indiscriminately. You will often find, to your great dissatisfaction, that your function F has a great number of local minima. In Figure 9.6.1, for example, there is likely to be a local minimum wherever the zero contours of f and g make a close approach to each other. The point labeled M is such a point, and one sees that there are no nearby roots.

However, we will now see that sophisticated strategies for multidimensional root finding can in fact make use of the idea of minimizing a master function F , by *combining* it with Newton's method applied to the full set of functions F_i . While 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

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

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.

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}_{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}_{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}_{new} to be greater than some fraction β of the rate of decrease of f at \mathbf{x}_{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 $\lambda_{\min} = 0.1$.

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 `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  $\Delta x$ .

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;
    for (i=1;i<=n;i++) {
        Compute  $\lambda_{\min}$ .
```

```

    temp=fabs(p[i])/FMAX(fabs(xold[i]),1.0);
    if (temp > test) test=temp;
}
alamin=TOLX/test;
alam=1.0;
for (;;) {
    for (i=1;i<=n;i++) x[i]=xold[i]+alam*p[i];
    *f=(*func)(x);
    if (alam < alamin) {
        for (i=1;i<=n;i++) x[i]=xold[i];
        *check=1;
        return;
    } else if (*f <= fold+ALF*alam*slope) return;
    else {
        if (alam == 1.0)
            tmplam = -slope/(2.0*(f-fold-slope));
        else {
            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;
        }
    }
    alam2=alam;
    f2 = *f;
    alam=FMAX(tmplam,0.1*alam);
}

```

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

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.

```

#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 δx ; STPMX is the scaled maximum step length allowed in line searches.

```

int nn;
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;}

```

Global variables to communicate with `fmin`.


```

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);
    nn=n;
    nrfuncv=vecfunc;
    f=fmin(x);
    test=0.0;
    for (i=1;i<=n;i++)
        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]);
    stpmax=STPMX*FMAX(sqrt(sum),(float)n);
    for (its=1;its<=MAXITS;its++) {
        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++) {
            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];
        fold=f;
        for (i=1;i<=n;i++) p[i] = -fvec[i];
        ludcmp(fjac,n,indx,&d);
        lubksb(fjac,n,indx,p);
        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;
        for (i=1;i<=n;i++)
            if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
        if (test < TOLF) {
            *check=0;
            FREERETURN
        }
        if (*check) {
            test=0.0;
            den=FMAX(f,0.5*n);
            for (i=1;i<=n;i++) {
                temp=fabs(g[i])*FMAX(fabs(x[i]),1.0)/den;

```

Define global variables.

fvec is also computed by this call.

Test for initial guess being a root. Use more stringent test than simply TOLF.

Calculate stpmax for line searches.

Start of iteration loop.

Compute ∇f for the line search.

Store x, and f.

Right-hand side for linear equations.

Solve linear equations by LU decomposition.

Test for convergence on function values.

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

```

        if (temp > test) test=temp;
    }
    *check=(test < TOLMIN ? 1 : 0);
    FREERETURN
}
test=0.0;                                Test for convergence on  $\delta x$ .
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 formula.
    }
    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 \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} ,

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

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 $f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F}$ (cf. equation 9.7.5). Thus the line search algorithm can fail to return a suitable step if \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 \mathbf{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;
float *fvec;
void (*nrfuncv)(int n, float v[], float f[]);

void broydn(float x[], int n, int *check,
    void (*vecfunc)(int, float [], float []))
```

Global variables to communicate with `fmin`.

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,resttr,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);
nn=n;
nrfuncv=vecfunc;
f=fmin(x);
test=0.0;
for (i=1;i<=n;i++)
    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]);
stpmax=STPMX*FMAX(sqrt(sum),(float)n);
restrt=1;
for (its=1;its<=MAXITS;its++) {
    if (restrt) {
        fdjac(n,x,fvec,r,vecfunc);
        qrdcmp(r,n,c,d,&sing);
        if (sing) nrerror("singular Jacobian in broydn");
        for (i=1;i<=n;i++) {
            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++) {
            r[i][i]=d[i];
            for (j=1;j<=n;j++) r[i][j]=0.0;
        }
    } else {
        for (i=1;i<=n;i++) s[i]=x[i]-xold[i];
        for (i=1;i<=n;i++) {
            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++) {
            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;
            else w[i]=0.0;
        }
        if (!skip) {
            for (i=1;i<=n;i++) {
                for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*w[j];
                t[i]=sum;
            }

```

Define global variables.

The vector fvec is also computed by this call.

Test for initial guess being a root. Use more stringent test than simply TOLF.

Calculate stpmax for line searches.

Ensure initial Jacobian gets computed.

Start of iteration loop.

Initialize or reinitialize Jacobian in r.

QR decomposition of Jacobian.

Form Q^T explicitly.

Form R explicitly.

Carry out Broyden update.

$s = \delta x$.

$t = R \cdot s$.

$w = \delta F - B \cdot s$.

$t = Q^T \cdot w$.

```

        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 QT.
        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++) {                            Right-hand side for linear equations is  $-\mathbf{Q}^T \cdot \mathbf{F}$ .
    for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*fvec[j];
    p[i] = -sum;
}

for (i=n;i>=1;i--) {                            Compute  $\nabla f \approx (\mathbf{Q} \cdot \mathbf{R})^T \cdot \mathbf{F}$  for the line search.
    for (sum=0.0,j=1;j<=i;j++) sum -= r[j][i]*p[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.
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 Jacobian.
    else {
        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  $\delta \mathbf{x}$ .
    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 \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]