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,

$$[x - (a + ib)]\,[x - (a - ib)] = x^2 - 2ax + (a^2 + b^2) \tag{9.5.1}$$

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

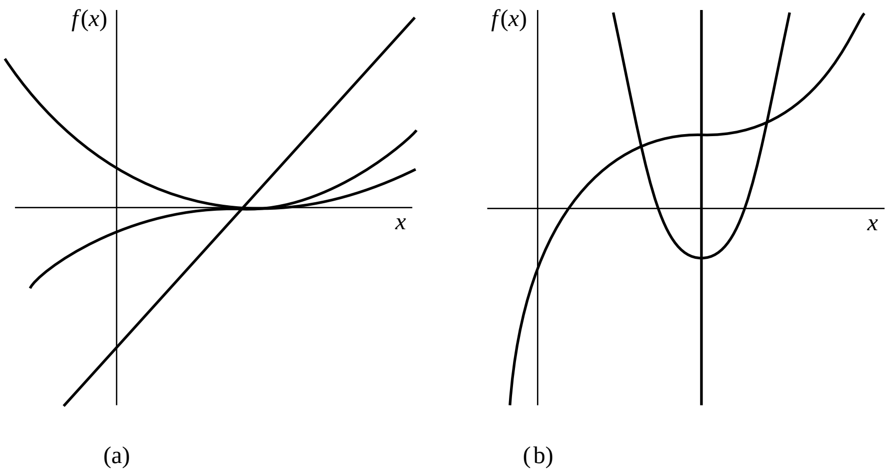

Figure 9.5.1. (a) Linear, quadratic, and cubic behavior at the roots of polynomials. Only under high magnification (b) does it become apparent that the cubic has one, not three, roots, and that the quadratic has two roots rather than none.

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 nth 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

$$P_n(x) = (x - x_1)(x - x_2)\dots(x - x_n) \tag{9.5.4}$$

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

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

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

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

$$x - x_1 = a \quad ; \quad x - x_i = b \quad i = 2, 3, \dots, n \tag{9.5.8}$$

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

$$\frac{1}{a} + \frac{n-1}{b} = G \tag{9.5.9}$$

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

which yields as the solution for a

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

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.
            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 occur-
        rence).
    }
    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 coeffi-
            laguer(a,m,&roots[j],&its);         cients.
    for (j=2;j<=m;j++) {                    Sort roots by their real parts by straight in-
        x=roots[j];                             sertion.
        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 **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*

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

is equivalent to the general polynomial

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

If the coefficients a_i are real, rather than complex, then the eigenvalues of **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, $\sum_{i=0}^{\mathtt{m}} \mathtt{a}(i)x^i$, 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]...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

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

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

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

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 zero.

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

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

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

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

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

which can be rewritten as

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

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

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

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 $R_1x + S_1$, then

$$\frac{\partial R}{\partial C} = -R_1 \qquad \frac{\partial S}{\partial C} = -S_1 \tag{9.5.21}$$

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

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

we get

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

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

Solve these two equations for the partial derivatives, using

$$x_+ + x_- = -B \qquad x_+ x_- = C \tag{9.5.25}$$

and find

$$\frac{\partial R}{\partial B} = BR_1 - S_1 \qquad \frac{\partial S}{\partial B} = CR_1 \tag{9.5.26}$$

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                                      At most ITMAX iterations.
#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];                                     First division r,s.
        r=rem[1];
        poldiv(q,(n-1),d,2,qq,rem);
        sb = -(*c)*(rc = -rem[1]);                    Second division partial r,s with respect to
        rb = -(*b)*rc+(sc = -rem[0]);                     c.
        dv=1.0/(sb*rc-sc*rb);                         Solve 2x2 equation.
        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);                     Coefficients converged.
            free_vector(qq,0,n);
            free_vector(q,0,n);
            return;
        }
    }
    nrerror("Too many iterations in routine qroot");
}
```

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

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

can be written as

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

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

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

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

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

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