ME 370B
Energy Systems II:
Modeling and Advanced Concepts

Chris Edwards

It is time to deal with Newton-Raphson. This is a powerful method—and really the only viable approach for many types of problems that you will encounter. As to speed, for the kinds of problems we deal with here, N-R can usually find the solution in 3-5 iterations in situations where bisection or false-position might take 10-50 iterations. As our procedures become increasingly complex, there is just no substitute.

If you are already familiar with N-R then all of this is obvious and you can just get on to using it. (Or you may have already been using it.) If you are not, there is a short excerpt from *Numerical Recipes* in this PDF file. Have a look at that to see the basic idea.

There are a couple of adjustments to the storyline from *Numerical Recipes* that are important to know. This first is that they (Press, Flannery, etc.) are mainly concerned with finding roots for analytical functions. We on the other hand are concerned with finding solutions that more often than not involve *procedures*. The distinction is that they have an equation and want to know where it is satisfied. We have a system of interlinked equations which must all be satisfied together and often with several implicit solution stages. As such it requires a sequence of steps to find the solution, not just manipulation of an equation. The reason this is important is that it leads to their pooh-poohing of the use of numerical derivatives in N-R in favor of analytical derivatives. Of course analytical derivatives are preferable—they are exact and cheap to compute. But they are not available in complex procedure solutions, so we must use numerical derivatives. This actually doesn't cost us that much—usually one or two more N-R iterations than with analytical derivatives—and it does allow us to use the fast convergence of the basic N-R approach. Note that in choosing the order of derivative to compute (central difference vs. forward, for example) it is not the derivative that we really want to know—it is the direction and magnitude of step that is important for the method. As such, you will often want to use first-order derivatives (requiring only one additional function evaluation) rather than second-order (requiring two more) as you would if you really wanted a good estimate of the slope.

The other thing to know about N-R is that you must ride-herd on it. It will try to run away from you. That is OK as long as you watch it and pull it back. This can be done by keeping track of where your solution is supposed to be (for example, between zero and unity mole fraction), and never letting N-R outside of the fences. My basic advice is to let N-R try to solve the problem—from the existing solution, ask N-R which way it wants to go and by how much—but then check to make sure: (1) that it is still within bounds, and (2) that the solution is really better at the new location than it was at the old one. The first of these provisions (bound checking) can be combined with bisection should you decide you don't like what N-R wants to do. The second leads to the idea of backtracking—accepting the direction that N-R wants to go, but progressively shorting the step size (I like halving it) until you get an answer that is improved—before actually accepting the new value of the iterate.

If you are wondering why I am talking about direction and step size when most of our problems are one-dimensional (so direction amounts to +/-), it is because everything I have said applies in multiple dimensions (as does N-R), so you can think the whole problem through in terms of a unit vector that sets the direction of decent towards the solution and whether you should take the full Newton step or backtrack along that vector to find an improved solution.

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,

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

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

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

Newton-Raphson is not restricted to one dimension. The method readily generalizes to multiple dimensions, as we shall see in §9.6 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. Like most powerful tools, Newton-Raphson can be destructive 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:

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

By the Newton-Raphson formula,

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

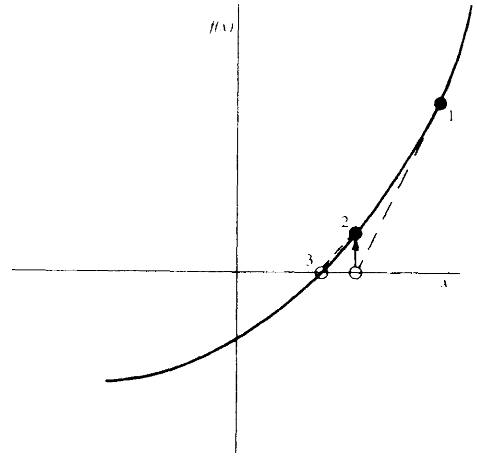

Figure 9.4.1. Newton's method extrapolates the local derivative to find the next estimate of the root. In this example it works well and converges quadratically.

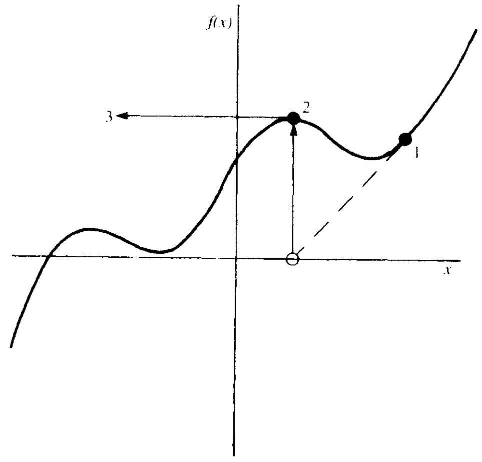

Figure 9.4.2. Unfortunate case where Newton's method encounters a local extremum and shoots off to outer space. Here bracketing bounds, as in **rtsafe**, would save the day.

so that

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

When a trial solution x_i differs from the true root by ϵ_i, we can use (9.4.3) to

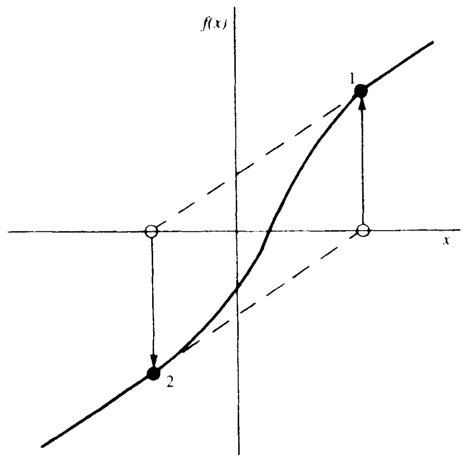

Figure 9.4.3. Unfortunate case where Newton's method enters a nonconvergent cycle. This behavior is often encountered when the function f is obtained, in whole or in part, by table interpolation. With a better initial guess, the method would have succeeded.

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

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

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 which 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,

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

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.)

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 only used 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(funcd,x1,x2,xacc)
float x1,x2,xacc;
void (*funcd)();    /* ANSI: void (*funcd)(float,float *,float *); */
```
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 provides both the function value and the first derivative of the function at the point x.
```
{
    int j;
    float df,dx,f,rtn;
    void nrerror();

    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)
            nrerror("Jumped out of brackets in RTNEWT");
        if (fabs(dx) < xacc) return rtn;    Convergence.
    }
    nrerror("Maximum number of iterations exceeded in RTNEWT");
}
```

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(funcd,x1,x2,xacc)
float x1,x2,xacc;
```

```
void (*funcd)();     /* ANSI: void (*funcd)(float,float *,float *); */
```

Using a combination of Newton-Raphson and bisection, find the root of a function bracketed between x1 and x2. The root, returned as the function value **rtsafe**, will be refined until its accuracy is known within ±**xacc**. **funcd** is a user-supplied routine that provides both the function value and the first derivative of the function.

```
{
    int j;
    float df,dx,dxold,f,fh,fl;
    float temp,xh,xl,rts;
    void nrerror();

    (*funcd)(x1,&fl,&df);
    (*funcd)(x2,&fh,&df);
    if (fl*fh >= 0.0) nrerror("Root must be bracketed in RTSAFE");
    if (fl < 0.0) {                    Orient the search so that f(xl) < 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 "step-size 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");
}
```

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.)

REFERENCES AND FURTHER READING:

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row), Chapter 2.

Ralston, Anthony, and Rabinowitz, Philip. 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).

9.5 Roots of Polynomials

Here we present a few methods for finding roots of polynomials. These will serve for most practical problems involving polynomals 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 in Acton's book.)

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

Henrici, P. 1974, *Applied and Computational Complex Analysis*, vol. 1 (New York: Wiley).

Peters G., and Wilkinson, J.H. 1971, *J. Inst. Math. Appl.*, vol. 8, pp. 16-35.

IMSL Library Reference Manual, 1980, ed. 8 (IMSL Inc., 7500 Bellaire Boulevard, Houston TX 77036).

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) which 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 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 hyperplanes, 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 has a good discussion of some of the particular strategies that can be tried.

Once, however, you identify the neighborhood of a root, or of a place where there *might* be a root, then the problem firms up considerably: It is time to turn to Newton-Raphson, which readily generalizes to multiple dimensions. This method gives you a very efficient means of converging to

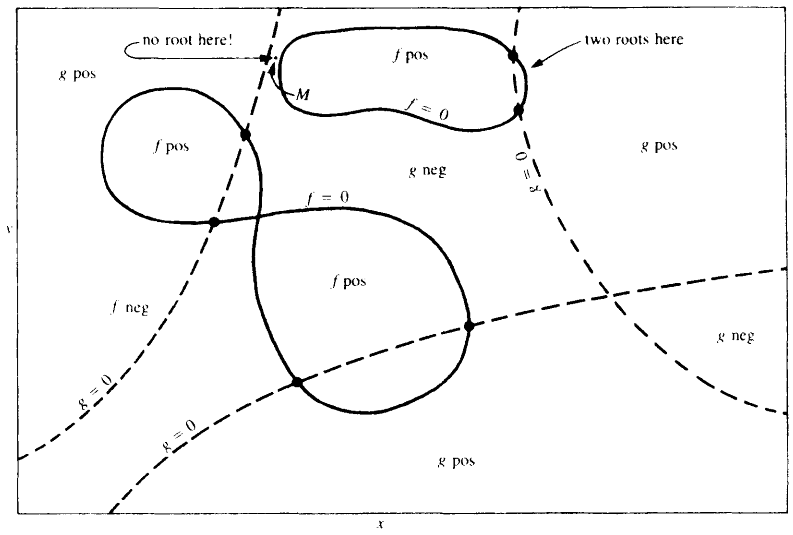

Figure 9.6.1. Solution of two nonlinear equations in two unknowns. Solid curves refer to $f(x,y)$, dashed curves to $g(x,y)$. Each equation divides the (x,y) plane into positive and negative regions, bounded by zero curves. The desired solutions are the intersections of these unrelated zero curves. The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

the root, if it exists, or of spectacularly failing to converge, indicating (though not proving) that your putative root does not exist nearby.

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

$$f_i(x_1,x_2,\ldots,x_N) = 0 \qquad i = 1,2,\ldots,N. \tag{9.6.2}$$

If we let $\mathbf{X}$ denote the entire vector of values x_i then, in the neighborhood of $\mathbf{X}$, each of the functions f_i can be expanded in Taylor series

$$f_i(\mathbf{X}+\delta\mathbf{X}) = f_i(\mathbf{X}) + \sum_{j=1}^{N} \frac{\partial f_i}{\partial x_j}\delta x_j + O(\delta\mathbf{X}^2). \tag{9.6.3}$$

By neglecting terms of order $\delta\mathbf{X}^2$ and higher, we obtain a set of linear equations for the corrections $\delta\mathbf{X}$ that move each function closer to zero simultaneously, namely

$$\sum_{j=1}^{N} \alpha_{ij}\delta x_j = \beta_i, \tag{9.6.4}$$

where

$$\alpha_{ij} \equiv \frac{\partial f_i}{\partial x_j} \qquad \beta_i \equiv -f_i \tag{9.6.5}$$

Matrix equation (9.6.4) can be solved by LU decomposition as described in §2.3. The corrections are then added to the solution vector,

$$x_i^{new} = x_i^{old} + \delta x_i \quad i = 1, \dots, N \tag{9.6.6}$$

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 matrix of partial derivatives α, and the negative of the function values β, as defined in (9.6.5). You should not make `ntrial` too big; rather inspect to see what is happening before continuing for some further iterations.

```
#include <math.h>

#define FREERETURN {free_matrix(alpha,1,n,1,n);free_vector(bet,1,n);\
    free_ivector(indx,1,n);return;}

void mnewt(ntrial,x,n,tolx,tolf)
int ntrial,n;
float x[],tolx,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`.

```
{
    int k,i,*indx,*ivector();
    float errx,errf,d,*bet,**alpha,*vector(),**matrix();
    void usrfun(),ludcmp(),lubksb(),free_ivector(),free_vector(),
        free_matrix();

    indx=ivector(1,n);
    bet=vector(1,n);
    alpha=matrix(1,n,1,n);
    for (k=1;k<=ntrial;k++) {
        usrfun(x,alpha,bet);              User function supplies matrix coefficients.
        errf=0.0;                         Check function convergence.
        for (i=1;i<=n;i++) errf += fabs(bet[i]);
        if (errf <= tolf) FREERETURN
        ludcmp(alpha,n,indx,&d);          Solve linear equations using LU decomposition.
        lubksb(alpha,n,indx,bet);
        errx=0.0;                         Check root convergence.
        for (i=1;i<=n;i++) {              Update solution.
            errx += fabs(bet[i]);
            x[i] += bet[i];
```

```
        }
        if (errx <= tolx) FREERETURN
    }
    FREERETURN
}
```

Multidimensional Root Finding Versus Multidimensional 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.

A popular idea for multidimensional root finding is to collapse all these dimensions into one, by adding 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 almost surely 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.

REFERENCES AND FURTHER READING:

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row), Chapter 14.

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).