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);                   Note which row corresponds to 0.
        for (i=1;i<=ISCR;i++) {                  Place an indicator at function height and
            scr[i][jz]=ZERO;                         0.
            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

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

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 $x \to \pm\infty$.