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:

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

\[
f(x) = \frac{1}{x - c}
\]  

(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 \(x \to \pm \infty\).
9.1 Bracketing and Bisection

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.
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 \( x_1-x_2 \) subdivide the interval into n equally
spaced segments, and search for zero crossings of the function. nb is input as the maximum
number 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++) { // Loop over all intervals
        fc=(*fx)(x += dx);
        if (fc*fp <= 0.0) { // If a sign change occurs then record values for the
            xb1[++nbb]=x-dx;
            xb2[nbb]=x;
            if(*nb == nbb) return;
        }
        fp=fc;
    }
    *nb = nbb;
}
9.1 Bracketing and Bisection

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 \( \epsilon_n \), then after the next iteration it will be bracketed within an interval of size

\[
\epsilon_{n+1} = \frac{\epsilon_n}{2}
\]

(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,

\[
n = \log_2 \frac{\epsilon_0}{\epsilon}
\]

(9.1.3)

where \( \epsilon_0 \) is the size of the initially bracketing interval, \( \epsilon \) 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,

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

(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 \( \epsilon \) 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 \( 2^{-40} \approx 10^{-12} \).
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

$$\lim_{k \to \infty} |\epsilon_{k+1}| \approx \text{const} \times |\epsilon_k|^{1.618}$$  \hspace{1cm}(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.