7
Trial and Error Searching

Many computer techniques, such as multiplying matrices, follow a well-defined set procedure. The same basic operation is repeated for each element in the matrix, and stops when we have covered all the elements. In contrast, some computational techniques are trial-and-error algorithms in which the program goes through some numerical procedure, and quits only when the error is acceptably small. (We already did some of this when we summed a power series until the terms became small.) This type of programming is usually interesting because we must think hard in order to foresee how to have the computer act intelligently for all possible situations. These "trial and error" programs are also interesting to run because, like experiments, they depend very much on the initial conditions, and it is hard to predict exactly what the computer will come up as it searches for a solution.

7.1
Quantum States in Square Well (Problem IA)

Maybe the most standard problem in quantum mechanics is to solve for the energies of a particle of mass m bound within a 1D square well of radius a:[1]

$$V(x) = \begin{cases} -V_0 & \text{for } |x| \le a \\ 0 & \text{for } |x| \ge a \end{cases} \tag{7.1}$$

As shown in quantum mechanics texts [8], the energies of the bound states $E = -E_B < 0$ within this well are solutions of the transcendental equations

$$\sqrt{2m(V_0 - E_B)}\ \tan\left[a\sqrt{2m(V_0 - E_B)}\right] = \sqrt{2mE_B} \qquad \text{(even)}, \tag{7.2}$$

$$\sqrt{2m(V_0 - E_B)}\ \text{cotan}\left[a\sqrt{2m(V_0 - E_B)}\right] = \sqrt{2mE_B} \qquad \text{(odd)} \tag{7.3}$$

[1] We solve this same problem in Chap. 16 using an approach that is applicable to most any potential, and which also provides the wave functions. The approach of this section only works for the eigenenergies of a square well.

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

ISBN: 978-3-527-40626-5

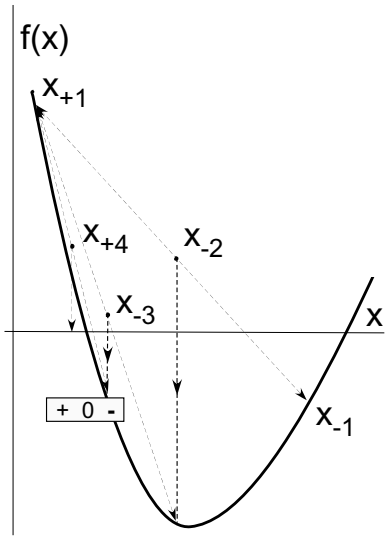

Fig. 7.1 A graphical representation of the steps involved in solving for a zero of $f(x)$ using the bisection algorithm. The bisection algorithm takes the midpoint of the interval as the new guess for x, and so each step reduces the interval size by one half. Four steps are shown.

where even and odd refer to the symmetry of the wave function, and where we have chosen units such that $\hbar = 1$. To have a numerical problem to solve, we set $2m = 1$, $a = 1$, and $V_0 = 10$, in which case there results

$$\sqrt{10 - E_B}\ \tan\left(\sqrt{10 - E_B}\right) = \sqrt{E_B} \qquad \text{(even)} \tag{7.4}$$

$$\sqrt{10 - E_B}\ \text{cotan}\left(\sqrt{10 - E_B}\right) = \sqrt{E_B} \qquad \text{(odd)} \tag{7.5}$$

Your **problem** is to

1. Find several bound state energies E_B for even wave functions, that is, solution of (7.4).

2. Notice that the "10" in these equations is proportional to the strength of the potential that causes the binding. See if making the potential deeper, say by changing the 10 to a 20 or a 30, produces a larger number of, or deeper bound states.

3. Although we have yet to discuss the methods of solution, after we have, we want you to compare a solution found with the Newton–Raphson algorithms to one found with the bisection algorithm.

7.2 Trial-and-Error Root Finding via Bisection Algorithm

The trial-and-error technique for root finding looks for a solution x of the equation $f(x) = 0$. Having the RHS $= 0$ is just a convention. Any equation such as $10 \sin x = 3x^3$ can easily be converted to $10 \sin x - 3x^3 = 0$. The general procedure is the one in which we start with a guess value for x, substitute that guess into $f(x)$ (the "trial"), and then see how far the RHS is from zero (the "error"). You then change x appropriately (a new guess) and try it out in $f(x)$. The procedure continues until $f(x) \simeq 0$ to some desired level of precision, or until the changes in x are insignificant. As a safeguard, you will also want to set a maximum value for the number of trials.

The most elementary trial-and-error technique is the *bisection algorithm*. It is reliable, but slow. If you know some interval in which $f(x)$ changes sign, then the bisection algorithm will always converge to the root by finding progressively smaller and smaller intervals in which the zero occurs. Other techniques, such as the Newton–Raphson method we describe next, may converge more quickly, but if the initial guess is not good, it may become unstable and fail completely.

The basis behind the bisection algorithm is shown in Fig. 7.1. We start with two values of x, between which we know a zero occurs. (You can determine these by making a graph or by stepping through different x values and looking for a sign change.) To be specific, let us say that $f(x)$ is negative at x_- and positive at x_+:

$$f(x_-) < 0 \qquad f(x_+) > 0 \tag{7.6}$$

Note that it may well be that $x_- > x_+$ if the function changes from positive to negative as x increases. Thus we start with the interval $x_+ \leq x \leq x_-$ within which we know a zero occurs. The algorithm then bisects this interval at

$$x = \frac{x_+ + x_-}{2} \tag{7.7}$$

and selects as its new interval the half in which the sign change occurs:

```
if ( f(x) f(xPlus)  > 0 ) xPlus = x
    else xMinus = x
```

The process continues until the value of $f(x)$ is less than a predefined level of precision, or until a predefined (large) number of subdivisions has occurred.

The example in Fig. 7.1 shows the first interval extending from $x_- = x_{+1}$ to $x_+ = x_{-1}$. We then bisect that interval at x, and since $f(x) < 0$ at the midpoint, we set $x_- \equiv x_{-2} = x$ and label it as x_{-2} to indicate the second step. We then use $x_{+2} \equiv x_{+1}$ and x_{-2} as the next interval and continue the process. We see that only x_- changes for the first three steps in this example, but that

for the fourth step, x_+ finally changes. The changes then get too small for us to show.

7.2.1 Bisection Algorithm Implementation

1. The first step in implementing any search algorithm is to get an idea what your function looks like. For the present problem you do this by making a plot of $f(E) = \sqrt{10 - E_B}\tan(\sqrt{10 - E_B}) - \sqrt{E_B}$ versus E_B. Note from your plot some approximate values at which $f(E_B) = 0$. Your program should be able to find more exact values for these zeros.
2. Write a program that implements the bisection algorithm and use it to find some solutions of (7.4).
3. *Warning:* because the tan function has singularities, you have to be careful. In fact, your graphics program (or Maple) may not function accurately near these singularities. One cure is to use a different, but equivalent form of the equation. Show that an equivalent form of (7.4) is
$$\sqrt{E}\cot(\sqrt{10 - E}) - \sqrt{10 - E} = 0 \tag{7.8}$$
4. Make a second plot of (7.8), which also has singularities, but at different places. Choose some approximate locations for zeros from this plot.
5. Compare the roots you find with those given by *Maple* or *Mathematica*.

7.3 Newton–Raphson (Faster Algorithm)

The Newton–Raphson algorithm finds approximate roots of the equation

$$f(x) = 0 \tag{7.9}$$

more quickly than the bisection method. As we see graphically in Fig. 7.2, this algorithm is the equivalent of drawing a straight line $f(x) \simeq mx + b$ tangent to the curve at an x value for which $f(x) \simeq 0$, and then using the intercept $x = -b/m$ of that line with the x axis as an improved guess for the root. If the "curve" were a straight line, the answer would be exact; otherwise it is a good approximation if the guess is close enough to the root for $f(x)$ to be nearly linear. The process continues until some set level of precision is reached. If a guess is in a region where $f(x)$ is nearly linear (Fig. 7.2), then the convergence is much more rapid than the bisection algorithm.

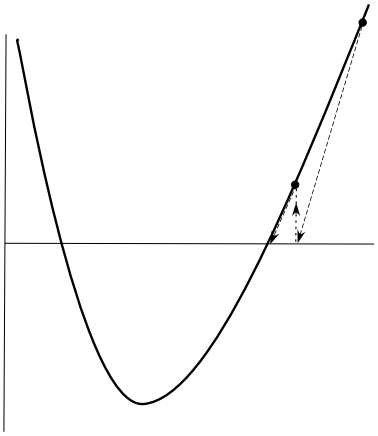

Fig. 7.2 A graphical representation of the steps involved in solving for a zero of $f(x)$ using the Newton–Raphson method. The Newton–Raphson method takes the new guess as the zero of the line tangent to $f(x)$ at the old guess. Because of the rapid convergence, only two steps can be shown.

The analytic formulation of the Newton–Raphson algorithm starts with an old guess x_0, and expresses the new guess x as a correction Δx to the old guess:

$$x_0 = \text{old guess} \qquad \Delta x = \text{unknown correction} \tag{7.10}$$

$$\Rightarrow \qquad x = x_0 + \Delta x = \text{(unknown) new guess.} \tag{7.11}$$

We next expand the known function $f(x)$ in a Taylor series around x_0 and keep only the linear terms:

$$f(x = x_0 + \Delta x) \simeq f(x_0) + \left.\frac{df}{dx}\right|_{x_0} \Delta x \tag{7.12}$$

We then determine the correction Δx by determining the point at which this linear approximation to $f(x)$ crosses the x axis:

$$f(x_0) + \left.\frac{df}{dx}\right|_{x_0} \Delta x = 0 \qquad \Rightarrow \qquad \Delta x = -\frac{f(x_0)}{df/dx|_{x_0}} \tag{7.13}$$

The procedure is repeated starting at the improved x until some set level of precision is obtained.

The Newton–Raphson algorithm (7.13) requires evaluation of the derivative df/dx at each value of x_0. In many cases you may have an analytic expression for the derivative and can built that into the algorithm. However, and especially for more complicated problems, it is simpler and less error prone to use a numerical, forward-difference approximation to the derivative:[2]

$$\frac{df}{dx} \simeq \frac{f(x + \delta x) - f(x)}{\delta x} \tag{7.14}$$

[2] We discuss numerical differentiation in Chap. 6.

where δx is some small change in x that you just chose (different from the Δx in (7.13)), the exact value not mattering once a solution is found. While a central-difference approximation for the derivative would be more accurate, it would require additional evaluations of the f's, and once you find a zero it does not matter how you got there. On the CD we give the programs `Newton_cd.java` (also Listing 7.1) and `Newton_fd.java`, which implement the derivative both ways.

Listing 7.1: The program `Newton_cd.java` uses the Newton–Raphson method to search for a zero of the function $f(x)$. A central-difference approximation is used to determine df/dx.

```
// Newton_cd.java: Newton–Raphson root finder, central diff deriv

public class Newton_cd  {

  public static double f(double x)       // Find zero of this function
   { return 2*Math.cos(x) – x; }

  public static void main(String[] argv) {

    double x = 2., dx = 1e–2, F= f(x), eps = 1e–6, df;
    int it, imax = 100;              // Max no of iterations permitted
                                                          // Iterate
    for ( it = 0;  it <= imax;  it++ )  {
      System.out.println("Iteration # = "+it+" x = "+x+" f(x) = "+F);
                                                  // Central diff deriv
      df = ( f(x + dx/2) – f(x–dx/2) )/dx;
      dx = –F/df;
      x += dx;                                                // New guess
      F = f(x);                                             // Save for use
                                              // Check for convergence
      if ( Math.abs(F) <= eps )  {
         System.out.println("Root found, tolerance eps = " + eps);
         break;
} } } }
```

7.3.1 Newton–Raphson with Backtracking

Two examples of possible problems with Newton-Raphson are shown in Fig. 7.3. On the left we see a case where the search takes us to an x value where the function has a local minimum or maximum, that is, where $df/dx = 0$. Because $\Delta x = -f/f'$, this leads to a horizontal tangent (division by zero), and so the next guess is $x = \infty$, from where it is hard to return. When this happens, you need to start your search off with a different guess and pray that you do not fall into this trap again. In those cases where the correction is very large, but maybe not infinite, you may want to try backtracking (described below) and hope that by taking a smaller step you will not get into as much trouble.

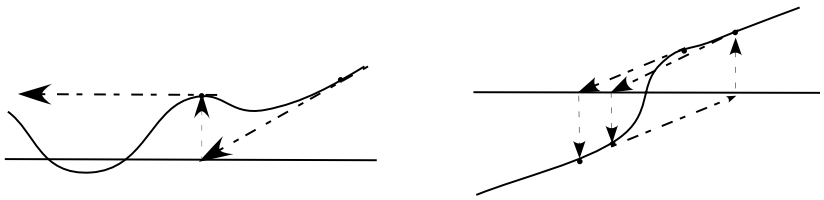

Fig. 7.3 Two examples of how Newton–Raphson may fail if the initial guess is not in the region where $f(x)$ can be approximated by a straight line. *Left:* A guess lands at a local minimum/maximum, that is, a place where the derivative vanishes, and so the next guess ends up at $x = \infty$. *Right:* The search has fallen into an infinite loop.

On the right of Fig. 7.3 we see a case where a search falls into an infinite loop surrounding the zero, without ever getting there. A solution to this problem is called *backtracking*. As the name implies, it says that in those cases where the new guess $x_0 + \Delta x$ leads to an increase in the magnitude of the function, $|f(x_0 + \Delta x)|^2 > |f(x_0)|^2$, you should backtrack somewhat and try a smaller guess, say $x_0 + \Delta x/2$. If the magnitude of f still increases, then you just need to backtrack some more, say by trying $x_0 + \Delta x/4$ as your next guess, and so forth. Because you know that the tangent line leads to a local decrease in $|f|$, eventually an acceptable small enough step should be found.

The problem in both these cases is that the initial guesses were not close enough to the regions where $f(x)$ is approximately linear. So again, a good plot may help produce a good first guess. Alternatively, you may want to start off your search with the bisection algorithm, and then switch to the faster Newton-Raphson when you get closer to the zero.

7.3.2 Newton–Raphson Implementation

1. Use the Newton–Raphson algorithm to find some energies E_B that are solutions of (7.4). Compare this solution with the one found with the bisection algorithm.

2. Again, notice that the "10" in this equation is proportional to the strength of the potential that causes the binding. See if making the potential deeper, say by changing the 10 to a 20 or a 30, produces more or deeper bound states. (Note that in contrast to the bisection algorithm, your initial guess must be close to the answer for Newton–Raphson to work.)

3. Modify your algorithm to include backtracking and then try it out on some problem cases.