Physics 305: Ch. 4

Finding the Roots of Functions

4.1 What is Root Finding

To find a root simply means to find where a curve or function goes to zero. For a 1-dimensional function, i.e. an equation that is dependent on only one variable, finding roots is a fairly straight forward matter. In the simplest case, a function can be factored so as to to explicitly show its roots. For example, the equation $f(x) = x^2 - 3x + 2$ can be rewritten as $f(x) = (x - 1)(x - 2)$, and it is obvious its roots are 1 and 2. Most functions, however, are not so cooperative. In these cases, one must employ computational methods to determine roots.

There are many methods which find the roots of 1-dimensional functions. All have their strong points as well as weak points. Choosing the best algorithm not only depends on the function for which you are trying to find roots for but also on computer time constraints and the accuracy that is required for the solution. In this set of notes, we will talk only about the bisection and the Newton-Raphson methods. The bisection method is slower, but with a proper set of initial conditions, bisection is guaranteed to find a root (or a singularity). The Newton-Raphson method is faster and more elegant, but does not always converge. However, with a bit of care, Newton-Raphson can be very stable and dependable and is generally the method of choice.

Methods to find the roots of multi-dimensional functions are fewer and require much more care in their application. Even for a 2-D set of equations, two equations with two unknowns, it can be extremely difficult to find the roots. It is nearly impossible to even determine if a root exits, let alone find it. For example, if we have the equations $f(x, y)$ and $g(x, y)$, a root x_r, y_r is defined by $f(x_r, y_r) = 0$ and $g(x_r, y_r) = 0$. There may be many values of x and y where one equation or the other goes to zero, but finding values where both go to zero is difficult.

We shall limit our discussion to 1-dimensional functions, which are not only easier to solve but have the added advantage of being easily graphed in order to gain insight into the shape of the function and the positions of its roots. An extensive discussion

of 1-dimensional methods as well as multi-dimensional root finding methods can be found in Numerical Recipes, chapter 9.

4.2 Bisection Method

Bisection is the brute-force method of determining roots. The key concept is the idea of "bracketing a root". A root is bracketed when we have two values of x for which the value of the function is positive at one and negative at the other. If the function is continuous, then the function must cross zero somewhere in between these two points, and hence there must be a root in the specified interval.

The algorithm then proceeds by iteratively reducing the size of the range. At each step, one divides the interval in half (hence the name bisection) and evaluates the function at the midpoint. One then replaces *one* of the bounding ends by the midpoint, picking the half that keeps the root bracketed. In other words, if the midpoint value is negative, then one replaces the bounding x that had a negative functional value; if the midpoint value is positive, then one replaces the bounding x with a positive functional value. Each step thereby halves the interval containing the root; after 20 or so iterations, the value of the root will be well constrained.

An important consideration is when to stop and declare victory. Since computers have finite precision, you will almost never find the exact value of the root. Instead, we must content ourselves with being close. The definition of "close" depends on the problem at hand, and it is not possible to give a completely general prescription. Often, one is interested in finding the root to within some tolerance of a "typical" value of the function. For example, one might say that the root has been found when you identify a point x for which

$$|f(x)| \leq \epsilon \times (|f(x_1)| + |f(x_2)|) \,, \tag{4.1}$$

where x_1 and x_2 are the initial bracketing values. Note that as you iterate, you do *not* want to use the current bounding values as x_1 and x_2, because the functional values of the boundaries will themselves approach zero as the iteration proceeds. The parameter ϵ is called the "tolerance" and might be picked to a value like 10^{-6}. Make sure you understand why we use absolute values in this equation!

Is this stopping criteria always safe? Not necessarily! If the function is incredibly steep near the root or if the value of ϵ was picked too small, then there may be no machine-representable value of x for which $f(x)$ satisfies the criteria. By machine-representable, we mean a number that can be exactly represented as a base-2 floating point number with the number of significant digits appropriate to a `float` or `double`. For example, consider the function $f(x) = \text{Arctan}(x) - 0.8$. This has a root at $x = \tan(0.8) \approx 1.029638557$. The criteria in equation (4.1) would plausibly be something like $|f(x)| \leq \epsilon$. However, if we are using type `float`, then we will find that there is no such x for ϵ less than 10^{-8}. Instead, the available floating point numbers

are $f(1.029638529) = -1.37 \times 10^{-8}$ and $f(1.029638648) = 4.41 \times 10^{-8}$. The bisection loop will reach this pair of numbers as the bracket, but after this point the midpoint of the two x values will be incorrectly calculated to be one of the endpoints themselves (there are no available `float` values in between!). If we ask for $\epsilon = 10^{-6}$, however, we would have found a solution. For type `double`, one can often reach $\epsilon \sim 10^{-13}$.

This is an example of a general numerical principle: beware of demanding too fine a tolerance! You need both to set your tolerance according to the required precision of the problem *and* to assess whether your method (and your chosen data type) can deliver that precision.

If the function is continuous, then the bisection method will always converge towards a bracketed root. However, if the function is discontinuous, there may not be a root in the bracketed region. For example, the function $f(x) = 1/x$ is negative for $x < 0$ and positive for $x > 0$ but has no root between $x = -1$ and $x = 1$. In such a case, the bisection method will converge to the singularity instead of a root. This is not necessarily bad, as sometimes the position of singularities is of interest. However, if the positions of roots are needed, singularities can easily be recognized and avoided.

4.3 Newton-Raphson Method (Supplemental Material)

The Newton-Raphson method is a much more elegant way to find roots in that it uses information about the shape of the function to determine an improved estimate of the value of a root. The assumption it makes is that the function is linear from the current guess to the root. Thus, a linear extrapolation from the current guess, using the slope of the function at that point, gives the next and hopefully better guess for the position of the root (see figure 4.1). This can be implemented using the Taylor series expansion of the function about the current position, x_i,

$$f(x) = f(x_i) + (x - x_i)f'(x_i) + \frac{(x - x_i)^2}{2!}f''(x_i) + ... \tag{4.2}$$

If we write the new position as x_{i+1} and assume the function is linear (ie.. that all derivatives higher than the first derivative are zero), then the equation above becomes

$$f(x_{i+1}) = f(x_i) + (x_{i+1} - x_i)f'(x_i) \tag{4.3}$$

The root of this straight line is where $f(x_{i+1}) = 0$ and thus the next guess for the position of the root is

$$x_{i+1} = x_i - f(x_i)/f'(x_i) \tag{4.4}$$

The root is said to be found if the function at the new position is smaller than some pre-determined tolerance.

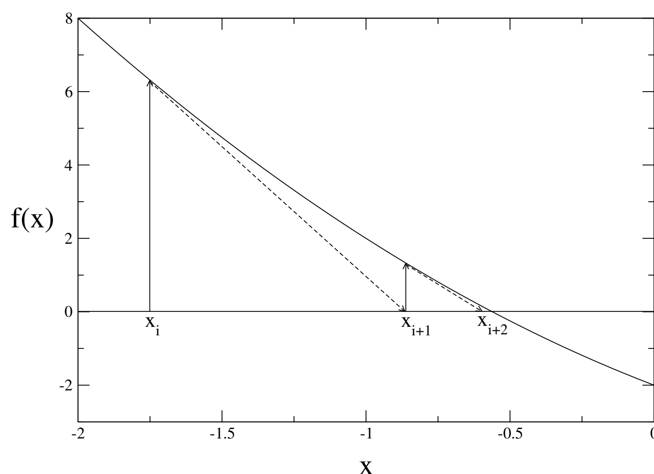

Figure 4.1: The Newton-Raphson method extrapolates the local derivative to find the next estimate of the root

The danger is that a straight line may be a poor approximation to the function and the new position, x_{i+1}, may not be any closer to the root than was the original position, x_i. In this case, the Newton-Raphson method may converge slowly at first but once a reasonably linear region is found near the root, convergence is rapid. However, it is also possible that Newton-Raphson will actually diverge and no roots will be found (see figure 4.2).

This is why a good first guess of the solution is vitally important. For 1-D problems, it is reasonable to graph the function, get an idea where a root exists, and choose a position nearby as the initial guess. This does not guarantee convergence to the root you had in mind, or even to any root, but chances are better than just a random first guess.

4.4 Combining bisection and Newton-Raphson (Supplemental Material)

An improved root finding scheme is to combine the bisection and Newton-Raphson methods. The bisection method guarantees a root (or singularity) and is used to limit the changes in position estimated by the Newton-Raphson method when the linear assumption is poor. However, Newton-Raphson steps are taken in the nearly linear regime to speed convergence.

In other words, if we know that we have a root bracketed between our two bounding points, we first consider the Newton-Raphson step. If that would predict a next

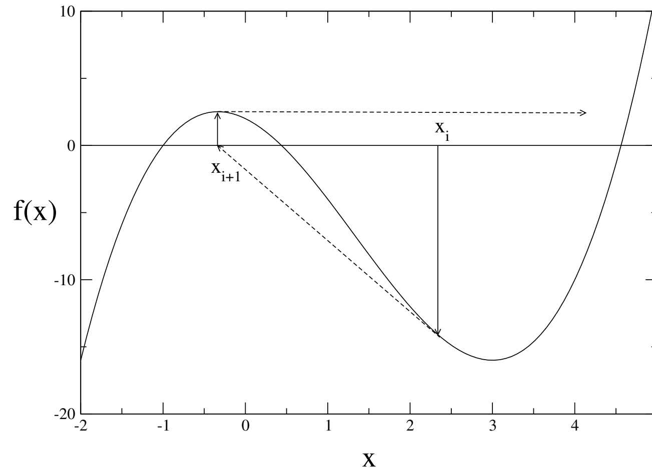

Figure 4.2: Unfortunate case where the Newton-Raphson method finds a local extremum and shoot off to some large incorrect value

point that is outside of our bracketed range, then we do a bisection step instead by choosing the midpoint of the range to be the next point. We then evaluate the function at the next point and, depending on the sign of that evaluation, replace one of the bounding points with the new point. This keeps the root bracketed, while allowing us to benefit from the speed of Newton-Raphson.

4.5 Minimization

To find roots in the previous sections we wanted to find where the value of the function goes to zero. In minimization, however, we wish to find where the function has a minimum, or equivalently where the first derivative goes to zero. To do this, we take the first derivative of the function and find its roots.

This can be written as

$$g(x) = f'(x) \tag{4.5}$$

where f(x) is our original function. Now bisection or Newton-Raphson can be used to determine the roots of $g(x)$ which are the positions of minima of the function $f(x)$. To find the roots of a function via Newton-Raphson, the function and its derivative were required. For minimization, both the first and second derivative of the original function are needed.

Note, that once a minimum has been found, there is no guarantee that this is a global minimum, the smallest value of $f(x)$ for any value of x. This minimum may

only be a local minimum. Again, for 1-D problems, a graph of the function can give insight to whether a global or local minimum has been found.

The same advantages and disadvantages exist for the two methods when determining a minimum as for roots, and a combination of the two methods will be faster and more reliable.

4.5.1 Algorithm for Bisection method

1. Declare variables.
2. Set maximum number of iterations to perform.
3. Set tolerance to a small value (eg. 1.0e-6).
4. Set the two initial bracket values.
5. Check that values given bracket a root or singularity.
    1. Determine value of function `fnct` at the two bracket values.
    2. Make sure product of functional values is less than 0.0. If not, then report this and stop.
    3. If the absolute value of one of the functional values is less than tolerance, then a root is found and write value to terminal and stop.
6. Set the counter of the number of iterations to zero.
7. Begin Bisection loop
    1. Find value midway between bracket values.
    2. Determine functional value at this midpoint.
    3. If the absolute value of functional value at midpoint is less than tolerance, then exit Bisection loop.
    4. If product of functional values at midpoint and at one of the endpoints is greater than zero, then replace this endpoint and its functional value with midpoint and its functional value.
    5. Otherwise replace the other endpoint and its functional value with midpoint and its functional value.
    6. Increment the count of the number of iterations.
    7. If we have exceeded the maximum number of iterations, then exit Bisection loop.

    End Bisection loop
8. If root was not found in maximum number of iterations, write a warning message to the terminal.
9. Write to terminal the value of root and number of iterations performed.

Function `fnct`: Given 1 argument (type `float`):

1. Declare any additional variables.
2. Calculate value of function at the given point.
3. Return value as a `float`.

4.5.2 Algorithm for Newton-Raphson method

1. Declare variables.
2. Set maximum number of iterations to perform.
3. Set tolerance to small value (eg. 1.0e-6).
4. Set an initial guess .
5. Set the counter of the number of iterations to zero.
6. Begin Newton-Raphson loop.
    1. Find next guess via equation.
       new_root = root - `fnct`(root)/`fnct_prime`(root)
    2. If absolute value of `fnct`(new_root) is less than tolerance, then exit Newton-Raphson loop.
    3. Increment the count of the number of iterations.
    4. If we have exceeded the maximum number of iterations, then exit Newton-Raphson loop.

   End Newton-Raphson loop.
7. If root was not found in maximum number of iterations, then write a warning message to the terminal.
8. Write to terminal the value of root and number of iterations performed.

Function `fnct`: Given 1 argument (type `float`)

1. Declare any additional variables.
2. Calculate the value of the function.
3. Return value as type `float`.

Function `fnct_prime`: Given 1 argument (type `float`)

1. Declare any additional variables.
2. Calculate the value of the derivative of the function.
3. Return value as type `float`.

4.5.3 Algorithm for Combination method

1. Declare variables.
2. Set maximum number of iterations to perform.
3. Set tolerance to small value (eg. 1.0e-6).
4. Set two initial bracket values.
5. Check that values given do bracket a root or singularity:
    1. Determine value of function `fnct` at the two bracket values.
    2. Make sure product of functional values is less than 0.0. If not, then report this and stop.
    3. If the absolute value of one of the functional values is less than tolerance, then a root is found and write value to terminal and stop.
6. Set one end of the bracket to be the initial guess for root.
7. Set the counter of the number of iterations to zero.
8. Begin Combo loop
    1. Find next guess via equation
       new_root = root - `fnct`(root)/`fnct_prime`(root)
    2. If new_root is inside bracket values, then use it to shrink bracket.
        1. Determine functional value at new_root.
        2. If the absolute value of functional value at new_root is less than tolerance, then exit Combo loop.
        3. If product of functional values at new_root and at one of the endpoints is greater than zero, then replace this endpoint and its functional value with new_root and its functional value.
        4. Otherwise, replace the other endpoint and its functional value with new_root and its functional value.

       Otherwise, use Bisection method

        1. Find value midway between bracket values.
        2. Determine functional value at this midpoint.
        3. If the absolute value of functional value at midpoint is less than tolerance, then exit Bisection/N-R loop.
        4. If product of functional values at midpoint and at one of the endpoints is greater than zero, then replace this endpoint and its functional value with midpoint and its functional value.
        5. Otherwise, replace the other endpoint and its functional value with midpoint and its functional value.
    3. Increment the counter of the number of iterations.
    4. If the maximum number of iterations has been exceeded, then exit Bisection/N-R loop.

   End Combo loop
9. If root was not found in maximum number of iterations, write a warning.

10. Write to terminal the value of root and number of iterations performed.

Function `fnct`: Given 1 argument (type `float`)

1. Declare any additional variables.
2. Calculate the value of the function.
3. Return value as type `float`.

Function `fnct_prime`: Given 1 argument (type `float`)

1. Declare any additional variables.
2. Calculate the value of the derivative of the function.
3. Return value as type `float`.

4.6 Homework 4

Due Monday, September 21, 10 pm

In this assignment, you will develop a program to find roots of equations by the bisection method. You will apply that program to three physics examples. One of the goals of the week is that you should be able to solve these three examples by only changing the function whose roots you are seeking. By putting this function in a separate file (well, three files), you should be able to solve all three examples with only changes of the compile command.

You will write two separate programs to use that function. First, you will write a program to generate a table of values of the function, suitable as input for your graphing program (simpleplot, graph, xmgrace, or whatever you want to use). You will need to plot the function in order to see its behavior and plan your attack on the roots.

Second, you will write a program to start at some point x_0 and then search along in steps of Δx looking for a bracketed root. Once you have found a Δx region that contains a root, take that bracket and refine it with the bisection method to at least 5 significant figures. You should print the value of the root and the value of the function at the root (which may be non-zero because of numerical precision or your choice of stopping criterion). Your code should then continue stepping along looking for additional roots, up to a number N_{root} supplied by the user.

Your code should prompt the user for the values of x_0, Δx, and N_{root}. You should develop your choices for these values based on the graph of the function. Your Δx has to be large enough that the linear search for brackets is efficient, but small enough that you don't have more than one root in a single step, as that would often be missed. Your choice of N_{root} should be based on the number of roots: you don't want to tell your code to keep looking for a second root when only one exists!

Your code should guard against infinite loops. There are two places where infinite loops could arise. First, in refining a root with bisection, you could reach a point where the function is non-zero and yet the computer cannot bisect any further because no floating point numbers exist between the two endpoints. You can avoid iterating forever in this situation by limiting the number of bisections to something like 30 or 50. Second, you may have supplied a x_0 or Δx that have sent the code looking for a root that doesn't exist. You can avoid this infinite loop by limiting yourself to some number of Δx steps, like 100 or 1000.

You should use double precision. Remember that scanf requires a format of `"%lf"` when reading a variable of type `double`.

Remember that a root is bracketed when the values of the function at the two end points are of opposite sign. This can be easily tested as $f(x_1) * f(x_2) < 0$. Note that you have to be watchful for the case where one of the endpoints has $f(x) = 0$; if that happens, you've found the root and can stop!

Make sure that your programs are commented so that we can see what you have done and why.

Because this problem is moderately complicated, we strongly recommend that you build your programs in several steps, debugging each along the way. The first step is to cast the physical application as a root finding problem. In other words, you need to manipulate the equations into the form $f(x) = 0$. The next step is to write a C function (in a separate file) for the function $f(x)$. Then write the program to call that function and produce a table of outputs. Graph the function and double check that it looks like what you expected.

Then you're ready to proceed to the root finding. Your code to refine the root by bisection should be in a separate function (with arguments for the lower and upper bracket value and perhaps some tolerance choice), so that it can be called from the function that is doing the stepping and search for brackets. You should probably start with a main() program that is hardwired to search for a single root with a known bracket, so that you can debug the bisection function first. Then you can implement the bracket search into main().

If this is confusing, you might want to start by writing an outline describing the different functions and how they'll fit together. Then solve the problem one step at a time!

The best implementations of bisection try to minimize the number of times the function f is called, on the assumption that f may be expensive to compute. This is easily done if you hold the values of f at the two endpoints in two variables, and update those as you update the endpoints when bisecting. That way you can use the value of f at a particular point multiple times without recomputing it. If this is confusing, don't worry about it at first; it may be more clear how to avoid recomputing f at a given x after you have the program working.

As a minor note for your program to generate the table of function values, you may want to prompt the user for the interval and spacing of the table. This prompting has the problem that when you redirect the output to a file, your printed prompt goes to the file instead of the screen, so the user doesn't know when or what to type. There are many possible work-arounds, but a simple one is to change the printing of the prompt to go to the `stderr` stream so that it prints on the screen instead of going to the file. This is easy to do: just replace your `printf()` calls for the prompt, e.g.,

```
printf(Please enter the lower bound for the table");
```

with

```
fprintf(stderr,"Please enter the lower bound for the table");
```

When you want to print the table, use `printf()` again so that those lines will go to `stdout` and be redirected to the file.

Keep your root finding program around. We will need it for at least one later assignment in this class, and it may come in handy in some of your other classes.

Use your program to solve the following problems.

1) When the wave equation is solved in spherical coordinates, the radial dependence of the solution involves a special function known as the spherical Bessel function $j_n(x)$. Here x is a rescaled radius. To solve the wave equation in a spherical cavity, we need to know the zeroes of these functions. The zeroth spherical Bessel function j_0 is simply $\sin(x)/x$, so its zeroes are trivial to find. The roots of the higher functions are harder to find.

For this problem, find the first three positive zeroes of the first spherical Bessel function

$$j_1(x) = \frac{\sin(x)}{x^2} - \frac{\cos(x)}{x} \tag{4.6}$$

to five significant figures.

Note: Bessel functions are actually in the standard C math library, just like exp *and* sin. *But for this assignment please implement the expression above, so that you can see what is going on.*

2) Ferromagnetism is the property in which the atomic spins of a material can spontaneously align, giving a large magnetic dipole moment even without an external magnetic field. Ferromagnets lose this ability at high temperature, as the heat causes the spins to misalign. Interesting, the transition from ferromagnetism to paramagnetism occurs as a phase transition at a temperature known as the Curie temperature.

The Ising model is a simple model for ferromagnetism. When the "mean field approximation" is used to analyze this model, for temperatures below the Curie temperature the magnetization is found by solving the equation

$$\tanh\left(\frac{6J}{k_B T} M\right) = M \tag{4.7}$$

where M is the magnetization (as a fraction of the maximum possible magnetization). J is the strength of the interatomic coupling, T is the temperature, and k_B is Boltzmann's constant. For temperatures below the Curie temperature, $6J/k_B T > 1$. Find M to five significant figures when $6J/k_B T = 1.2$. Find a nonzero solution — the trivial solution at $M = 0$ doesn't count. Note that we are just treating this as an algebra problem — you don't have to know what ferromagnetism or the Ising model is in order to do this problem.

3) Here is another real physics example where for the moment you don't have to understand how we got the equation. In elementary quantum mechanics, you may be asked to find the ground state energy for a particle moving in a one dimensional "square well" potential, $V(x) = -V_0$ when $|x| < a$, and $V(x) = 0$ when $|x| > a$. The ground state energy is found by solving the equation

$$\frac{\sqrt{\lambda - \xi^2}}{\xi} = \tan(\xi) \tag{4.8}$$

This equation is to be solved for ξ, using

$$\lambda = \frac{2(mc^2)V_0a^2}{(\hbar c)^2} \tag{4.9}$$

ξ is related to the ground state energy by

$$\xi = \sqrt{\lambda\left(\frac{V_0 + E}{V_0}\right)} \tag{4.10}$$

Note that for a bound state energy, $E < 0$. In fact, we know that $-V_0 < E < 0$.

Consider a square well potential with $V_0 = 1$ electron volt (eV) and $a = 1$ Å. Use your root finding program with Eq 4.8 to find the smallest positive root for ξ. Then use this to compute the ground state energy in electron volts. (Once you find ξ, you use Eq. 4.10 find E analytically!)

Use $\hbar c = 1970$ eV-Å and $mc^2 = 5.11 \times 10^5$ eV. (1 Å$= 10^{-8}$ cm., but you don't need to convert to cm. – just use the units given in the problem.) Just to get you started, you should find $\lambda = 0.263$.

Reference: *Quantum Mechanics*, by Amit Goswami, section 4.2 (W.C. Brown, 1997), or almost any other elementary quantum mechanics text.