§7 Solutions of Nonlinear Equations

In high school, we have learned how to find the roots of a quadratic equation $ax^2 + bx + c = 0$. This is a particular case of solving nonlinear equations $f(x) = 0$. In general, the function $f(x)$ can be a polynomial, transcendental, or a combination of different functions, e. g. $f(x) = \exp(x)\ln(x) - \cos(x)$. In science we encounter many types of nonlinear equations besides the quadratic ones. As a rule, it is difficult or not feasible to find analytic solutions of these equations. A very short computer program can find numerical solutions instantly. You may already have some experience for solving nonlinear equations with a programmable graphical calculator. In this chapter, we consider a few simple but powerful methods for solving nonlinear equations, starting from the single variable case.

7.1 The Relaxation Method

Suppose we want to find the value of a single variable x that satisfies the single nonlinear equation:

$$x = 3 - e^{-x}. \tag{7.1}$$

There is no known analytical method for solving this equation. So we turn to computational methods. Our elementary method that gives good answers in many cases is simply to iterate the equation. In other words, we guess an initial value x_0 for the unknown variable x, plug it into the right-hand side of the equation, and get a new value x_1 on the left-hand side. For example, in this case we might guess an initial value $x_0 = 1$ and plug it into the right-hand side of Eq. (7.1) to obtain

$$x_1 = 3 - e^{-1} \approx 2.632$$

Then we repeat the process, taking this value and feeding it in on the right-hand side again to obtain

$$x_2 = 3 - e^{-2.632} \approx 2.928$$

and so on. If we keep on doing this, and if we are lucky, then the value will converge to a **fixed point** of the equation, meaning it stops changing. That is exactly what happens in this particular case. Below is the Python code performing ten iterations of such calculation:

```
from math import exp
x = 1.0
for k in range(10):
    x = 3 - exp(-x)
    print("{0:12.11f}".format(x))
```

and here is the output:

```
2.63212055883
2.92807422244
2.94650003196
2.94747678580
2.94752806301
2.94753075355
2.94753089473
2.94753090213
2.94753090252
2.94753090254
```

Obviously, the result seems like converging to a value around `2.9475`. Indeed, if we continue this process for a short while, then it will converge to `2.94753090254` and stop changing. If a process like this converges to a fixed point, then the value of x you get is necessarily a solution to the original equation — you feed that value in on one side of the equation and get the same value on the other side. This is the definition of a solution.

This method is called the **relaxation method**. When it works, it's often a good way to get a solution. It is easy to program (the code above has only five lines) and runs reasonably fast (it took about ten steps to reach a solution accurate to twelve significant figures in this case).

The relaxation method does have its own limitations. First, the equation you are solving needs to be in the simple form $x = f(x)$, where $f(x)$ is some known function, and this is not always the case. But even when the given equation is not in this form, it is in most cases a trivial task to rearrange it to this form. For instance, if you have the equation $\ln x + x^2 - 1 = 0$, then you can take exponential on both sides and rearrange to get $x = e^{1-x^2}$. Moreover, an equation may have more than one solution. Sometimes the relaxation method may converge to one solution and not to another, in which you only learn about that one solution. You may able to lighten this problem somewhat by your choice of the starting value of x. Depending on the starting value, the iteration process may converge to more than one fixed point. So you can find multiple solutions to the equation. Typically, if the process does converge to a given solution, then you can find it by choosing a starting value near that solution. So if you have some approximate idea of the position of the solution that you are looking for, then you should choose a starting value near that position.

Nevertheless, there are some solutions to some equations that you cannot find by the relaxation method no matter what starting value you choose. Even with a starting value close to such a solution, the method will not converge to this solution. For example, consider the equation

$$x = e^{1-x^2} \tag{7.2}$$

which was discussed above. Again there is no analytical method for solving this equation. But in this case we can see by inspection that the solution is simply $x = 1$. If we try to find that solution by the relaxation method starting with $x = 1/2$, then we get the following values of x for the first ten iterations:

```
2.11700001661
0.03075541907
2.71571183275
0.00170346518
2.71827394058
0.00167991310
2.71827415718
0.00167991112
2.71827415720
0.00167991112
```

We can see that the result is oscillating back and forth between two different values instead of approaching a fixed point. It keeps on doing this no matter how long we wait so that we never get a solution to our equation. The relaxation method has failed in this case.

A useful trick in such case is trying to find an alternative way of rearranging the equation to give the value of x. For instance, if we take logs on both sides of Eq. (7.2) and then rearrange, we find the following alternative form for the equation

$$x = \sqrt{1 - \ln x}. \tag{7.3}$$

If we apply the relaxation method to this form starting with $x = 1/2$, then we get the following values of x for the first ten iterations:

```
1.30120989105
0.85831549149
1.07367757795
```

```
0.96379990441
1.01826891043
0.99090663593
1.00455709697
0.99772403758
1.00113862994
0.99943084694
```

The method is now converging to the solution at $x = 1$. If we go on for a few more steps, then we get a very good approximation to that solution.

There is a mathematical theory behind the relaxation method that explains why this rearranging trick works. Assume we have an equation of the form $x = f(x)$ that has a solution at $x = x^*$ and let us consider the relaxation method when x is close to x^*. Performing a Taylor series expansion, the value x_i after an iteration of the method is given in terms of the previous value x_{i-1} by

$$x_i = f(x_{i-1}) = f(x^*) + (x_{i-1} - x^*)f'(x^*) + \dots \tag{7.4}$$

However, by definition, x^* is a solution of the original equation so that $x^* = f(x^*)$. Thus Equation (7.4) can be rewritten as

$$x_i - x^* = (x_{i-1} - x^*)f'(x^*) \tag{7.5}$$

where we have neglected the higher order terms.

Equation (7.5) tells us that the distance $x_i - x^*$ to the true solution of the equation gets multiplied on each iteration of the method by a factor of the derivative $f'(x^*)$. If the absolute value of this derivative is less than one, then the distance will get smaller on each iteration which means we are converging to the solution. On the other hand, if it is greater than one, then we are getting farther away from the solution on each step and the method will not converge. Therefore, the relaxation method will converge to a solution at x^* if and only if $|f'(x^*)| < 1$. This explains why the method failed for Eq. (7.2). In this case, we have $f(x) = e^{1-x^2}$ and $x^* = 1$ and so

$$|f'(x^*)| = \left|-2xe^{1-x^2}\right|_{x=x^*} = 2$$

Thus the method will not converge in this case.

Suppose we find ourselves in a situation where the relaxation method does not converge because $|f'(x^*)| > 1$. Let us rearrange the equation $x = f(x)$ by inverting the function f to obtain $x = f^{-1}(x)$, where f^{-1} is the functional inverse of f. Then the method will converge if the derivative of f^{-1} is less than one at x^*. To calculate the derivative of f^{-1}, we define $u = f^{-1}(x)$ so that the derivative we want is du/dx. But in this case we also have $x = f(u)$. So

$$\frac{dx}{du} = f'(u)$$

But we have $u = f^{-1}(x^*) = x^*$ when $x = x^*$. Then taking reciprocal gives us the derivative we want:

$$\frac{du}{dx} = \frac{1}{f'(x^*)}$$

So the derivative of f^{-1} is simply the reciprocal of the derivative of f. It implies that if the derivative of f is greater than one, then the derivative of f^{-1} must be less than one. Hence, if the relaxation method fails to converge for $x = f(x)$, then it will succeed for the equivalent form $x = f^{-1}(x)$. As a result, if the method fails for a particular equation, then you should invert the equation and the method will work. This is exactly what we have done for Eq. (7.3).

Unfortunately, not all equations can be inverted. For example, if your function $f(x)$ is a tenth-order polynomial, then inversion is not possible. In such cases, it may still be possible to rearrange and get a different equation for x. But the above theory no longer applies and convergence is no longer guaranteed. You might not be lucky enough to get a convergent calculation. For instance, consider the equation

$$x = x^2 + \sin 2x \tag{7.6}$$

This equation has a solution at $x^* = 0$. However, if we apply the relaxation method directly, then we will not able to find that solution. The method fails to converge as we can prove by calculating $|f'(x^*)|$ for $x^* = 0$, which turns out to be equal to 2 in this case. However, the function $f(x) = x^2 + \sin 2x$ on the right-hand side of Eq. (7.6) cannot be inverted. But we can still rearrange equation to obtain

$$x = \frac{1}{2}\sin^{-1}(x - x^2)$$

Since this is not a true inversion of the original equation, we are not guaranteed that this equation will converge better than the original one. But in fact it does. If we calculate $|f'(x^*)|$ for $x^* = 0$ again, then we find that it's now equal to 1/2 and so the relaxation method will work. This time we are lucky.

In summary, the relaxation method does not always work. But between the cases where it works directly for the equation, the cases where it works by inverting the equation, and the cases where some other rearrangement turns out to work by good luck, the method is useful for a wide range of problems.

An important question about the relaxation method is how fast it converges to a solution, assuming it does converge. We can answer this question by looking again at Eq. (7.5). This equation tells us that the distance to the solution gets smaller by a factor of $|f'(x^*)|$ on each iteration when the method converges. In other words, the distance decreases exponentially with the number of iterations we perform. Since the exponential is a rapidly decaying function, the relaxation method converges to a solution quickly.

This is a nice result as far as it goes. But it isn't very practical. In a typical application of the relaxation method, what we really want to know is "When can I stop iterating? When is the answer I have good enough?". One simple way to answer this question is simply look at the solutions obtained in successive iterations and observe when they stop changing. For instance, if you want an answer accurate to five significant figures, then you continue until a few iterations have passed without any changes in the first five significant figures of x. If you want a quick answer to a problem and accuracy is not a big issue, then this is actually not a bad approach as it is easy and usually works.

But sometimes we may want to know the exact accuracy of our answer or to stop immediately as soon as a required target accuracy is reached. For example, if a single iteration of the relaxation method takes a long time (minutes or hours) due to the involvement of complex calculations, then we may not want to perform even one more iteration than necessary. We can take the following approach in such cases.

Let ϵ_i be the error of our current estimate x_i of the solution to the equation, i. e. the true solution x^* is related to current estimate x_i by $x^* = x_i + \epsilon_i$. Similarly, let ϵ_{i+1} be the error of the next estimate so that $x^* = x_{i+1} + \epsilon_{i+1}$. Then Equation (7.5) tells us that close to x^* we have

$$\epsilon_{i+1} = \epsilon_i f'(x^*)$$

$$\Rightarrow\ x^* = x_i + \epsilon_i = x_i + \frac{\epsilon_{i+1}}{f'(x^*)}$$

Equating the above equation with $x^* = x_{i+1} + \epsilon_{i+1}$ and rearranging for ϵ_{i+1}, we obtain an expression for the error of the new estimate:

$$\epsilon_{i+1} = \frac{x_i - x_{i+1}}{1 - 1/f'(x^*)} \approx \frac{x_i - x_{i+1}}{1 - 1/f'(x_i)} \tag{7.7}$$

where we have made use of the fact that x_i is close to x^* so that $f'(x_i) \approx f'(x^*)$. If we know the form of the function $f(x)$, then we can calculate its derivative and use this formula to estimate the error ϵ_{i+1} of the new value x_{i+1} for the solution at each step. As a result, we can simply repeat the iteration until the magnitude of this estimated error falls below some target value, ensuring that an answer that is as accurate as we want is obtained without wasting any time on additional iterations.

However, there are some cases that we don't know the full formula for $f(x)$. For instance, $f(x)$ might be given as the output of another program that performs some complicated calculation instead of a mathematical formula. In such cases, we cannot calculate the derivative $f'(x)$ analytically. But we can estimate it using a numerical derivative like those discussed in Section 6.7. As we have seen, calculating a numerical derivative involves taking the difference of the values of f at two different points. In the most common application of this idea to the relaxation method, we choose the points to be values of x at successive steps of the iteration.

Suppose we have three successive estimates of x denoted as x_i, x_{i+1}, and x_{i+2}. We woul like to calculate the error of the most recent estimate x_{i+2} . According to Eq. (7.7), this error is given by:

$$\epsilon_{i+2} = \frac{x_{i+1} - x_{i+2}}{1 - 1/f'(x^*)} \approx \frac{x_{i+1} - x_{i+2}}{1 - 1/f'(x_i)} \tag{7.8}$$

Next, we approximate $f'(x_i)$ as

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i+1})}{x_i - x_{i+1}}$$

But by definition $x_{i+1} = f(x_i)$ and $x_{i+2} = f(x_{i+1})$. Therefore,

$$f'(x_i) \approx \frac{x_{i+1} - x_{i+2}}{x_i - x_{i+1}}$$

Substituting this into Eq. (7.8), we find that the error of the third and most recent one of our estimates of the solution is given approximately by

$$\epsilon_{i+2} \approx \frac{x_{i+1} - x_{i+2}}{1 - (x_i - x_{i+1})/(x_{i+1} - x_{i+2})} = \frac{(x_{i+1} - x_{i+2})^2}{2x_{i+1} - x_i - x_{i+2}}$$

As a result, if we keep track of these three successive estimates of x at each stage of the calculation, then we can estimate the error even when we cannot calculate a derivative of $f(x)$ directly.

◇◇

Example 7.1

In the mean field theory of ferromagnetism, the strength M of magnetization of a ferromagnetic material like iron depend on its temperature T according to the formula

$$M = \mu \tanh\left(\frac{JM}{k_B T}\right)$$

where μ is the magnetic dipole moment, J is the coupling constant, and k_B is the Boltzmann's constant. For convenience, we make the substitutions $m = M/\mu$ and $C = \mu J/k_B$ so that

$$m = \tanh\left(\frac{Cm}{T}\right) \tag{7.9}$$

Obviously, this equation has a solution at $m = 0$ which implies a material that is not magnetized at all. But are there solutions with $m \neq 0$? There is no known method of solving such equation exactly. However, we can find them using the computer. Let's assume that $C = 1$ for simplicity and look for solutions as a function of T accurate to within $\pm 10^{-6}$ of the true answer. In this case, since we know the full functional form of the equation we are solving, we can evaluate the derivative in Eq. (7.7) explicitly and show that the error is given by

$$\epsilon_{i+1} \approx \frac{m_i - m_{i+1}}{1 - T\cosh^2(m_i/T)}$$

Below is the Python program `ferromag.py` that finds the solutions of Eq. (7.9) and makes a plot of these solutions as a function of the temperature as shown in Figure 7.1.

```python
# ferromag.py
# This program finds the solutions of the equation for the strength
# of the magntization of a ferromagnetic material and make a plot of
# the solutions versus its temperature. For convenience, we assume
# that C = mu*J/kB = 1 where mu is the magnetic dipole moment, J is
# the coupling constant, and kB is the Boltzmann's constant.
# Last Update on 30 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

# Parameters for the calculation
Tmax = 2.0
points = 1000
accuracy = 1e-6

# Set up the data lists for plotting
y = []
```

```
temp = np.linspace(0.01, Tmax, points)

# Find the solutions of the equation
for T in temp:
    m = 1.0
    error = 1.0
    # Loop until the error is small enough
    while error > accuracy:
        mo = m
        m = np.tanh(mo/T)
        error = abs((mo - m)/(1 - T*np.cosh(mo/T)**2))
    y.append(m)

# Plot the graph of the solutions
fig, ax = plt.subplots()
ax.plot(temp, y)
ax.set_ylim(-0.1, 1.1)
ax.set_xlabel("Temperature")
ax.set_ylabel("Magnetization/dipole moment")
plt.show()
```

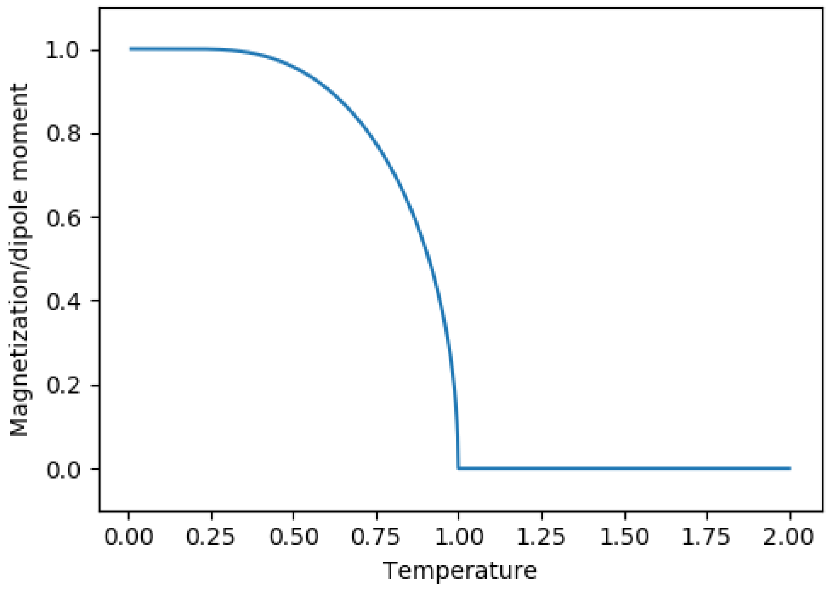

Figure 7.1: The solutions of Eq. (7.9) found by the program `ferromag.py` as a function of the temperature.

We can see that the program finds non-zero solutions of the equation only for values of the temperature below $T = 1$. As we approach $T = 1$ from below, the value of the magnetization m falls off. Above $T = 1$, the program only finds the solution $m = 0$. This is a real physical phenomenon observed in experiments on real magnets. We would observe that the magnetization doesn't decrease gradually to zero but instead passes through a **phase transition** at which it disappears suddenly and completely. In our model system, this transition occurs at $T = 1$ which is known as the **critical temperature** of the magnet.

◇◇

7.2 The Bisection Method

The **bisection method**, also known as **binary search**, is a more robust method for solving nonlinear equations for a single variable. In this method, one specifies an interval in which one wants to search for a solution to an equation. If a single solution exists in that interval, then this method will always find it. In this sense, the bisection method is superior to the relaxation method, although it is a little bit more complicated to program.

A nonlinear equation for a single variable x can always be rearranged to the form $f(x) = 0$. Thus finding the solutions to such an equation is equivalent to finding zeros or roots of $f(x)$. The bisection method works by finding the roots.

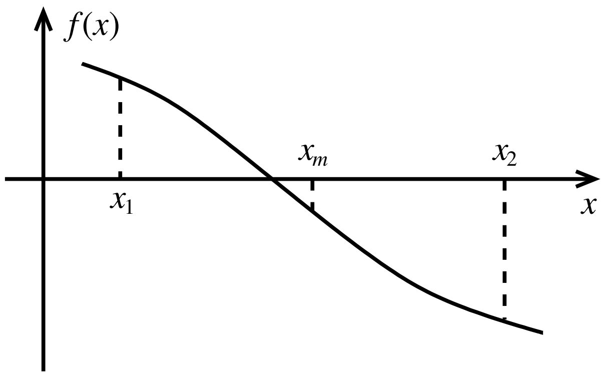

Figure 7.2: If $f(x_1)$ and $f(x_2)$ have different signs and $f(x)$ is continuous between x_1 and x_2, then there must be at least one root between x_1 and x_2.

Suppose we want to find a root of the function $f(x)$ in the interval between x_1 and x_2 if it exists. We begin by calculating the values $f(x_1)$ and $f(x_2)$ of $f(x)$ at these two points. Assume that one of these values is positive while the other is negative as depicted in Figure 7.2. If the function $f(x)$ is continuous, then there must be at least one point somewhere between x_1 and x_2 where $f(x)$ equals zero and so there must exist at least one root of the function in the interval. Then we say that the points x_1 and x_2 *bracket* a root of the function $f(x)$. We can also make the inverse statement that if there is exactly one root between x_1 and x_2, then $f(x_1)$ and $f(x_2)$ must have opposite signs. Thus if a single root exists in the interval, then we will know it.

Now suppose we take a new point $x_m = (x_1 + x_2)/2$ halfway between x_1 and x_2 and evaluate $f(x_m)$. If we are incredibly lucky, then x_m may fall exactly at the root of the function such that $f(x_m) = 0$, in which case we have found our root and our job is done. However, it's much more likely that $f(x_m)$ is non-zero. So $f(x_m)$ must be either positive or negative, meaning that it must have the same sign as $f(x_1)$ or $f(x_2)$ and the opposite sign to the other. Suppose it has the same sign as $f(x_2)$ and the opposite sign to $f(x_1)$ as in Figure 7.2. By the same argument as before, x_1 and x_m must bracket a root of the function $f(x)$. But notice that x_1 and x_m are closer together by a factor of two than x_1 and x_2. Thus we have narrowed the range in which our root lies. Similarly, if $f(x_m)$ and $f(x_2)$ have opposite signs, then x_m and x_2 bracket our root and are closer together than x_1 and x_2. No matter $f(x_1)$ or $f(x_2)$ has opposite sign to $f(x_m)$, this approach allows us to find two points that bracket our root and are closer together than the original two points by a factor of two.

Now we repeat this process, finding the midpoint of our two new points, and so forth. In each step, we narrow the range in which our root lies by a factor of two. We repeat the process until we have narrowed the range to the desired accuracy. For example, if we want to find a root to an accuracy of 10^{-6}, then we repeat until the distance between our points is 10^{-6} or less. Then we take the midpoint of the final two points as our answer.

Here is an outline of the whole procedure of finding solution to $f(x) = 0$ using the bisection method:

1. Given an initial pair of points x_1 and x_2, check that $f(x_1)$ and $f(x_2)$ have opposite signs. Also choose a target accuracy for your answer.
2. Calculate the midpoint $x_m = (x_1 + x_2)/2$ and evaluate $f(x_m)$.
3. If $f(x_m)$ has the same sign as $f(x_1)$, then set $x_1 = x_m$; otherwise set $x_2 = x_m$.
4. If $|x_1 - x_2|$ is larger than the target accuracy, repeat from step 2. Otherwise, calculate the midpoint x_m once more and this is the final estimate of the position of the root.

Just like the relaxation method, the accuracy of the bisection method improves exponentially as the method progresses since the error decreases by a factor of two on each step of the calculation. This again means that the method is relatively fast because exponentials decay towards zero quickly. Indeed, we can easily estimate how many steps the method will take to find a solution of a desired accuracy. Suppose our initial points x_1 and x_2 are a distance d apart. The distance is halved at each step of the bisection method so that the distance is $d/2^N$ after N steps. The calculation ends when this distance reaches our target accuracy ϵ. Therefore, the number of steps we have to take is given by

$$\epsilon = \frac{d}{2^N} \qquad \Rightarrow\ N = \log_2\left(\frac{d}{\epsilon}\right)$$

The logarithm is a very slowly growing function of its argument. So even if d is much bigger than ϵ, the number of steps N is still quite small. For instance, if our two starting points are a distance $d = 10^{10}$ apart and we want an accuracy of 10^{-10} (which is quite generous), then we have

$$N = \log_2\left(\frac{10^{10}}{10^{-10}}\right) \approx 66.4$$

So it will take 67 steps to find the root. A computer can comfortably execute millions of instruction per second that sixty or seventy is nothing. This method will work very fast.

However, the bisection method does have its disadvantages. First, if there is no root between the initial points x_1 and x_2, then of course it will not find a root. In such case, $f(x_1)$ and $f(x_2)$ will have the same sign which tells us in advance that the method will not work. Besides, if there is an even number of roots so that $f(x)$ cross the zero line an even number of times as shown in Figure 7.3, then the method will also fail. In such case, $f(x_1)$ and $f(x_2)$ will again have the same sign and there is no easy way to distinguish this situation from the one in which no roots exist in the interval.

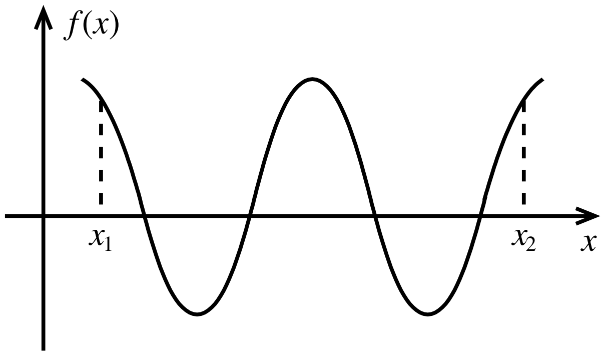

Figure 7.3: If the points x_1 and x_2 bracket an even number of roots (four in this case), then $f(x_1)$ and $f(x_2)$ have the same sign and the bisection method fails.

In fact, one does not necessarily have an initial interval in mind to search for a root using the bisection method. It's very often that one is simply interested in finding a root wherever it may be hiding. In that case, any initial pair of points x_1 and x_2 such that $f(x_1)$ and $f(x_2)$ have different signs is good enough. However, finding such a pair is not always easy and there is no universal method that will always work. Below are a couple of rules of thumb that are useful in practice:

1. If you know some information about your function that can help you bracket a root, then this is usually the best place to start. For instance, if you know specific values where the function must be positive or negative due to physical reason, then make use of these values.

2. If you are looking for a root near a particular value of x, starting with values x_1 and x_2 closely spaced around that value and check to see if they bracket the root. If they do not, double their distance from that value in either direction and check again. Keep doubling the distance until you find values that bracket the root.

But these techniques can fail on occasions. Initial bracketing of a root is one of those problems that can require a creative solution.

A more subtle, though rarer, problem with the bisection method is that it cannot find even-order polynomial roots of functions, i. e. double roots, quadruple roots, and so forth that fall in the same place such as the root of $(1 - x)^2$ or $(3 - 2x)^4$. Functions with such roots only touch the horizontal axis at the position of the root but don't cross it (see Figure 7.4). So it is impossible to bracket the root and the method fails.

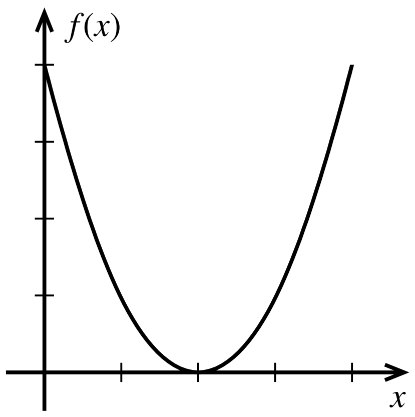

Figure 7.4: A function such as $(1 - x)^2$ that has two roots (or any even number of roots) in the same place only touches the horizontal axis but does not cross it.

Finally, a notable drawback of the bisection method is that it does not extend to the solution of simultaneous equations in multiple variables. Unlike the relaxation method, it works only for the solution of single equation in one variable.

◇◇

Example 7.2

Planck's formula tells us that the intensity of radiation per unit wavelength λ emitted from a blackbody at temperature T is

$$I(\lambda) = \frac{2\pi hc^2}{\lambda^5}\frac{1}{(1 - e^{hc/\lambda k_B T})}$$

where h is Planck's constant, c is the speed of light in vacuum, and k_B is Boltzmann's constant. Differentiating this expression with respect to the wavelength reveals that the wavelength λ_m at which the emitted radiation is strongest is the solution of the equation

$$5e^{-hc/\lambda k_B T} + \frac{hc}{\lambda k_B T} - 5 = 0$$

Using the substitution $x = hc/\lambda k_B T$, one can show that the wavelength of maximum radiation λ_m obeys the Wien's displacment law:

$$\lambda_m = \frac{b}{T}$$

where $b = hc/k_B x$ is the so-called Wien's displacement constant and x is the solution of the non-linear equation

$$5e^{-x} + x - 5 = 0.$$

Below is the Python program `wien.py` which solves this equation to an accuracy of $\epsilon_0 = 10^{-6}$ using the bisection method and hence finds the value of the Wien's displacement constant.

```
# wien.py
# This program solves the nonlinear equation 5exp(-x) + x - 5 = 0 to
# an accuracy of 10^{-6} using the bisection method and hence finds
# the value of the Wien's displacement constant b = hc/(kB*x).
# Written on 7 Jan 2022 by F K Chow

from math import exp
from scipy.constants import h, c, k

def f(x):
    """ Compute the function f(x) = 5exp(-x) + x - 5 """
    return 5*exp(-x) + x - 5.0
```

```
a, b = 1.0, 5.0
if f(a)*f(b) > 0:
    print("Bisection method fails for this case!")
else:
    # Use the bisection method to find the root of f(x)
    x1, x2 = a, b
    tol = 1e-6
    error = 1
    while error > tol:
        xm = 0.5*(x1 + x2)
        if f(xm)*f(x1) > 0:
            x1 = xm
        else:
            x2 = xm
        error = abs(x1 - x2)
    xm = 0.5*(x1 + x2)
    b = h*c/(k*xm)
    print("The root of the equation 5e^(-x) + x - 5 = 0 is ", end="")
    print("x = {0:.6f}.".format(xm))
    print("So the value of the Wien's displacement constant ", end="")
    print("is b = {0:.6f}.".format(b))
```

Running this program produces the output:

```
The root of the equation 5e^(-x) + x - 5 = 0 is x = 4.965114.
So the value of the Wien's displacement constant is b = 0.002898.
```

which agrees with the experimental result $b = 2.898 \times 10^{-3}\,\mathrm{m} \cdot \mathrm{K}$.

◇◇

7.3 The Newton's Method

In this section, we examine a third common method for solving nonlinear equations known as **Newton's method** or **Newton-Raphson method**. It addresses some of the problems with the relaxation and bisection methods although it has its own set of problems too.

Figure 7.5 shows a graphical representation of Newton's method for a single variable.

Just like the bisection method, we convert the problem of solving an equation into one of finding the root of a function $f(x)$ represented by the curve in the figure. We start with a single guess x_i and then use the slope at that position to extrapolate and make another guess x_{i+1} which will hopefully better than the previous guess x_i.

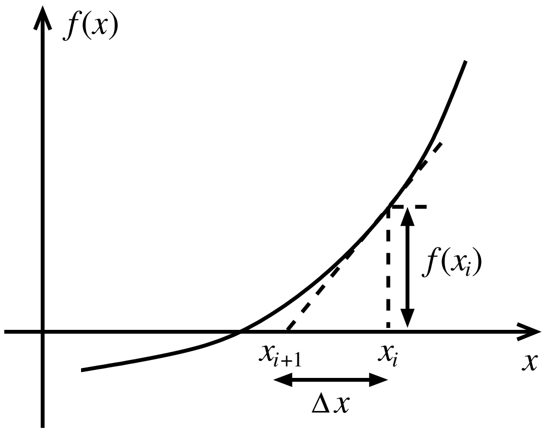

Figure 7.5: Newton's method takes a single estimate x_i of the root of a function $f(x)$ and uses the slope of $f(x)$ at that point to extrapolate a better estimate x_{i+1}.

From the figure, we can see that the slope at x_i is

$$f'(x_i) = \frac{f(x_i)}{\Delta x}$$

The formula for our new guess x_{i+1} of the root is thus

$$x_{i+1} = x_i - \Delta x = x_i - \frac{f(x_i)}{f'(x_i)} \tag{7.10}$$

The drawback of this approach is that it requires us to know the derivative $f'(x_i)$. If the derivative is known, then putting the method to work on the computer is straightforward. We just start with a guess x_i and use Eq. (7.10) repeatedly to get better and better estimates of the position of the root. For functions with more than one root, Newton's method usually converges to a root near the starting value of x. So one can use this method to find different roots by choosing different starting points.

How long should we go on to iterating Eq. (7.10) before we are satisfied with our answer? If one is not highly concerned with running time or accuracy, then it's often adequate to simply repeat the process until the value of x stop changing or at least until its first few digits stop changing. For example, if we want an answer accurate to six significant digits, then we just iterate until a few iterations pass without any changes in those six figures. On the other hand, if we want more exact solution, then we can make an estimate of the accuracy of the method as follows.

Suppose x^* represents the true solution of the root of interest. Performing a Taylor series expansion about our estimate x_i of the root, we obtain

$$f(x^*) = f(x_i) + (x^* - x_i)f'(x_i) + \frac{1}{2}(x^* - x_i)^2 f''(x_i) + \ldots \tag{7.11}$$

But by definition $f(x^*) = 0$ since x^* is a root of $f(x)$. So the left-hand side of this equation vanishes. Then dividing both sides by $f'(x_i)$ and rearranging yields

$$x^* = \left[x_i - \frac{f(x_i)}{f'(x_i)}\right] - \frac{1}{2}(x^* - x_i)^2 \frac{f''(x_i)}{f'(x_i)} + \ldots$$

Comparing with Eq. (7.10), we see that the quantity in square brackets [ ] on the right-hand side of this equation is just our new estimate x_{i+1} of the root. So we have

$$x^* = x_{i+1} - \frac{1}{2}(x^* - x_i)^2 \frac{f''(x_i)}{f'(x_i)} + \ldots \tag{7.12}$$

In other words, the root x^* is equal to our new estimate x_{i+1} plus an error term proportional to $(x^* - x_i)^2$ with higher order terms neglected.

Let us define the error ϵ_i on our old estimate x_i of the root in the standard fashion by $x^* = x_i + \epsilon_i$, and the error on our next estimate x_{i+1} similarly by $x^* = x_{i+1} + \epsilon_{i+1}$. Then Equation (7.12) tells us that

$$\epsilon_{i+1} \approx \left[-\frac{f''(x_i)}{2f'(x_i)}\right] \epsilon_i^2 \tag{7.13}$$

That is to say, if the size of the error is ϵ_i on one step of the method, then it will be around ϵ_i^2 on the next step. So we say that Newton's method has **quadratic convergence**.

The above result implies that Newton's method converges *extremely* fast. The relaxation and bisection methods are not bad — both have an error that shrinks exponentially with passing iterations of the calculation. But Newton's method does considerably better. Assuming that the quantity $c = -f''(x_i)/2f'(x_i)$ in Eq. (7.13) is roughly constant in the vicinity of the root, the error ϵ_N after N iterations will be approximately $\epsilon_N \approx (c\epsilon_0)^{2^N}/c$, where ϵ_0 is the error on the initial guess. It implies that the error varies with N as an exponential of an exponential, which is very fast indeed.

To estimate the error on our current estimate of the root, we can write $x^* = x_i + \epsilon_i$ and $x^* = x_{i+1} + \epsilon_{i+1} = x_{i+1} + c\epsilon_i^2$ for some constant c, and equate the values of x^* to get

$$x_{i+1} - x_i = \epsilon_i(1 - c\epsilon_i) \approx \epsilon_i$$

if ϵ_i is small. So a simple estimate of the error ϵ_i on the value of x_i is just the change $x_{i+1} - x_i$ from one iteration to the next. Alternatively speaking, Newton's method converges so rapidly that to all intents and purposes the value of x_{i+1} is a good enough approximation to the true root that $x_{i+1} - x_i \approx x^* - x_i = \epsilon_i$. Note that this gives us the error ϵ_i on the old estimate x_i. You need to perform one more iteration to get the error of the new estimate x_{i+1}. From the above equation, we can see that a good rule of thumb for implementing Newton's method is to go on iterating Eq. (7.10) until the difference between two successive value of x is comparable or smaller than the desired accuracy of the solution.

Newton's method has two main disadvantages. The first one is that it requires us to know the derivative $f(x)$, which is sometimes impossible. However, we can solve this problem by using a numerical derivative as described in the next section. A more serious problem is that Newton's method doesn't always converge like the relaxation method. In particular, if the value of $f'(x_i)$ is very small, then Eq. (7.13) can give an error that actually gets larger rather than smaller upon iteration of the method. Then we are moving further away from the root we want and not closer. Newton's method can also fail to converge if the shape of the function $f(x)$ is unfavorable so that it slopes in the wrong direction as shown in Figure 7.6.

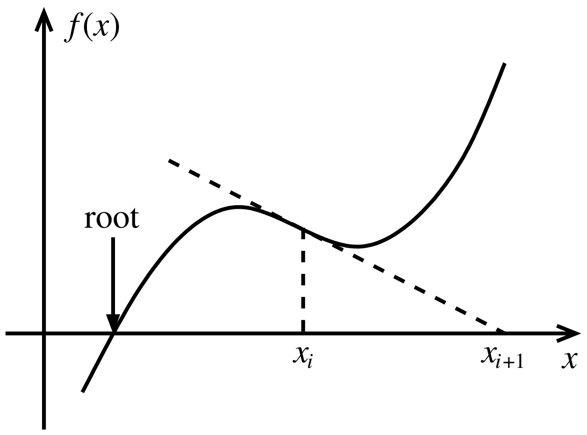

Figure 7.6: Newton's method can fail to converge if the shape of the function $f(x)$ is unfavorable.

Situation like these are usually quite easy to spot in practice. If your Newton's method calculation seems to be giving strange answers, then you may be having convergence problems of this kind. In the end, the important lesson is that no root finding method is perfect. Each has its good and bad points and each is better than the others in some cases. Newton's method is certainly not perfect for every problem although its speed and flexibility make it a good choice for many. One of the keys to do good computational physics is to have a wide range of methods at your disposal so that if one works poorly for a particular problem you have others that can be utilized.

◇◇

Example 7.3

Let us use Newton's method to calculate the inverse hyperbolic tangent of a number u. By definition, $\tanh^{-1} u$ is the number x such that $u = \tanh x$. That is is to say, x is a root of the equation $\tanh x - u = 0$. Recalling that the derivative of $\tanh x$ is $1/\cosh^2 x$, Eq. (7.10) for this problem becomes

$$x_{i+1} = x_i - (\tanh x_i - u)\cosh^2 x_i$$

Below is the Python program `atanh.py` which calculates the inverse hyperbolic tangent function using Newton's method starting with an initial guess $x_0 = 0$ and then makes a plot of this function. The resulting plot is depicted in Figure 7.7.

```python
# atanh.py
# This program calculates the inverse hyperbolic tangent function
# using Newton's method starting with an initial guess of 0 and make
# a plot of this function.
# Last Update on 1 Dec 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

accuracy = 1e-12

def arctanh(u):
    """ Function to compute the inverse hyperbolic tangent using
        Newton's method starting with an initial guess of 0. """
    x = 0.0
    delta = 1.0
    while abs(delta) > accuracy:
        delta = (np.tanh(x) - u)*np.cosh(x)**2
        x -= delta
    return x

# Generate a list of the arctanh function
upoints = np.linspace(-0.99, 0.99, 100)
xpoints = []
```

```
for u in upoints:
    xpoints.append(arctanh(u))

# Plot a graph of the arctanh function
fig, ax = plt.subplots()
ax.plot(upoints, xpoints)
ax.set_xlabel("u")
ax.set_ylabel("arctan(u)")
plt.show()
```

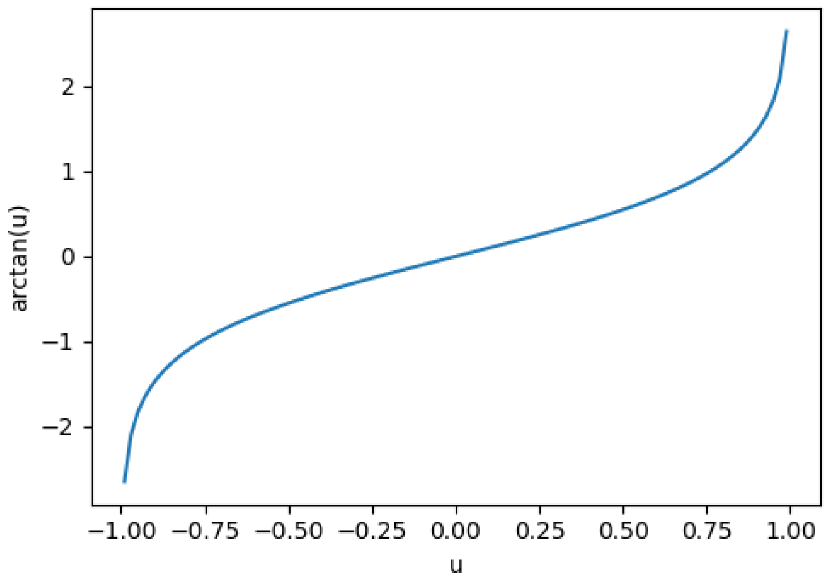

Figure 7.7: The inverse hyperbolic tangent computed using Newton's method starting with an initial guess $x_0 = 0$.

◇◇

7.4 The Secant Method

Newton's method requires us to know the derivative of the function $f(x)$. If we don't have analytical formula for the derivative, then we cannot use Newton's method directly as in the previous section. But we can instead calculate the derivative using the techniques discussed in Section 6.7. As we saw there, calculating a numerical derivative involves taking the difference of f at two closely spaced points. In the most common application of this idea to Newton's method, we choose the points to be the points x_i and x_{i+1} at two successive steps of the method which avoid us from evaluating $f(x)$ at additional values of x. This version of Newton's method is called the **secant method**. Let us study its details in the following.

Let us start with two points x_i and x_{i+1}. We now calculate an approximation to the derivative of f at x_{i+1} from the formula

$$f'(x_{i+1}) \approx \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

Then we substitute this value into Eq. (7.10) to get a new and hopefully better estimate x_{i+2} for the position of the root

$$x_{i+2} = x_{i+1} - f(x_{i+1}) \left[\frac{x_{i+1} - x_i}{f(x_{i+1}) - f(x_i)} \right]$$

This new estimate is based on the *two* previous values in the series. It is different from the ordinary Newton's method in which each estimate was based on only one previous value. It also makes use of the values $f(x_{i+1})$ and $f(x_i)$ that we would have to calculate these values in the process of carrying out the normal Newton's method. Thus the secant method involves just little extra work. Apart from these differences, the secant method is similar to the ordinary Newton's method. It has the same fast convergence and can break down in the same way if we are unlucky with the shape of the function $f(x)$. It can also be generalized to the solution of simultaneous nonlinear equations. But the generalization is quite complicated which involves the calculation of an entire matrix of numerical derivatives rather than just a single one.

7.5 Methods for Two or More Variables

The relaxation method can be extended easily to the solution of simultaneous nonlinear equations in two or three variables. For example, suppose that we are given two equations to solve for the two variables x and y. First we rearrange those two equations into the form

$$x = f(x, y), \qquad y = g(x, y),$$

where f and g are known functions of x and y. Then we guess initial starting values for both variables, substitute them into the equations to get new values, and repeat. If we converge to a fixed point so that the values for both variables are unchanging, then those values must be the solution to the two simultaneous equations.

In general, if we have any number N of simultaneous equations in N unknown variables

x_1, x_2, ..., x_N, then we can write them in the form

$$x_1 = f_1(x_1, x_2, \ldots, x_N),$$
$$\vdots$$
$$x_N = f_N(x_1, x_2, \ldots, x_N).$$

Then we can choose starting values for all the variables and apply the equations repeatedly to find a solution. Like the relaxation method for a single variable, the method is not guaranteed to converge to a solution. Depending on the exact form of the equation, it will sometimes converge and sometimes not. If it does not converge, then it may be possible to rearrange the equations into a different form that will converge. So a little experimentation is sometimes required to make the method work.

Newton's method can also be used for the solution of simultaneous nonlinear equations. Any set of simultaneous equations for N unknown variables x_1, x_2, ..., x_N can be written in the form

$$f_1(x_1, x_2, \ldots, x_N) = 0,$$
$$\vdots$$
$$f_N(x_1, x_2, \ldots, x_N) = 0,$$

for some set of functions f_1, f_2, ..., f_N. Note that the number of equations and the number of variables must be the same for the equations to be solvable in general.

Suppose the equations have a root at x_1^*, x_2^*, ..., x_N^* . Then we can write the equivalent of the Taylor series expansion in Eq. (7.11):

$$f_j(x_1^*, x_2^*, \ldots, x_N^*) = f_j(x_{1,i}, x_{2,i}, \ldots, x_{N,i}) + \sum_{k=1}^{N} (x_k^* - x_{k,i}) \left.\frac{\partial f_j}{\partial x_k}\right|_{x_1=x_{1,i}, x_2=x_{2,i}, \ldots, x_N=x_{N,i}} + \ldots$$

where $x_{j,i}$ is the estimate of the root for x_j in the ith iteration. In vector notation, the above equation can be written as

$$\mathbf{f}(\mathbf{x}^*) = \mathbf{f}(\mathbf{x}_i) + \nabla\mathbf{f} \cdot (\mathbf{x}^* - \mathbf{x}_i) \tag{7.14}$$

where $\nabla\mathbf{f}$ is the $N \times N$ matrix with elements $\partial f_j / \partial x_k$. Since $\mathbf{x}^*$ is a root of the equations, we have $f(\mathbf{x}^*) = 0$ by definition. So the left-hand side of Eq. (7.14) is zero. Neglecting the higher-order terms and defining $\Delta\mathbf{x} = \mathbf{x}_i - \mathbf{x}^*$, we then obtain

$$\nabla\mathbf{f} \cdot \Delta\mathbf{x} = \mathbf{f}(\mathbf{x}_i). \tag{7.15}$$

This is a set of ordinary linear simultaneous equations of the form $\mathbf{A}\mathbf{x} = \mathbf{v}$. Thus we could solve for $\Delta\mathbf{x}$ by using Gaussian elimination or the Python function `solve` from the module `numpy.linalg`. Once we have solved for $\Delta\mathbf{x}$, our new estimate of the position of the root is

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \Delta\mathbf{x}. \tag{7.16}$$

Thus applying Newton's method for more than one variable involves evaluating the matrix of the first derivatives $\nabla\mathbf{f}$ at the point $\mathbf{x}_i$, then solving Eq. (7.15) for $\Delta\mathbf{x}$, and using the result to calculate the new estimate of the root via Eq. (7.16). This is much more complex situation than the single variable case. We can also calculate the derivatives numerically leading to a multivariable version of the secant method.