CHAPTER 4

ROOT FINDING

Finding roots of a function is a ubiquitous mathematical procedure in physics. For example, an equilibrium position of a particle in a potential field is determined as the place where the net force is zero, i.e., $U'(x) = 0$. In many cases, the equation is transcendental and no analytical solution is available. Even when analytical solutions are known, numerical evaluation of them may suffer from the round-off error. In this chapter, we discuss how to find a root of a given equation $f(x)$ within acceptable error ϵ (tolerance). There are various methods depending on available information. If the derivative of the function is known, some methods definitely utilize it. Other methods do not require the derivative and use other information. We need to choose a good algorithm for a given situation.

4.1 Quadratic, Cubic, and Quartic Polynomials

The roots of quadratic, cubic and quartic polynomials can be expressed in analytic expressions and we just evaluate them numerically. However, in some extreme cases, numerical errors become too large and a special care must be taken.

4.1.1 Quadratic Polynomials

The roots of quadratic equation:

$$ax^2 + bx + c = 0, \qquad a \neq 0 \tag{4.1}$$

are well known as

$$x_{1,2} = \frac{-b \mp \sqrt{b^2 - 4ac}}{2a} \tag{4.2}$$

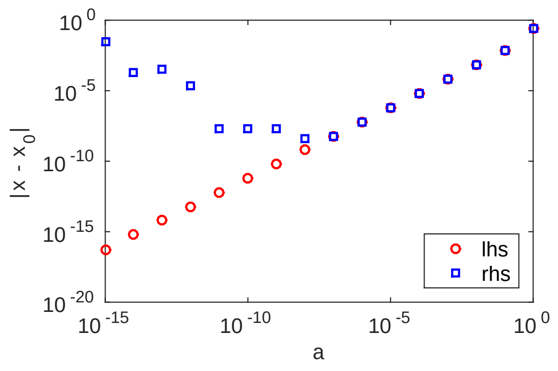


Figure 4.1: A root of the quadratic equation $\epsilon x^2 + x + 1/4 = 0$ is evaluated with two different methods with $\epsilon < 1$. The left hand side of Eq. (4.4) is initially approaching to the correct limit as ϵ decreases. However, it goes erroneous below $\epsilon = 10^{-5}$. On the other hand, the right hand side steadily converges to the right answer.

However, when $ac \ll b^2$, one of the solution suffers from the bit-off error as discussed in Chapter 1. Instead, the use of the following formula is recommended:

$$x_1 = \frac{-b - \text{sgn}(b)\sqrt{b^2 - 4ac}}{2a} \tag{4.3a}$$

$$x_2 = \frac{2c}{-b - \text{sgn}(b)\sqrt{b^2 - 4ac}} = \frac{c}{a x_1} \tag{4.3b}$$

where the signum function $\text{sgn}(x)$ is defined by

$$\text{sgn}(x) = \begin{cases} +1 & x > 0 \\ -1 & x < 0 \end{cases}$$

As demonstrated in Example 4.1, these expressions do not suffer from the round-off error.

EXAMPLE 4.1 Root of quadratic polynomials

In Eq. (4.3a), we used the equality.

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-2c}{b + \sqrt{b^2 - 4ac}} \tag{4.4}$$

When $|ac| \ll b^2$, the left hand side becomes numerically inaccurate. On the other hand, the right hand side has no such problem. We want to know how bad the left hand side is. Evaluate the both sides of the equality for $\epsilon x^2 + x + \dfrac{1}{4} = 0$ where $\epsilon = 0.1, 0.01, \cdots$. Reduce the value of ϵ until it hits the machine epsilon. Note that the exact answer for $\epsilon = 0$ is $x = -\dfrac{1}{4}$. Which side of Eq. (4.4) approaches to the correct limit? Program 4.1 evaluate the both sides of Eq. (4.4). The results are plotted in Fig. 4.1. The left hand side of Eq. (4.4) is initially approaching to the correct limit as ϵ decreases. However, it goes erroneous below $\epsilon = 10^{-5}$. On the other hand, the right hand side steadily converges to the correct answer. (This example is a solution to Problem 1.1.)

4.1.2 Cubic Polynomials

The roots of cubic polynomial

$$ax^3 + bx^2 + cx + d = 0, \quad a \neq 0 \tag{4.5}$$

can be obtain analytically. There are various mathematical descriptions of the solutions but most of them are not convenient for numerical calculations. The following procedure is simple and does not use complex numbers.[1] Without losing generality, we can assume $a = 1$. [The roots do not change if Eq (4.5) is divided by a.]

First, we evaluate the following quantities:

$$F = \frac{3c - b^2}{3}, \quad G = \frac{2b^3 - 9bc + 27d}{27}, \quad H = \frac{G^2}{4} + \frac{F^3}{27} \tag{4.6}$$

where H is a discriminant for Eq. (4.5). If $H < 0$, then there are three real roots:

$$x_1 = P + 2JM, \qquad x_2 = P - J(M + N), \qquad x_3 = P - J(M - N) \tag{4.7}$$

where

$$\begin{aligned} I &= \sqrt{\frac{G^2}{4} - H}, & J &= \sqrt[3]{I}, & K &= \arccos(-G/2I) \\ M &= \cos(K/3), & N &= \sqrt{3}\sin(K/3), & P &= -\frac{b}{3}. \end{aligned} \tag{4.8}$$

If $H = 0$, the above solutions are still valid but two of solutions are degenerate. If $H > 0$, then there is one real root and two others are complex:

$$x_1 = S + U - \frac{b}{3}, \quad x_{2,3} = -\frac{S + U}{2} - \frac{b}{3} \pm i\frac{(S - U)\sqrt{3}}{2} \tag{4.9}$$

where

$$S = \sqrt[3]{\sqrt{H} - \frac{G}{2}}, \quad U = -\sqrt[3]{\sqrt{H} + \frac{G}{2}} \tag{4.10}$$

In some cases, we don't get a good result due to the round-off error. Use an iterative method explained in Section 4.2 to see if the round-off error is small enough. Since the numerical results obtained by the present method is already close to the exact one, the iterative method should converge quickly (a few iterations are enough for most cases).

EXAMPLE 4.2 Cubic Polynomials

Let's try to find all roots of $x^3 - 9x^2 + 23x + 15 = 0$ using the formula given above. It can be factorized to $(x - 1)(x - 3)(x - 5) = 0$ and thus the exact solution is $x = 1$, 3, and 5. Program 4.2 finds them correctly.

```
Answer = 1.00000,   3.00000,   5.00000
```

4.1.3 Quartic Polynomials

The roots of quartic polynomial

$$ax^4 + bx^3 + cx^2 + dx + e = 0 \tag{4.11}$$

can be also solved analytically. Assuming $a = 1$ as before, we evaluate the following quantities:

$$F = -\frac{3b^2}{8} + c, \quad G = \frac{b^3}{8} - \frac{bc}{2} + d, \quad H = -\frac{b^4}{256} + \frac{b^2c}{16} - \frac{bd}{4} + e \tag{4.12}$$

If $G = 0$, then the four roots are

$$x_{1,2,3,4} = -\frac{b}{4} \pm \sqrt{\frac{-F \pm \sqrt{F^2 - 4H}}{2}} \tag{4.13}$$

If $G \neq 0$, then

$$x_{1,2} = -\frac{b}{4} + \frac{W \pm \sqrt{-\left(3F + 2Y + \frac{2G}{W}\right)}}{2} \tag{4.14a}$$

$$x_{3,4} = -\frac{b}{4} - \frac{W \pm \sqrt{-\left(3F + 2Y - \frac{2G}{W}\right)}}{2} \tag{4.14b}$$

where

$$Y = \begin{cases} -\dfrac{5F}{6} + U - \dfrac{P}{3U} & \text{if } U \neq 0 \\ -\dfrac{5F}{6} - \sqrt[3]{Q} & \text{if } U = 0 \end{cases}, \qquad W = \sqrt{H + 2Y} \tag{4.15}$$

$$P = -\frac{F^2}{12} - H, \qquad Q = -\frac{F^3}{108} + \frac{FH}{3} - \frac{G^2}{8}, \qquad U = \sqrt[3]{-\frac{Q}{2} \pm \sqrt{\frac{Q^2}{4} + \frac{P^3}{27}}} \tag{4.16}$$

Similar to quadratic and cubic equations, sever round-off error can happen, for example, when $H \ll F$ in Eq (4.13). Use an iterative method to improve the results.

4.2 Iterative Methods

In this section, we solve $f(x) = 0$ for x where $f(x)$ is a continuous function. There may be multiple solutions. Any root finding method needs a rough idea of the location of root you are looking for. The first step is to *bracket* the target root x^* between x_L and x_U such that $x_L < x^* < x_U$. It is important that there is only one root between x_L and x_U. It turns out that this is not an easy task for computer. Searching is something difficult even for computers to accomplish. On the other hand, human eyes can find the bracket easily if you are able to plot the function. Computers do not have an eye to view the whole curve. Therefore, the best practice is to plot the function and bracket the target root by visual inspection. However, in some cases it is desirable to have a robust numerical method to find the bracket. For example, when root finding is required many times during long computer simulation, you can't stop the simulation to visually inspect the bracket. There are simple algorithms of finding the bracket[2] but unfortunately no method guarantees the outcome. In this chapter, we assume that the bracket is done by direct visual inspection.

Any iterative method needs a criteria to stop the iteration. Ultimately, we stop it when the error is smaller than the tolerance. However, in practice we never know the exact error. If we knew it, we have the exact root! Therefore, we must carefully choose an ending criteria.

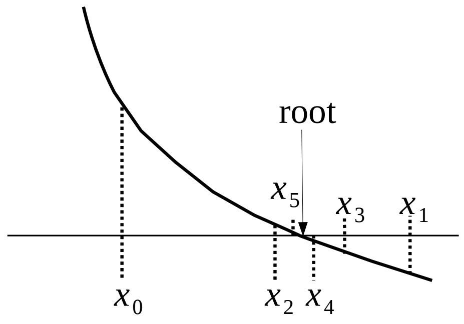


Figure 4.2: Bisection method. Starting with initial bracket (x_0, x_1), the bracket is at each iteration halved to (x_2, x_1), (x_2, x_3), (x_2, x_4), (x_5, x_4), $\cdots$.

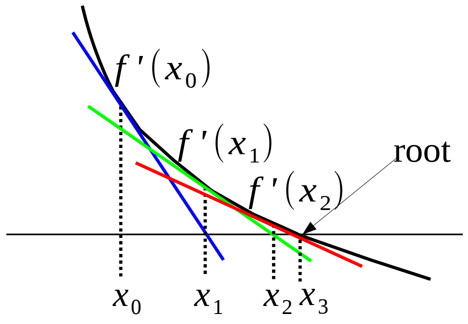


Figure 4.3: Newton-Raphson method. Starting with initial guess x_0, the line tangent to the curve at the current point x_n is used to find a new imroved root x_{n+1}. If the initial guess x_0 is close enough to the true root, this procedure rapidly converges to it.

4.2.1 Bisection method

Suppose we evaluate the function at two different points x_L and x_U. If $f(x_\text{L})f(x_\text{U}) < 0$, then we are sure that there is at least one root between the two points. This is a sufficient condition but not a necessary condition. When there is no root between the bracket, $f(x_\text{L})f(x_\text{U}) > 0$. However, when there are even number of roots in the bracket, the same inequality is valid. In the following we assume that a good bracket is found by visual inspection or other means so that there is only one root within the bracket.

Consider a mid point x_M between x_L and x_U. If $f(x_\text{L})f(x_\text{M}) < 0$, the root must be between x_L and x_M. Now we have a new bracket x_L and x_M. Otherwise, the root must be between x_M and x_U, which is the new bracket. Repeating this procedure, the root is isolated in a small region. The error must be smaller than $x_\text{U} - x_\text{L}$ and thus the iteration is terminated when $x_\text{U} - x_\text{L} <$ tolerence. This is the method of bisection. Figure 4.2 demonstrate how the bisection method works. The procedure of this iterative method is as follows.

Algorithm 4.1 Bisection method

1. Get a initial bracket x_L and x_U and a tolerance ϵ.
2. Make it sure that $f(x_\text{L})f(x_\text{U}) < 0$. Otherwise, stop and check the initial bracket.
3. Evaluate the function at the mid point $x_\text{M} = \frac{1}{2}(x_\text{L} + x_\text{U})$.
4. If $x_\text{U} - x_\text{L} < \epsilon$, x_M is the root and stop here. Otherwise continue.
5. If $f(x_\text{L})f(x_\text{M}) < 0$, then the root is between x_L and x_M. Let $x_\text{U} = x_\text{M}$ and go to step 3. Otherwise continue.
6. The root must be between x_M and x_U. Let $x_\text{L} = x_\text{M}$ and go to Step 3.

This method is generally robust and steadily approaches to the root. However, it is not a smart one and iterates many times before reaching the root. This method fails when the root is located at the edge of support since the root cannot be bracketed. For example, the bisection method is not able to solve $\sqrt{x-a} = 0$ simply because the function cannot be evaluated for $x < a$. It turns out that not only the bisection method but many other methods also fail for this function.

EXAMPLE 4.3 Bisection Method

Find a root of $x^3 - 9x^2 + 23x + 15 = 0$ between $x_1 = 1.5$ and $x_2 = 4$. Program 4.3 implement the above Bisection algorithm. With tolerance 10^{-6}, the bisection method converges to a root after 20 iterations. The root agrees with the exact answer. (See Example 4.2.)

```
Answer = 3.00000  (iteration= 20)
```

Exercise 4.1 Find two other roots.

Exercise 4.2 Find all roots using $x_\mathrm{U} - x_\mathrm{L}$ as the measure of error.

4.2.2 Newton-Raphson method

If the derivative of the function is known, there are faster methods utilizing it. Newton-Raphson method is one of those. Starting with an initial guess x_0, we approximate the function using Taylor expansion

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2 + \cdots \tag{4.17}$$

If x_0 is close to the root, we can ignore the higher order terms. Keeping only the first order term, the original equation $f(x) = 0$ is replaced by

$$f(x_0) + f'(x_0)(x - x_0) = 0 \tag{4.18}$$

and the root of this approximted equation is

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{4.19}$$

which should be closer to the root than x_0. Repeating this procedure, we have a recursive equation

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.20}$$

We stop the iteration when $|x_{n+1} - x_n| < \epsilon$ (tolerance). Is this a reasonable criteria? Using Taylor expansion around x_n,

$$f(x^*) = f(x_n) + (x^* - x_n) f'(x_n) \tag{4.21}$$

where x^* is the exact root. By definition, $f(x^*) = 0$. Then, the error is given by $|x^* - x_n| = \left| \frac{f(x_n)}{f'(x_n)} \right| = |x_{n+1} - x_n|$. As long as x_n is close to x^*, the criteria measures the error correctly.

Note that this method does not require initial bracketing. However, this method works great only if the initial point is sufficiently close to the root. Otherwise it is not guaranteed to converge to the target root. It might reach another root which you are not looking for. In order to avoid this failure, find a reasonably accurate root using the bisection method and switch to the Newton-Raphson method for the further improvement. While the bisection method is relatively robust many iterations are needed. On the other hand, the Newton-Raphson method converges faster. Therefore, such a hybrid method make a sense.

Algorithm 4.2 Newton-Raphson method

```
1. Set a tolerance $\epsilon$.
2. Choose an initial guess $x_0$ and let $n = 0$.
3. Estimate a new candidate by $x_{n+1} = x_n - \frac{f(x_n)}{f'(x)}$.
4. If $|x_{n+1} - x_n| < \epsilon$, then $x_{n+1}$ is the root. Stop the iteration.
5. If not, increment $n$ and go to step 3.
```

Exercise 4.3 The solution to $\sqrt{x-a} = 0$ is $x = a$. Can we find it by the Newton-Raphson method? Starting with $x_0 > a$, try to find x_1 and x_2. Verify that x_2 does not exist.

4.2.3 Secant method

The Newton-Raphson method requires the analytic expression of the first order derivative, which limits the range of applications. However, the same idea can be used without the analytic expression. If we replace the analytic derivative with the numerical derivative, the Newton-Raphson procedure still works. Using the backward finite difference method [Eq. (2.3)], the recursive equation (4.20) becomes

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n) \tag{4.22}$$

This method is commonly known as the secant method. Use the bisection method before starting the secant method to ensure the convergence.

One problem is that this method needs two points to start the iteration. However, this is not a big issue since we can pick any point close to the initial guess. If the root is already bracketed between a and b, use $x_1 = x_0 + \Delta$ with $\Delta \ll b - a$. If the secant method is preceded by the bisection method, use the final bracket from the bisection calculation. Then, $\Delta = (b-a)/10$ is good enough.

Algorithm 4.3 Secant method

```
1. Find bracket $a$ and $b$. Set a tolerance $\epsilon$.
2. Choose two initial points $x_0$ and $x_1 = x_0 + \frac{b-a}{10}$ and let $n = 1$
3. Estimate a new candidate by $x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$.
4. If $|x_{n+1} - x_n| < \epsilon$, then $x_{n+1}$ is the root. Stop the iteration.
5. If not, increment $n$ and go to step 3.
```

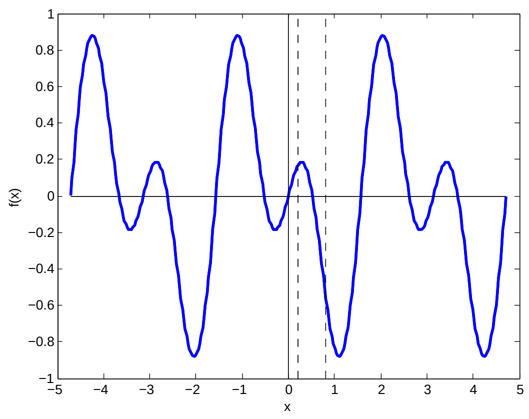


Figure 4.4: The function used in Example 4.4. The smallest positive root is bracketed between 0.2 and 0.8 (between the dashed lines) by visual inspection.

EXAMPLE 4.4 Multiple roots

Find the smallest positive root of $f(x) = \cos 3x \sin x$. Since this function has infinitely many roots, we have to be careful about choosing an initial guess. Figure 4.4 plots the function. By visual inspection, it is safe to say that the smallest positive root is between 0.2 and 0.8. We seek the solution with tolerance $\epsilon = 10^{-8}$. If only the bisection method were used, it takes 24 iterations and the answer is 0.52359877.

Program 4.4 runs 10 iterations of the bisection method and the Newton-Raphson and Secant methods follow. The output is shown below. The bisection method gets 0.52373047 which is close but not sufficient. Subsequent Newton-Raphson method iterates only two times and obtained 0.52359878. Similarly, the secant method also iterates twice and reaches the same answer as the Newton-Raphson. The exact answer is $\frac{\pi}{6} \approx 0.52359878$. All methods arrive the correct answer. However, the total number of iterations is 12 if the combination of the bisection and Newton-Raphson/secant methods is used, which is a half of the steps needed by the bisection method only. For complicated equations, the time saving can be significant.

```
Bisection      = 0.52373047  (iteration= 10)
Newton-Raphson = 0.52359878  (iteration= 2)
Secant         = 0.52359878  (iteration= 2)
```

Exercise 4.4 Find the second smallest positive root.

4.3 Applications in Physics Problems

4.3.1 Magnetic Phase Transition

The Ising model explains the ferromagnetic phase transition.[3] Using the mean field approximation, the magnetization per atom satisfies the following transcendental equation

$$m = \mu \tanh\left(\frac{Cm}{k_{\mathrm{B}}T}\right) \tag{4.23}$$

where C is a positive constant determined by the spin interaction between atoms, T is temperature, and μ is the magnetic moment each spin carries. We want to know the magnetization as a function of temperature by solving Eq. (4.23). By direct inspection $m = 0$ is always a solution. However, depending on temperature, there are more solutions. We want to plot all roots as functions of temperature. First, we simplify the expression by introducing normalized magnetization $\tilde{m} = m/\mu$ and normalized temperature $\tilde{T} = \frac{k_{\mathrm{B}}T}{\mu C}$. The normalized quantities are dimensionless and various constants disappear. We omit tilde in the normalized quantities to avoid cluttering. Then, we solve

$$f(m) \equiv m - \tanh(m/T) = 0 \tag{4.24}$$

utilizing its derivative

$$f'(m) = 1 - \frac{\operatorname{sech}^2(m/T)}{T} \tag{4.25}$$

To use the Newton-Raphson method, we need to start with a good initial guess. As Fig 4.5 shows, $\tanh(m) \to 1$ for $m \gg 1$. Therefore, the root must be below $m = 1$. As T approaches zero, the root approaches one. Therefore, $m = 1$ is a good starting point for low temperature case. As temperature increases, the root gradually decreases toward zero. Therefore, when temperature is lowered a little bit, the previous root can be a very good starting point of the next root finding process. We don't have to bracket the root in this case. The results are shown in Fig. 4.5b.

EXAMPLE 4.5 Bifurcation of magnetization

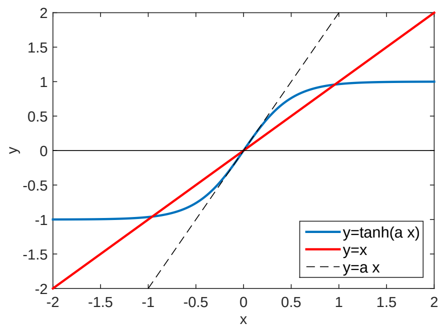
Program 17.2 finds the roots of Eq. (4.23) by Newton-Raphson method and plots them. The result is shown in Fig. 4.5b. Above the critical temperature $S_{\mathrm{C}} = 1$, there is only one root and the magnetization vanishes. Below S_{C}, positive and negative magnetization spontaneously appear. The root at $m = 0$ still exists but it is no thermodynamically not stable.

Exercise 4.5 Spontaneous vs induced magnetization

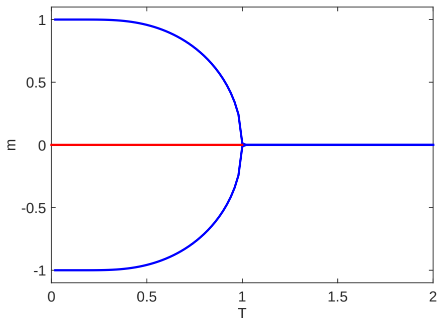
When external magnetic field H is applied, the mean-field equation becomes

$$m = \mu \tanh\left(\frac{\mu H + Cm}{k_{\mathrm{B}}T}\right) . \tag{4.26}$$

Modify Program 17.2 and plot the magnetization as a function of temperature with a fixed value of $H \neq 0$. Try various different values of H. How is the spontaneous magnetization shown in Fig. 4.5b modified by the magnetic field? [Use the normalized magnetic field $\tilde{H} = H/C$.]



(a) The magnetization is determiend by the crossing of $y = x$ and $y = \tanh(ax)$ where $a = 1/T$. When $a < 1$ $x = 0$ is only the solution. For $a > 1$, two more solutions appear. $a = 1$ corresponds to the critical point of the ferromagnetic transition.



(b) Bifurcation of magnetization. For $T > 1$ there is only one magnetic state $m = 0$. When temperature is lowered below a critical temperature $T_C = 1$, two stable states, one with positive m and the other with negative m. $m = 0$ is still a solution but unstable one. At $T = 0$, the magnetization reaches the maximum possible value $m = \pm 1$ where all electrons have the same spin state.

Figure 4.5: Ferromagnetic Phase Transition

4.3.2 Energy of a Quantum Particle in a Square Potential

A particle of mass m is trapped in a square well potential of depth V_0 and width $2a$. Energy eigenvalue E is determined by Schödinger equation

$$\left[-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x)\right]\psi(x) = E\psi(x) \tag{4.27}$$

Analytical calculation ends up with the following transcendental equations[4]

$$z\tan(z) - \sqrt{z_0^2 - z^2} \quad = \quad 0 \qquad \text{for symmetric states} \tag{4.28a}$$

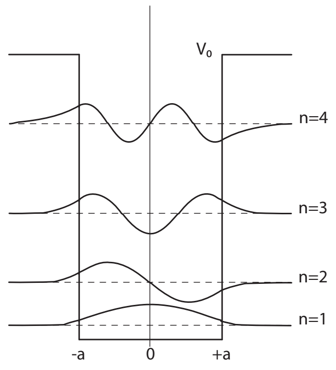
$$z\cot(z) + \sqrt{z_0^2 - z^2} \quad = \quad 0 \qquad \text{for anti-symmetric states} \tag{4.28b}$$

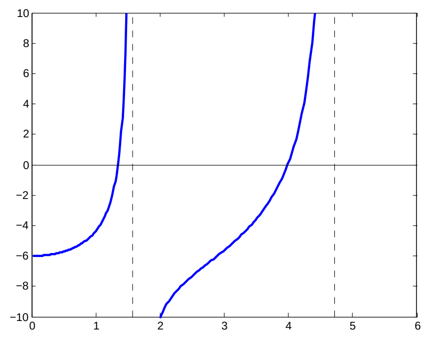
where

$$z = \sqrt{\frac{2ma^2E}{\hbar^2}} \tag{4.29a}$$

$$z_0 = \sqrt{\frac{2ma^2V_0}{\hbar^2}} \tag{4.29b}$$

The roots of Eqs (4.28) determines the energy eigenvalues. WE want to know all eigenvalues. However, the number of roots depends on the system parameters, which makes it difficult to bracket a root. We need to inspect the location of the roots. In Fig. the left hand side of Eq (4.28a) is plotted. First of all, we note that the function is defined for $z \in (0, z_0)$ and thus all roots must be between 0 and z_0. Considering $\tan x$ diverges at $2\pi(n \pm 1/4)$, each root must be between the diverging points. Hence, we bracket the roots in $(0, \frac{\pi}{2} - \delta)$, $(\frac{\pi}{2} + \delta, \frac{3\pi}{2} - \delta)$, $\cdots$, $2\pi(n + \frac{1}{4}) + \delta, z_0)$. Here δ is a small positive quantity to avoid the singularities. Program





(a) A square well potential with depth V_0 and width $2a$. Two symmetric and two anti-symmetric energy eigenstates are plotted.

(b) The left hand side of Eq (4.28a) is plotted for $z_0 = 6$. There are two roots. Clearly the root is bracketed by the dashed lines at $x = \frac{\pi}{2}$ and $\frac{3\pi}{2}$.

Figure 4.6: Quantum particle in a finite suare well

4.6 finds all roots for given z_0. Ten iterations of the bisection methods followed by the secant method find two roots.

```
root=1.34475105
root=3.98582621
```

4.3.3 Classical Turning Points

In Section 3.6.1, we tried to calculate the period of oscillation but we were not quite ready at that time because we did not know how to find the turning points. Now we can find them using a numerical root finding method. Suppose that a particle is bound in a potential

$$U(x) = U_0 \left[\sin\left(\frac{2\pi x}{L}\right) - \frac{1}{4}\sin\left(\frac{4\pi x}{L}.\right)\right] \tag{4.30}$$

We want to find the period of the bound motion using the formula (3.26). The turning points are the roots of $U(x) = E$. The potential is simple and we can find the analytic expression of its derivative. Hence, the Newton-Raphson method can be used to find the roots.

Exercise 4.6 Modify Program 4.4 and find a pair of the classical turning points of the potential (18.39). Using the results, find the period of oscillation.

4.3.4 Closest Approach in Scattering

In Section 3.6.2, we discussed how to evaluate the scattering angle (3.35). In order to carry out the integral, we still have to find the closest approach r_0 determined by Eq. (3.36). For the screened Coulomb potential, we are able to calculate the first order derivative of the equation. The Newton-Raphson method seems a good choice.

Exercise 4.7 Solve Eq. (3.36) for the Yukawa potential. Then, calculate the scattering angle, Eq. (3.35), using the method discussed in Section 3.6.2.

4.4 Problems

4.1 The interaction energy between two atoms is often described by the Morse potential[5]

$$U(r) = D_e \left[\mathrm{e}^{-2(r-r_0)/a} - 2\mathrm{e}^{-\alpha(r-r_0)/a} \right] \tag{4.31}$$

where r is the distance between the atoms. The parameters, D_e. r_0, and a are, dissociation energy, equilibrium distance, and the range of interaction, respectively. To simplify numerical calculation, we normalize energy and distance by D_e and a. In addition, we use the displacement from the equilibrium position as the coordinate. Then, the potential is simply

$$U(x) = \mathrm{e}^{-2x} - 2\mathrm{e}^{-x} \,. \tag{4.32}$$

were $x = (r - r_0)/a$. Find the classical turning points for $E = -0.5$ (This means the actual energy is $E = -D_e/2$ since we use D_e as the unit of energy.) Do not forget to plot the potential and visually inspect the turning point before using numerical root finding methods.

4.2 Find the eigenvalue of all asymmetric state by solving Eq. (4.28b) for $z_0 = 6$.

4.3 A quantum ball of mass m is bouncing on a hard floor under a uniform gravity g.[6] Schrödinger equation for the particle is

$$-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2\psi(y)}{\mathrm{d}y^2} + mgy\,\psi(y) = E\psi(y). \tag{4.33}$$

The wavefunction ψ must vanish at the floor (boundary condition). To find the energy eigenvalue, we first simplify the Schrödinger equation. Introducing a new variable $z = \alpha(y - E/mg)$ where

$$\alpha = \left(\frac{2m^2 g}{\hbar^2}\right)^{1/3} \tag{4.34}$$

Eq. (4.33) is rewritten as

$$\frac{\mathrm{d}^2\psi(z)}{\mathrm{d}z^2} = z\psi(z) \tag{4.35}$$

This equation is known as Airy equation and its solution is Airy function $\mathrm{Ai}(z)$. Hence, the energy eigenfunction is $\psi(z) = N\,\mathrm{Ai}(z)$ where N is a normalization constant. Now, we apply the boundary condition. At $y = 0$. $z = -\alpha E/mg$ and thus $\mathrm{Ai}(-\alpha E/mg) = 0$. This boundary condition indicates that the energy E is determined by the roots of the Airy function. Denoting the n-th root as z_n, the energy is $E_n = -mgz_n/\alpha$. The bouncing quantum ball has discrete energy! Find the smallest root of the Airy function and corresponding energy E_1. Use the built-in Airy function `airy()` in MATLAB. For C or Fortran, GNU Scientific Library (GSL)* provides Airy function.

*You can find GSL at `https://www.gnu.org/software/gsl/`.

MATLAB Source Codes

Program 4.1

```
%**********************************************************************
%*     Example  4.1                                                   *
%*     filename: ch04pr01.m                                           *
%*     program listing number: 4.1                                    *
%*                                                                    *
%*     This program finds roots of a quadratic equation               *
%*               a x^2 + x + 1/4 = 0                                  *
%*     for which the quadratic equation formula                       *
%*            x = (-1 + sqrt(1 - a))/(2a)                             *
%*     fails for small value of a.                                    *
%*                                                                    *
%*     Programed by Ryoichi Kawai for Computational Physics Course.   *
%*     Last modification:  10/13/2013.                                *
%**********************************************************************
clear all;

b=1;  % fixed parameters
c=0.25;

n=1;
a(1)=1;
while (a>eps())
  d = b^2-4*a(n)*c;
  x1(n) = (-b+sqrt(d))/(2*a(n));
  x2(n) = -2*c/(b+sqrt(d));
  fprintf('a= %22.16e, lhs=%22.16e, rhs=%22.16e \n',a(n),x1(n),x2(n));
  n=n+1;
  a(n)=a(n-1)/10;
end

% plot the results
p=loglog(a(1:n-1),abs(x2+0.25),'o',a(1:n-1),abs(x1+0.25),'s');
set(p(1),'Linewidth',2,'Color','red')
set(p(2),'Linewidth',2,'Color','blue')
xlabel('a','Fontsize',14)
ylabel(texlabel('|x - x_0|'),'Fontsize',14)  % x_0 = root at a=0
legend('lhs','rhs')
legend('Location','Southeast')
```

▲▲▲

Program 4.2

```
%**********************************************************************
%*     Example  4.2                                                   *
%*     filename: ch04pr02.m                                           *
%*     program listing number: 4.2                                    *
%*                                                                    *
%*     This program finds roots of a cubic equation                   *
%*              a*x^3 + b*x^2 + c*x + d = 0                           *
%*                                                                    *
%*     Programed by Ryoichi Kawai for Computational Physics Course.   *
%*     Last modification:  10/13/2013.                                *
%**********************************************************************
clear all;
```

```
% coefficients
b=-9; c=23; d=-15;

% formula for cubic polynomials
F=(3*c-b^2)/3;
G=(2*b^3 - 9*b*c + 27*d)/27;
H=G^2/4 + F^3/27;

yes = false;

if H>0
    S= (sqrt(H)-G/2)^(1./3.);
    U=-(sqrt(H)+G/2)^(1./3.);
    x = S+U-b/3;
    fprintf('Answer = %.5f\n',x);

else
    I=sqrt(G^2/4-H);
    J=I^(1./3.);
    K=acos(-G/(2*I));
    M=cos(K/3);
    N=sqrt(3)*sin(K/3);
    P=-b/3;
    x(1) = P-J*(M+N);
    x(2) = P-J*(M-N);
    x(3) = P+2*J*M;
    fprintf('Answer = %.5f,  %.5f,  %.5f \n',x);
end
```

▲▲▲

Program 4.3

```
%***********************************************************************
%*     Example  4.3                                                    *
%*     filename: ch04pr03.m                                            *
%*     program listing number: 4.3                                     *
%*                                                                     *
%*     This program finds roots of a cubic equation                    *
%*               a*x^3 + b*x^2 + c*x + d = 0                           *
%*     using the bisection method.                                     *
%*                                                                     *
%*     Programed by Ryoichi Kawai for Computational Physics Course.    *
%*     Last modification:  01/24/2018.                                *
%***********************************************************************
clear all;

% define a cubic equation
b = -9;
c = 23;
d =-15;

f = @(x) x^3+b*x^2+c*x+d;

% tolerance
epsilon=input('Enter tolerance =');

% initial bracket
x1=1.5;
x2=4;
f1 = f(x1);
f2 = f(x2);
```

```
% iteration counter
n=0;

% mid point
xm = (x1+x2)/2;
fm = f(xm);
while abs(fm) > epsilon
    if f1*fm < 0  % root in the lower half
        x2=xm;
        f2=fm;
    else          % root in the upper half
        x1=xm;
        f1=fm;
    end
    xm = (x1+x2)/2;  % new mid point
    fm = f(xm);
    n=n+1;
end

fprintf('Answer = %.5f  (iteration= %i)\n',xm,n);
```

▲▲▲

Program 4.4

```
%***********************************************************************
%*     Example 4.4                                                     *
%*     filename: ch04pr04.m                                            *
%*     program listing number: 4.4                                     *
%*                                                                     *
%*     This program seeks the root of                                  *
%*             cos(3*x)*sin(x)=0                                       *
%*     between x=0.2 and 0.8 using bisection, Newton-Raphson, and      *
%*     secant methods.                                                 *
%*                                                                     *
%*     Programed by Ryoichi Kawai for Computational Physics Course     *
%*     Revised on 01/07/2014.                                          *
%***********************************************************************
clear all;

% define function and its derivative
f=@(x) cos(3*x)*sin(x);
df=@(x) -3*sin(3*x)*sin(x) + cos(3*x)*cos(x);

% initial bracket
x1=0.2;
x2=0.8;
f1=f(x1);
f2=f(x2);
if f1*f2 > 0
   error('Bracket is incorrect');
end

% tolerance
epsilon=1e-8;

% bisection 10 iterations

% iteration counter
n=0;
```

```
% Bisection method (10 iterations)
xm = (x1+x2)/2;
fm = f(xm);
while n<10
    if f1*fm < 0  % root in the lower half
        x2=xm;
        f2=fm;
    else          % root in the upper half
        x1=xm;
        f1=fm;
    end
    xm = (x1+x2)/2;  % new mid point
    fm = f(xm);
    n=n+1;
end

fprintf('Bisection      = %.8f  (iteration= %i)\n',xm,n);

% Newton-Raphson method
x = xm;
fx = fm;
n = 0;
while abs(fx)> epsilon
    dfx = df(x);
    x = x - fx/dfx;
    fx = f(x);
    n = n+1;
end

fprintf('Newton-Raphson = %.8f  (iteration= %i)\n',x,n);

% Secant method
dx = (x2-x1)/10;
x1 = xm;
f1 = fm;
x2 = x1 + dx;
f2 = f(x2);
n = 0;
while abs(f2)> epsilon
    x = x2 - (x2-x1)/(f2-f1)*f2;
    x1 = x2;
    f1 = f2;
    x2 = x;
    f2 = f(x);
    n = n+1;
end

fprintf('Secant         = %.8f  (iteration= %i)\n',x,n);
```

Program 4.5

```
%***********************************************************************
%*     Example 4.5                                                     *
%*     filename: ch04pr05.m                                            *
%*     program listing number: 4.5                                     *
%*                                                                     *
%*     This program calculates magnetization as a function of          *
%*     temperature using the mean field Ising model:                   *
%*         x = x tan(x/S)                                              *
%*     where the variables are normalized as                           *
%*        x = m/m_0 and S=k*T/(C*m_0) .                                 *
%*                                                                      *
%*     Programed by Ryoichi Kawai for Computational Physics Course      *
%*     Revised on 01/07/2014.                                           *
%***********************************************************************
clear all;
N=101; % number of data points
tol=10^(-6);  % tolerance

dS = 2/(N-1); % step size in temperature

for i=1:N
    S=(i-1)*dS;  % temperature
    a=1/S;
    f=@(x) x - tanh(a*x);  % function
    df=@(x) 1 - a*sech(a*x)^2;  % derivative
    % Newton-Raphson method
    x=2;  % initial guess
    fx=f(x);
    while fx>tol
      x=x-fx/df(x);
      fx=f(x);
    end
    % store the magnetization and temperature
    m(i)=x;
    t(i)=S;
end

p=plot(t,m,t,-m,[0,1],[0,0]);
xlabel('T','fontsize',14);
ylabel('m','fontsize',14);
set(p(1),'linewidth',2,'color','blue');
set(p(2),'linewidth',2,'color','blue');
set(p(3),'linewidth',2,'color','red');
axis([0 2 -1.1 1.1]);
```

▲▲▲

Program 4.6 (a)

```
%***********************************************************************
%*     Secion 4.6.                                                      *
%*     filename: ch04pr06.m                                             *
%*     program listing number: 4.6(a)                                   *
%*                                                                      *
%*     Require: rootfinding.m                                           *
%*                                                                      *
%*     This program finds trhe energy eigenvalues of a pqrticle         *
%*     in a finite square well potential using the secant root          *
%*     finding methods.                                                 *
%*                                                                      *
%*     Programed by Ryoichi Kawai for Computational Physics Course      *
%*     Revised on 01/07/2014.                                           *
%***********************************************************************
clear all;

z0=6;  %system parameter

f=@(z) z*tan(z) - sqrt(z0*z0-z*z);  %define function

%control parameters
N=100;
```

```
K=ceil(z0/pi);  %The number of roots

% initial bracket
z1=0;
z2=pi/2;

for k=1:K
    dz = (z2-z1)/N;  % small shit
    z = rootfinding(f,z1+dz,z2-dz,10,10^(-6));
    fprintf('root=%.8f\n',z);
    z1 = z2;
    z2 = min([z0,z1 + pi]);
end
```

Program 4.7 (b)

```
%**************************************************************************
%*     Secion 4.6.                                                        *
%*     filename: rootfinding.m                                            *
%*     program listing number: 4.6(b)                                     *
%*                                                                        *
%*     Inputs                                                             *
%*         f = function name                                              *
%*         x1, x2 = bracket                                               *
%*         N = interations of bisection                                   *
%*         tol = tolerance for scant method                               *
%*     Output:                                                            *
%*         x = root                                                       *
%*                                                                        *
%*     This program find a root of a given funcion f(x)=0                *
%*     using the bisection and secant root finding method                 *
%*     finding methods.                                                   *
%*                                                                        *
%*     Programed by Ryoichi Kawai for Computational Physics Course        *
%*     Revised on 01/07/2014.                                             *
%**************************************************************************

function x=rootfinding(f,x1,x2,N,tol)

% check if the initial bracket satisfies the necessary condition.
f1=f(x1);
f2=f(x2);
if f1*f2 > 0
   error('Bracket is incorrect');
   display(x1,x2,f1,f2);
end

% Bisection method (N iteration)
n=0; % iteration counter

% mid point
xm = (x1+x2)/2;
fm = f(xm);
n=0;

while n<N
    if f1*fm < 0  % root in the lower half
        x2=xm;
        f2=fm;
    else          % root in the upper half
        x1=xm;
        f1=fm;
    end
    xm = (x1+x2)/2;  % new mid point
    fm = f(xm);
    n=n+1;
end

% Secant method
dx = (x2-x1)/10;
x1 = xm;
f1 = fm;
x2 = x1 + dx;
f2 = f(x2);
n = 0;
while abs(f2)> tol
    x = x2 - (x2-x1)/(f2-f1)*f2;
    x1 = x2;
    f1 = f2;
    x2 = x;
    f2 = f(x);
    n = n+1;
end
```

Python Source Codes

Program 4.1

```
#!/usr/bin/env python3
"""
%*************************************************************************
%*     Example  4.1                                                      *
%*     filename: ch04pr01.py                                             *
%*     program listing number: 4.1                                       *
%*                                                                       *
%*     This program finds roots of a quadratic equation                  *
%*               a x^2 + x + 1/4 = 0                                     *
%*     for which the quadratic equation formula                          *
%*            x = (-1 + sqrt(1 - a))/(2a)                                *
%*     fails for small value of a.                                       *
%*                                                                       *
%*     Programed by Ryoichi Kawai for Computational Physics Course.      *
%*     Last modification:  01/07/2017.                                   *
%*************************************************************************
"""
import numpy as np
import matplotlib.pyplot as plt

b=1.0
c=0.25
a=1.0
x=np.zeros(50)
y1=np.zeros(50)
y2=np.zeros(50)
n=0
while a > np.finfo(float).eps:
    x[n]=a
    d=b**2-4*a*c
    y1[n]=(-b+np.sqrt(d))/(2.0*a)
    y2[n]=-2*c/(b+np.sqrt(d))
    print("a={0:22.16e}, regular={1:22.16e}, smart={2:22.16e}"
          .format(x[n],y1[n],y2[n]))
    n+=1
    a=a/10.

plt.ioff()
plt.figure(figsize=(6,5))
plt.loglog(x[0:n],abs(y2[0:n]+0.25), 'ob', label='smart')
plt.loglog(x[0:n],abs(y1[0:n]+0.25), 'or', label='regular')
plt.legend(loc=4)
plt.xlabel('a')
plt.ylabel('x')
plt.show()
```

Program 4.2

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%***********************************************************************
%*  Example  4.2                                                       *
%*  filename: ch04pr02.py                                              *
%*  program listing number: 4.2                                        *
%*                                                                     *
%*     This program finds roots of a cubic equation                    *
%*               a*x^3 + b*x^2 + c*x + d = 0                           *
%*                                                                     *
%*     Programed by Ryoichi Kawai for Computational Physics Course.    *
%*     Last modification:  01/14/2017.                                 *
%***********************************************************************
"""
import numpy as np
# coefficients
b=-9.0; c=23.0; d=-15.0
k=0

# formula for cubic polynomials
F=(3.0*c-b**2)/3.0
G=(2.0*b**3 - 9.0*b*c + 27.0*d)/27.0
H=G**2/4.0 + F**3/27.0

yes = False

if H>0.0:
    S= (np.sqrt(H)-G/2.0)**(1./3.)
    U=-(np.sqrt(H)+G/2.0)**(1./3.)
    x1 = S+U-b/3.0
    print("Answer = {0:10.5f}".format(x1))

else:
    I=np.sqrt(G**2/4.0-H)
    J=I**(1./3.)
    K=np.arccos(-G/(2.0*I))
    M=np.cos(K/3.0)
    N=np.sqrt(3)*np.sin(K/3.0)
    P=-b/3.0
    x3=np.zeros(3)
    x3[0] = P-J*(M+N)
    x3[1] = P-J*(M-N)
    x3[2] = P+2.0*J*M
    print("Answer = {0:10.5f}, {1:10.5f}, {2:10.5f}"
          .format(x3[0],x3[1],x3[2]))
```

▲▲▲

Program 4.3

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*********************************************************************
%*     Example  4.3                                                  *
%*     filename: ch04pr03.m                                          *
%*     program listing number: 4.3                                   *
%*                                                                   *
%*     This program finds roots of a cubic equation                  *
%*               a*x^3 + b*x^2 + c*x + d = 0                         *
%*     using the bisection method.                                   *
%*                                                                   *
%*     Programed by Ryoichi Kawai for Computational Physics Course.  *
%*     Last modification:  10/13/2013.                               *
%*********************************************************************
"""
import numpy as np

# define a cubic equation
def f(x):
    return x**3-9.0*x**2+23.0*x-15

if __name__ == "__main__":

    tol=input("Enter tolerance =")  # Read a tolerabce from the console
    epsilon=np.float(tol)

    # initial bracket
    x1=1.5
    x2=4.0
    f1 = f(x1)
    f2 = f(x2)

    # iteration counter
    n=0

    # mid point
    xm = (x1+x2)/2.0
    fm = f(xm)

    while x2-x1 > epsilon:
        if f1*fm < 0:  # root in the lower half
            x2=xm
            f2=fm
        else:          # root in the upper half
            x1=xm
            f1=fm

        xm = (x1+x2)/2.0      # new mid point
        fm = f(xm)
        n+=1

    print("Answer = {0:7.5f}, (iteration = {1:3d})"
          .format(xm,n))
```

▲▲▲

Program 4.4

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*********************************************************************
%*     Example 4.4                                                   *
%*     filename: ch04pr04.m                                          *
%*     program listing number: 4.4                                   *
%*                                                                   *
%*     This program seeks the root of                                *
%*             cos(3*x)*sin(x)=0                                     *
%*     between x=0.2 and 0.8 using bisection, Newton-Raphson, and    *
%*     secant methods.                                               *
%*                                                                   *
%*     Programed by Ryoichi Kawai for Computational Physics Course   *
%*     Revised on 01/14/2017.                                        *
%*********************************************************************
"""
import sys
import numpy as np

# define function and its derivative
def f(x):
    return np.cos(3.0*x)*np.sin(x)

def df(x):
    return -3.0*np.sin(3*x)*np.sin(x) + np.cos(3.0*x)*np.cos(x);

if __name__ == "__main__":
    # initial bracket
    x1, x2 = input("Initial blacket (a pair o numbers) = ").split()
    x1, x2 = [np.float(x1), np.float(x2)]

    f1=f(x1)
    f2=f(x2)
    if f1*f2 > 0:
        sys.exit('Bracket is incorrect')

    # tolerance
    epsilon = input("Tolerance = ")
    epsilon = np.float(epsilon)

    # bisection 10 iterations
    n=0 # iteration counter

    xm = (x1+x2)/2.0
    fm = f(xm)

    while n<10:
        if f1*fm < 0:  # root in the lower half
            x2=xm
            f2=fm
        else:          # root in the upper half
            x1=xm
            f1=fm

        xm = (x1+x2)/2.0  # new mid point
        fm = f(xm)
        n+=1

    print("Bisection      = {0:12.8f}  (iteration= {1:4d})"
          .format(xm,n))

    # Newton-Raphson method
    x = xm
    fx = fm
    n = 0
    while abs(fx)> epsilon:
        dfx = df(x)
        x = x - fx/dfx
        fx = f(x)
        n+=1

    print("Newton-Raphson = {0:12.8f}  (iteration=  {1:4d})"
          .format(x,n))

    # Secant method
    dx = (x2-x1)/10.
    x1 = xm
    f1 = fm
    x2 = x1 + dx
    f2 = f(x2)
    n = 0
    while abs(f2)> epsilon:
        x = x2 - (x2-x1)/(f2-f1)*f2
        x1 = x2
        f1 = f2
        x2 = x
        f2 = f(x)
        n+=1

    print("Secant         = {0:12.8f}  (iteration=  {1:4d})"
          .format(x,n))
```

Program 4.5

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*********************************************************************
%*  Example 4.5                                                      *
%*  filename: ch04pr05.py                                            *
%*  program listing number: 4.5                                      *
%*                                                                   *
%*     This program calculates magnetization as a function of        *
%*     temperature using the mean field Ising model:                 *
%*         x = x tan(x/S)                                            *
%*     where the variables are normalized as                         *
%*         x = m/m_0 and S=k*T/(C*m_0) .                             *
%*                                                                   *
%*     Programed by Ryoichi Kawai for Computational Physics Course   *
%*     Revised on 01/14/2017.                                        *
%*********************************************************************
"""
import numpy as np
import matplotlib.pyplot as plt

def f(x,a):
    return x - np.tanh(a*x)

def df(x,a):
    return 1.0 - a/np.cosh(a*x)**2

if __name__ == "__main__":
    N=100 # number of data points
    tol=1.0e-6  # tolerance

    m=np.zeros(N+1)
    t=np.zeros(N+1)
    dS = 2.0/N   # step size in temperature


    for k in range(0,N):
        S=(k+1)*dS  # temperature
        a=1.0/S
        # Newton-Raphson method
        x=2.0    # initial guess
        fx=f(x,a)
        while fx>tol:
            x=x-fx/df(x,a)
            fx=f(x,a)

        # store the magnetization and temperature
        m[k]=x
        t[k]=S

    plt.ioff()
    plt.figure(figsize=(12,5))
    plt.plot(t,m, '-r')
    plt.plot(t,-m, '-r')
    plt.plot([0,1],[0,0],'--w')
    plt.ylim([-1.2,1.2])
    plt.xlabel('T')
    plt.ylabel('m')
    plt.savefig('ch04r05.pdf')
    plt.show()
```

Program 4.6 (a)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*************************************************************************
%*  Secion 4.6.                                                          *
%*  filename: ch04pr06.py                                                *
%*  program listing number: 4.6-1                                        *
%*                                                                       *
%*     Require: rootfinding.py                                           *
%*                                                                       *
%*     This program finds trhe energy eigenvalues of a pqrticle          *
%*     in a finite square well potential using the secant root           *
%*     finding methods.                                                  *
%*                                                                       *
%*     Programed by Ryoichi Kawai for Computational Physics Course       *
%*     Revised on 01/07/2014.                                            *
%*************************************************************************
"""

import numpy as np
import rootfinding as rf
```

```
def f(z):
    global z0
    return z*np.tan(z) - np.sqrt(z0*z0-z*z)

if __name__ == "__main__":
    global z0
    z0=6  #system parameter
    #control parameters
    N=100
    K=np.int(np.ceil(z0/np.pi))  #The number of roots

    # initial bracket
    z1=0.0;
    z2=np.pi/2.0

    for k in range(1,K+1):
        dz = (z2-z1)/N  # small shit
        z = rf.findaroot(f,z1+dz,z2-dz,10,1.0e-6)
        print("root={0:12.8f}".format(z))
        z1 = z2
        z2 = min([z0,z1 + np.pi])
```

Program 4.7 (b)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*************************************************************************
%*   Secion 4.6.                                                         *
%*   filename: rootfinding.py                                            *
%*   program listing number: 4.6-2                                       *
%*                                                                       *
%*     Inputs                                                            *
%*         f = function name                                             *
%*         x1, x2 = bracket                                              *
%*         N = interations of bisection                                  *
%*         tol = tolerance for scant method                              *
%*     Output:                                                           *
%*         x = root                                                      *
%*                                                                       *
%*     This program find a root of a given funcion f(x)=0                *
%*     using the bisection and secant root finding method                *
%*     finding methods.                                                  *
%*                                                                       *
%*     Programed by Ryoichi Kawai for Computational Physics Course       *
%*     Revised on 01/07/2014.                                            *
%*************************************************************************
"""
def findaroot(f,x1,x2,N,tol):

    f1=f(x1)
    f2=f(x2)
    if f1*f2 > 0:
        exit('Bracket is incorrect')

    # Bisection method (N iteration)
    n=0  # iteration counter

    # mid point
    xm = (x1+x2)/2.0
```

```
fm = f(xm)

while n<N+1:
    if f1*fm < 0.0:  # root in the lower half
        x2=xm
        f2=fm
    else:            # root in the upper half
        x1=xm
        f1=fm

    xm = (x1+x2)/2.0  # new mid point
    fm = f(xm)
    n+=1

# Secant method
dx = (x2-x1)/10.0
x1 = xm
f1 = fm
x2 = x1 + dx
f2 = f(x2)
n = 0
while abs(f2)> tol:
    x = x2 - (x2-x1)/(f2-f1)*f2
    x1 = x2
    f1 = f2
    x2 = x
    f2 = f(x)
    n+=1

return x
```

Bibliography

[1] Daniel Zwillinger. *CRC Stanbdard Mathematical Tables and Formula.* CRC Press, 35th edition, 2012. Chapter 2.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 3rd edition, 2007.

[3] David Chandler. *Introduction to Modern Statistical Mechanics.* Oxford University Press, 1987. Section 5.4.

[4] David Griffiths. *Introduction to Quantum Mechanics.* Pearson Prentice Hall, 2nd edition, 2005. Section 2.6.

[5] Donald A. McQuarrie. *Physical Chemistry: A Molecular Approach.* University Science Books, 1997. Chapter 5.

[6] Julio Gea-Banacloche. A quantum bouncing ball. *American Journal of Physics*, 67:776, 1999. See also references therein.