COMPUTATIONAL PHYSICS

ADITYA MEHTA

JULY 2023

Contents

1 Introduction

Computational physics is an interdisciplinary tool that involves use of computer programming to solve physics problems. It involves utilizing computers and numerical techniques to simulate, model, and analyze physical systems that are often too complex to be solved analytically. Computational physics has become an integral part of modern research in physics.

In condensed matter physics, computational methods enable scientists to investigate the behavior of materials at the atomic and molecular scale. Properties of substances, such as electrical conductivity, thermal behavior, and magnetic properties can be studied using simulations.

In the realm of astrophysics, researchers employ computational simulations to study the formation and evolution of galaxies, the behavior of black holes, and the generation of gravitational waves. These simulations provide a deeper understanding of the cosmos and aid in the interpretation of astronomical observations.

Computational physics also plays a crucial role in the study of high-energy particle physics. Large amount of data is generated in Particle accelerators such as the Large Hadron collider which requires sophisticated computational techniques to analyze. These simulations and data analysis tools aid in the discovery of new particles, the exploration of the fundamental forces and particles that make up the universe, and the validation of theoretical models.

Moreover, computational physics finds applications in fields like fluid dynamics, quantum mechanics, nuclear physics, and even biological systems. It enables the modeling of fluid flows, the understanding of quantum mechanical phenomena, the simulation of nuclear reactions, and the study of biological processes at the molecular level. The versatility of computational physics allows researchers to tackle complex problems across multiple disciplines.

In summary, computational physics is a powerful tool that combines the principles of physics with advanced computational methods. It enables scientists to simulate, model, and analyze complex physical systems, leading to a deeper understanding of the natural world. Computational physics continues to revolutionize scientific research, pushing the boundaries of our knowledge and contributing to technological advancements in numerous fields.

2 Integration

It is not possible to integrate every function. So, we often need to use approximate methods for integrals. For numerical integration, different methods are used such as trapezoidal rule, Simpson's rule etc. We will discuss a few of them.

2.1 Trapezoidal rule

2.1.1 Formula

Say we have an interval [a,b]. Let us divide this interval into n equal pieces. Therefore, the length of each piece is

$$h = (b-a)/n \tag{1}$$

and

$$x = a + jh, j = 0, 1, ...n \tag{2}$$

Now, the integral can be written as:

$$\begin{aligned} I(f) &= \int_a^b f(x)dx \\ &= \int_{x_0}^{x_n} f(x)dx \\ &= \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} f(x)dx \end{aligned} \tag{3}$$

Using trapezoidal rule,

$$I(f) = \int_a^b f(x)dx = \frac{h}{2}(f(a) + f(b)) \tag{4}$$

we can approximate the integral as

$$\begin{aligned} I(f) &= \sum_{j=0}^{n-1} \frac{h}{2}(f(x_j) + f(x_{j+1})) \\ &= h\left(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + ... + \frac{f(x_n)}{2}\right) \end{aligned} \tag{5}$$

2.1.2 Error in Trapezoidal rule

Let $f \in C^2[a,b]$. The error in Trapezoidal rule is given by

$$ME_T(f) = -\frac{f''(\eta)(b-a)^3}{12} \tag{6}$$

2.2 Simpson's rule

2.2.1 Formula

Simpson's rule is another way of approximating an integral. It gives exact result for polynomial of degree less than or equal to 2. The expression is given by:

$$I(f) = \int_a^b f(x)dx = \frac{b-a}{6}\left\{f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right\} \tag{7}$$

We can divide interval [a,b] into 2n equal parts where $n \in N$. For k = 0,1,2 ... 2n, define $x_k :== a + kh$ where $h = (b-a)/2n$. Now, applying Simpson's rule on interval $[x_{2i}, x_{2i+2}]$, we get

$$\int_{x_{2i}}^{x_{2i+2}} f(x)dx \approx \frac{2h}{6}\left\{f(x_{2i}) + 4f(x_{2i+1}) + f(x_{2i+2})\right\}$$

We now sum over $i = 0, 1, ...n-1$

$$\int_a^b f(x)dx \approx \frac{2h}{6}\sum_{i=0}^{n-1}\left\{f(x_{2i}) + 4f(x_{2i+1}) + f(x_{2i+2})\right\}$$

Therefore, the composite Simpson's rule takes the form:

$$I_S^n(f) = \frac{h}{3}\left[f(x_0) + f(x_{2n}) + 2\sum_{i=1}^{n-1} f(x_{2i}) + 4\sum_{i=0}^{n-1} f(x_{2i+1})\right] \tag{8}$$

2.2.2 Error in Simpson's rule

Let $f \in C^4[a,b]$. The error in Simpson's rule is given by

$$ME_S(f) = -\frac{f^{(4)}(\eta)(b-a)^5}{2880} \tag{9}$$

3 Differentiation

We often need to approximate derivatives in differential equation. Different formula such as forward difference formula, backward difference formula and central difference formula are used.

3.1 Approximations of first derivative

3.1.1 Forward difference formula

We use the definition of derivative of function

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{10}$$

It is approximated as

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \coloneqq D_h^+ f(x) \tag{11}$$

for sufficiently small $h > 0$.

3.1.2 Error in forward difference formula

Let $f \in C^3[a, b]$. The mathematical error in forward difference formula is given by

$$f'(x) - D_h^+ f(x) = -\frac{h}{2} f"(\eta) \tag{12}$$

where $\eta \in (x, x+h)$

3.1.3 Backward difference formula

The derivative of a function can also be written as

$$f'(x) = \lim_{h \to 0} \frac{f(x) - f(x-h)}{h} \tag{13}$$

It is approximated as

$$f'(x) \approx \frac{f(x) - f(x-h)}{h} \coloneqq D_h^- f(x)$$

for sufficiently small $h > 0$.

3.1.4 Error in backward difference formula

Let $f \in C^2[a, b]$. The mathematical error in backward difference formula is given by

$$f'(x) - D_h^- f(x) = \frac{h}{2} f"(\eta) \tag{14}$$

where $\eta \in (x-h, x)$

3.1.5 Central difference formula

In this case, we write the derivative of function as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h} \tag{15}$$

It is approximated as

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} := D_h^0 f(x) \tag{16}$$

for sufficiently small $h > 0$.

3.1.6 Error in central difference formula

Let $f \in C^3[a, b]$. The mathematical error in central difference formula is given by

$$f'(x) - D_h^+ f(x) = -\frac{h^2}{6} f'''(\eta) \tag{17}$$

where $\eta \in (x-h, x+h)$

It can be seen that central difference formula has better accuracy compared to forward difference and backward difference formula.

4 Non-Linear Equations

One of the most important problems in physics is to find solution to the equation:

$$f(x) = 0$$

More often than not, it is not possible to solve such a system exactly (for e.g. higher order polynomial equations). Hence, we resort to approximation methods. Suppose, r is a root for the equation. So, we tend to find r' such that $|r' - r| < \epsilon$ where ϵ is a very small number.
For solving such equations we use itreative method:

1. **Starting step:** Take one or more points as per the procedure $x_i \in [a, b](i = 0, 1, 2, ...m, m \in N)$. Consider x_m as an approximation to the root of $f(x) = 0$.

2. **Improvisation Step:** If x_m is not close to the required root, then repeat the above procedure until the root is close to actual root.

The iterative methods can be classified as open domain methods and closed domain methods.

1. **Closed domain methods:** In these methods, the location of root should be known in advance and we need to choose an interval around it. This guarantees that the iterative solution will converge to actual root.

2. **Open domain methods:** In these methods, there is no well-defined method to choose interval. An interval is chosen and iterative procedure is applied. However, in this method the iterative solution may not converge to actual solution.

4.1 Closed domain methods

4.1.1 Bisection Method:

This is by far the simplest method to find root. We simply have to keep dividing the interval into two parts until we converge to a solution. The following is the algorithm:

Algorithm:

1. Select an interval, say $[a_o, b_o]$ which includes the root

2. Find the value of function f(x) (whose root is to be determined) at points a, b and the midpoint of interval x_1

$$x_1 = \frac{a + b}{2} \tag{18}$$

3. Either $f(a)f(x_1) < 0$ or $f(b)f(x_1) < 0$. Chose the interval in which the product is less than zero. Label that interval as $[a_1, b_1]$.

4. Select a stopping criteria listed below and keep iterating until the criteria is satisfied.

4.2 Open Domain Methods

4.2.1 Secant Method

Since this is an open domain method, we don't need to know the location of root beforehand. We choose two points a_o and b_o such that $f(x_o) \neq f(x_1)$.

Algorithm:

1. For $n = 0, 1, 2...$ the iterative sequence is given by:

$$x_{n+1} = \frac{f(x_n)x_{n-1} - f(x_{n-1})x_n}{f(x_n) - f(x_{n-1})} \tag{19}$$

2. Choose a stopping criteria listed below and keep iterating until the criteria is satisfied.

The above expression can be re-arranged as:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \tag{20}$$

4.2.2 Newton-Raphson Method

In secant method, as n increases, x_{n-1} approaches x_n. Thus, for sufficiently large n,

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \tag{21}$$

Thus, if f(x) is differentiable, then on substituting the above expression in the iterative solution of secant method, we get the following iterative solution for Newton-Raphson method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{22}$$

4.3 Stopping Criteria

Stopping Criteria 1: Fix a $K \in N$ and stop when x_k is calculated.

Stopping Criteria 2: Find a real number $\epsilon > 0$ and a natural number N. Stop the iterations when $|x_k - x_{k-N} < \epsilon|$. It is usually more convenient to take N = 1 in which it becomes $|x_k - xk - 1| < \epsilon$.

Stopping Criteria 3: Find a real number $\epsilon > 0$ and a natural number N. Stop the iterations when

$$\left|\frac{x_k - x_{k-N}}{x_k}\right| < \epsilon \tag{23}$$

As above, we can conveniently $N = 1$

5 Differential Equations

Suppose we have a differential equation of the form:

$$y' = f(x, y) \tag{24}$$

where y(x) is unknown, x is independent variable and the function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. is a known function. We need to find a solution y for all x ∈ [a, b] subjected to initial condition:

$$y(x_{o)}) = y_o, \;\; x_o \in [a, b] \tag{25}$$

This is known as initial value problem.
Differential Equations are of immense use in physics. Examples include Newton's second law of motion, Simple Harmonic motion, Schrodinger Equation, Heat Equation, Wave Equation, Radioactive decay of isotopes etc. Many of the Ordinary Differential Equations we know cannot be solved exactly. Hence, we resort to approximation methods.

5.1 Euler Method

Suppose we have a first order differential equation of the form:

$$y'(x) = f(x_o, y(x_o)) \tag{26}$$

Say that we know the value of function at $x = x_o$. We can then replace the derivative on LHS with forward difference formula:

$$\frac{y(x_1) - y(x_0)}{h} \approx f(x_o, y(x_o) \tag{27}$$

which on rearranging gives

$$y(x_1) \approx y(x_o) + hf(x_o, y(x_o)) \tag{28}$$

This is known as Euler's method. Similarly, using backward difference formula will give us

$$y(x_1) \approx y(x_o) - hf(x_o, y(x_o)) \tag{29}$$

5.2 Runge-Kutta Method

The following is the formula for Runge-Kutta method of order two:

$$y_{j+1} = y_j + \frac{h}{2}(k_1 + k_2) \tag{30}$$

where

$$k_1 = f(x_j, y_j) \tag{31}$$
$$k_2 = f(x_{j+1}, y_j + hk_1) \tag{32}$$

6 Examples

Some of the examples are mentioned below. Few more were tried out but are not included in report. These example problems are borrowed from Computational Physics, a book by Mark Newman.

6.1 Plotting heat capacity of solids

According to Debye theory, the heat capacity of solid at a temperature T is given by

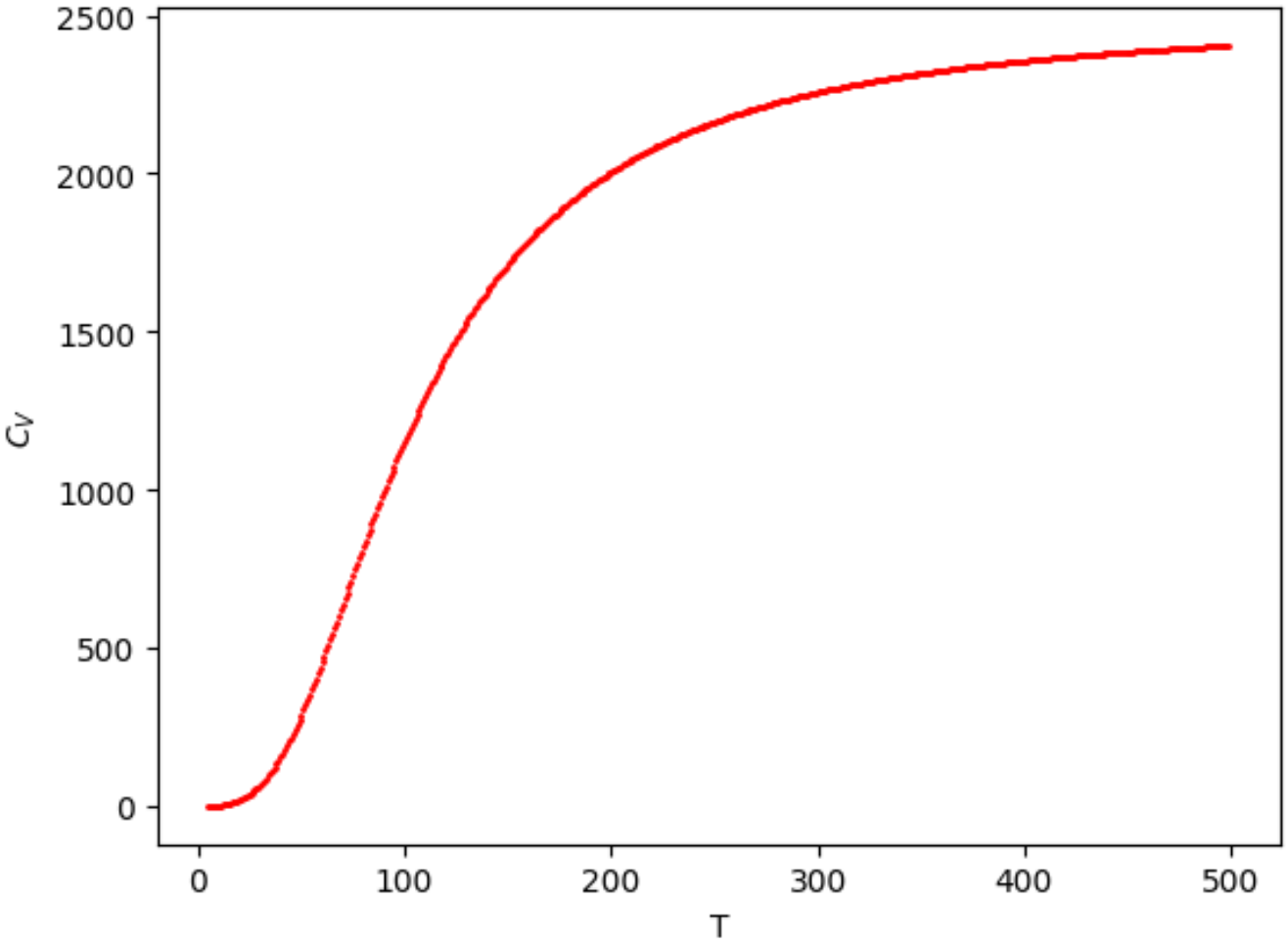
$$C_V = 9V\rho k_B \left(\frac{T}{\theta_D}\right)^3 \int_0^{\theta_D/T} \frac{x^4 e^x}{(e^x-1)^2} dx$$

where V is the volume of the solid, ρ is the density of the solid, k_B is the Boltzmann constant and θ_D is the Debye temperature, a property that depends on their density and speed of sound. The following is a code written to calculate the above integral:

```
import math
import matplotlib.pyplot as plt
import numpy as np

deb_temp = 428
V = 0.001
rho = 6.022e28
k_B = 1.380649e-23

for T in range (5, 500):
    h = 0.00001
    x = 0.000001
    integral = 0
    while x < deb_temp/T:
        integral = integral + ((x**4*math.e**x)/(math.e**x - 1)**2)
        x = x + h
    integral = integral * h
    C_V = 9*V*rho*k_B*T**3*integral/deb_temp**3
    plt.plot(T, C_V, marker = 'o', markersize = 1, color = 'red')
    plt.xlabel("T")
    plt.ylabel("$C_{V}$")
```



The above is a plot of C_V vs T. It can be compared with actual graph given in Debye theory of specific heat and be verified that it matches with great accuracy.

6.2 Calculating Stefann-Boltzmann constant

The total energy per unit area emitted by black body is given by

$$W = \frac{k_B^4 T^4}{4\pi^2 c^2 \hbar^3} \int_0^\infty \frac{x^3}{e^x - 1} dx \tag{33}$$

where T is the temperature of the solid, k_B is the Boltzmann constant, c is the speed of light and $\hbar$ is $\frac{h}{2\pi}$

The following code calculates the value of above integral. Temperature is the only parameter on RHS. Excluding that, the remaining term constitues a constant known as Stefann-Boltzmann constant, σ. Here is the code:

```
import numpy as np
import math

k_B = 1.3806452e-23
c = 299792458
h_cross = 1.054571817e-34

h = 0.0000001
x = 0.0000001
integral = 0

while x < 50:
    integral = integral + (x**3)/(math.e**x - 1)*h
    x = x + h
print(integral)

sigma = (k_B**4*integral)/(4*math.pi**2*c**2*h_cross**3)
print(sigma)
```

which gives the value of σ as $5.6703119989675985 * 10^{-8}$ which is quite close to the actual value. Here, we have not taken care of significant digits. The value of constants mentioned are not up to same decimal places and also the integral is not till infinity. So the answer does not match exactly. Nonetheless, it is still a powerful code as it gives a pretty close value.

6.3 Derivative of a function

Suppose we want to calculate the derivative of $f(x) = 1 + \frac{1}{2}tanh(2x)$ using central difference formula. The following is the code and also graph of derivative is also attached. Also, the derivative is calculated analytically and plotted on the same graph.
Here is the code:

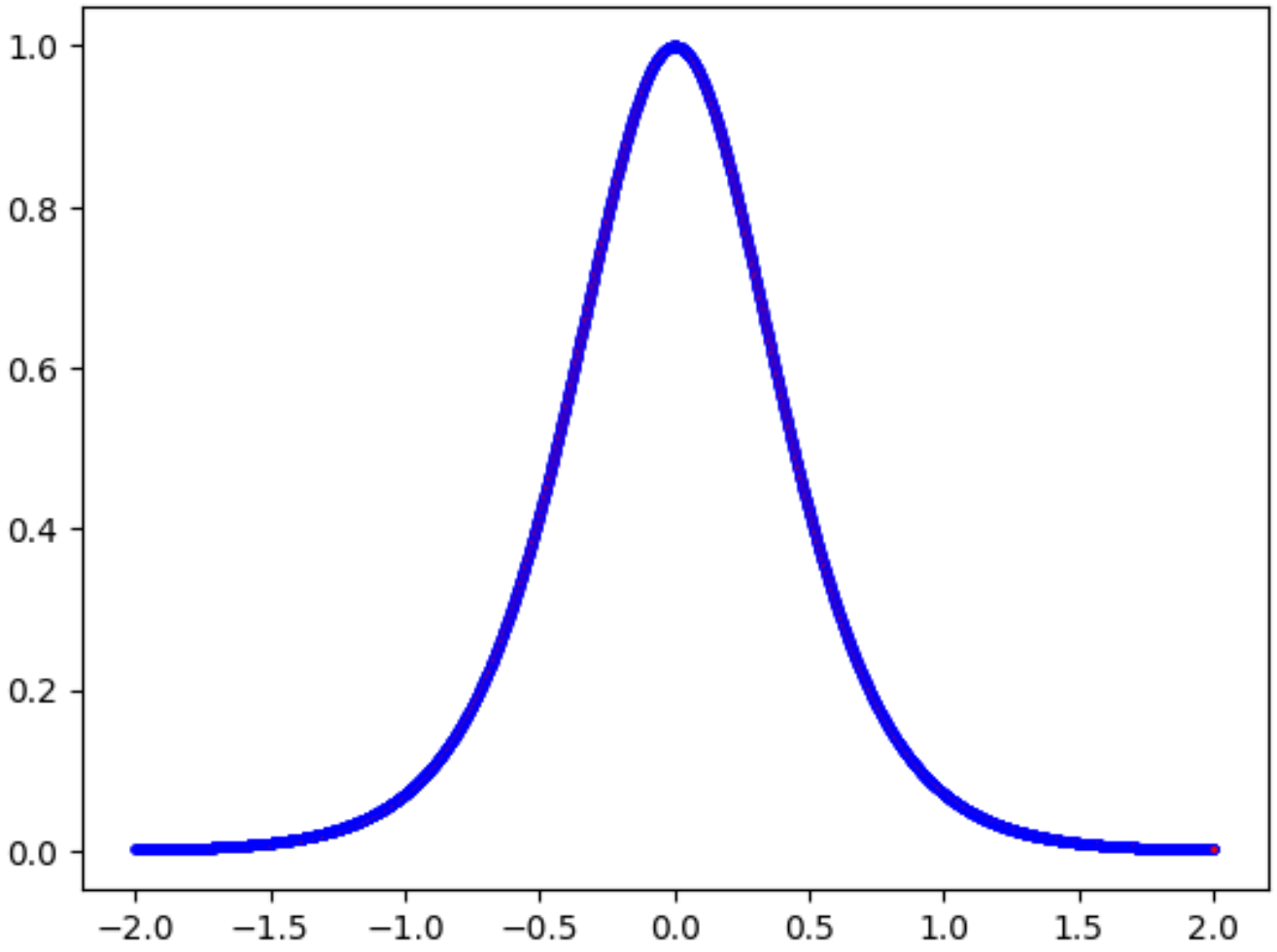
```
import matplotlib.pyplot as plt
import numpy as np
import math

h = 0.001
x = -2

def f(y):
    return 1 + (0.5*(math.tanh(2*y)))

while x < 2:
    f_1 = 0.5*(f(x+h) - f(x-h))/h
    f_analytic = 1 - (math.tanh(2*x))**2
    plt.plot(x, f_1, marker = 'o', markersize = '2', color = 'blue')
    plt.plot(x, f_analytic, marker = 'o', markersize = '1', color = 'red')
    x = x + h
```

The graph is shown on next page.



In this graph, we can the analytic curve cannot be seen as it completely overlaps with the numerical graph which shows that central difference formula works with a great accuracy.

7 2D Ising Model

Ising Model is a mathematical model for ferromagnetism in statistical mechanics. The Hamiltonian is given by:

$$H(\sigma) = -\sum_{<ij>} J_{ij}\sigma_i\sigma_j \tag{34}$$

where J_{ij} is the coupling constant between nearest neighbours. In this discussion, we are not considering any external magnetic field, although it can be included in the code easily. Here is the code for 2D Ising Model using Metropolis algorithm:

```
import numpy as np
import matplotlib.pyplot as plt
import random
import math

def algorithm(N, J, beta):

    E = 0
    E_new = 0
    dE = 0
    spin = np.zeros((N,N))

 #Generating random spin configuration
    for i in range (0, N):
        for j in range (0, N):
            spin[i][j] = random.randrange(-1, 2, 2)

#Calculating the energy of configuration
    def energy(spin_arr):
        E_val = 0
        for k in range (0, N):
            for l in range (0, N-1):
                E_val = E_val - J*spin_arr[k][l]*spin_arr[k][l+1]
        for n in range (0, N):
            for m in range (0, N-1):
                E_val = E_val - J*spin_arr[m][n]*spin_arr[m+1][n]
        return E_val

#Performing iterations
    for p in range (0, 10000):
        dE = 0
        x = random.randint(0, N-1)
        y = random.randint(0, N-1)
```

```
#Change in energy
        if x > 0:
            dE = dE + 2*J*spin[x-1][y]*spin[x][y]
        if x < N-1:
            dE = dE + 2*J*spin[x][y]*spin[x+1][y]
        if y > 0:
            dE = dE + 2*J*spin[x][y-1]*spin[x][y]
        if y < N-1:
            dE = dE + 2*J*spin[x][y]*spin[x][y+1]

        spin_i = spin[x][y]

        E = energy(spin)
        E_new = E + dE

#Comparing energy values
        if((E_new <= E) or (math.exp(-beta*(E_new-E)) >= np.random.random())):
            E = E_new
            spin[x][y] = spin_i * (-1)

        spin_avg = np.sum(spin)/N**2

#Plotting results
        plt.plot(p, spin_avg, marker = 'o', markersize = '3', color = 'red')
        plt.xlabel("Number of iterations")
        plt.ylabel("Average spin")

algorithm(10, 1, 0.1)
```

8 Heisenberg Model

Quantum Heisenberg Model is used to study critical points and phase transitions of magnetic systems in which spins are treated quantum mechanically. The following is the Hamiltonian for Heisenberg Model:

$$\hat{H} = -J\sum_{j=1}^{N}\sigma_j\sigma_{j+1} - h\sum_{j=1}^{N}\sigma_j \tag{35}$$

The following is a code for the exact diagonalisation of 1D quantum chain:

```
import numpy as np
#import matplotlib as plt

N = 12
basis = [[1 for x in range(0, N)]  for y in range(0, int(pow(2, N)))]

#defining basis
for pos in range (0, N):
    for i in range(0, int(pow(2, N))):
        if((i % int(pow(2, N-pos))) < int(pow(2, N-1-pos))): basis[i][pos] = 0

#print("The basis is:")
#print(basis)

H = np.zeros((int(pow(2, N)), int(pow(2, N))))
#print(H)

#reversing spin
def spin_reverse(spin_value):
    if(spin_value == 1): return 0
    elif(spin_value == 0): return 1

#converting binary to decimal
def bin_to_dec(arr):
    dec = 0
    for a in range(0, N):
        dec = dec + (arr[N-1-a]*int(pow(2, a)))
    return(dec)


#-ve sign should be adjusted with J_z and J_xy

#initialising Hamiltonian
def init_H(J_z, J_xy, h):
```

```
#Hamiltonian off-diagonal terms
for l in range(0, int(pow(2, N))):
    for p in range(0, N-1):

        offdiag_term = 0
        temp_array = [0 for a in range(0, N)]
        #count = 0

        #defining temporary array
        for m in range(0, N):
            value = basis[l][m]
            temp_array[m] = value
        #print(temp_array)

        if(temp_array[p] != temp_array[p+1]):
            #count = count + 1
            offdiag_term =  offdiag_term + 0.5

            spin_val = temp_array[p]
            spin_val_new = spin_reverse(spin_val)
            temp_array[p] = spin_val_new

            spin_val = temp_array[p+1]
            spin_val_new = spin_reverse(spin_val)
            temp_array[p+1] = spin_val_new

            H[bin_to_dec(temp_array)][l] =
            H[bin_to_dec(temp_array)][l] + J_xy*offdiag_term

    if(basis[l][N-1] != basis[l][0]):

        #defining temporary array
        for k in range(0, N):
            value = basis[l][k]
            temp_array[k] = value
        #print(temp_array)

        #count = count + 1
        offdiag_term = 0.5

        spin_val = temp_array[0]
        spin_val_new = spin_reverse(spin_val)
        temp_array[0] = spin_val_new

        spin_val = temp_array[N-1]
```

```
            spin_val_new = spin_reverse(spin_val)
            temp_array[N-1] = spin_val_new

            H[bin_to_dec(temp_array)][l] =
            H[bin_to_dec(temp_array)][l] +  J_xy*offdiag_term
            print("valid l = ", l)

    #Hamiltonian diagonal terms
    for j in range(0, int(pow(2, N))):
        diag_term = 0
        for k in range(0, N-1):
            if(basis[j][k] == basis[j][k+1]): diag_term = diag_term + 0.25
            else: diag_term = diag_term - 0.25
            #print("j = ", j, "diag term = ", diag_term)
            #periodic boundary conditions
        if(basis[j][0] == basis[j][N-1]): diag_term = diag_term + 0.25
        elif(basis[j][0] != basis[j][N-1]): diag_term = diag_term - 0.25
        H[bin_to_dec(basis[j])][bin_to_dec(basis[j])] = J_z*diag_term
    print("Hamiltonian after filling diagonal terms:")
    print(H)


    #External field

    for y in range(0, int(pow(2, N))):
        net_spin = 0
        for x in range(0, N):
            if(basis[y][x] == 1): net_spin = net_spin + 1
            elif(basis[y][x] == 0): net_spin = net_spin - 1
        H[y][y] = H[y][y] + h*net_spin


init_H(1, 1, 0)

eigval = []
eigvec = []
eigval, eigvec = np.linalg.eigh(H)
```

9 References

- Introduction to Numerical Analysis, S. Baskar and S. Sivaji Ganesh (for theory)
- Computational Physics, Mark Newman (for example problems)
- For 2D Ising Model refer `https://youtu.be/K--1hlv9yv0`
- For Exact Diagonalisation of 1D quantum chain refer `https://www.ryanlevy.science/physics/Heisenberg1D-ED/`