Ramakrishna Mission Vivekananda Centenary College
Rahara, Kolkata, West Bengal - 700118
College with Potential for Excellence (CPE)
Accredited by NAAC with Grade A++

PHSA CC-XIV: Statistical Mechanics

Laboratory Notebook

Rohan Deb Sarkar
Undergraduate Student
Department of Physics
Ramakrishna Mission Vivekananda Centenary College

End Semester Examination

Subject Code:	**PHSA CC-XIV**
Registration Number:	**A01-1112-111-020-2021**
Eamination Roll Number:	**2024161001**

April, 2024

Contents

1 Black Body Radiation

[1]:
```
from numpy import array, linspace, empty, exp, log, linalg
from matplotlib import pyplot as plt
from scipy.constants import h, c, k, pi
from scipy.integrate import simpson
import scienceplots

# Default configuration for matplotlib
plt.style.use(["science", "ieee", "grid"])
plt.rcParams["figure.figsize"] = (10, 5)
```

1.1 Planck's Distribution Law

The spectral distribution of of a black body radiation at a specific temperature T is given by,

$$\rho(\lambda, T) = \frac{8\pi hc}{\lambda^5} \frac{1}{\exp(hc/\lambda k_B T) - 1}$$

where,

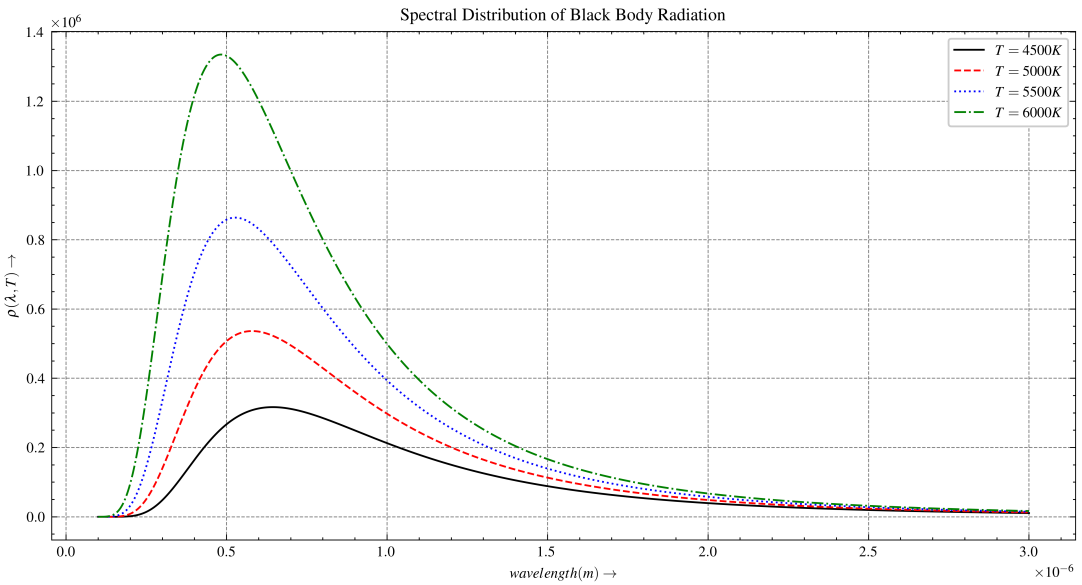
ρ represents energy density or spectral distribution,

λ represents wavelength of the radiation,

T represents the absolute temperature of the black body

[2]:
```
# Function to calculate the spectral distribution of a black body using Planck's distribution law
def planck_distribution(T, wavelength_i, wavelength_f, n=1000):
    wavelength = linspace(wavelength_i, wavelength_f, n)
    distribution = ((8 * pi * h * c) / wavelength**5) * (
        1 / (exp((h * c) / (wavelength * k * T)) - 1)
    )

    return wavelength, distribution
```

[3]:
```
wavelength_i = 0.1e-6
wavelength_f = 3e-6

T = 4500, 5000, 5500, 6000

for i in range(len(T)):
    wavelength, distribution = planck_distribution(T[i], wavelength_i, wavelength_f)

    plt.plot(wavelength, distribution, label=f"$T={T[i]}K$")

plt.title("Spectral Distribution of Black Body Radiation")
plt.xlabel("$wavelength (m) \\rightarrow$")
plt.ylabel("$\\rho (\\lambda, T) \\rightarrow$")
plt.legend()
plt.show()
```



1.2 Stefan-Boltzmann Law

The Stefan-Boltzmann law states that the total radiation emitted across all wavelengths is proportional to to the temperature T to the power N.

$$\int_0^\infty \rho(\lambda, T) d\lambda \propto T^N$$

$$R(T) = \int_0^\infty \rho(\lambda, T) d\lambda = \sigma\, T^N$$

If we take logarithm on both sides,

$$\log R = \log \sigma + N \log T$$

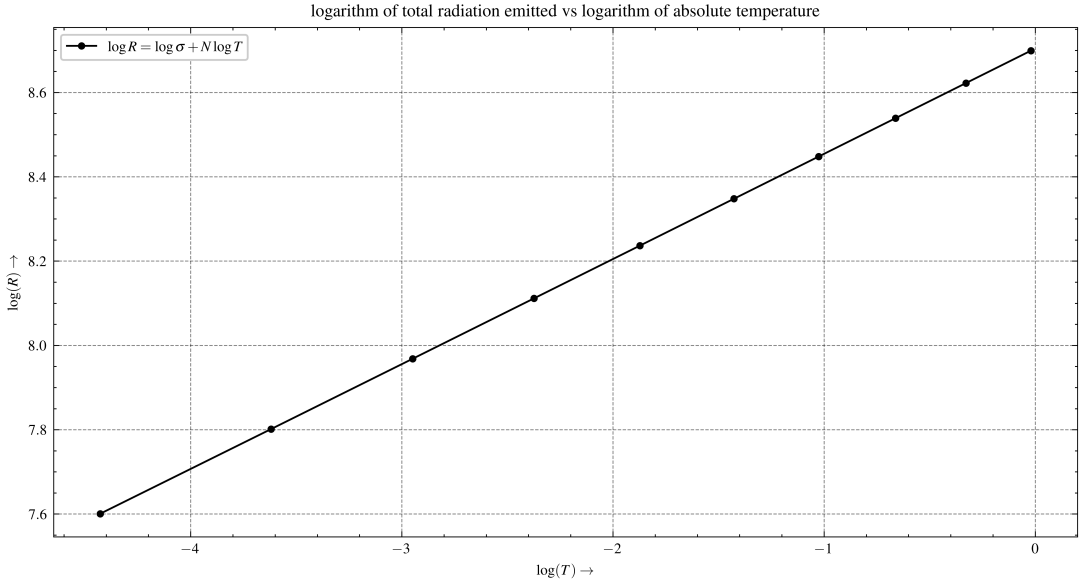
Plotting $\log R$ vs $\log T$, will give a straight line with slope N.

[4]:
```
# Function to integrate the spectral distribution
def integrate_planck_distribution(T_i, T_f, wavelength_i, wavelength_f, n=10):
    T = linspace(T_i, T_f, n)
    R = empty(len(T))

    for i in range(len(T)):
        wavelength, distribution = planck_distribution(T[i], wavelength_i,␣
 ↪wavelength_f)

        R[i] = simpson(distribution, x=wavelength)

    return T, R
```

[5]:
```
# Function to evaluate least square fit for linear data points
def least_square_fit(y, x):
    A = array([[sum(x**2), sum(x)], [sum(x), len(x)]])
```

```
    B = array([sum(x * y), sum(y)])

    m, c = linalg.solve(A, B)

    return m, c
```

[6]:
```
T_i = 2000
T_f = 6000

wavelength_i = 0.1e-6
wavelength_f = 10e-6

T, R = integrate_planck_distribution(T_i, T_f, wavelength_i, wavelength_f)

x, y = log(R), log(T)

N, lnS = least_square_fit(x, y)

plt.plot(x, y, ".-", label=f"$\\log R = \\log \\sigma + N \\log T$")
plt.title("logarithm of total radiation emitted vs  logarithm of absolute␣
 ↪temperature")
plt.xlabel("$\\log(T) \\rightarrow$")
plt.ylabel("$\\log(R) \\rightarrow$")
plt.legend()
plt.show()
```



[7]:
```
print("Value of N is:", N)
```

```
Value of N is: 4.011214109611678
```

2 Central Limit Theorem

The Central Limit Theorem is a fundamental concept in statistics that states that the distribution of the sum (or average) of a large number of independent, identically distributed random variables will be approximately normally distributed, regardless of the original distribution of the variables.

In simpler terms, it suggests that when you add up many random variables, even if they don't follow a normal distribution individually, their sum will tend to follow a normal distribution.

[1]:
```
import numpy as np
from matplotlib import pyplot as plt
from math import ceil, floor
import scienceplots

# Default configuration for matplotlib
plt.style.use(["science", "ieee", "grid"])
plt.rcParams["figure.figsize"] = (10, 5)
```

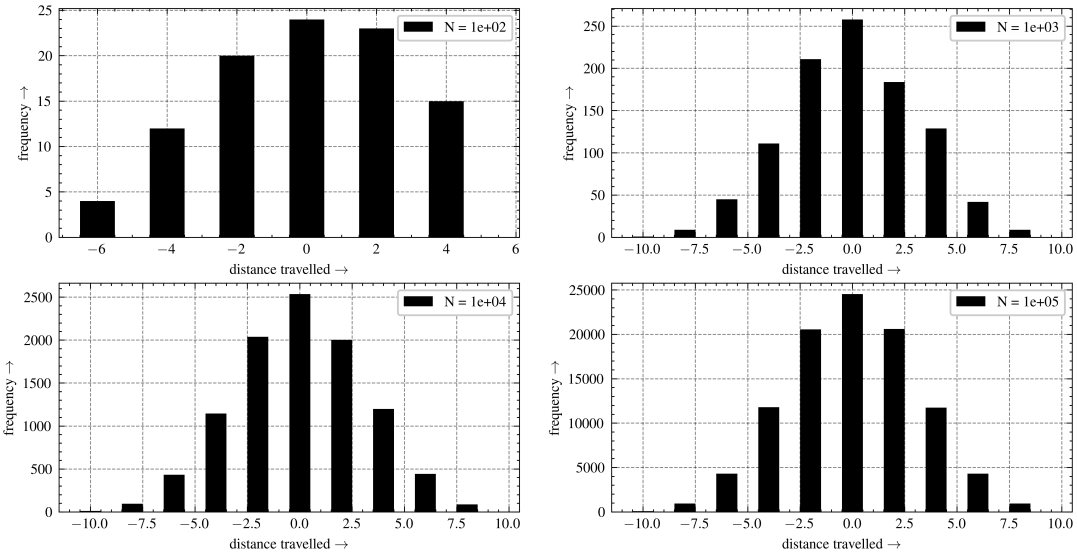
2.1 Random Walk in One-Dimension

Random walk in one-dimension is a mathematical concept describing a path where each step taken is determined by a random process, involving equal probabilities.

The summation of the distance traveled by a particle exhibiting random walk resembles the normal distribution when the experiment is repeated a large number of times.

[2]:
```
possible_steps = [+1, -1]
```

[3]:
```
n = 10  # Number of random walk samples taken in each experiment
N = 1e2, 1e3, 1e4, 1e5  # Number of experiments

for i in range(len(N)):
    # Randomly choosing `n` random walk samples
    rand_steps = np.random.choice(possible_steps, (int(N[i]), n))

    # Calculating distance travelled
    distance = np.sum(rand_steps, axis=1)

    plt.subplot(ceil(len(N) / 2), ceil(len(N) / 2), i + 1)

    # Plotting a histogram of distance travelled
    plt.hist(
        distance,
        bins=np.arange(floor(min(distance)) - 0.5, ceil(max(distance)) + 0.5, 1),
        label=f"N = {N[i]:.0e}",
    )
    plt.legend()
    plt.xlabel("distance travelled $\\rightarrow$")
    plt.ylabel("frequency $\\rightarrow$")

plt.show()
```



2.2 Collatz Conjecture

Consider the following operations for an arbitrary positive integer n,

$$n = \begin{cases} n/2, & \text{n in even} \\ 3n+1, & \text{n is odd} \end{cases}$$

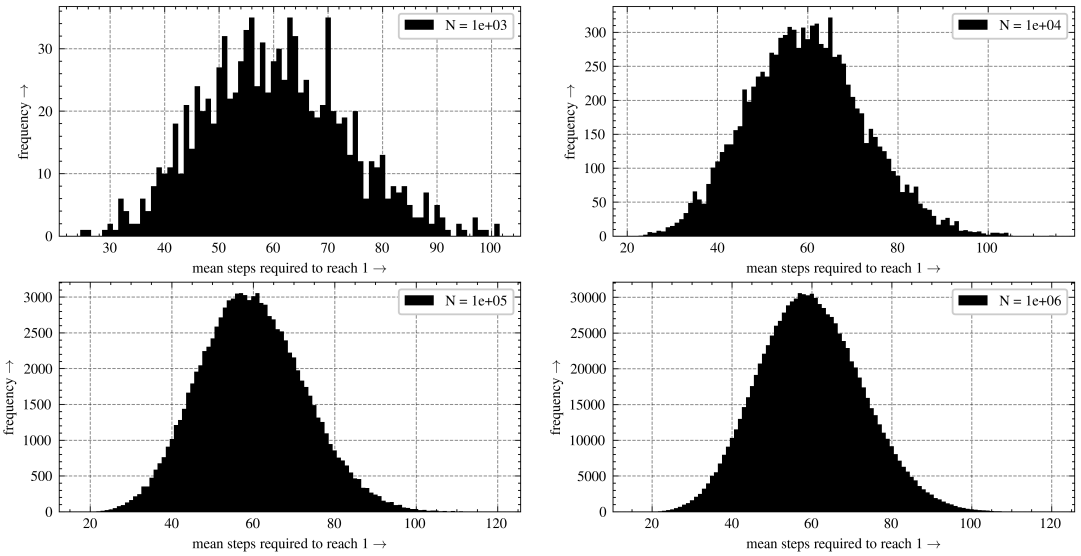
The conjecture states that repeating this process will eventually reach 1, regardless of which positive integer is chosen initially.

The steps required to reach 1 is calculated for set of positive integers. A random sample is chosen from them and the mean of steps required for each number in that sample is calculated. This experiment when repeated a large number of times resembles a normal distribution.

[4]:
```
# Function to calculate the number to steps required to reach 1
@np.vectorize
def collatz_steps_to_1(n):
    count = 0
    while n > 1:
        count += 1
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1

    return count


min_int = 5  # Starting positive integer for calculating required steps
max_int = 1000  # Ending positive integer for calculating required steps

# Steps required to reach 1 in Collatz conjecture from `min_int` to `max_int`
collatz_steps = collatz_steps_to_1(np.arange(min_int, max_int, 1))
```

[5]:
```
n = 10  # Number of samples taken in each experiment
N = 1e3, 1e4, 1e5, 1e6  # Number of experiments

for i in range(len(N)):
```

```
    # Randomly choosing `n` samples
    rand_steps = np.random.choice(collatz_steps, (int(N[i]), n))

    # Calculating the mean of the steps required of the `n` samples
    mean_steps = np.mean(rand_steps, axis=1)

    plt.subplot(ceil(len(N) / 2), ceil(len(N) / 2), i + 1)

    # Plotting a histogram of the mean steps required
    plt.hist(
        mean_steps,
        bins=np.arange(floor(min(mean_steps)) - 0.5, ceil(max(mean_steps)) + 0.5,␣
↪1),
        label=f"N = {N[i]:.0e}",
    )
    plt.legend()
    plt.xlabel("mean steps required to reach 1 $\\rightarrow$")
    plt.ylabel("frequency $\\rightarrow$")

plt.show()
```



3 Ising Model

The Ising model is a mathematical model used in statistical mechanics to study phase transitions in physical systems, particularly in magnetism. It was proposed by the physicist Ernst Ising in 1925 to describe the behaviour of magnetic spins in a lattice.

In the Ising model, each lattice site can be in one of two states, typically represented as "up" or "down" spins. The interactions between neighbouring spins are taken into account, described by an energy function that depends on the alignment of neighbouring spins. When spins are aligned, the energy is lowered, while misaligned spins increase the energy.

The energy function (hamiltonian) is given by,

$$H = -J \sum_{\langle i,j \rangle} s_i s_j - h \sum_i s_i$$

where,

J represents a constant specifying the strength of interaction,

h represents the magnitude of the external magnetic field applied,

s_i represents the spin of the i^{th} lattice point

[1]:
```
import numpy as np
from itertools import product
from matplotlib import pyplot as plt
import scienceplots

# Default configuration for matplotlib
plt.style.use(["science", "ieee", "grid"])
plt.rcParams["figure.figsize"] = (10, 5)

# Configuration for numpy prints
np.set_printoptions(formatter={"int": "{:+}".format})
```

3.1 Lattice configurations

1D Lattice structure with 4 lattice points:

s_0	s_1	s_2	s_3

where, $s_i \in \{+, -\}$

[2]:
```
possible_spins = [+1, -1]
lattice_points = 4  # Number of lattice points

T = 0.1 # Absolute temperature

# Determining all the possible lattice configurations
configurations = np.array(list(product(possible_spins, repeat=lattice_points)))

for i in range(len(configurations)):
    print(configurations[i])
```

```
[+1 +1 +1 +1]
[+1 +1 +1 -1]
[+1 +1 -1 +1]
[+1 +1 -1 -1]
[+1 -1 +1 +1]
[+1 -1 +1 -1]
```

```
[+1 -1 -1 +1]
[+1 -1 -1 -1]
[-1 +1 +1 +1]
[-1 +1 +1 -1]
[-1 +1 -1 +1]
[-1 +1 -1 -1]
[-1 -1 +1 +1]
[-1 -1 +1 -1]
[-1 -1 -1 +1]
[-1 -1 -1 -1]
```

3.2 Energy values for each lattice configuration

$$E_r = \frac{1}{N}\left[-J\sum_{\langle i,j\rangle} s_i s_j - h\sum_i s_i\right]$$

For our numerical computation, $J = 1$ and $h = 0.1$

[3]:
```python
# Function to calculate the absolute energy of a configuration
def absolute_energy(configurations, J, h):
    H = np.zeros(len(configurations))

    for i in range(len(configurations)):
        for j in range(len(configurations[i])):
            H[i] -= (
                J * configurations[i][j - 1] * configurations[i][j]
                + h * configurations[i][j]
            )

    return H / len(configurations)


J = 1
h = 0.1

energy = absolute_energy(configurations, J, h)

for i in range(len(configurations)):
    print(f"{configurations[i]} \t {energy[i]:+.3f}")
```

```
[+1 +1 +1 +1]    -0.275
[+1 +1 +1 -1]    -0.013
[+1 +1 -1 +1]    -0.013
[+1 +1 -1 -1]    +0.000
[+1 -1 +1 +1]    -0.013
[+1 -1 +1 -1]    +0.250
[+1 -1 -1 +1]    +0.000
[+1 -1 -1 -1]    +0.013
[-1 +1 +1 +1]    -0.013
[-1 +1 +1 -1]    +0.000
[-1 +1 -1 +1]    +0.250
[-1 +1 -1 -1]    +0.013
[-1 -1 +1 +1]    +0.000
[-1 -1 +1 -1]    +0.013
[-1 -1 -1 +1]    +0.013
[-1 -1 -1 -1]    -0.225
```

3.3 Partition function

$$\mathcal{Z} = \sum_r \exp(-\beta E_r)$$

where, $\beta = \frac{1}{k_B T}, k_B = 1$

[4]:
```
# Boltzmann constant
k = 1

# Function to calculate the partition function
def partition_function(configurations, T):
    energy = absolute_energy(configurations, J, h)

    Z = 0
    for i in range(len(configurations)):
        Z += np.exp(-energy[i] / (k * T))

    return Z


print("Partition Function:", partition_function(configurations, T))
```

```
Partition Function: 37.35711914040018
```

3.4 Probabilities for each lattice configuration

$$P_r = \frac{\exp(-\beta E_r)}{\mathcal{Z}}$$

[5]:
```
# Function to calculate the probabilities for each configuration
def configurations_probabilities(configurations, T):
    energy = absolute_energy(configurations, J, h)
    Z = partition_function(configurations, T)

    probabilities = np.empty(len(configurations))
    for i in range(len(configurations)):
        probabilities[i] = np.exp(-energy[i] / (k * T)) / Z

    return probabilities


probabilities = configurations_probabilities(configurations, T)

for i in range(len(configurations)):
    print(f"{configurations[i]} \t {probabilities[i]:+.3f}")
```

```
[+1 +1 +1 +1]    +0.419
[+1 +1 +1 -1]    +0.030
[+1 +1 -1 +1]    +0.030
[+1 +1 -1 -1]    +0.027
[+1 -1 +1 +1]    +0.030
[+1 -1 +1 -1]    +0.002
[+1 -1 -1 +1]    +0.027
[+1 -1 -1 -1]    +0.024
[-1 +1 +1 +1]    +0.030
[-1 +1 +1 -1]    +0.027
[-1 +1 -1 +1]    +0.002
[-1 +1 -1 -1]    +0.024
[-1 -1 +1 +1]    +0.027
```

```
[-1 -1 +1 -1]    +0.024
[-1 -1 -1 +1]    +0.024
[-1 -1 -1 -1]    +0.254
```

3.5 Average Energy

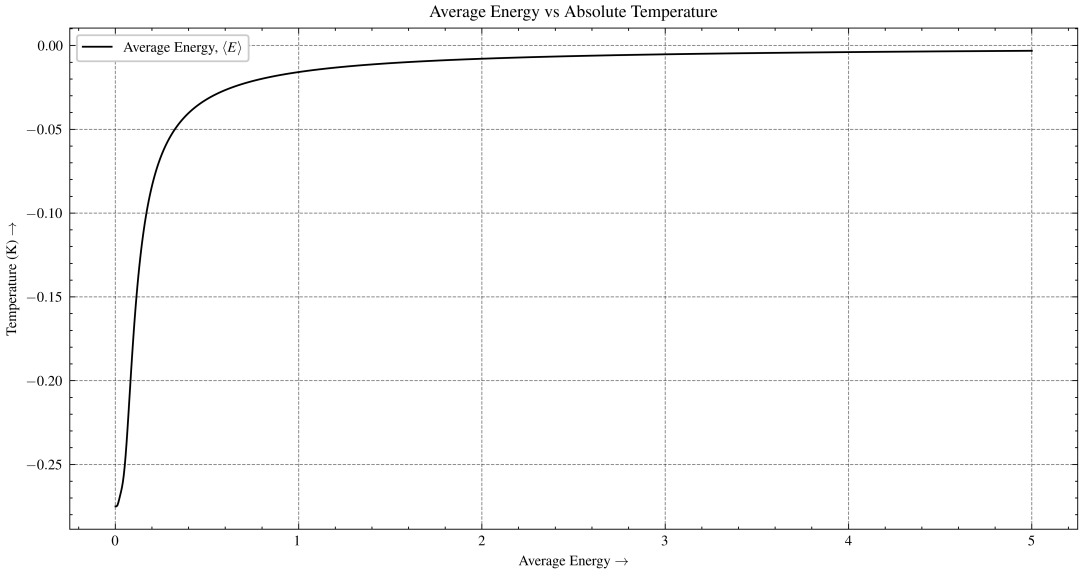
$$\langle E \rangle = \sum_r E_r P_r$$

[6]:
```
# Function to calculate the average energy
def average_energy(configurations, T):
    energy = absolute_energy(configurations, J, h)
    probabilities = configurations_probabilities(configurations, T)

    E = 0
    for i in range(len(configurations)):
        E += energy[i] * probabilities[i]

    return E


print("Average Energy:", average_energy(configurations, T))
```

```
Average Energy: -0.1715323493083448
```

3.5.1 Variation of average energy with absolute temperature

[7]:
```
T_i = 1e-3
T_f = 5
n = 1000

TT = np.linspace(T_i, T_f, n)

EE = np.empty(len(TT))

for i in range(len(TT)):
    EE[i] = average_energy(configurations, TT[i])

plt.plot(TT, EE, label="Average Energy, $\\langle E \\rangle$")
plt.title("Average Energy vs Absolute Temperature")
plt.xlabel("Temperature (K) $\\rightarrow$")
plt.ylabel("Average Energy $\\rightarrow$")
plt.legend()
plt.show()
```



3.6 Magnetization values for each lattice configuration

$$M_r = \frac{1}{N} \sum_i s_i$$

[8]:
```
# Function to calculate the absolute magnetisation for each configuration
def absolute_magnetization(configurations):
    M = np.empty(len(configurations))

    for i in range(len(configurations)):
        M[i] = np.sum(configurations[i])

    return M / len(configurations)


M = absolute_magnetization(configurations)

for i in range(len(configurations)):
    print(f"{configurations[i]} \t {M[i]:+.3f}")
```

```
[+1 +1 +1 +1]    +0.250
[+1 +1 +1 -1]    +0.125
[+1 +1 -1 +1]    +0.125
[+1 +1 -1 -1]    +0.000
[+1 -1 +1 +1]    +0.125
[+1 -1 +1 -1]    +0.000
[+1 -1 -1 +1]    +0.000
[+1 -1 -1 -1]    -0.125
[-1 +1 +1 +1]    +0.125
[-1 +1 +1 -1]    +0.000
[-1 +1 -1 +1]    +0.000
[-1 +1 -1 -1]    -0.125
[-1 -1 +1 +1]    +0.000
[-1 -1 +1 -1]    -0.125
[-1 -1 -1 +1]    -0.125
[-1 -1 -1 -1]    -0.250
```

3.7 Average magnetization

$$\langle M \rangle = \sum_r M_r P_r$$

[9]:
```
# Function to calculate the average magnetisation
def average_magnetization(configurations, T):
    magnetization = absolute_magnetization(configurations)
    probabilities = configurations_probabilities(configurations, T)

    M = 0
    for i in range(len(configurations)):
        M += magnetization[i] * probabilities[i]

    return M


print("Average magnetization:", average_magnetization(configurations, T))
```

```
Average magnetization: 0.04454438204788995
```

3.7.1 Variation of average magnetization with absolute temperature

[10]:
```
T_i = 1e-3
T_f = 3
n = 1000

TT = np.linspace(T_i, T_f, n)

MM = np.empty(len(TT))

for i in range(len(TT)):
    MM[i] = average_magnetization(configurations, TT[i])

plt.plot(TT, MM, label="Average magnetization$, \\langle M \\rangle$")
plt.title("Average magnetization vs Absolute Temperature")
plt.xlabel("Temperature (K) $\\rightarrow$")
plt.ylabel("Average magnetization $\\rightarrow$")
plt.legend()
plt.show()
```