ising_ensemble_avg

March 21, 2024

1 Ising Model

For a graph, $G = (E, V)$, defined by a set of edges, E, and vertices, V, we want to represent an Ising model, where the edge weights, w_{ij} are given by the spin interactions, i.e., $w_{ij} = J_{ij}$.

Given a configuration of spins (e.g., ↑↓↓↑↓) we can define the energy using what is referred to as an Ising Hamiltonian:

$$\hat{H} = \sum_{(i,j) \in E} J_{ij} s_i s_j$$

where, $s_i = 1$ if the i^{th} spin is `up` and $s_i = -1$ if it is `down`, and the sumation runs over all edges in the graph. **Note:** As we saw before, this Hamiltonian operator is simple, in that a single `BitString` returns a single energy. This is because the matrix representation of the Hamiltonian operator in the computational basis (i.e., basis of all possible `BitString`'s) is a diagonal matrix. However, most quantum mechanical Hamiltonians will not be diagonal, and in that case applying H to a single `BitString` would generate multiple `BitString`'s.

2 Thermodynamic averages

In the previous notebook, we used the Hamiltonian (which was defined as a graph) to find the lowest "energy" configuration (`BitString`). However, often times we want to compute average values of an observable over all possible configurations. Imagine that you have a bag containing some mixture of `BitString`'s. If we reach into the back and pull out a `BitString` at random, the probability of observing the specific `BitString` ket α will be denoted as $P(\alpha)$. Each possible `BitString` has its own probability.

Given this situation, what is the average energy in the bag? To answer this, we could just pull out each `BitString`, measure it's energy, add them all up, and divide by the total number of `BitString`s. Or if we knew the probabilty of observing each possible `BitString`, we could equivalently, add up the probabilities times the energy, $E(\alpha)$, of each possible `BitString`:

$$\langle E \rangle = \sum_{\alpha} P(\alpha) E(\alpha)$$

In this sense, the average energy (or any average quantity) depends on the given probability distribution in the bag.

While there are an infinite number of possible probability distributions one might interact with, a very common distribution (and the one we will focus on) is the `Gibbs Distribution`, also called the `Boltzmann Distribution`:

$$P(\alpha) = \frac{e^{-\beta E(\alpha)}}{Z} = \frac{e^{-\beta E(\alpha)}}{\sum_{\alpha'} e^{-\beta E(\alpha')}}$$

where β sometimes has a physical meaning of $\beta = 1/kT$, where k is the Boltzmann constant, $k = 1.38064852 \times 10^{-23} J/K$ and T is the temperature in Kelvin. We generally refer to the normalization constant Z as the partition function.

This expression, defines the probability of observing a particular configuration of spins, α. As you can see, the probability of pulling α out of your bag decays exponentially with increasing energy of α, $E(\alpha)$. This expression governs the behavior of the vast majority of physical systems, meaning that in nature at low temperatures, one typically expects to observe the lowest possible configuration of a system.

If the population (e.g., the bag of `BitString`s) is known to form a Boltzmann distribution, the expectation value of any quantity, `A`, can be defined as:

$$\langle A \rangle = \frac{\sum_\alpha e^{-\beta E(\alpha)} A(\alpha)}{Z}.$$

3 Properties

For any fixed state, α, the `magnetization` (M) is proportional to the *excess* number of spins pointing up or down while the energy is given by the Hamiltonian:

$$M(\alpha) = N_{\text{up}}(\alpha) - N_{\text{down}}(\alpha).$$

As a dynamical, fluctuating system, each time you measure the magnetization, the system might be in a different state (α) and so you'll get a different number! However, we already know what the probability of measuring any particular α is, so in order to compute the average magnetization, $\langle M \rangle$, we just need to multiply the magnetization of each possible configuration times the probability of it being measured, and then add them all up!

$$\langle M \rangle = \sum_\alpha M(\alpha) P(\alpha).$$

In fact, any average value can be obtained by adding up the value of an individual configuration multiplied by it's probability:

$$\langle E \rangle = \sum_\alpha E(\alpha) P(\alpha).$$

This means that to exactly obtain any average value (also known as an `expectation value`) computationally, we must compute the both the value and probability of all possible configurations. This becomes extremely expensive as the number of spins (N) increases.

The expectation values we will compute in this notebook are

$$\text{Energy} = \langle E \rangle \tag{1}$$

$$\text{Magnetization} = \langle M \rangle \tag{2}$$

$$\text{Heat Capacity} = \left(\langle E^2 \rangle - \langle E \rangle^2 \right) T^{-2} \tag{3}$$

$$\text{Magnetic Susceptibility} = \left(\langle M^2 \rangle - \langle M \rangle^2 \right) T^{-1} \tag{4}$$

4 Expectation values for Boltzmann Distribution

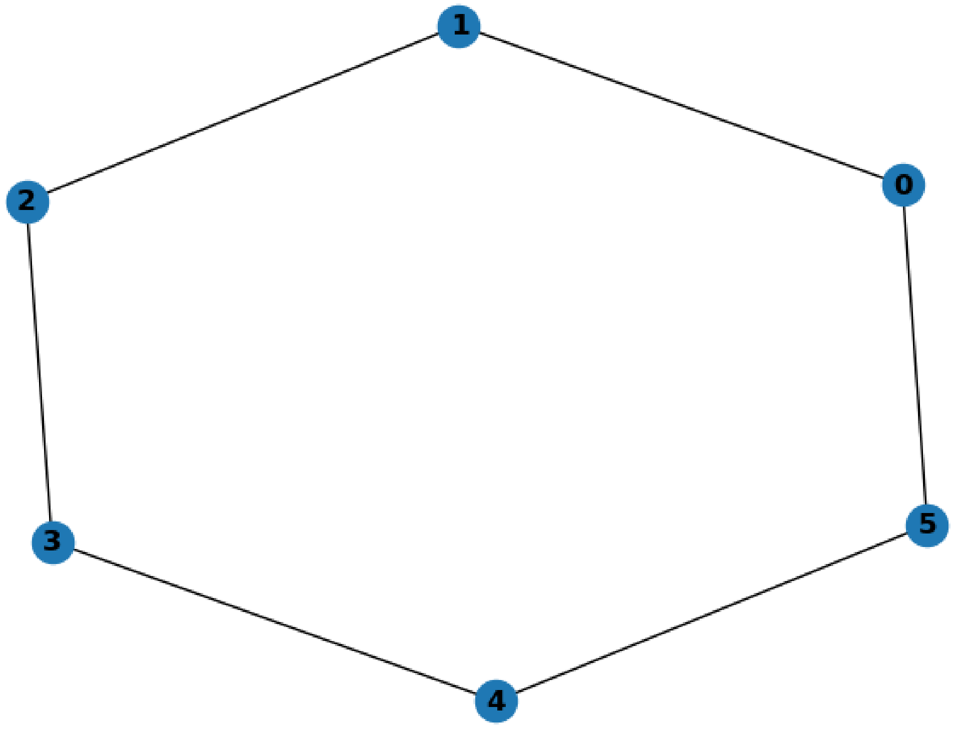
In this notebook, we will write code to compute the expectation values of a few different properties, at a given temperature. We will then see how these change with temperature.

4.1 Load packages

[ ]:
```python
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import random
import math
random.seed(2)
```

4.2 Create a graph that defines the Ising interactions

[ ]:
```python
N = 6
Jval = 2.0
G = nx.Graph()
G.add_nodes_from([i for i in range(N)])
G.add_edges_from([(i,(i+1)% G.number_of_nodes() ) for i in range(N)])
for e in G.edges:
    G.edges[e]['weight'] = Jval

# Now Draw the graph.
plt.figure(1)
nx.draw(G, with_labels=True, font_weight='bold')
plt.show()
```



4.3 Add your BitString class here:

[ ]:
```
class BitString:
    """
    Simple class to implement a config of bits
    """
    def __init__(self, N):
        self.N = N
        self.config = np.zeros(N, dtype=int)

    def __repr__(self):
        selfString = ''
        for bit in self.config:
            selfString += str(bit)
        return selfString

    def __eq__(self, other):
        return (self.config == other.config).all()

    def __len__(self):
        return self.N

    def on(self):
        num_on = 0
        for bit in self.config:
            if bit == 1:
                num_on += 1
        return num_on

    def off(self):
        num_off = 0
        for bit in self.config:
            if bit == 0:
                num_off += 1
        return num_off

    def flip_site(self,i):
        self.config[i] ^= 1

    def int(self):
        return int(str(self), 2)

    def set_config(self, s:list[int]):
        self.config = s

    def set_int_config(self, dec:int):
        # bin_string = bin(dec)[2:]
        # bin_string = bin_string.zfill(self.N)
        # for i in range(self.N-1):
        #     self.config[i] = bin_string[i]

        self.config = np.zeros(self.N, dtype=int)

        i = 1
        while dec != 0:
            self.config[-i] = dec % 2
            dec = dec // 2
            i += 1
```

4.4 Write your energy function here:

```
[ ]: def energy(bs: BitString, G: nx.Graph):
         """Compute energy of configuration, `bs`

             .. math::
                 E = \\left<\\hat{H}\\right>
```

```
    Parameters
    ----------
    bs    : Bitstring
        input configuration
    G     : Graph
        input graph defining the Hamiltonian
    Returns
    -------
    energy  : float
        Energy of the input configuration
    """
    # energy = sum of J_ij * S_i * S_j = J_ij * translation_thingy(S_i, S_j)
    # J represents whether or not the qubits are adjacent

    # translating 0 and 1 to -1 and 1
    def z1_to_n1(bit1, bit2):
        if bit1 == bit2:
            return 1
        else:
            return -1

    # array_J = nx.adjacency_matrix(G).todense()
    nrg = 0
    # for i in range(bs.N):
    #     for j in range(i, bs.N):
    #         nrg += array_J[i,j] * z1_to_n1(bs.config[i], bs.config[j])
    for (i,j) in G.edges:
        nrg += G.edges[(i,j)]['weight'] * z1_to_n1(bs.config[i], bs.config[j])

    return nrg
```

4.5 Write function to compute the thermodynamic averages

```
[ ]: def compute_average_values(bs:BitString, G: nx.Graph, T: float):
    """
    Compute the average value of Energy, Magnetization,
    Heat Capacity, and Magnetic Susceptibility

        .. math::
            E = \\left<\\hat{H}\\right>

    Parameters
    ----------
    bs    : Bitstring
        input configuration
    G     : Graph
        input graph defining the Hamiltonian
    T    : float
        temperature of the system
    Returns
    -------
    energy  : float
    magnetization  : float
    heat capacity  : float
    magnetic susceptibility  : float
    """

    # k = 1.38064852 * math.pow(10,-23)
    k = 1
    beta = 1/(k * T)

    ''' Probability of any bs '''
    def prob(bs):
        return math.exp(-beta * energy(bs, G))

    ''' Defining Z and E using the same loop'''
    Z = 0
    E = 0
    for i in range(2 ** bs.N):
        bs.set_int_config(i)
        Z += math.exp(-beta * energy(bs, G))
        E += prob(bs) * energy(bs, G)
    E /= Z

    ''' Defining magnetism: M(bs) = N_up(bs - N_down(bs)'''
    def mag(bs):
        x = 0
        for i in bs.config:
            x += 2*i - 1
        return x

    '''Expectation value function'''
    '''
    def ex_val(a:BitString, function(a)): # idk about this part chief
        EV = 0
        for i in range(2 ** a.N):
            bs.set_int_config(i)
            EV += prob(a) * EV(a)
        EV /= Z
        return EV
    '''

    '''Calculating M'''
    M = 0
    for i in range(2 ** bs.N):
        bs.set_int_config(i)
        M += prob(bs) * mag(bs)
    M /= Z

    '''Calculating heat capacity'''
    '''HC = (E2 - E**2) / T^2'''
    E2 = 0
    for i in range(2 ** bs.N):
        bs.set_int_config(i)
        E2 += prob(bs) * energy(bs, G) ** 2
    E2 /= Z
    HC = (E2-E**2) / (T**2)

    '''MS = (M2 - M**2) / T'''
    M2 = 0
    for i in range(2 ** bs.N):
        bs.set_int_config(i)
        M2 += prob(bs) * mag(bs)**2
    M2 /= Z
    MS = (M2-M**2) / T

    return E, M, HC, MS
```

[ ]:
```
# Define a new configuration instance for a 6-site lattice
conf = BitString(N)

# Compute the average values for Temperature = 1
E, M, HC, MS = compute_average_values(conf, G, 1)


print(" E  = %12.8f" %E)
print(" M  = %12.8f" %M)
print(" HC = %12.8f" %HC)
print(" MS = %12.8f" %MS)

assert(np.isclose(E,  -11.95991923))
assert(np.isclose(M,   -0.00000000))
assert(np.isclose(HC,   0.31925472))
assert(np.isclose(MS,   0.01202961))
```

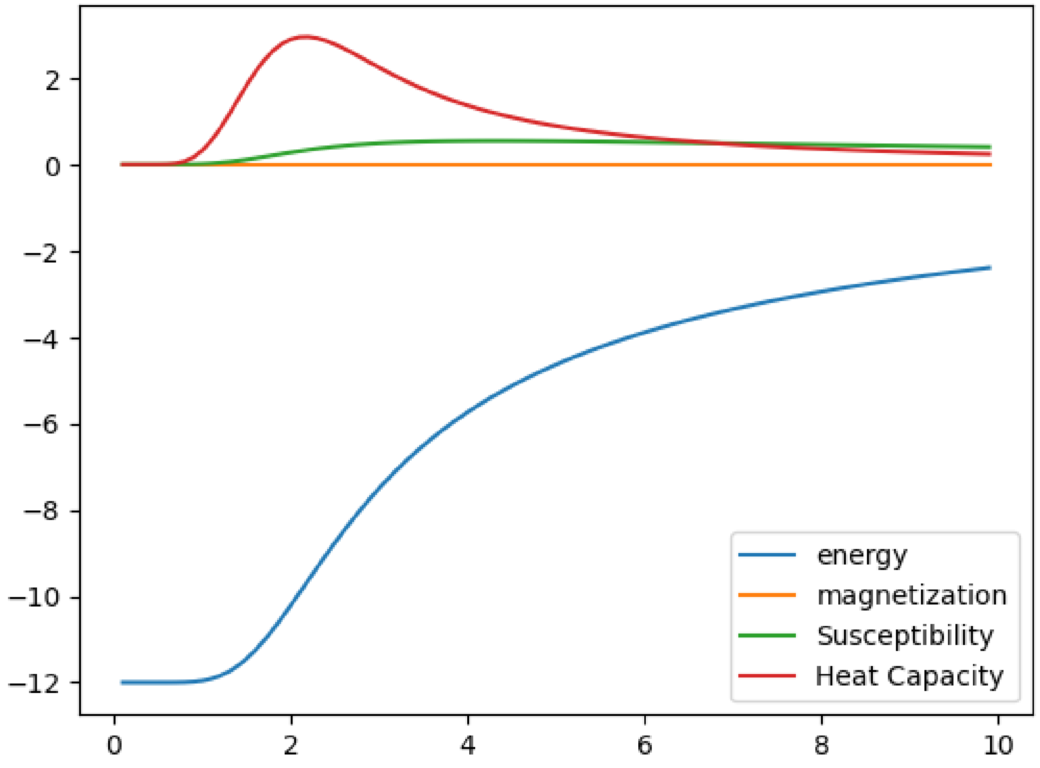
```
 E  = -11.95991923
 M  =  -0.00000000
 HC =   0.31925472
 MS =   0.01202961
```

5 Properties vs Temperature (exact)

```
[ ]: # Initialize lists that we will fill with the property vs. temperature data
e_list = []
e2_list = []
m_list = []
m2_list = []
T_list = []

# Create BitString
conf = BitString(N)
print(" Number of configurations: ", 2**len(conf))


for Ti in range(1,100):
    T = .1*Ti

    E, M, HC, MS = compute_average_values(conf, G, T)

    e_list.append(E)
    m_list.append(M)
    e2_list.append(HC)
    m2_list.append(MS)
    T_list.append(T)


plt.plot(T_list, e_list, label="energy");
plt.plot(T_list, m_list, label="magnetization");
plt.plot(T_list, m2_list, label="Susceptibility");
plt.plot(T_list, e2_list, label="Heat Capacity");
plt.legend();

Tc_ind = np.argmax(m2_list)
print(" Critical Temperature: %12.8f " %(T_list[Tc_ind]))
print("     E:  %12.8f" %(e_list[Tc_ind]))
print("     M:  %12.8f" %(m_list[Tc_ind]))
print("     HC: %12.8f" %(e2_list[Tc_ind]))
print("     MS: %12.8f" %(m2_list[Tc_ind]))
Tc2 = T_list[np.argmax(e2_list)]
print(" Critical Temperature: %12.8f" %(Tc2))

print(" E = %12.8f @ T = %12.8f"% (e_list[T_list.index(2.00)], 2.0))
```

```
 Number of configurations:  64
 Critical Temperature:   4.30000000
     E:   -5.36028889
     M:   -0.00000000
```

```
    HC:    1.18893365
    MS:    0.54308001
Critical Temperature:    2.20000000
E = -10.21957820 @ T =    2.00000000
```


energy
magnetization
Susceptibility
Heat Capacity