Report of FFR120 HM1: Chapter 1-4

Siyu Hu, gushusii@student.gu.se, ID: 19950910-3702

2024-11-11

1 Exercise 1 Brownian Model

Note: I have worked on this exercise for several days but I am sorry for that I don't have time to put code and figure in the report , I will finish before correction.

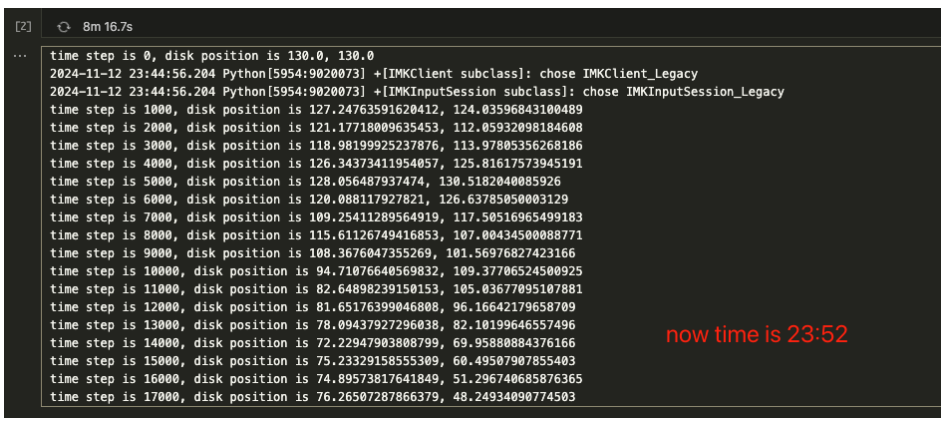


Figure 1: I don't have time to finish 80000 time steps before 23:59,

2 Exercise 2 Ising Model

2.1 Task1 P1

Plot magnetization as a function of temperature as Figure below:

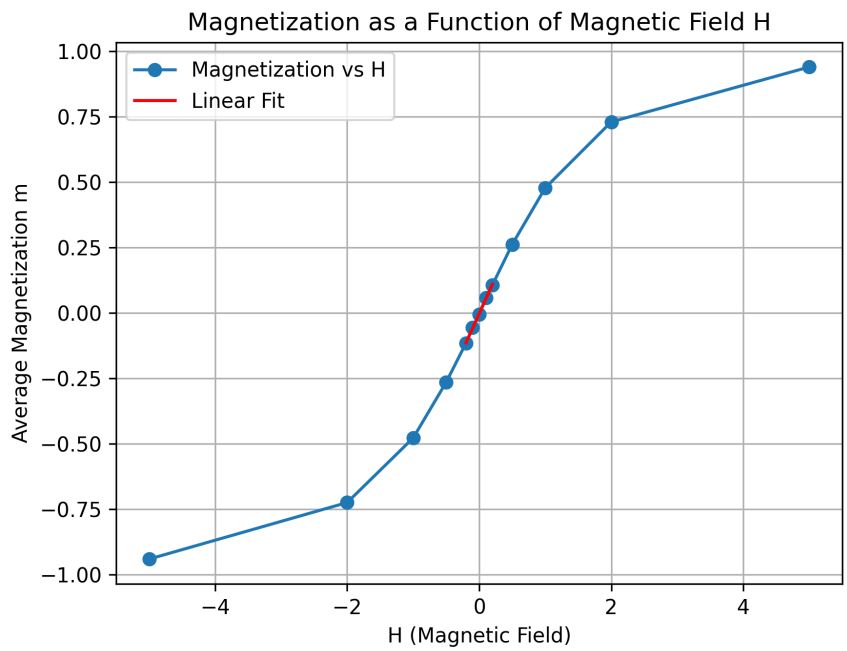


Figure 2: Curve of m(H), magnetization as a function of magnetic field

2.2 Task1 Q1

For small H, linear fit function is $m(H) = \chi * H$, with χ**: 0.5577459999999999**

2.3 Task2 P2

Plot magnetization as a function of temperature T as below. One can tell the critical temperature is about 2.8, because when temperature ¿= 2.8, the magnetization is closely to zero, which means that the system transform to a disorder state.

The theoretical critical temperature is 2.269 which has some differnce with this report. There are some possibly reasons for the error of simulation, in this report:

- N is equal to 200, which can not stand for larger and more complicate systems in real world.
- the time steps is equal to 5000, which is smaller than 10^5 in textbook Figure 2.5. This means that the system doesn't have enough time to reach stable state, especially near the critical temperature, so that the magnetization computation is not precisely.

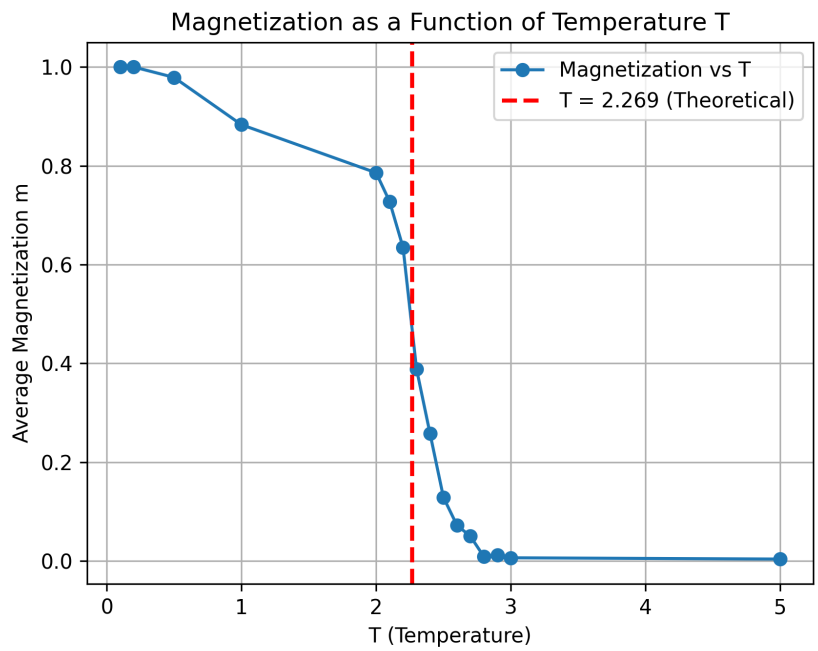


Figure 3: Curve of m(T), magnetization as a function of temperature

3 Exercise 3 Forest Fire

3.1 Notes of programming Setting

- For each N = [16, 32, 64, 128, 256, 512], repeat simulation 10 times; for N = 1024, repeat simulation twice.
- For each N, reached 300 times fire propagation.

3.2 P1

The exponents α_N as a function of N^{-1}, as shown below.

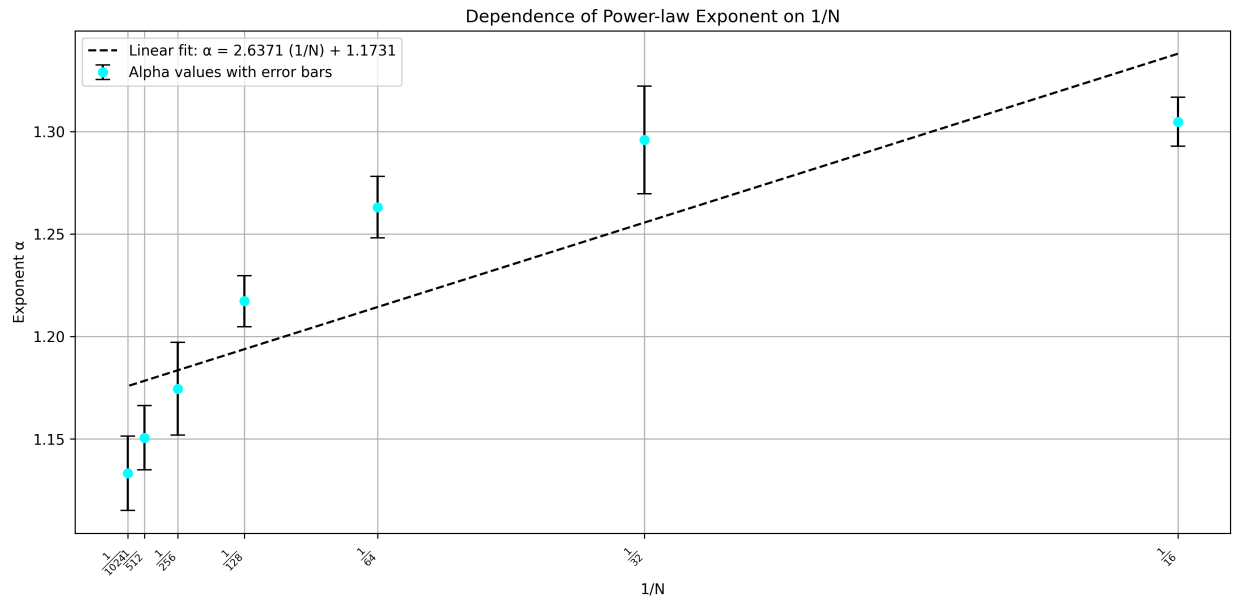


Figure 4: Denpendence of Power-law exponent over 1/N

3.3 Q1

The fit function of power-law exponent on 1/N is a linear function, using 7 points to fit.

$$\alpha = 2.6371 * (1/N) + 1.1731$$

When $N \to \infty$ $(1/N \to 0)$, $\alpha = 1.1731$.

Compare to the Figure 3.6 in the textbook, one can find that:

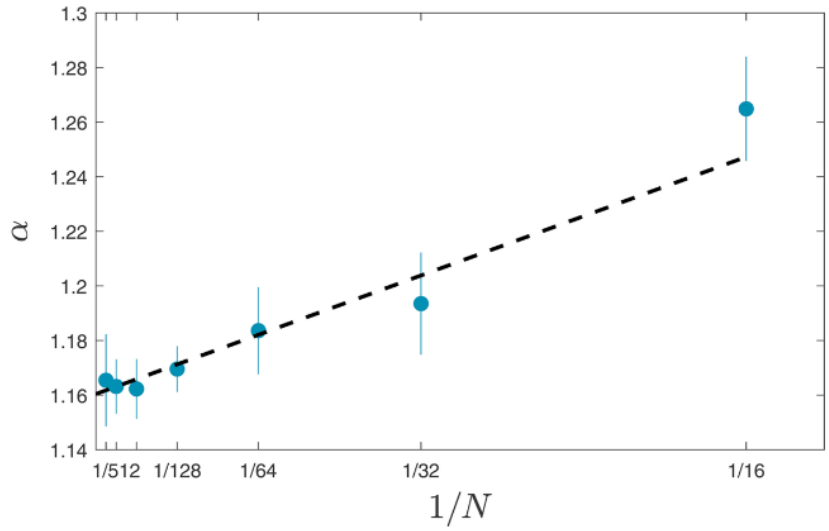


Figure 3.6. Dependence of the power-law distribution exponent as a function of the inverse of the forest size. The

Figure 5: Textbook Figure 3.6

- Figure 3.6 in the book, the fitted line has a gentle slope, with α_∞ close to 1.160. In this report, the fitted line has a steeper slope, with higher (α_∞ , around 1.173.
- Figure 3.6 in the book, data points are more concentrated, especially at smaller 1/N values (larger N), with relatively small error bars. In this report, Data points show significant variation at larger 1/N values (smaller N), especially has larger error bar at N = 32.

Possible Causes: The number of fire ignitions and the number of simulation repetitions can affect the randomness of the simulation data. Random fluctuations in the data can lead to variations in the calculated cCDF values.

Increasing the number of fire ignitions or repetitions may could make the linear extrapolation of alpha more closely match what is shown in Figure 3.6 of the textbook.

3.4 Q2

The $\alpha_\infty = 1.1731$ in this report has little differ from estimate $\alpha = 1.15$ given in Ref. [15].

Possible Causes:

- Insufficient Number of Simulations: For larger values of N, the number of simulations may not be sufficient, leading to instability in the estimated value of N
- Effect of Errors: Large standard deviations could contribute to insufficient fitting precision.

Despite some errors, the result in this report aligns with the value in the literature, validating the model's accuracy.

4 Exercise 4 Game of Life

4.1 Notes of programming Setting

- Size of area N * N = 200 * 200 , time step = 0.05, total simulation time T = 300
- Simulation repeats 5 times.

4.2 Task1 P1

Plot A(t) for the different runs

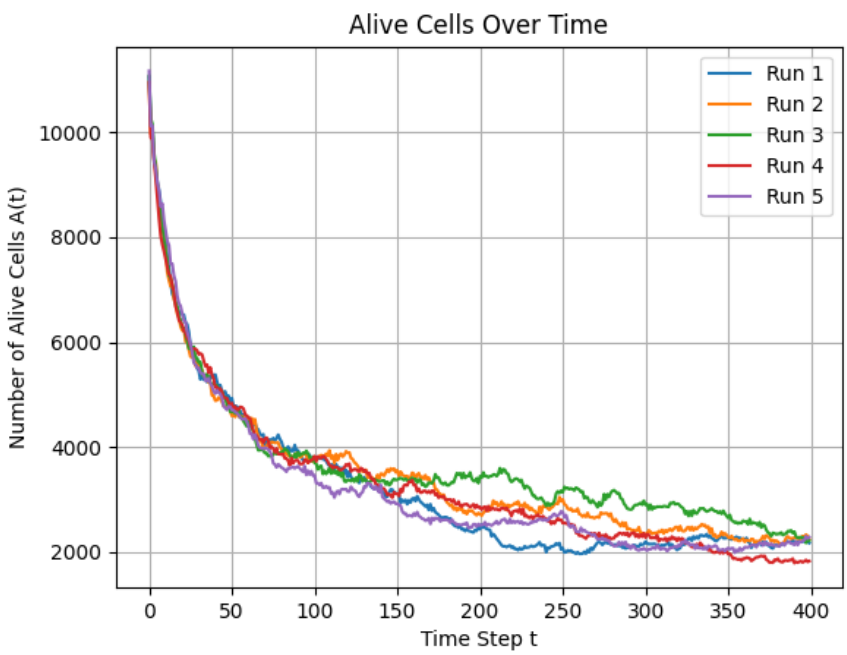


Figure 6: Number of alive cells over time

4.3 Task1 Q1

The average density of alive cell per unit area : 0.062476666666666666 (estimate for 5 runs and only calculate the density after steady state iteration¿=200).

4.4 Task1 Q2

Read from Figure above, after approximately **200 iterations** a steady state with average density is reached, from initial random configuration.

4.5 Task2 P2

plot changed cells C(t) over time for the different runs as Figure below. **Note that after 200 time steps, the the initial transient has passed and the configuration is settle down around its average density.**

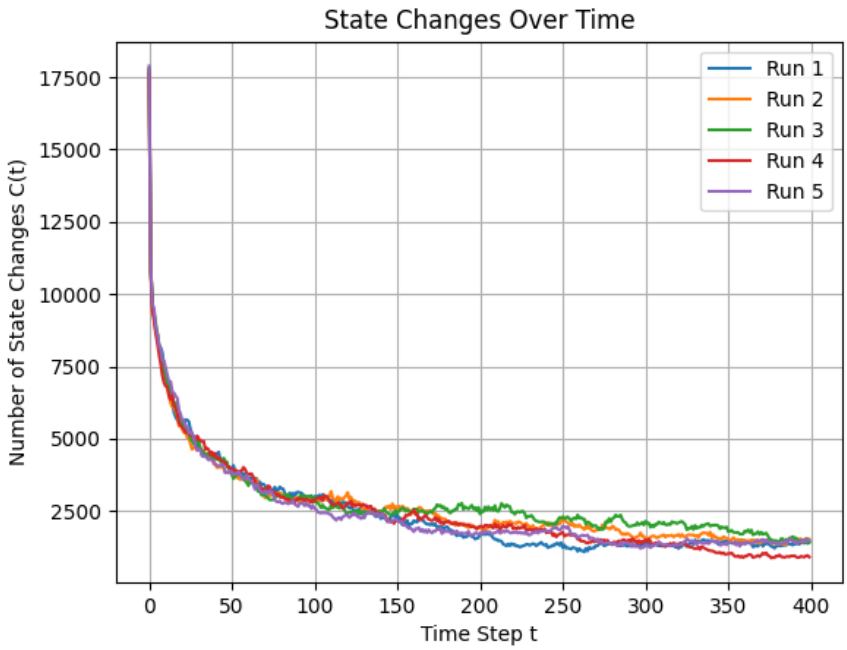


Figure 7: Number of state-changed cells over time

ising_model_final

November 11, 2024

[2]:
```
import random
import time
from tkinter import *
import numpy as np
import matplotlib.pyplot as plt


def neighboring_spins(i_list, j_list, sl):
    """
    Function returning the position of the neighbouring spins of a list of
    spins identified by their positions in the spin lattice.

    Parameters
    ==========
    i_list : Spin position first indices.
    j_list : Spin position second indices.
    sl : Spin lattice.
    """

    Ni, Nj = sl.shape  # Shape of the spin lattice.

    # Position neighbors right.
    i_r = i_list
    j_r = list(map(lambda x:(x + 1) % Nj, j_list))

    # Position neighbors left.
    i_l = i_list
    j_l = list(map(lambda x:(x - 1) % Nj, j_list))

    # Position neighbors up.
    i_u = list(map(lambda x:(x - 1) % Ni, i_list))
    j_u = j_list

    # Position neighbors down.
    i_d = list(map(lambda x:(x + 1) % Ni, i_list))
    j_d = j_list
```

```
    # Spin values.
    sl_u = sl[i_u, j_u]
    sl_d = sl[i_d, j_d]
    sl_l = sl[i_l, j_l]
    sl_r = sl[i_r, j_r]

    return sl_u, sl_d, sl_l, sl_r
```

[3]:
```
def energies_spins(i_list, j_list, sl, H, J):
    """
    Function returning the energies of the states for the spins in given
    positions in the spin lattice.

    Parameters
    ==========
    i_list : Spin position first indices.
    j_list : Spin position second indices.
    sl : Spin lattice.
    """

    sl_u, sl_d, sl_l, sl_r = neighboring_spins(i_list, j_list, sl)

    sl_s = sl_u + sl_d + sl_l + sl_r

    E_u = - H - J * sl_s
    E_d =   H + J * sl_s

    return E_u, E_d
```

[4]:
```
def probabilities_spins(i_list, j_list, sl, H, J, T):
    """
    Function returning the energies of the states for the spins in given
    positions in the spin lattice.

    Parameters
    ==========
    i_list : Spin position first indices.
    j_list : Spin position second indices.
    sl : Spin lattice.
    """

    E_u, E_d = energies_spins(i_list, j_list, sl, H, J)

    Ei = np.array([E_u, E_d])

    Z = np.sum(np.exp(- Ei / T), axis=0)  # Partition function.
    pi = 1 / np.array([Z, Z]) * np.exp(- Ei / T)  # Probability.
```

```
    return pi, Z
```

0.1 Task 1

[13]:
```
N = 200  # Size of the splin lattice.
H_values = [-5, -2, -1, -0.5, -0.2, -0.1, 0, 0.1, 0.2, 0.5, 1, 2, 5]
J = 1  # Spin-spin coupling.
T = 5  # Temperature. Temperatura critica ~2.269.

f = 0.05  # Number of randomly selected spins to flip-test..
total_steps = 500

magnetizations = []

for H in H_values:
    sl = 2 * np.random.randint(2, size=(N, N)) - 1 # Initialize N*N self-spin␣
 ↪lattice ( +1 or -1)
    Nspins = np.size(sl) # Total number of spins in the spin lattice.
    Ni, Nj = sl.shape
    S = int(np.ceil(Nspins * f))  # Number of randomly selected spins.

    step = 0
    magnetization_list = []
    running = True  # Flag to control the loop.

    while running and step < total_steps:
        ns = random.sample(range(Nspins), S)
        i_list = list(map(lambda x: x % Ni, ns))
        j_list = list(map(lambda x: x // Ni, ns))

        pi, Z = probabilities_spins(i_list, j_list, sl, H, J, T)

        rn = np.random.rand(S)

        for i in range(S):
            if rn[i] > pi[0, i]:
                sl[i_list[i], j_list[i]] = -1
            else:
                sl[i_list[i], j_list[i]] = 1

        # record magnetization
        if total_steps - 300 <= step < total_steps - 100:
            magnetization = np.sum(sl) / (N * N)
            magnetization_list.append(magnetization)

        step += 1
```

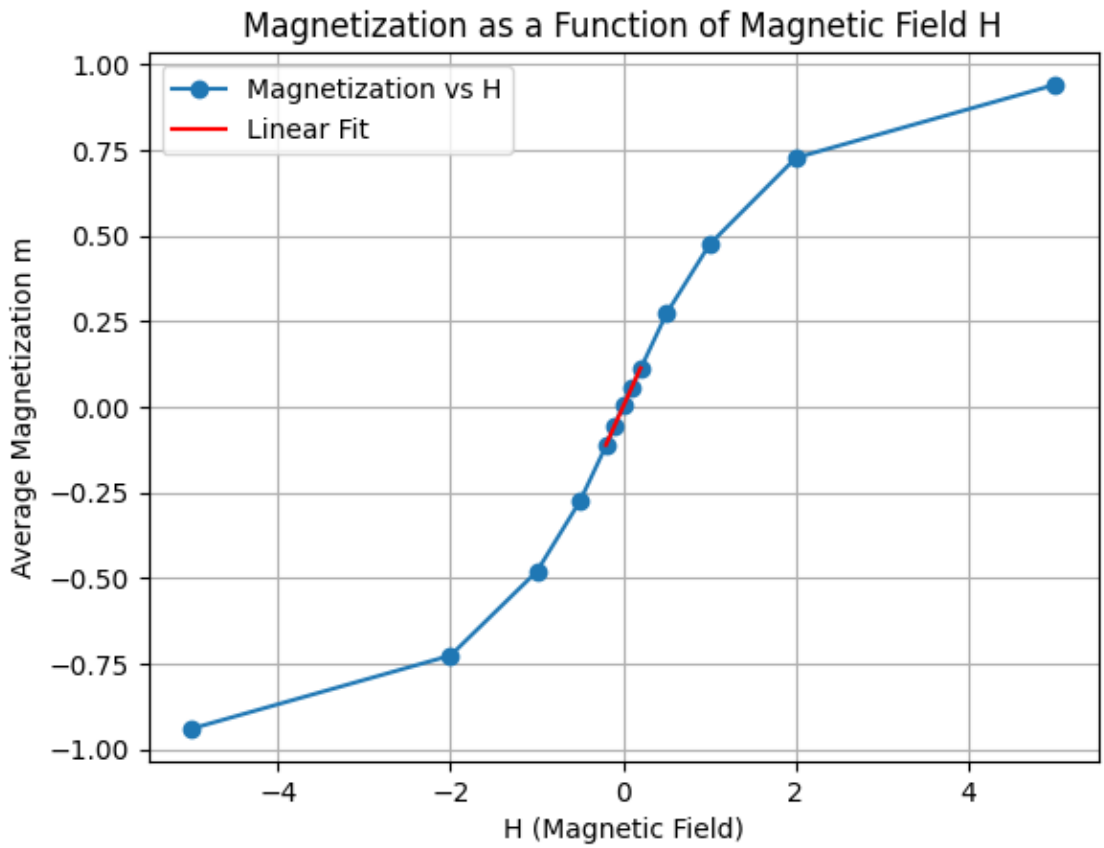
```
        if step >= total_steps:
            running = False

    print(f'H = {H}, Ising Model simulation done')

    avg_magnetization = np.mean(magnetization_list)
    magnetizations.append(avg_magnetization)
```

```
H = -5, Ising Model simulation done
H = -2, Ising Model simulation done
H = -1, Ising Model simulation done
H = -0.5, Ising Model simulation done
H = -0.2, Ising Model simulation done
H = -0.1, Ising Model simulation done
H = 0, Ising Model simulation done
H = 0.1, Ising Model simulation done
H = 0.2, Ising Model simulation done
H = 0.5, Ising Model simulation done
H = 1, Ising Model simulation done
H = 2, Ising Model simulation done
H = 5, Ising Model simulation done
```

Plot m(H) and compute linear funciton for small H values

[15]:
```
small_H_values = [-0.2, -0.1, 0, 0.1, 0.2]
small_magnetizations = [magnetizations[H_values.index(h)] for h in␣
 ↪small_H_values]
fit_params = np.polyfit(small_H_values, small_magnetizations, 1)  # linear fit
 = fit_params[0]
print(f'Calculated magnetic susceptibility  : {}')

fit_function = np.poly1d(fit_params)
fit_H_values = np.linspace(min(small_H_values), max(small_H_values), 100)
fit_magnetizations = fit_function(fit_H_values)

# Plot m(H)
plt.figure()
plt.plot(H_values, magnetizations, 'o-', label='Magnetization vs H')
plt.plot(fit_H_values, fit_magnetizations, 'r-', label='Linear Fit')
plt.xlabel('H (Magnetic Field)')
plt.ylabel('Average Magnetization m')
plt.title('Magnetization as a Function of Magnetic Field H')
plt.legend()
plt.grid(True)
plt.savefig('Magnetization_as_Function_of_Magnetic_Field_H.png', format='png',␣
 ↪dpi=300)
plt.show()
```

```
Calculated magnetic susceptibility  : 0.56006475
```



0.2 Task 2

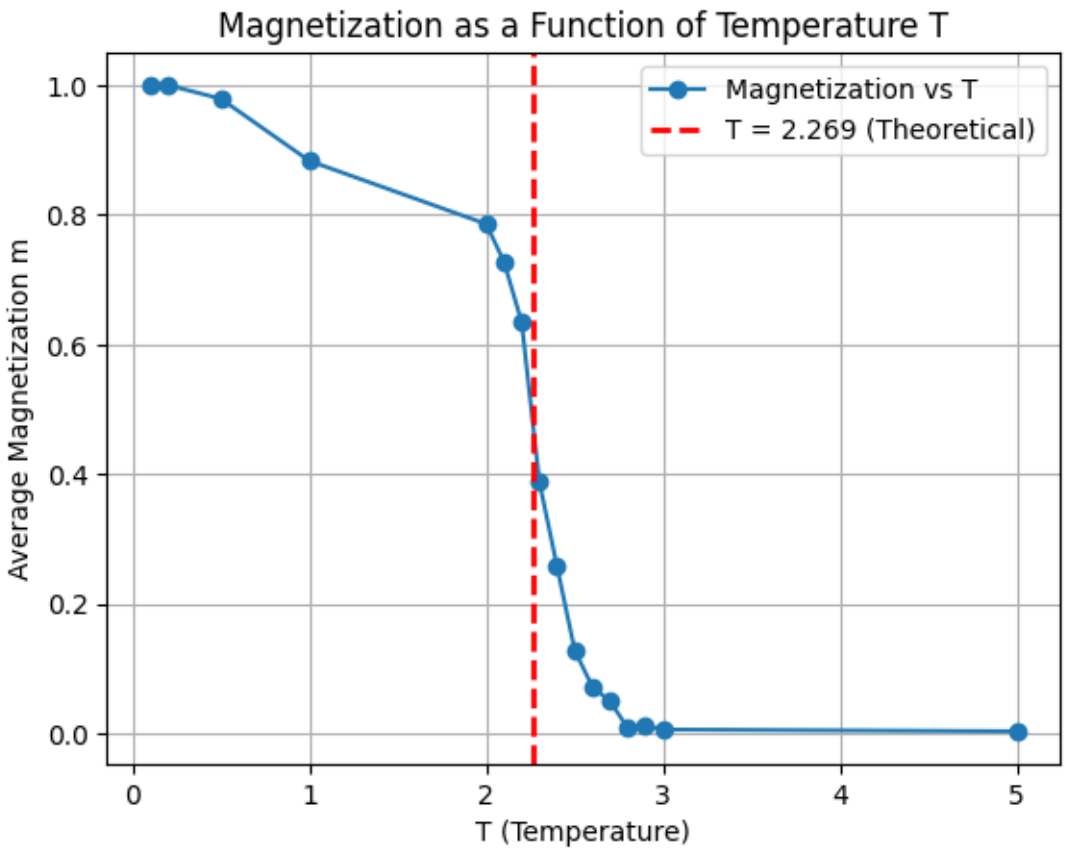
[17]:
```
N = 200  # Size of the splin lattice.

J = 1  # Spin-spin coupling.
T_values = [0.1, 0.2, 0.5, 1, 2, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9,␣
 ↪3, 5] # Temperature. Temperatura critica ~2.269.

f = 0.05  # Number of randomly selected spins to flip-test.
total_steps = 5000

magnetizations = []

for T in T_values:
    sl = 2 * np.random.randint(2, size=(N, N)) - 1 # Initialize N*N self-spin␣
 ↪lattice ( +1 or -1)
    Nspins = np.size(sl) # Total number of spins in the spin lattice.
    Ni, Nj = sl.shape
    S = int(np.ceil(Nspins * f))  # Number of randomly selected spins.
```

```
    step = 0
    magnetization_list = []
    running = True  # Flag to control the loop.

    while running and step < total_steps:

        if step <= 300:
            H = 0.1
        else:
            H = 0

        ns = random.sample(range(Nspins), S)
        i_list = list(map(lambda x: x % Ni, ns))
        j_list = list(map(lambda x: x // Ni, ns))

        pi, Z = probabilities_spins(i_list, j_list, sl, H, J, T)
        rn = np.random.rand(S)

        for i in range(S):
            if rn[i] > pi[0, i]:
                sl[i_list[i], j_list[i]] = -1
            else:
                sl[i_list[i], j_list[i]] = 1

        # record magnetization

        if (total_steps - 300) <= step < (total_steps - 100):
            magnetization = np.sum(sl) / (N * N)

            if H == 0:
                magnetization = np.abs(magnetization)

            magnetization_list.append(magnetization)


        step += 1
        if step >= total_steps:
            running = False

    print(f'T = {T}, Ising Model simulation done')
    avg_magnetization = np.mean(magnetization_list)
    magnetizations.append(avg_magnetization)
```

```
T = 0.1, Ising Model simulation done
T = 0.2, Ising Model simulation done
T = 0.5, Ising Model simulation done
T = 1, Ising Model simulation done
T = 2, Ising Model simulation done
```

```
T = 2.1, Ising Model simulation done
T = 2.2, Ising Model simulation done
T = 2.3, Ising Model simulation done
T = 2.4, Ising Model simulation done
T = 2.5, Ising Model simulation done
T = 2.6, Ising Model simulation done
T = 2.7, Ising Model simulation done
T = 2.8, Ising Model simulation done
T = 2.9, Ising Model simulation done
T = 3, Ising Model simulation done
T = 5, Ising Model simulation done
```

[18]:
```
# Plot m(T)
plt.figure()
plt.plot(T_values, magnetizations, 'o-', label='Magnetization vs T')
plt.axvline(x=2.269, color='r', linestyle='--', linewidth=2, label='T = 2.269␣
↪(Theoretical)')
plt.xlabel('T (Temperature)')
plt.ylabel('Average Magnetization m')
plt.title('Magnetization as a Function of Temperature T')
plt.legend()
plt.grid(True)
plt.savefig('Magnetization_as_Function_of_Temperature_T.png', format='png',␣
↪dpi=300)
plt.show()
```


Magnetization as a Function of Temperature T
Magnetization vs T
T = 2.269 (Theoretical)
Average Magnetization m
T (Temperature)
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5

Game_of_life

November 11, 2024

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import time

def neighbors_Moore(status):
    """
    Function to return the number of neighbors for each cell in status.

    Parameters
    ==========
    status : Current status.
    """

    # Initialize the neighbor count array
    n_nn = (
        np.roll(status, 1, axis=0) +  # Up.
        np.roll(status, -1, axis=0) +  # Down.
        np.roll(status, 1, axis=1) +  # Left.
        np.roll(status, -1, axis=1) +  # Right.
        np.roll(np.roll(status, 1, axis=0), 1, axis=1) +  # Up-Left.
        np.roll(np.roll(status, 1, axis=0), -1, axis=1) +  # Up-Right
        np.roll(np.roll(status, -1, axis=0), 1, axis=1) +  # Down-Left
        np.roll(np.roll(status, -1, axis=0), -1, axis=1)  # Down-Right
    )

    return n_nn


def apply_rule_2d(rule_2d, status):
    """
    Function to apply a 2-d rule on a status. Return the next status.

    Parameters
    ==========
    rule_2d : Array with size [2, 9]. Describe the CA rule.
    status : Current status.
    """
```

```
    Ni, Nj = status.shape  # Dimensions of 2-D lattice of the CA.
    next_status = np.zeros([Ni, Nj])

    # Find the number of neighbors.
    n_nn = neighbors_Moore(status)
    for i in range(Ni):
        for j in range(Nj):
            next_status[i, j] = rule_2d[int(status[i, j]), int(n_nn[i, j])]

    return next_status


# Apply Game of life
N = 200
repeat = 5

rule_2d = np.zeros([2, 9])

# Game of Life's rules.
rule_2d[0, :] = [0, 0, 0, 1, 0, 0, 0, 0, 0]  # New born from empty cell.
rule_2d[1, :] = [0, 0, 1, 1, 0, 0, 0, 0, 0]  # Survival from living cell.

T  = 400
alive_counts = []  # record A(t)
changed_counts = []  # record C(t)

all_alive_counts = []
all_changed_counts = []
average_density = []

average_density_all_runs = []

for run in range(repeat):

    alive_counts = []  # record A(t)
    average_density_per_run = []  # record density
    changed_counts = []  # record C(t)
    gol = np.random.randint(2, size=[N, N]) # Random initial state.

    for step in range(T):
        prev_status = gol.copy()
        gol = apply_rule_2d(rule_2d, gol)   # update cells

        # num of survive cells
        alive_counts.append(np.sum(gol))
```

```
        # density of every time step
        average_density_per_run.append(np.sum(gol)/ (N**2))

        # num of changed cells
        changed_counts.append(np.sum(gol != prev_status))


        time.sleep(0.05)  # timestep


    all_alive_counts.append(alive_counts)
    all_changed_counts.append(changed_counts)


    average_density_all_runs.append(average_density_per_run)
    print(f'Run = {run + 1}, A(t) C(t) have saved.')

# Task1 Q2

steady_state_densities = []

for run in average_density_all_runs:
    steady_state_density = np.mean(run[250:]) # steady state iteration >= 250
    steady_state_densities.append(steady_state_density)

mean_density = np.mean(steady_state_densities)
print(f'Q1: the average density of alive cell per unit area is {mean_density}')

# Task1 P1
plt.figure(1)
for run in range(repeat):
    plt.plot(range(T), all_alive_counts[run], label=f'Run {run + 1}',␣
 ↪color=f'C{run}')
plt.xlabel('Time Step t')
plt.ylabel('Number of Alive Cells A(t)')
plt.title('Alive Cells Over Time')
plt.legend()
plt.grid(True)
plt.savefig('alive_cells_over_time.png')
plt.pause(2)
plt.close()


# Task2 P2
plt.figure(2)
for run in range(repeat):
```

```
    plt.plot(range(T), all_changed_counts[run], label=f'Run {run + 1}',␣
 ↪color=f'C{run}')
plt.xlabel('Time Step t')
plt.ylabel('Number of State Changes C(t)')
plt.title('State Changes Over Time')
plt.legend()
plt.grid(True)
plt.savefig('state_changes_over_time.png')
plt.pause(2)
plt.close()
```

Forest_fire

November 11, 2024

[ ]:
```
import numpy as np

def complementary_CDF(f, f_max):
    """
    Function to return the complementary cumulative distribution function.

    Parameters
    ==========
    f : Sequence of values (as they occur, non necessarily sorted).
    f_max : Integer. Maximum possible value for the values in f.
    """

    num_events = len(f)
    s = np.sort(np.array(f)) / f_max  # Sort f in ascending order.
    c = np.array(np.arange(num_events, 0, -1)) / (num_events)  # Descending.

    c_CDF = c
    s_rel = s

    return c_CDF, s_rel
```

[ ]:
```
def grow_trees(forest, p):
    """
    Function to pgrow new trees in the forest.

    Parameters
    ==========
    forest : 2-dimensional array.
    p : Probability for a tree to be generated in an empty cell.
    """

    Ni, Nj = forest.shape  # Dimensions of the forest.

    new_trees = np.random.rand(Ni, Nj)

    new_trees_indices = np.where(new_trees <= p)
    forest[new_trees_indices] = 1
```

```
    return forest
```

[ ]:
```
def propagate_fire(forest, i0, j0):
    """
    Function to propagate the fire on a populated forest.

    Parameters
    ==========
    forest : 2-dimensional array.
    i0 : First index of the cell where the fire occurs.
    j0 : Second index of the cell where the fire occurs.
    """

    Ni, Nj = forest.shape  # Dimensions of the forest.

    fs = 0  # Initialize fire size.

    if forest[i0, j0] == 1:
        active_i = [i0]  # Initialize the list.
        active_j = [j0]  # Initialize the list.
        forest[i0, j0] = -1  # Sets the tree on fire.
        fs += 1  # Update fire size.

        while len(active_i) > 0:
            next_i = []
            next_j = []
            for n in np.arange(len(active_i)):
                # Coordinates of cell up.
                i = (active_i[n] + 1) % Ni
                j = active_j[n]
                # Check status
                if forest[i, j] == 1:
                    next_i.append(i)  # Add to list.
                    next_j.append(j)  # Add to list.
                    forest[i, j] = -1  # Sets the current tree on fire.
                    fs += 1  # Update fire size.

                # Coordinates of cell down.
                i = (active_i[n] - 1) % Ni
                j = active_j[n]
                # Check status
                if forest[i, j] == 1:
                    next_i.append(i)  # Add to list.
                    next_j.append(j)  # Add to list.
                    forest[i, j] = -1  # Sets the current tree on fire.
                    fs += 1  # Update fire size.
```

```
                # Coordinates of cell left.
                i = active_i[n]
                j = (active_j[n] - 1) % Nj
                # Check status
                if forest[i, j] == 1:
                    next_i.append(i)  # Add to list.
                    next_j.append(j)  # Add to list.
                    forest[i, j] = -1  # Sets the current tree on fire.
                    fs += 1  # Update fire size.

                # Coordinates of cell right.
                i = active_i[n]
                j = (active_j[n] + 1) % Nj
                # Check status
                if forest[i, j] == 1:
                    next_i.append(i)  # Add to list.
                    next_j.append(j)  # Add to list.
                    forest[i, j] = -1  # Sets the current tree on fire.
                    fs += 1  # Update fire size.

            active_i = next_i
            active_j = next_j

    return fs, forest
```

[ ]:
```
#main.py

import matplotlib.pyplot as plt
from grow_trees import grow_trees
from propagate_fire import propagate_fire
from complementary_CDF import complementary_CDF


N_values = [16, 32, 64, 128, 256, 512, 1024]
p = 0.01  # prob. of fire propagating
f = 0.2 # prob. of one tree fired ( lightning occurs)
repeats = 10


alpha_results = []
target_num_fires = 300
num_fires = 0

for N in N_values:

    if N == 1024:
        repeats = 2
```

```python
alpha_results_for_N = []
r = 0 # count repeat times  for debug

for _ in range(repeats):

    forest = np.zeros([N, N])  # Empty forest.
    fire_size = []  # Empty list of fire sizes.
    fire_history = []  # Empty list of fire history.

    num_fires = 0
    while num_fires < target_num_fires:

        forest = grow_trees(forest, p)  # Grow new trees.
        Ni, Nj = forest.shape
        p_lightning = np.random.rand()

        if p_lightning < f:  # Lightning occurs.
            i0 = np.random.randint(Ni)
            j0 = np.random.randint(Nj)

            fs, forest = propagate_fire(forest, i0, j0) # fs = firesize

            if fs > 0:
                fire_size.append(fs)
                num_fires += 1

            fire_history.append(fs)

        else:
            fire_history.append(0)

        forest[np.where(forest == -1)] = 0

    print(f'N = {N}', f'Target of {target_num_fires} fire events reached')

    c_CDF, s_rel = complementary_CDF(fire_size, forest.size)
    min_rel_size = 1e-3
    max_rel_size = 1e-1

    is_min = np.searchsorted(s_rel, min_rel_size)
    is_max = np.searchsorted(s_rel, max_rel_size)

    # Note!!! The linear dependence is between the logarithms
    fit_result = np.polyfit(np.log(s_rel[is_min:is_max]),
                np.log(c_CDF[is_min:is_max]), 1)
    beta = fit_result[0]
```

```
        alpha = 1 - beta
        alpha_results_for_N.append(alpha)
        r  = r + 1
        print(f'Repeat times = {r}')

    alpha_mean = np.mean(alpha_results_for_N )
    alpha_std = np.std(alpha_results_for_N )
    alpha_results.append((alpha_mean, alpha_std))
    print(f'After {r} times repeat, empirical cCDF has an exponent alpha =␣
 ↪{alpha_results[-1]}')


# # Note loglog plot!
# plt.loglog(s_rel, c_CDF, ".-", color='k', markersize=5, linewidth=0.5)
# plt.title('Empirical cCDF')
# plt.xlabel('relative size')
# plt.ylabel('c CDF')
# plt.show()

inv_N = 1 / np.array(N_values)
alpha_means = [result[0] for result in alpha_results]
alpha_errors = [result[1] for result in alpha_results]

# Q1 Extrapolate results to 1/N : 0 -->> find fit function
inv_N_fit = inv_N[:7]
alpha_means_fit = alpha_means[:7]
fit_result = np.polyfit(inv_N_fit, alpha_means_fit, 1)
a, b = fit_result
fit_line = a * inv_N + b

xticks_positions = inv_N
xticks_labels = [f'$\\frac{{1}}{{{N}}}$' for N in N_values]  #   LaTeX

plt.figure(figsize=(12, 6))
plt.plot(inv_N, fit_line, 'k--', label=f'Linear fit:   = {a:.4f} (1/N) + {b:.
 ↪4f}')  #  line of fit funcion
plt.errorbar(inv_N, alpha_means, yerr=alpha_errors, fmt='o', color='cyan',␣
 ↪ecolor='black', capsize=5, label='Alpha values with error bars')
plt.xticks(xticks_positions, xticks_labels)
plt.xticks(xticks_positions, xticks_labels, rotation=45, ha='right',␣
 ↪fontsize=10)

plt.xlabel('1/N')
plt.ylabel('Exponent  ')
plt.title('Dependence of Power-law Exponent on 1/N')
plt.grid(True)
plt.legend()
```

```
plt.tight_layout()
plt.savefig('power_law_exponent_vs_inv_N.png', format='png', dpi=300)
plt.show()
```