The final problem set involves Monte Carlo sampling of the Ising model in one and two dimensions. If you really don't want to use python and would rather write your own code from scratch, you can skip problems 1 and 2. Otherwise, problems 1 and 2 should help walk you through the programming.

1. **Monte Carlo simulation of the one-dimensional Ising model.**

   Open Ising1D.ipynb to find my code for simulating and visualizing a 1D Ising model at inverse temperature β. This code simulates $L = 40$ spins with periodic boundary conditions. Let's pass through the code block by block.

   First, you import some packages. Nothing significant to say here.

```
# Import some packages that we will use
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
plt.style.use('default')
```

   (i) Next, you define lists of 40 spins to represent the configuration.

```
# Store three different types of configurations as a demonstration

# down_Config will be a configuration with every spin equaling -1
# up_Config will be a configuration with every spin equally +1
# random_Config will be a configuration with every spin randomly set equal to
# either +1 or -1

L = 40; # Length of the 1d grid

# Make the down_Config. Note that np.zeros(L) will return a vector of all 0's
# and Python is smart enough to know that subtracting the scalar 1 from the
# vector should really subtract 1 from each element of the zero vector.
down_Config = np.zeros(L) - 1;

# Make the up_Config based on the down_Config
up_Config = np.copy(down_Config) + 2;

# Make the random_Config. I start by copying the down_Config because it is a
# quick and easy way to make the random_Config vector be the right have the
# correct size
random_Config = np.copy(down_Config);
for i in range(L):
    ### EXPLAIN THE NEXT LINE ###
    random_Config[i] = 2 * np.random.randint(2) - 1

print('Here is a configuration of all down spins:')
print(down_Config)
print('Here is a configuration of all up spins:')
print(up_Config)
print('Here is a configuration of random spins:')
print(random_Config)
```

   Explain what is happening in line 24. What is randint doing and why are we multiplying by 2 and subtracting one?

   (ii) Next we define a function that graphically plots the spins to make them easier to visualize.

```
# Write a function to plot the spin configuration
def plotConfig(config, L):
  ### EXPLAIN THE NEXT LINE ###
  plt.imshow(np.resize((config + 1)/2, (1,L)), vmin=0, vmax=1, \
             cmap=plt.cm.Greys);
  plt.axis('off');

# Use the plotting function
plotConfig(random_Config, L)
```

Explain what is happening in lines 4-5. Specifically, why do I use (random_Config + 1)/2 and not just random_Config?

(iii) To compute the energy of a spin configuration, we will need to be able to sum over the nearest neighbors. Toward this end, it would be nice to have a function that returns the location of the neighboring spins. For example, I'd like to be able to know that spin number 5 is next to spins 4 and 6. It seems this function would be trivial, it should just return $i + 1$ and $i - 1$, but the following code is a little more complicated. Here is one way.

```
# Write a function to compute the index for spin i's neighbors
def neighbors1d(i, L):
  ### EXPLAIN THE NEXT LINES ###
  if i==0:
    neighbor1 = L-1
  else:
    neighbor1 = i-1
  if i==L-1:
    neighbor2 = 0
  else:
    neighbor2 = i+1
  return [neighbor1, neighbor2]
```

Explain the purpose of the if/else statements. Why did we not just set neighbor1 to $i - 1$ and set neighbor2 to $i + 1$?

(iv) Next we define two different energy functions.

```
# Compute the contribution of spin i to the total energy
def energy1spin(i, L, h, J, config):
  energy = -h * config[i]
  for neighbor in neighbors1d(i,L):
    # Notice that we have J/2 rather than J since we will count up
    # the i-j interaction for spin i *and* for spin j.
    energy -= (J/2) * config[i]*config[neighbor]
  return energy

# Compute the total energy of a configuration
def energy(L, h, J, config):
  energy = 0
  for i in range(L):
    energy += energy1spin(i, L, h, J, config)
  return energy
```

The first energy function computes the contribution from spin i. The second function sums over all of the possible i values to give the total energy. You will see why I separately defined energy1spin later. For now, we check that the energy function is working properly.

```
# Let's check that the energy function is working correctly

```

```
LCheck = 4
config1 = [1, 1, 1, 1]
config2 = [1, -1, 1, -1]
config3 = [1, -1, -1, -1]

print('When h = 0 and J = 1...')
print('The energy of ', config1, ' is ', energy(LCheck, 0, 1, config1))
print('The energy of ', config2, ' is ', energy(LCheck, 0, 1, config2))
print('The energy of ', config3, ' is ', energy(LCheck, 0, 1, config3))

print('\nWhen h = 1 and J = 0...')
print('The energy of ', config1, ' is ', energy(LCheck, 1, 0, config1))
print('The energy of ', config2, ' is ', energy(LCheck, 1, 0, config2))
print('The energy of ', config3, ' is ', energy(LCheck, 1, 0, config3))
```

Work out the energy of the 4-spin configurations config1, config2, and config3 by hand to compare with the output of the code.

(v) Finally we run the main Monte Carlo loop.

```
# This is the main Monte Carlo loop!

# Start with a random configuration
config = np.copy(random_Config);

# Set up the Hamiltonian you want to sample
J = 1;
h = 0;
beta = 1;

# Store the initial energy
old_energy = energy(L, h, J, config);

# Configurations will be added to this empty list so you can make a movie
# of the visited configurations
stored_configs = []

# Perform 100000 Monte Carlo steps
num_mc_steps = 100000;

# Create a vector to store the values of m
mag_values = np.zeros(num_mc_steps);

# Loop over the Monte Carlo steps
for mcstep in range(num_mc_steps):

  # Pick a random spin to flip
  index_for_spin_flip = np.random.randint(L);
  # Flip that spin
  config[index_for_spin_flip] *= -1;

  # Compute the energy for the new configuration
  new_energy = energy(L, h, J, config);

  # Compute the change in energy due to the spin flip
  delta_energy = new_energy - old_energy;

  # Calculate the probability of accepting the move
  acceptance_probability = min(1, np.exp(-beta * delta_energy))
```

```
40
41    # Draw a random number between 0 and 1 to determine whether or not to
42    # accept the move
43    if np.random.random() < acceptance_probability:
44      # we already flipped the spin, so we only need to update the energy
45      old_energy = new_energy;
46    else:
47      # Since the spin flip is rejected we retain the old energy but need to
48      # flip the spin back
49      config[index_for_spin_flip] *= -1;
50
51    # compute m for the configuration
52    mag_values[mcstep] = np.sum(config) / len(config);
53
54    # Store every 1000 configurations. You could adjust this if you feel like you
55    # have too many or too few frames
56    if (mcstep % 1000 == 0):
57      stored_configs.append(np.copy(config))
```

Discuss what line 43 is doing.

(vi) The remaining blocks of code plot the magnetization and generate a movie of configurations that are visited during the Monte Carlo sampling. My default parameters used $\beta = 1, J = 1, h = 0$. **Based on last week's problem set (with the transfer matrices), do you expect the typical configurations to be ordered** ($m = \pm 1$) **or disordered** ($m = 0$)**?** Run the code to check.

2. **Monte Carlo simulation of the two-dimensional Ising model.** Open Ising2D.ipynb to find the basic framework for code that will simulate and visualize a 2D Ising model at inverse temperature β. This code simulates $L^2 = 20 \times 20$ spins with periodic boundary conditions. Let's pass through the code block by block. This time there will be some parts that you need to write to make it work!

   We start with a very similar structure but now the list of spins must have L^2 elements. I store the spins in a single 1D list, which will require us to think a little bit when figuring out which spins are neighbors on the 2d grid. The following block of code would print out some of these lists for the configurations with 20×20 spins.

```
1  # Import some packages that we will use
2  import matplotlib as mpl
3  import matplotlib.pyplot as plt
4  import numpy as np
5  plt.style.use('default')
6
7  # Store three different types of configurations as a demonstration
8
9  # down_Config will be a configuration with every spin equalling -1
10 # up_Config will be a configuration with every spin equally +1
11 # random_Config will be a configuration with every spin randomly set equal to
12 # either +1 or -1
13
14 L = 20; # Length of the 2d grid
15
16 # Make the down_Config. Note that np.zeros(L*L) will return a vector of all 0's
17 # and Python is smart enough to know that subtracting the scalar 1 from the
18 # vector should really subtract 1 from each element of the zero vector.
19 down_Config = np.zeros(L*L) - 1;
20
21 # Make the up_Config based on the down_Config
22 up_Config = np.copy(down_Config) + 2;
```

```

# Make the random_Config. I start by copying the down_Config because it is a
# quick and easy way to make the random_Config vector be the right have the
# correct size
random_Config = np.copy(down_Config);
for i in range(L*L):
    random_Config[i] = 2 * np.random.randint(2) - 1

print('Here is a configuration of all down spins:')
print(down_Config)
print('Here is a configuration of all up spins:')
print(up_Config)
print('Here is a configuration of random spins:')
print(random_Config)
```

I also updated the plotting function that will let you look at a 2D grid of spins.

```
# Write a function to plot the spin configuration
def plotConfig(config, L):
  # The only thing that has changed from the 1D code is that (1, L)
  # became (L, L)
  plt.imshow(np.resize((config + 1)/2, (L,L)), vmin=0, vmax=1, \
             cmap=plt.cm.Greys);
  plt.axis('off');

# Use the plotting function
plotConfig(random_Config, L)
```

(i) As I indicated, it will be a little harder to compute the neighbors in 2D. We can leverage the 1D function that we already wrote in the following manner.

```
# Write a function to compute the index for spin i's neighbors
def neighbors1d(i, L):
  if i==0:
    neighbor1 = L-1
  else:
    neighbor1 = i-1
  if i==L-1:
    neighbor2 = 0
  else:
    neighbor2 = i+1
  return [neighbor1, neighbor2]

# Write a function to compute the index for spin i's neighbors
def neighbors2d(i, L):
  row = int(np.floor(i / L))
  column = i - (row * L)

  [leftcol, rightcol] = neighbors1d(column, L)
  [downrow, uprow] = neighbors1d(row, L)

  upneighbor = (uprow * L) + column;
  downneighbor = # FILL THIS IN;
  leftneighbor = # FILL THIS IN;
  rightneighbor = (row * L) + rightcol;
  return [leftneighbor, rightneighbor, upneighbor, downneighbor]
```

Fill in the right hand side of lines 22 and 23 to complete the neighbors2d function. You may want to run the function with some test cases to confirm that it works properly.

(ii) The energy computation can be implemented in a manner that is *very* similar to 1D case.

```
# Compute the contribution of spin i to the total energy
def energy1spin(i, L, h, J, config):
  energy = -h * config[i]
  for neighbor in neighbors2d(i,L):
    # Notice that we have J/2 rather than J since we will count up
    # the i-j interaction for spin i *and* for spin j.
    energy -= (J/2) * config[i]*config[neighbor]
  return energy

# Compute the total energy of a configuration
def energy(L, h, J, config):
  energy = 0
  for i in range(L*L):
    energy += energy1spin(i, L, h, J, config)
  return energy
```

Identify and explain every difference between these two energy functions and the 1D versions of Problem 1(iv).

(iii) As in Problem 1(iv), we want to check that our energy is computed correctly. Since you're really using my energy function from part (ii), you can view this as a test of your nearest neighbor function from part (i).

```
# Let's check that the energy function is working correctly

config1 = np.zeros(9)+1
config2 = np.copy(config1)
for spin_index in range(9):
  config2[spin_index] = np.random.randint(2)*2 - 1

print('When h = 0 and J = 1...')
print('The energy of ', config1, ' is ', energy(3, 0, 1, config1))
plotConfig(config1, 3)
plt.show();
print('The energy of ', config2, ' is ', energy(3, 0, 1, config2))
plotConfig(config2, 3);
plt.show();

print('\nWhen h = 1 and J = 0...')
print('The energy of ', config1, ' is ', energy(3, 1, 0, config1))
plotConfig(config1, 3);
plt.show();
print('The energy of ', config2, ' is ', energy(3, 1, 0, config2))
plotConfig(config2, 3);
plt.show();
```

Work out the energy of the 9-spin configurations config1 and config2 by hand to compare with the output of the code.

(iv) The Monte Carlo steps proceed in a manner that is very similar to the 1D code, but we will want to make one critical change. It will be wasteful to compute the full energy before and after a spin flip. The reason for this is that every computation of an energy requires you to add up the contribution from spin 1, from spin 2, from spin 3, etc., but in a single move you are only changing a single spin! We should be able to use our function energy1spinflip to more rapidly determine delta_energy.

```
# This is the main Monte Carlo loop!

```

```
# Start with a random configuration
config = np.copy(random_Config);

# Set up the Hamiltonian you want to sample
J = 1;
h = 0;
beta = 1;

# Store the initial energy
old_energy = energy(L, h, J, config);

# Configurations will be added to this empty list so you can make a movie
# of the visited configurations
stored_configs = []

# Perform 100000 Monte Carlo steps
num_mc_steps = 100000;

# Create a vector to store the values of m
mag_values = np.zeros(num_mc_steps);

# Loop over the Monte Carlo steps
for mcstep in range(num_mc_steps):

  # Pick a random spin to flip
  index_for_spin_flip = ### FILL THIS IN ###

  # Compute the old contribution to the energy from spin i
  old_energy = energy1spin(index_for_spin_flip, L, h, J, config)

  # Flip that spin
  ### FILL THIS IN ###

  # Compute spin i's contribution to energy for the new configuration
  new_energy = ### FILL THIS IN ###

  # Compute the change in energy due to the spin flip
  # Notice the factor of 2, which was needed because we have to count the way
  # flipping i changes the (J/2) factor that energy1spin attributed to spin i
  # but also the (J/2) factor that energy1spin attributed to spin j
  delta_energy = 2 * (new_energy - old_energy);

  # Calculate the probability of accepting the move
  ### FILL THIS IN ###

  # Draw a random number between 0 and 1 to determine whether or not to
  # accept the move

  ### FILL THIS IN (Several lines) ###

  # compute m for the configuration
  mag_values[mcstep] = np.sum(config) / len(config);

  # Store every 1000 configurations. You could adjust this if you feel like you
  # have too many or too few frames
  if (mcstep % 1000 == 0):
    stored_configs.append(np.copy(config))
```

Fill in the missing code to perform Metropolis Monte Carlo sampling for the 2D Ising model.

To check your code, run the following block to see how the magnetization per site evolves as a function of the number of Monte Carlo steps.

```
# Plot the values of the magnetization, which have been stored throughout the
# Monte Carlo sampling in the vector mag_values
plt.plot(mag_values)
plt.xlabel(r'Monte Carlo Step', fontsize=18);
plt.ylabel(r'$m$', fontsize=18);
plt.title(r'Fluctuating values of the magnetization per site', fontsize=20);
```

Turn in your plot but not your code.

(v) In analogy with Problem 1(vi), you can run the final blocks of code in Ising2DFramework.ipynb to see the probability distribution for the magnetization per site and to generate a movie of typical configurations. If you look carefully, you will see that I only collect the samples after 40,000 Monte Carlo steps to make sure that the initial configuration does not strongly bias the sampling. **Discuss the major difference between the behavior of the 2D Ising model and the 1D behavior you observed (at the same temperature) in Problem 1(vi).**

3. **Using your Ising simulation.**

(i) In the absence of an external magnetic field (i.e., $h = 0$), the transition to a ferromagnetic state upon decreasing temperature is second order. In other words, the average magnetization per spin $\langle m \rangle$ changes continuously but not smoothly at $T = T_{\mathrm{c}}$. Try to confirm this fact by computing $\langle m \rangle$ as a function of T from several Monte Carlo trajectories above and below T_{c}. Estimate the sampling error associated with each computed value of $\langle m \rangle$, and describe the procedure you use to do so. **Provide a plot of m as a function of T (showing error bars), along with snapshots of the system at a temperature below T_{c}, a temperature very close to T_{c}, and a temperature above T_{c}.**

(ii) Your results will differ from the exact behavior of this system (determined mathematically by Lars Onsager) in several respects. For example, the apparent transition in your simulated system should not occur precisely at $k_{\mathrm{B}}T_{\mathrm{c}} = 2.269J$. (Note that, in the code, the unit of temperature is J/k_{B}.) The transition should also appear smooth (rather than exhibiting a sharp kink). **Explain both of these deviations.**

(iii) [Optional] Singularities at the critical point arise from correlated fluctuations of spins separated by macroscopically large distances. Try to confirm the existence of long-ranged correlations at the critical point by computing $c(r_{ij}) = \langle s_i s_j \rangle - \langle s_i \rangle \langle s_j \rangle$ as a function of $r_{ij} = |\mathbf{r}_i - \mathbf{r_j}|$ (the distance between spins i and j) at several temperatures above and below the critical point. **Plot your results.** [Note: To accurately sample with $T \approx T_{\mathrm{c}}$ you have to generate quite a few Monte Carlo samples, so it's a little harder to carry out this problem. I've made it optional, but I encourage those who expect to use computers in their research to embrace the challenge.]