Lab 8

Apr 7, 2017

Entropy
(Based on chapter 7 of "Computational Physics" by Giordano and Nakanishi)

Here, we will consider the free diffusion of a particle in 2D and calculate its entropy as a function of time.

We have modeled the free motion of a particle in 2D as a random walk, and have simulate the diffusion of a drop of cream in the coffee by simulating the trajectory of a large number of random walkers in 2D.

The second law of thermodynamics states that the entropy of a closed system either increases or stays the same over time. In this lab, we will calculate the entropy of the system numerically.

The entropy of the system can be written as:

$$S = -\sum_{i} P_i ln P_i$$

where P_i is the probability of finding the system in a state i, and the sum is over all possible states. In this example, where we have a 2D diffusion, we can define each state to correspond to a position in the x-y plane. P_i at each time n, is the probability of finding a random walker (with n steps) at this position. To calculate this probability, we need to repeat the random walk a large number of times, and calculate the number of times that the walker is at this position after n steps.

We will use the python script for the 2D random walker (random_walk_2d_histogram_template.py) and modify it to calculate the entropy. You can download the script along with these instructions from the blackboard. A brief summary of the script and necessary modifications are provided below:

- Define the following constants:
```
num_steps = 500
num_walkers = 1000
```

- Define the following two arrays to store the coordinates of each walker over time:
```
x_coor = np.zeros([num_walkers, num_steps])
y_coor = np.zeros([num_walkers, num_steps])
```

- To create a random walk for each walker, we will define a loop and execute the following steps for each iteration:

  - Create two arrays called x_step and y_step to store the random steps taken by each walker in each direction. We will do this in two steps.

    ```
    x_step = rng(num_steps)
    ```

    will define an array that has 500 (num_steps) elements and each element is chosen randomly between 0 and 1.

    Add a similar command to define an array called y_steps for the steps taken in the y-direction.

    The rng function will generate random numbers that are uniformly distributed between 0 and 1. To create a binary distribution of random steps with equal length, for each element that is > 0.5, we will assume a move in the positive direction and for every element that is < 0.5, we will assume a move in the negative direction. This is achieved by the following commands:

    ```
    x_step = 2 * (x_step > 0.5) -1
    ```

    where the value of the argument in parentheses is 1 if True and is -1 if False. Define a similar command for y_step.

    The two arrays, x_step and y_step represent the steps taken by the random walker. To find the position of the random walker at each time step, for example at time n, we need to add the steps taken prior to that. The following command will create an array with the cumulated sum of all the elements in x_step:

    ```
    x = np.cumsum(x_step)
    ```

    define a similar command to define y.

    Now, (x,y) will represent the coordinates of the random walker at each time. We will store these coordinates into the corresponding arrays x_coor and y_coor for walker j:

    ```
    x_coor[j,:] = x
    y_coor[j,:] = y
    ```

  - You can plot each random walk by running the following command:

    ```
    plt.plot(x,y)
    ```

inside the loop. However, having too many random walkers may result in a plot that is not very clear to understand.

Here we will plot the position of the random walkers at the last step. You can plot this after all the coordinates have been generated in the loop:

```
plt.plot(x_coor[:,num_steps-1],y_coor[:,num_steps-1],'o')
```

to plot the distribution of the walkers at any other step, for example after num_steps/2 steps, you can create a new plot and run:

```
plt.figure()
plt.plot(x_coor[:,int(num_steps/2)],y_coor[:,int(num_steps/2)],'o')
```

- To calculate the entropy at each time step, we need to calculate the distribution of the walkers on the surface.

  We will use the built-in python function np.histogram2d for this purpose. The histogram function in 2d (there is also a np.histogram function in 1d) acts on an array (2d or 1d) and calculates its probability distribution over a certain range. For example, if you have a 1d list of numbers such as:

  mylist = [1,5,2,0,8,2,3,9,10,2,6,9]

  np.hist(mylist) will create two arrays for you. It will determine the range of numbers in mylist, in this case going from 0 to 9, and create an array that goes from 0 to 9 and has 10 elements. This list will be considered your grid points (edges of the bins).

  It then calculates the number of times each value appear in my list and stores them in the first list, e.g., in this case the grid points are:

  [0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.]

  and the histogram is:

  [1, 1, 3, 1, 0, 1, 1, 0, 1, 3]

  meaning that there is only one value in mylist that appears in the first bin (first bin is the interval from 0 to 1, [0,1)). There is one value within [1,2) (second bin) and 3 values that falls in the third bin [2,3), and so on.

  A similar function np.histogram2d acts on 2 dimensional matrices and returns the probability of finding a number on a certain grid point. By default, the histogram divides the range of numbers into 10 bins, and calculates the probability of occurrence at each bin. However, in many cases it is more desirable to have more than 10 bins. The number of bins or the bin edges can also be specified in the histogram function if needed.

In this case, we would like to calculate the distribution of the walkers at each time step. For this purpose, we will divide our 2dimensional surface into a 64 x 64 grid, and use the np.histogram2d function to calculate a histogram of the walkers position at each time step.

We will define the following variables:

```
    nbins = 64      #number of bins in each dimension
    xedges = np.linspace(-200,200,nbins+1)
    yedges = np.linspace(-200,200,nbins+1)
```

The two variables xedeges and yedges store the coordinates of the grid points in x and y dimensions. You can also define xedegs and yedges in terms of num_steps:

```
xedges = np.linspace(-1*num_steps,num_steps,nbins+1)
yedges = np.linspace(-1*num_steps,num_steps,nbins+1)
```

We will define a variable called entropy that stores the entropy at each time step and initialize it to be zero:

```
Entropy = np.zeros(num_steps)
```

We will then run a loop over time steps (num_steps). The coordinates of the walkers at each time step are read and are used to calculate their histograms. You can define the loop as follows:

```
    for s in range(num_steps):
        --- calculate the histohrams
```

The coordinates of the walkers are accessible through the following elements: x_coor[:,s] and y_coor[:,s] where s is the timestep.

To calculate the histograms, run the following commands <u>inside the loop</u>:

```
h=np.histogram2d(x_coor[:,s],y_coor[:,s],bins=(xedges,yedges))
```

This command calculates the histogram and writes it into a variable called h. Now, if you inspect h, you can see that it consists of three arrays, h[0], h[1], h[2].

h[1] and h[2] contain the coordinates of the bins in x and y directions, respectively. In this case, they each have 64 elements.

h[0] is a 64x64 matrix. Each element of h[0], e.g., element [i,j] will represent the number of times that the random walkers have been in position i,j.

The sum of all the elements of h[0] should be equal to the number of walkers, and thus, we can define the probability of finding the walkers at position [i,j] as follows:

```
p = h[0]/sum(sum(h[0]))
```

Now that p is defined at each timestep, we can use it to calculate the entropy at each time step. Again, the following commands all have to be executed inside the same loop (for timestep).

To calculate the sum:

$$S = -\sum_{i,j} P_{i,j} ln P_{i,j}$$

run the following commands inside:

```
v = 0
for i in range(nbins):
    for j in range(nbins):
        if (p[i,j]>0):
        v += -p[i,j]*np.log(p[i,j])
```

v is the entropy of the system at timestep s, and we can store it in entropy[s]:

```
entropy[s]=v
```

Once this loop runs over all timesteps, we have calculated the entropy for all time steps, and it can be plotted with:

```
plt.figure()
plt.plot(entropy)
```

Plot the entropy as a function of timestep. You should see that entropy fluctuates but on average it will increase by time.

Can you explain whether or not these fluctuations are consistent with the second law of thermodynamics and why?

- Now, change this code such that it represents the free diffusion of particles in a confined space. An example would be diffusion of a drop of cream in a cup of coffee. The cream particles diffuse freely until they reach the edges of the cup.

  Modify your code, such that the random walk steps are accepted only if the coordinates of the walker fall within ($-100 < x < 100$) and ($-100 < y < 100$).

  Plot the distribution of the particles at a few time steps and show that the initial drop in fact diffuses in the cup. Increase the timesteps, to show that in fact the particles do not cross the borders that you have defined. Compare the same snapshot with a snapshot of the free diffusion (first example) at the same timestep, showing that while in the firs example particles diffuse out of the cup, your new code does not allow the particles to diffuse past the walls of the coffee cup.

- Plot the entropy of this system over time and compare it with the first example.