Lab 11
PHYS 1200 - Spring 2025

In this lab, you'll use Python to generate two-dimensional random walks, plot their trajectories, and look at the distribution of end points for a large number of random walkers. You will also use numerical experiments to study the statistics of rare events. Our goals are to:

- Generate random walk trajectories that begin at the origin and take random diagonal steps:

$$x_{n+1} = x_n \pm 1 \qquad \text{and} \qquad y_{n+1} = y_n \pm 1 \qquad (11.1)$$

- Plot individual trajectories side by side and compute statistics for many trajectories.

- Generate a Poisson distribution and analyze a Poisson process.

1- Brownian motion

1-1 Generating and plotting trajectories

First, review Lab 8 Example 4.

Our first task is to create a random walk of 1000 steps, each given by Equation 11.1 . Each trajectory will be a list of 1000 x values and 1000 y values. It's good programming practice to define the size of the simulation at the beginning of a script: **num_steps = 1000**

Now you can set the size of all arrays by using **num_steps**.

Assignment:

A. Make a random walk trajectory, and then plot it. To remove any distortion, use the command **plt. axis ('square')** or **plt.axis ('equal')** after making the plot.

B. Now make four such trajectories, and look at all four side by side. Use **plt.figure()** to create a new figure window. You can access the individual subplots by using commands like **plt.subplot (2, 2, 1)** before the first **plt.plot** command, **plt.subplot (2, 2, 2)** before the second **plt.plot** command, and so on. Python may give each plot a different magnification. Consult **help(plt.xlim)** and **help(plt.axis)** to find out how to give each of your plots the same x and y limits?

1-2 Plotting the displacement distribution

Run your script several times, and compare the resulting trajectories. Your plots always look different. Sometimes the walker wanders off the screen; sometimes it remains near the origin. And yet, there is some family resemblance among them. Let us begin to understand in what sense they are similar.
How far does the random walker get after 1000 steps? More precisely, what is the distance from the starting point (0, 0) to the ending point (x_{1000}, y_{1000}) for each of many random walks? We can use Python to give a statistical answer to such questions. Instead of four walks, we now need many, say, 100. You could manually examine all 100 plots, but it would be hard to see the common features. Instead, we'll ask Python to generate all of these random walks, but show us only a summary.
One straightforward way to make 100 walks is to take the code you already wrote and embed it inside of a for loop. You could create three arrays, **x_final**, **y_final**, and **displacemen**t, to store the ending x and y positions and distance to the origin. Then, just before the end of the loop, you could add something like:

```
x_final[i] = x[-1]
y_final[i] = y[-1]
displacement[i] = np.sgrt (x[-1]**2 + y[-1]**2)
```

(You can also solve the problem by using vectorized operations instead of a for loop. Try to figure out how. The vectorized approach is faster than a loop if the arrays are not too large-that is, if **num_walks*num_steps** is smaller than 10^7.)
We can summarize the results in at least three ways. You have a lot of end points (**x_ final, y_final** pairs), so you can make a scatter plot by using plt.plot or plt.scatter. Alternatively, you can examine the lengths of the final displacement vectors, or their squares.

Assignment:

A. Once you have a code that works, increase the number of random walks from 100 to 1000. Make a scatter plot of the end points.
B. Use **plt.hist** to make a histogram of the displacement values.
C. Make a histogram of the *quantity* **displacement**2**.
D. Your result from C may inspire a guess as to the mathematical form of the histogram. Use semilog and log-log axes to test for exponential or power law relationships.
E. Use **np.mean** to find the average value of **displacement**2** (the mean-square displacement) for a random walk of 1000 steps.

F. Find the mean-square displacement of a 4000-step walk. If you wish to carry the analysis further, see if you can determine how the mean-square displacement depends on the number of steps in a random walk.

It turns out that random walks are partially predictable after all. Out of all the randomness comes regular statistical behavior, partly visible in your answers to (B-F). Experimental data also agree with these predictions. The random walk, although stripped of much of the complexity of real Brownian motion, nevertheless captures nontrivial aspects of Nature that are not self-evident from its formulation. See if your output qualitatively resembles the experimental data shown in the figure below for the diffusion of a micrometer-size particle in water.

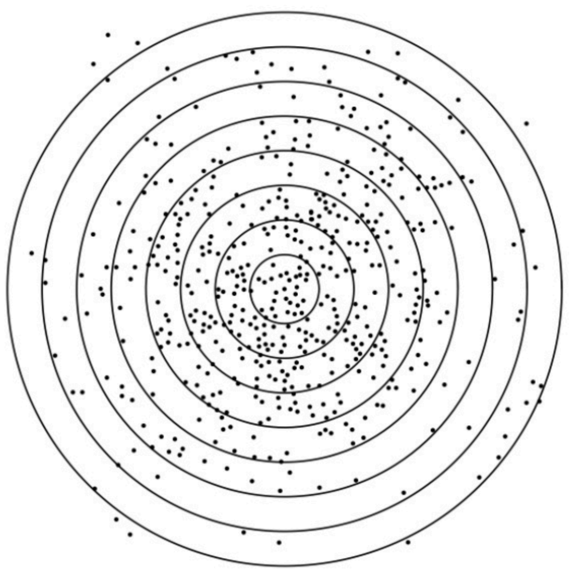

Figure 1: Experimental data for Brownian motion. Data from Perrin 1948

2 RARE EVENTS

2.1 The Poison distribution

Imagine an extremely unfair coin that lands heads with probability λ equal to 8% (not 0.5). Each trial consists of flipping the coin 100 times. You might expect that we'd then get "about 8" heads in each trial, although we could, in principle, get as few as 0, or as many as 100. The **Poison distribution** is a discrete probability distribution that applies to rare events. For our extremely unfair coin, it predicts that the probability of the coin coming up heads l times in 100 flips is:

$$\rho(l, \lambda) = \frac{e^{-\lambda}\lambda^{l}}{l!} \qquad (11.2)$$

where l is an integer greater than or equal to 0.

Assignments:

A. Before you start flipping coins, plot this function for some interesting range of l values. You may ind the following helpful:
   - The factorial function can be imported from scipy. special.
   - You need not take l all the way out to infinity. You'll see that $\rho(l)$ quickly gets negligibly small.
   - In Python, the elements of a vector are always numbered 0, 1, 2, 3, ..., and l is also an integer starting from zero, so l is a good array index.
   - The value of 8^{l} can get very large- larger than the largest integer NumPy can store. (By default NumPy uses 64-bit integers, so the largest number it can store is $2^{63} - 1$.) To avoid numerical over flow - and erroneous results - use an array of floats instead of integers. Consult **help(np.arange)**, and read about the *dtype* keyword argument.

B. Perform N coin flip trials, each consisting of 100 flips of a coin that lands heads only 8% of the time. [Good practice: Eventually you may want to take N to be a huge number. While developing your code, make it not so huge, say, N = 1000, so that your code will run fast.]

C. Get Python to count the number of heads, M, for each trial. Then, use **plt.hist** to create a histogram of the frequency of getting M heads in N trials. If you don't like what you see, consult **help(plt. hist)**. (For example, **plt.hist** may make a poor choice about how to bin the data.)

D. Make a graph of the Poisson distribution (Equation 11.2 above) multiplied by N. What's the most probable outcome? Graph this plot on the same axes as the histogram in C.

E. Repeat (B-D) for N = 1000,000, and comment. (This may take a while.)

Run the script over and over for N = 1000, and observe that the distribution is a bit different every time, and yet each plot has a general similarity to the others.

2.2 Waiting times

If we flip our imagined coin once every second, then our string of heads and tails becomes a time series called a Poisson process, or shot noise. Flipping heads is a rare event, because $\lambda = 0.08$. We expect long strings of tails, punctuated by occasional heads. This raises an interesting question: After we get a heads, how many flips go by before we get the next heads? More precisely, what's the distribution of the waiting times from one heads to the next?

Here's one way to use Python to answer that question. We can make a long list of ones and zeros, then search it for each occurrence of a 1 with NumPy's **np.nonzero** function. This function takes an array of numbers and returns an array of the indices of its nonzero elements. Consult **help(np.nonzero)** and experiment with small arrays like **np.nonzero([1, 0, 0, -1])** to understand its behavior.

Each waiting time is the length of a run of zeros, plus one. You can subtract successive entries in the array returned by **np.nonzero** to find the waiting time between successive heads, then make a graph showing the frequencies of those intervals. NumPy's **np.diff** function will take the difference of successive entries in an array. Consult **help(np.diff)** for more information about that function. Flatten the array returned by **np.diff** before plotting. See Numpy documentations for **flatten** for details on flattening arrays.

Try to guess what this distribution will look like before you compute the answer. Someone might reason as follows: "Because heads is a rare outcome, once we get a tails we're likely to get a lot of them in a row, so short strings of zeros will be less probable than medium-long strings. But eventually we're bound to get a heads, so very long strings of zeros are also less common than medium-long strings." Think about it. Is this sound reasoning? Now, compute the distribution. If your output is not what you expected, try to figure out why.

Assignment:

A. Construct a random string of 1's and 0's representing 1000 flips of the unfair coin. Then, plot the frequencies of waiting times of length 0, 1, 2, ..., as outlined above. Also make semilog and log-log plots of these frequencies. Is this distribution a familiar-looking function?

B. What is the average waiting time between heads?

C. Repeat A and B for 1000,000 flips of the coin.