Physics 2300
Spring 2018

Name _____

Lab partner _____

Project 6: Random Processes

Many physical systems, such as the molecules modeled in the previous project, are so chaotic that the motions might as well be random. But in that case, why should we even bother to calculate the actual trajectories? For many purposes we can get away with substituting fictitious "random" behavior for the actual behavior of a system. Randomness is also an intrinsic property of quantum mechanical measurements, for instance, of the time when a nucleus decays.

In this project you will write several short computer programs that use computer-generated "random" numbers to model simple physical systems. Programs that use random numbers to choose among various possible outcomes are called *Monte Carlo simulations*, after the famous European gambling resort.

In this project you will also leave GlowScript behind, and learn to work in a more traditional Python environment. There is no need for 3D graphics, or to draw the modeled systems in physical space at all. Nor will you need to animate the modeled processes, in order to watch them play out in time. Instead you will accumulate data in lists and then plot the data, all at once, using static graphs.

A traditional Python environment

The traditional way to use Python is not through a web browser but through a set of software tools that you install on your local hard drive. These tools include the Python interpreter itself, a variety of add-on packages, and software for editing, launching, and debugging your programs. The possible combinations of these tools are practically unlimited, and developers are constantly adding to these possibilities, creating new ways to adapt Python to new uses. Python's adaptability is both a blessing and a curse. On one hand, it accounts for much of Python's popularity among scientists. On the other hand, it makes it difficult to communicate with someone whose Python environment is different from your own, and it can make the initial installation and setup process rather daunting.

To mitigate these difficulties, I recommend that you take advantage of a free product called the Anaconda Python distribution, which makes many of the most widely used scientific Python tools available through a single download-install process. You may already be using a computer with the Anaconda distribution installed. To install it on your own computer, point your browser to `https://www.anaconda.com/download/` and follow the instructions. Choose the Python version 3 installer (currently version 3.6), rather than version 2.7 (which is provided only for compatibility with older code). Once the installation process is complete, test it out

by opening Anaconda Prompt (on Windows) or Terminal (on MacOS) and typing `python`. You should see a message confirming that you are using Python version 3 as configured by Anaconda. Type `exit()` to get out of the Python interpreter and back to the system-level prompt.

To create your Python programs you will also need a text editing application. A good programmer's text editor for Windows is Notepad++, which you can download from `https://notepad-plus-plus.org/`. A good programmer's text editor for MacOS is BBEdit, which you can download from `https://www.barebones.com/products/bbedit/`. Many other editing applications will work equally well, so if you already have a favorite programmer's text editor, just use that.

Next you'll want to create a folder on your hard drive to hold your Python programs. After doing that, launch your text editor and use it to create a Python program containing the single line of code `print("Hello, world!")`. Save this program in your newly created folder under the name `Hello.py`. Then go back to your command-line window (Anaconda Prompt or Terminal) and use the `cd` command ("change directory") to navigate to your folder (for example, on Windows, type something like `cd \Users\username\Desktop`). Then type `python Hello.py` to launch your program, and check that you see the printed greeting.

(There are many things that can go wrong during the process that I've just described, but it's hard to describe every possible problem—let alone their solutions—here in print. If you encounter difficulties, just ask your instructor or some other expert for help.)

Now that you're no longer working in the cloud, you'll also need a way to back up your programs. I recommend a simple USB stick, but you could also manually copy your programs to a cloud storage site, or just email them to yourself. Be sure to back up your work at the end of every coding session!

A traditional Python program

Now that you have some traditional Python infrastructure, let's talk about how your programs will differ from the ones you created in GlowScript.

A trivial difference is that you won't include the line `GlowScript 2.7 VPython` at the top of each program. Instead, just start right off with the usual comments that give the program name, your name, the date, and what the program does.

The next lines of your program will *import* whatever Python packages it needs. In a traditional Python environment you can't do much at all without packages. In this project you'll need three packages:

- `math` for common math functions like `sqrt` and `exp`;
- `random` for Python's pseudo-random number generator; and
- `matplotlib.pyplot` to produce professional-quality graphics.

(Some Python packages are more properly called *modules*, but I'll call them all packages for simplicity.)

Adding to the confusion, Python provides multiple *ways* to import a package. Here are four different ways to import some or all of the `math` package:

```
import math            # all math functions via "math." prefix
import math as m       # all math functions via "m." prefix
from math import *     # all math functions with no prefix
from math import sqrt, exp     # only certain math functions
```

In the first case you would say `math.sqrt(x)` to take the square root of `x`, while in the second case you would say `m.sqrt(x)`. In the third and fourth cases you would just say `sqrt(x)`, as in GlowScript. Although that might seem easiest, it can be dangerous to import too many functions in this way because their names can start conflicting with your own functions, and/or with each other.

Here's the package-importing code that I recommend for the programs you'll write in this project:

```
from math import sqrt, exp, factorial
from random import random
import matplotlib.pyplot as plt
```

The first line should take care of the math functions you need. I'll explain the other two when you're ready to use them, below.

Once you've imported the packages you need, much of your code will look the same as in GlowScript. But you will not be using any VPython features: no boxes, spheres, or cylinders; no vectors; and no rate functions to control the speed of animation loops. (If you ever want to use VPython features from a traditional Python environment, there is a VPython package that provides these features. It's not part of the Anaconda distribution, so you would need to install it as a separate step. Unfortunately, I've found it pretty awkward to use. Fortunately, it is not needed for this project.)

You'll also need to adapt to some restrictions that are present in true Python but not in GlowScript—due to the fact that GlowScript is actually a thin Python veneer on top of JavaScript:

- Standard Python won't automatically convert numbers to strings when you try to combine them with the + symbol, as in `print("x = " + x)`.
- Standard Python doesn't allow the `!` symbol in place of `not`, as in `running = !running`.
- Standard Python won't let you add an element to a list by simply giving it an initial value, as in `x[1] = 0` (if `x[1]` doesn't already exist).

- Standard Python won't let you use an expression as a list index if that expression might not evaluate to an integer, as in `x[i/2]`. But you can say `x[int(i/2)]`.
- Standard Python requires that a function definition appear *above* any code that calls the function.

With these constraints in mind, let's now get on with the project.

Computer-generated "random" numbers

The `random` package provides a function called simply `random()`, which returns a "random" real number between 0 and 1 each time it is called. Before using these numbers in physics simulations, it's a good idea to get some feel for what they're like.

Exercise: Write a short Python program called `RandomTest.py` that prints (to the screen) the values of 20 successive numbers returned by `random()`. Run your program from the Anaconda Prompt or Terminal window as before. Do the numbers appear random to you? How can you tell?

Question: If you run your `RandomTest` program a second time, do you get the same sequence of numbers, or a different sequence?

Often you'll want random numbers that span a range other than 0 to 1; a common need is to choose a random integer within a certain range. Although the `random` package provides a separate function for this purpose, it's often easier to just multiply the result of `random()` by n and round down using `int()`, to obtain a random integer between 0 and $n - 1$, inclusive. (The `random()` function never returns exactly 1, so when you multiply by n and round down, you'll never get n.)

Exercise: Add code to your `RandomTest` program (without deleting the code that's already there) to obtain and print the values of 50 random numbers (digits) in the range 0 to 9 inclusive. Count the number of times each digit occurs, and write the results of your counts below.

Another common need is to generate random points in some region of space, in two or more dimensions. To try this out in two dimensions, you can plot the points using the following code:

```
plt.plot(x, y, marker="o", markersize=2, color="red",
            linestyle="None")
plt.show()
```

(I'm assuming that you've already said `import matplotlib.pyplot as plt`.) Here `x` and `y` are *lists* of the points to plot, which you must build in advance; the `matplotlib` package is intended for static graphs, not for graphs that you build up, point by point, as a simulation progresses. By default it will connect the points with straight line segments, but in the code above I've overridden this behavior and instead told it to use red circles with a width of two printers' points.

Exercise: Add code to your `RandomTest` program (without deleting the code that's already there) to generate 1000 random real-number (x, y) pairs, with each random coordinate ranging from 0 to 10, and display them using the graphics code given above. Temporarily try omitting the `linestyle="None"` option, and try changing the size and color of the markers. Also try replacing the marker shape code (`o`) with some of the following: `s v ^ D x +`.

Besides the plotting options that you just explored, `matplotlib` provides a host of separate functions for adjusting the overall appearance of a graph. Here are several to try:

```
plt.axis("scaled")
plt.xlim(0,10)
plt.ylim(0,10)
plt.xticks(range(11))
plt.yticks(range(11))
plt.grid()
plt.title("1000 random points")
plt.xlabel("x")
plt.ylabel("y")
```

The first of these options forces the scales of the x and y axes to be the same. The next two lines remove the default buffer region around the edges. The rest, I hope, are reasonably self-explanatory. To see a complete list of `matplotlib` functions and options, you can look at the documentation at `https://matplotlib.org/api/pyplot_summary.html`. Just try not to be too horrified by the complexity, or by the way that whoever wrote the documentation assumes that you're already an expert. Poorly written documentation is disturbingly common for Python packages.

Exercise: Insert all of the code lines shown above, between the calls to `plt.plot` and `plt.show`. When you are happy with the appearance of the graph, use the

interactive controls to save it, then open the saved image in another application and print it. How many points, on average, do you expect to appear in each of the 100 grid squares of your graph? What are the approximate maximum and minimum numbers of points that actually appear within the squares? (Please circle the corresponding squares on your printout.)

Now that you've gotten a feel for what the `random()` function does, here's the bad news: The numbers that it returns are not actually random. In fact, each of these numbers is computed from the previous one in a completely deterministic way. (The computer's clock is used to get the sequence started, so you get a different sequence each time you run the program. If you ever want to get the same sequence every time, you can do so using the `random.seed` function.) But the function used to generate each number from the last is chaotic, so that even a tiny change in the input produces a substantial change in the output. The function also has the property that over the long term, it generates numbers that are distributed evenly throughout the interval from 0 to 1. But the function is not guaranteed to return numbers that are sufficiently random for all purposes. In particular, despite the appearance of your plot, this function is not guaranteed to generate evenly distributed points in higher-dimensional spaces. In recognition of these imperfections, computer-generated "random" numbers are often called *pseudo-random numbers*.

In this course we'll ignore this potential pitfall and assume that the numbers returned by `Math.random` are sufficiently random for our purposes. For research-quality work, however, you should always make sure your pseudo-random number generator is adequate for the task at hand. For an excellent discussion of this problem and several examples of much better generators (coded in C++), I recommend the book *Numerical Recipes* by Press, *et al.*

Expansion of a gas

Consider the situation shown in Figure 1: A box is divided into two sections of equal size, separated by a partition. On one side of the partition is a gas of n molecules; on the other side is a vacuum. We then puncture the partition and allow the molecules to pass from one side to the other. What happens to the number of molecules on each side as time passes?

To answer this question we *could* write a full-blown molecular dynamics simulation. Or we could just recognize that for our purposes the motions will be essentially random, and say that each of the n molecules has an equal probability of switching sides during any short time interval.

Exercise: Write a program called `TwoBoxes.py` to simulate the behavior of this system. Use the variables `n` for the total number of molecules and `nLeft` for the

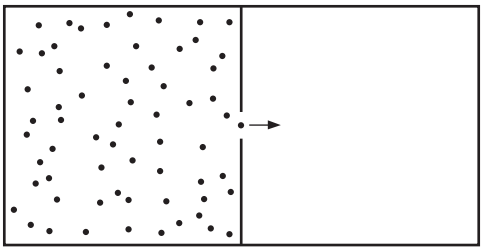

Figure 1: All the molecules start on the left side of the partition. What happens when we puncture the partition and let the molecules travel back and forth?

number that are currently on the left side of the box. In each step of "time," choose a molecule at random (by generating a suitable random number) and move it to the other side of the box (by increasing or decreasing `nLeft` by 1). To determine which side your randomly chosen molecule is on, you can simply assume that the first `nLeft` molecules are on the left and the rest are on the right; then there is no need to create any lists to represent the molecules. The output of your program should be a plot of `nLeft` vs. "time." This means that you'll need to build lists of the values to plot; you might call them `tList` and `nLeftList`. Adjust the appearance of the plot appropriately, being sure give it a suitable title and labels on both axes. Turn in a printout of this plot for `n` = 1000.

Question: Why did I put the word "time" in quotes? Can you use this simulation to determine the amount of time (in seconds) before about half the molecules will be on each side of the box?

Notice from your simulation that even after roughly half the molecules are on each side of the box, the number on the left side fluctuates significantly. It's interesting to study these fluctuations in a little more detail.

Exercise: To study fluctuations around equilibrium, modify your simulation to start with exactly half of the molecules on each side. Run the simulation for `n` equal to 10, 100, and 1000, and in each case, estimate (from the graph) the typical amount by which `nLeft` fluctuates away from `n/2`. Write down these estimated numbers and discuss them briefly. (There is no need to print these graphs.)

Exercise: Comment-out the plotting code in your `TwoBoxes` program, and modify it to instead create a histogram plot, with `nLeft` on the horizontal axis and the number of times that that value of `nLeft` occurs on the vertical axis. To store the histogram you'll need a list whose index runs over all possible values of `nLeft`. You can create this list, and initialize all its values to zero, with a statement like `hist = [0] * (n+1)`. After each step of the simulation, add 1 to the appropriate element of this list. It's customary to display the histogram as a bar graph; you can do this by using the `plt.bar` function instead of `plt.plot`. Be sure to give the plot a title and to label both axes. Run your simulation for `n` equal to 10, 100, and 1000, and briefly describe the appearance of the histogram plots. (Don't print any of these plots yet.)

If you've studied a little probability theory, you may have already realized that over the long term, the probabilities of the various possible `nLeft` values should be given by the *binomial distribution*:

$$\text{Probability} = \frac{1}{2^n}\frac{n!}{(n_{\text{left}}!)(n-n_{\text{left}})!} = \frac{1}{2^n}\frac{n!}{(n_{\text{left}}!)(n_{\text{right}}!)}. \tag{1}$$

For a sufficiently long run, therefore, the values in your histogram should be approximately equal to this formula times the number of steps in the simulation.

Exercise: Add a function definition to your program to compute and return the value of the binomial distribution, as a function of `n` and `nLeft`. Use Python's built-in `factorial` function, which is part of the `math` package. Test your binomial function in a couple of simple cases, and write the results below.

Exercise: Now use your binomial distribution function to plot the "theoretical" (long-term average) distribution of `nLeft` values on your histogram plot. Use `plt.plot` for the theoretical distribution, drawing it as a continuous line (no markers), in a color that contrasts with the histogram bars. To add a legend to the graph, insert

```
label="Monte Carlo results"
```

as a parameter in the `plt.bar` function call, and insert a similar parameter into the `plt.plot` function call. Then, before calling `plt.show`, insert the function call `plt.legend()`. Run the simulation for `n` = 10 and 100, with enough "time" steps to produce reasonably good agreement between the actual and theoretical distributions. Print the plot for `n` = 100.

Random walks

Next let us consider the erratic motion of a single microscopic particle. The particle could be a gas molecule, a dust grain suspended in a fluid, or a conduction electron in a copper wire. In all these cases the particle collides frequently with neighboring particles and therefore moves back and forth in a way that appears mostly or entirely random. Such motion is referred to as a *random walk*.

The simplest example of a random walk is in one dimension, with steps all the same size, each step equally likely to be one way or the other. This example is mathematically similar to the previous simulation of a gas in two boxes.

Exercise: Write a new Python program called `RandomWalk.py` to model a random walk in the x direction. During each time step the particle should move one unit of distance, with a 50-50 chance of moving in the positive or negative direction. Start the particle at $x = 0$. Plot a graph of the particle's position as a function of time for some fixed number of steps, then repeat the simulation several (20 or more) times over, plotting all the results on the same graph. (You can do this by simply making multiple calls to `plt.plot`, all before calling `plt.show`.) Describe the results in the space below, and turn in a printed graph from a typical run of the program. As you increase the number of steps in the walk, what happens to the typical net distance traveled?

Exercise: To quantify your answer to the previous question, modify your program to calculate the root-mean-square (rms) net distance traveled by all of your random walkers. This is the square root of the average of the squares of the final positions. After calculating this quantity, simply print it to the screen. Run your program for several different values of the number of steps in the walk, and record the results below. Can you guess an approximate formula for the rms displacement as a function of the number of steps?

Question: Why not simply compute the *average* net displacement of the random walkers, rather than the rms displacement?

Nuclear decay

A classic example of random behavior is nuclear decay. Each radioactive isotope has a certain intrinsic probability of decaying per unit time. As far as we can tell, the time when any particular nucleus decays is truly random.

Exercise: Write a new Python program called `Decay.py` to model the decay of a collection of n radioactive nuclei of the same isotope. Let the probability of each nucleus decaying per unit time be 0.001. During each time step, for each remaining nucleus, generate a random number to determine whether that nucleus decays. Once you have calculated how many nuclei decay during a given time step, subtract that number from n and then repeat. Run the simulation long enough for at least 90% of the nuclei to decay. The output of the program should be a graph of n vs. time, labeled as usual. Run your program with n initially equal to 100, and keep a printout of the graph. Repeat for $n = 10,000$.

Exercise: Use one of your graphs to determine the approximate half-life of this isotope, that is, the average time for half of the nuclei to decay. Mark this value on the graph and write the result below.

Question: When $n = 10,000$, how many nuclei do you expect to decay (on average) during the very first time interval of this simulation? Explain.

Exercise: Write a new simulation program (call it `DecayDist.py`) to answer the previous question in more detail. The program should simulate only the very first time interval of the decay of 10,000 nuclei, where each nucleus has a probability to decay of 0.001. However, the program should repeat this one-time-interval "experiment" a hundred or more times, and plot a histogram of the results, so you can see in detail how the actual number of decays tends to fluctuate around the average expected number.

After a sufficiently large number of trials, the histogram plotted by this program should take on a well-defined shape. And as you might guess, there is a fairly simple

formula for this shape. If the average expected number of decays is λ, then the probability of actually getting k decays is given by the *Poisson distribution*:

$$\text{Probability} = \frac{1}{k!}\,\lambda^k\,e^{-\lambda}. \tag{2}$$

Exercise: Using pencil and paper (in the space below), sum the Poisson distribution over all possible k values, and interpret the result. To carry out the sum, you'll need to know the so-called Taylor series for the function e^x, so look that up if necessary.

Exercise: Add a function to your `DecayDist` program to compute the Poisson distribution. (The name `lambda` has a special meaning in Python, so you'll have to use a different variable name.) Using this function, plot the "theoretical" shape of your histogram on the same graph as the actual histogram, using a legend to label the plot as before. Once everything works, print the graph and turn it in with your lab report.

Congratulations! The computer programs for this project are now complete. Be sure to check that all of your code is clearly written and adequately commented. Turn in all five programs (`RandomTest`, `TwoBoxes`, `RandomWalk`, `Decay`, and `DecayDist`) as attachments to an email message to your instructor. Then answer the following question, and turn in this lab report with the printed graphs attached at the end, in order.

Question: Briefly summarize how you worked with your lab partner on this project. How did you divide the work, and what fraction of the work did you do yourself? Any other comments?