Lab 9

Learning Objectives:

I. **What characters do I type next?** *What are the most important functional elements of classes and objects in Python?* This lab will give you the opportunity to learn the major syntax and functionality associated with classes in Python.

II. **How do I plan a project and execute my plan?** *How do I organize and write object-oriented code?* This class will give you the opportunity to practice organizing code around classes and objects.

III. **Write and Test, Write and Test....** *Write code in small bits that you can easily test.* If you code an enormous project and then test it all at once, how do you know where your error is? This lab gives you the opportunity to practice testing your code at regular intervals. Object-oriented programming makes it particularly easy to organize code in ways that can be tested.

Part 0: Starter Code

Click on the link https://classroom.github.com/a/LsYmLEmi; the lab is then located at https://github.com/physics91si/lab09-*username*. Note that there you will not be working with .ipynb files today - only .py files.

Part 1: Writing Classes for a Diatomic Molecule

Today's lab guides you through creating a simulation of the motion of a diatomic molecule. In this part, **your task is to create a Molecule class that to store and calculate information about diatomic molecules.**

For this part of the lab you will be writing code to simulate **two particles connected together by a compressible bond**. You will be creating and using a Molecule class that contains information about the entire two-particle system as well as a Particle class which contains basic information about each subparticle.

Begin by examining the already implemented Particle class in particle.py. See what data is stored in each Particle object and how you would access it. Try creating a particle by calling the constructor. Then try modifying its attributes and confirm that they changed.

You need to create a Molecule class in molecule.py that contains the following attributes and methods:

- Two attributes p1 and p2 that are Particle objects (representing the two subparticles);
- An attribute k, the spring constant of the bond connecting the two subparticles;
- An attribute L0, the equilibrium length of the bond

- A constructor that accepts the following:
    - The initial positions of the two particles
    - The masses of the two particles
    - The spring constant and equilibrium length of the molecule
- A method get_displ that returns the vector pointing from the location of particle p1 to the location of particle p2. This should be a one-liner.
- A method get_force that returns the force (vector) due to the spring on particle p1.

You should create this file by yourself. It's not scary! You can use your text editor of choice in the terminal, or you can initialize it and then edit it using the jupyter notebook viewer.

Implement this class and make sure that your get_force function it is working before continuing onward!

Part 2: Using Your Molecule Class for Calculating Dynamics

Now that you've written the Molecule class, **your task is to use it to simulate the motion of a diatomic molecule.**

We have provided you the file dynamics.py, which is set up to animate the movement of the particle. You will need to do a few things within this file:

- In init_molecule, create and return an instance of your Molecule class. The initial position should satisfy $x_0, y_0 \in [0,1]$.
- In time_step, update the positions of the two particles in molecule mol after a time dt. **There's no need to return anything; the starter code can look inside the object when animating!** We suggest you use the following algorithm for updating (called *leapfrog integration*).
    - **(Optional) Background on leapfrog integration:** Newton's Second Law is a 2nd-order ODE, so we can split it up into two coupled first order ODEs:
    $$\begin{cases} \frac{dv}{dt} = \frac{F(x)}{m} \\ \frac{dx}{dt} = v \end{cases}$$
    Using Euler's method, we can approximate:
    $$\begin{cases} \Delta v = \frac{F(x)}{m}\Delta t \\ \Delta x = v\Delta t \end{cases}$$
    Since these finite differences are most accurate about halfway between the endpoints used to calculate them, we'll calculate x and $a$at integer multiples of Δt and v at half-integer multiples of Δt.
    - **(Optional) The algorithm, in depth:**
        - For time step i,
            - Increment time by $\frac{1}{2}\Delta t$ and recalculate velocity:
            $$v(t_{i+1/2}) = v(t_{i-1/2}) + \frac{F(x(t_i))}{m}\Delta t$$

- Increment time by $\frac{1}{2}\Delta t$ again, and recalculate position:

$$x(t_{i+1}) = x(t_i) + v(t_{i+1/2})\Delta t$$

- In short, **this is nothing more than alternately updating x and v.** Your job is to implement this using the objects you've created!

Of course you'll want to test your code. Try using

- starting positions (0.2, 0.2) and (0.8, 0.8)
- masses 1 and 2
- spring constant k = 1
- equilibrium length L0 = 0 .5

Things to look for: Does the molecule behave like a spring? Does the heavier particle move less than the
lighter particle?

Part 3: Documentation

Document the Molecule class in molecule.py and its get_force function. Verify that you can access your documentation by importing it in iPython and using the ? after the function to see the documentation.

Part 4: Submit Your Lab

Don't forget to add, commit, and push your created+edited python files!