Phys 3510 Final Exam
Dec 15, 8am – 10am
100 pts total

In this project, we will simulate particles interacting with a Lennard-Jones potential in 2 dimensions (2D). The program describes the dynamics of particles moving according to classical Newtonian mechanics, where we will evolve the system in time using small timesteps dt, using the 2nd order Runge-Kutta ODE solver we've already developed in class. The module ode.py is provided. The code will be similar in structure with the harmonic-2body.py code in Assignment #7,

As we implemented in Part 2 of the course, we will combine all positions $x1$, $z1$, $x2$, $z2$ … and velocities $vx1$, $vz1$, $vx2$, $vz2$ … into a single numpy array $y = (x1, vx1, z1, vz1, x2, vx2, z2, vz2, \ldots)$ and evolve it with time, so that the program can be efficiently vectorized and cleanly coded for any number of particles. Task 1 will be a "tutorial level" where we will keep some of the individual labels like *x1* and *vx1* for clarity. In Tasks 2 and 3 we will stick with array operations and will not be allowed to refer to the positions and velocities of individual particles. All positions will evolve according to their corresponding velocities; all velocities will evolve according to their corresponding accelerations.

The tasks will involve a lot of array slicing – if you're not sure of the shape of any array, you can always check the docstrings where the shapes of inputs and outputs are provided.

Task 1. Finish the program to simulate a two-particle LJ system in 2D

a. **(10 pts)** Complete the *flj()* function using an expression of the Lennard–Jones force. You may copy it from our homework assignment or re-derive it using the Lennard–Jones potential provided here.

$$U^{LJ}(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

b. **(10 pts)** Inside *f()*, the function that calculates time derivatives, all the accelerations needed in *ydot* for the two particles have been provided. Finish populating the rest of *ydot* with velocities according to the usual format we used in class; the format for *ydot* is also explained in the docstring of *f()*.

c. **(10 pts)** Inside *animate()* is where we advance timesteps in this task. The most important line

y = ode.move_RK2(f, y, dt)

is already provided. Immediately following that, set up reflective boundary conditions so that particles bounce back from walls at $x = \pm R$ and $y = \pm R$.

d. **(10 pts)** Test your simulation to see if it works! Are the two particles moving towards or away from each other? Comment on why that happens.

Task 2. Generalize the program to many particles

Using the code from Task 1 as a template, finish Task 2 where we generalize to 16 particles.

a. **(20 pts)** Copy your *flj()* function from Task 1. Use array slicing to complete *f()*. This step follows the *flj()* structure of Task 1, so compare the two to get started.

b. **(10 pts)** To initialize velocities, we use the random number generator

rng = np.random.default_rng()

Do a web search to find out how to set up a seed, so that the random initial velocities created in the next line will always be the same in all your simulations.

c. **(20 pts)** Similar to Task 1, set up reflective boundary conditions inside *animate()*. Test your simulation to see if it works!

Task 3. Particle statistics

(10 pts) Create a separate copy of the Task 2 code and remove all animation sections. Write a simple for loop that advances time for 1000 timesteps, in replacement for the animation section. The for loop will mostly have the same contents as the *animate()* function, including the *ode.move_RK2()* part and the reflective boundary conditions part.

Evolve the system for 1000 timesteps. At the last timestep, calculate the average kinetic energy of all the particles (a single number) and the 95% confidence interval of kinetic energies (two numbers bracketing the average).