KEM381 project II
Classical Mechanics and Numerical Analysis

Shafayet Islam
Iris Tamsalu
University of Helsinki

11 March, 2025

Introduction

The focus of the project II is on classical mechanics, specifically how to solve classical mechanics problems using numerical methods. The simulations are carried out using the leap-frog algorithm.

In the first two tasks, we code a model and study the behavior of two-particle harmonic systems. The first exercise deals with a one-dimensional harmonic oscillator, while the second exercise extends the problem to two dimensions.

In the third exercise, we simulate interactions between particles using the Lennard-Jones potential. We also implement periodic boundary conditions (PBC) in the Lennard-Jones simulation.

With these three mentioned models we study the effect of the timestep on the accuracy of the simulation. For example, how well are linear and angular momentum or total energy conserved? Another aim is to find the optimal timestep that balances these factors, minimizing both error and computational time.

1 Theoretical Background

This chapter includes relevant theoretical background and equations used in the simulations.

1.1 Newton's Second Law of Motion

In classical mechanics, the motion of a particle is described by Newton's second law of motion, which states that the force applied to an object is equal to the rate of change of its momentum [2]. The equation for the force F acting on a particle is given by, where m is the mass and $\vec{a}$ is the acceleration of the particle:

$$\vec{F} = m\vec{a}$$

By computing the forces at each time step and updating positions and velocities accordingly, we can simulate how the system evolves over time. Therefore Newton's second law of motion is the core concept of the simulation.

1.2 Hooke's Law

In both task one and task two, we simulate two particles interacting through a harmonic potential. The interaction between these particles can be described by Hooke's Law, which describes how the spring force is related to the displacement from its equilibrium position [2].

In the case of the 1D two-particle system, the force (F) acting between the two particles is given by the formula, where k is the spring constant, x_1 and x_2 are coordinates of particles, and x_{eq} is the equilibrium length of the spring:

$$F = -k\left(x_1 - x_2 - x_{\text{eq}}\right)$$

For the 2D system, the force is described by the equation, where $\vec{r_1}$ and $\vec{r_2}$ are position vectors of particles and C is the equilibrium position vector:

$$F = -k\left(|\vec{r_1} - \vec{r_2}| - C\right)$$

1.3 Momentum and Energy Conservation

Momentum and energy conservation are indicators of the stability and accuracy of a simulation. In our simulations, there are two types of momentum: linear and angular [3]. Linear momentum is relevant in both 1D and 2D systems and reflects the linear motion of the particles.

Linear momentum $\vec{p}$ in the two-particle system, where m_1 and m_2 are the masses of particles, and $\vec{v}_1$ and $\vec{v}_2$ are velocities:

$$\vec{p} = m_1\vec{v}_1 + m_2\vec{v}_2$$

Angular momentum, on the other hand, is only relevant in 2D simulations, as it involves rotational motion around an axis. Angular momentum (L) in the two-particle system, where x_1, x_2, y_1 and y_2 are coordinates and v_1x, v_2x, v_1y, v_2y are the velocity components in x- and y-axis direction:

$$L = m_1(x_1 v_{1y} - y_1 v_{1x}) + m_2(x_2 v_{2y} - y_2 v_{2x})$$

Alongside momentum, the total energy of a system should be conserved as well since, in our simulation, there aren't any external forces [2, 3]. The total energy includes both kinetic and potential energy. In the 2D two-particle simulation, the following equations describe the system. C is the natural length of the spring:

$$E_{\text{total}} = E_{\text{kinetic}} + E_{\text{potential}}$$
$$E_{\text{kinetic}} = \frac{1}{2} m_1 \left(v_{1x}^2 + v_{1y}^2\right) + \frac{1}{2} m_2 \left(v_{2x}^2 + v_{2y}^2\right)$$
$$E_{\text{potential}} = \frac{k}{2} \left(|\vec{r_1} - \vec{r_2}| - C\right)^2$$

Monitoring the conservation of these quantities throughout the simulation helps verify the correctness of the numerical method and detect any accumulating errors over time.

1.4 Root Mean Square Deviation (RMSD)

In the case of a 1D harmonic oscillator, to evaluate the accuracy of the numerical solution, it is compared to the analytical solution, and the root mean square deviation (RMSD) is calculated. The mean RMSD is also calculated to find the optimal time step length that considers accuracy without forgetting computational efficiency. RMSD quantifies the average deviation between the simulated and analytically calculated particle positions at each time step:

$$\text{RMSD} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left((x_{1,\text{sim},i} - x_{1,\text{ana},i})^2 + (x_{2,\text{sim},i} - x_{2,\text{ana},i})^2\right)}$$

Here, N is the total number of time steps, $x_{1,\text{sim},i}$ and $x_{2,\text{sim},i}$ are the simulated positions of the two particles, and $x_{1,\text{ana},i}$ and $x_{2,\text{ana},i}$ are the corresponding analytical positions.

1.5 Analytical Solution for the 1D Harmonic Oscillator

The 1D two-particle system connected by a spring has an analytical solution based on center-of-mass and relative motion. The relative displacement oscil-

lates with angular frequency, where μ is the reduced mass, while the center of mass moves linearly:

$$\omega = \sqrt{k/\mu}$$

The particle positions are given by:

$$x_1(t) = x_{\mathrm{CM}}(t) + \frac{1}{2}\left(x_{\mathrm{rel}}(t) + x_0\right)$$

$$x_2(t) = x_{\mathrm{CM}}(t) - \frac{1}{2}\left(x_{\mathrm{rel}}(t) + x_0\right)$$

2 Methods

2.1 Numerical Integration Using the Leap-Frog Method

In every task of this project, we employed the leap-frog algorithm, a numerical integration method used to approximate solutions to second-order differential equations — in our case, Newton's equations of motion.

The logical steps of the algorithm are illustrated in figure 1 [2]. Coordinate and force are updated every full timestep, whereas velocity is updated every half timestep. Note: According to Newton's Third Law, the force exerted on one particle is equal and opposite to the force on the other. In our program, this is accounted for by subtracting $\frac{1}{2}\Delta t F(x)/m$ from one particle's velocity while adding it to the other.

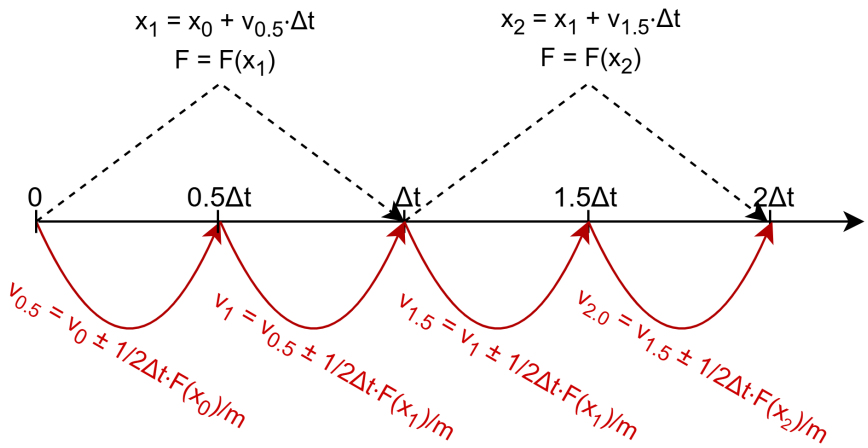


Figure 1: Computation scheme of leap-frog algorithm in our program. Above the axis, there are coordinate and force updates at full timesteps (Δt), and below the axis velocity updates at half-timesteps ($\frac{1}{2}\Delta t$).

2.2 Lennard-Jones potential

The Lennard-Jones potential is commonly used to model interactions between neutral atoms or molecules [2, 3]. One aim of the project is to study the behavior of particles in a 2D box with and without **periodic boundary conditions (PBCs)** and to analyze the energy conservation. The Lennard-Jones potential describes the interaction between two particles as:

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

where:

- r is the distance between two particles,
- ϵ is the depth of the potential well (set to 1 in this simulation),
- σ is the distance at which the potential is zero (set to 1 in this simulation).

The force between two particles is derived from the potential:

$$F(r) = -\frac{dU}{dr} = 24\epsilon \left[2 \left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \frac{1}{r}.$$

Periodic Boundary Conditions (PBCs)

- Without PBCs, particles scatter away as they move out of the box.
- With PBCs, particles that move out of the box reappear on the opposite side, simulating an infinite system.

3 Implementation

This section describes the developed simulation models and gives running instructions. You can find the `Python` programs for all three tasks on the **GitHub** repository. These are the program files:

- `harmonic_oscillator_1D.py`
- `harmonic_oscillator_2D.py`
- `lennard_jones.py`

There, you can also find a **README** file with running instructions and a **requirements.txt** file, which contains a list of packages and libraries needed to run the programs.

3.1 Two-Particle Harmonic System in 1D

3.1.1 Program Structure

Detailed descriptions of program functions are written in `Appendix C`. Figure 2 represents a simplified flow diagram of the program.

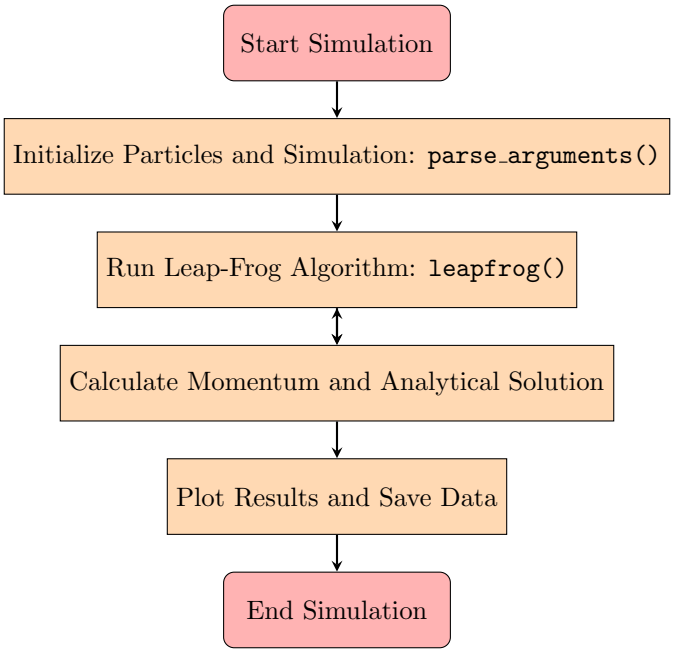


Figure 2: Flow Diagram of 1D Harmonic Oscillator Simulation.

3.1.2 Usage Instructions

The user gives input from the command line. The user must give the duration of the simulation and time step length (Δt). Here is a command line structure and an example:

```
> python <file name> --time <simulation time> --dt <time step>
> python harmonic_oscillator_1D.py --time 30 --dt 0.1
```

Initial velocities and positions, the mass of particles, the equilibrium position, and the spring constant are, in principle, modifiable, but they are currently fixed in our program according to the instructions on the slides. The initial velocities are generated randomly with `NumPy random` function in the range [-1, 1).

```
mass = 1                              # Particle mass
spring_constant = 1                   # Spring constant
x_equilibrium = 1                     # Equilibrium position
x1, x2 = 0, 0                         # Initial positions
v1,v2 = np.random.uniform(-1, 1, 2) # Initial velocities
```

3.2 Two-Particle Harmonic System in 2D

3.2.1 Program Structure

Detailed descriptions of program functions are written in `Appendix D`. Figure 3 represents a simplified flow diagram of the program.

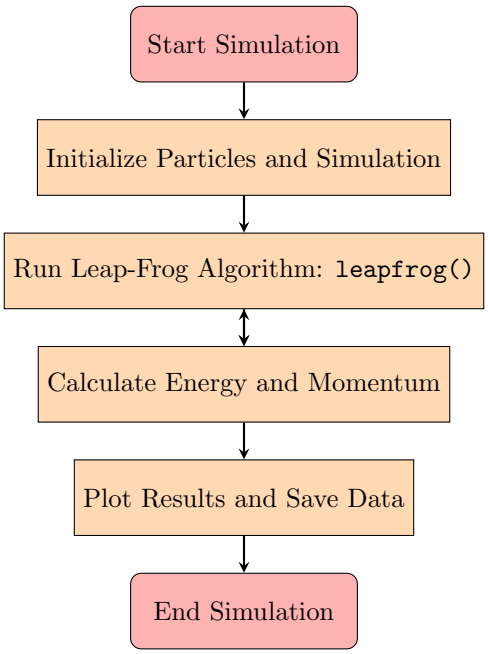


Figure 3: Flow Diagram of 2D Harmonic Oscillator Simulation.

3.2.2 Usage Instructions

The command line structure to run the two-particle simulation in 2D is the same as for the 1D simulation:

```
> python <file name> --time <simulation time> --dt <time step>
> python harmonic_oscillator_2D.py --time 30 --dt 0.01
```

Again, the initial velocities and positions, the mass of particles, the equilibrium position (C), and the spring constant are, in principle, modifiable, but they are currently fixed in our program according to the instructions on the slides. The initial velocities are generated randomly with `NumPy random` function in the range [-1, 1).

```
mass = 1                          # Particle mass
spring_constant = 1               # Spring constant
C = 0.5                           # Natural spring length
x1, x2 = 0, 1                     # Initial x-coordinates
y1, y2 = 0, 0                     # Initial y-coordinates
v1_x,v2_x=np.random.uniform(-1, 1, 2) #Initial velocities
v1_y,v2_y=np.random.uniform(-1, 1, 2) #Initial velocities
```

3.3 Lennard-Jones Simulation

3.3.1 Program Structure

In the Lennard-Jones simulation, the density is set to 1, meaning the box size L is determined by $L = \sqrt{N/\rho}$, where N is the number of particles and ρ is the density. The interaction cutoff is set to 2.5σ. Beyond this distance, the potential and force are set to zero. Particles are initialized with random positions within the box and small random velocities. The below function actually runs the whole Lennard-Jones simulation:

```
simulate(N=20, density=0.8, dt=0.001, steps=5000, use_pbc=True,
       desired_temperature=298.0, filename="lj_trajectory.xyz")
```

A detailed description of this function's parameters is given in the **appendix B** section. Details of all the functions in this simulation are given in the **appendix A**. Also, a flow diagram of how the Lennard-Jones simulation engine works is given below in **Figure 4**.

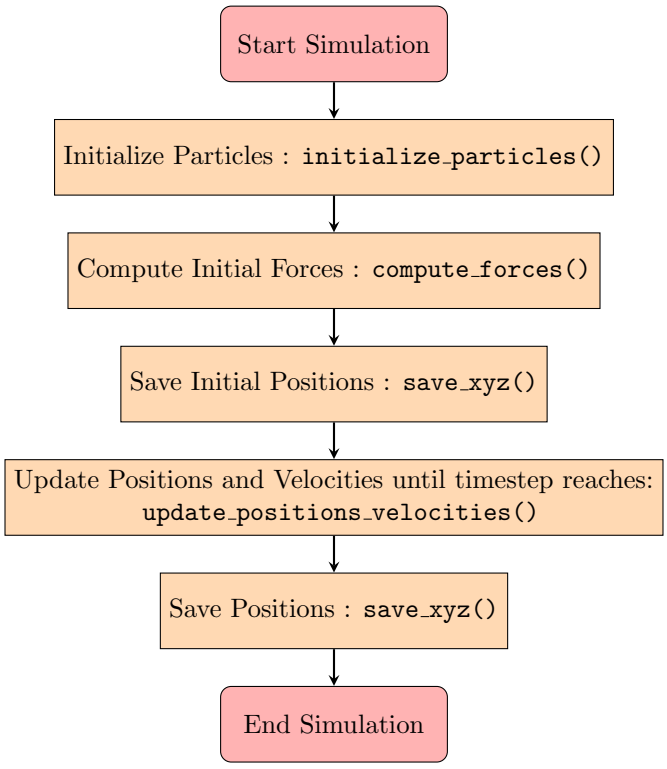


Figure 4: Flow diagram of the Lennard-Jones simulation engine.

3.3.2 Usage Instructions

The code is written in the file named `lennard_jones.py`. If you want to check the options type:

```
> python lennard_jones.py -h
usage: lennard_jones.py [-h] [--steps STEPS] [--density DENSITY]
[--file_pbc FILE_PBC] [--file_no_pbc FILE_NO_PBC] [--dt DT] [--N N]

Lennard-Jones simulation

options:
  -h, --help            show this help message and exit
  --steps STEPS         Number of simulation steps
  --density DENSITY     Density of particles
  --file_pbc FILE_PBC   PBC Output filename
  --file_no_pbc FILE_NO_PBC
                        No PBC Output filename
  --dt DT               Timestep
  --N N                 Number of particles
```

Which will show you all the arguments in the simulation. If you want a quick run, then you can write directly:

```
> python lennard_jones.py
```

That will run the simulation with default parameters. A short description of the default parameters is as follows:

```
N=20
density=0.0006
dt=0.002
steps=5000
file_pbc=lj_trajectory_pbc.xyz
file_no_pbc=lj_trajectory_no_pbc.xyz
```

`file_pbc` and `file_no_pbc` are output trajectory filenames. This will run two runs of the Lennard-Jones simulation. One is with PBC, and the other is without PBC. Detailed descriptions of N, density, and steps are found in the `Appendix B` section.

3.4 Output Files

The programs generate different output plots (trajectory, momentum, and energy conservation) made with `matplotlib`. Also, for the 1D harmonic oscillator, RMSD plots helped to visualize the leap-frog deviation from the analytical solution and to find the optimal timestep, taking into account both accuracy and computational time. The `.mp4`-files are generated using `VMD` movie maker and

later used `ffmpeg` to convert the format. You can find the videos on the **GitHub** repository under video material.

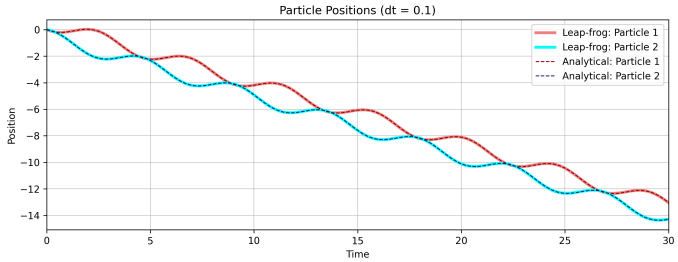
4 Results and Analysis

4.1 Two-Particle Harmonic System in 1D

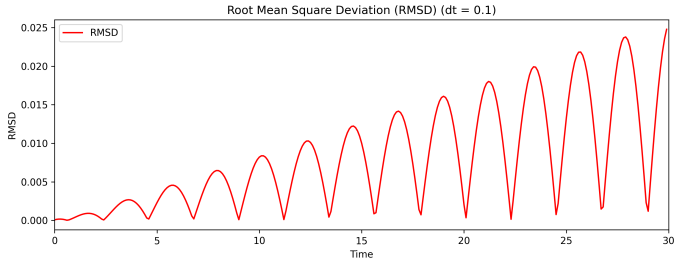
The aim of the first task was to simulate two particles interacting through harmonic potential using the leap-frog algorithm in one dimension. To evaluate the accuracy of the leap-frog solution, we also solved the harmonic oscillation analytically and checked the conservation of linear momentum throughout the simulation time. Please find the code for a two-body harmonic system in 1D on the `GitHub` repository: `harmonic_oscillator_1D.py`

Two possible trajectories are visible in figures 5a and 6a. The dashed line symbolizes the analytical solution. As can be seen from the plots, at $\Delta t = 0.1$, the leap-frog deviation from the analytical solution is not visible to the naked eye. Also, the RMSD in figure 5b increases over time but the deviations remain reasonably small. When the time step is further reduced, then every reduction of Δt by 10-fold results in around 100 times lower RMSD values.

However, with the increasing Δt, the leap-frog solution visibly starts to deviate and falls out of phase from the analytical solution. See figure 6b. With an even longer Δt, the oscillation curves become more distorted and, therefore, more inaccurate.

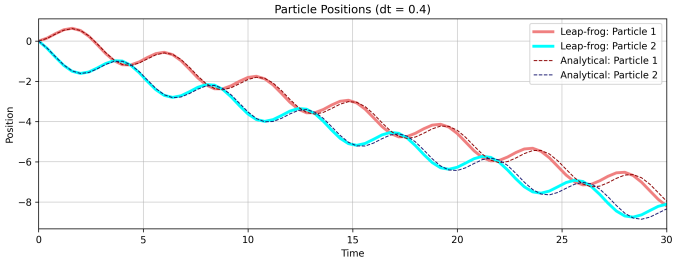


(a) Trajectories of particles.

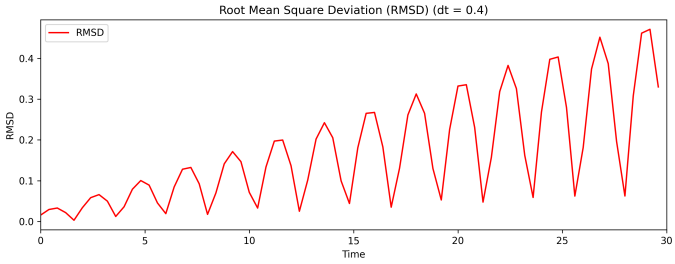


(b) RMSD between analytical and simulated positions.

Figure 5: Simulation results with $\Delta t = 0.1$ for task one.



(a) Trajectories of particles.



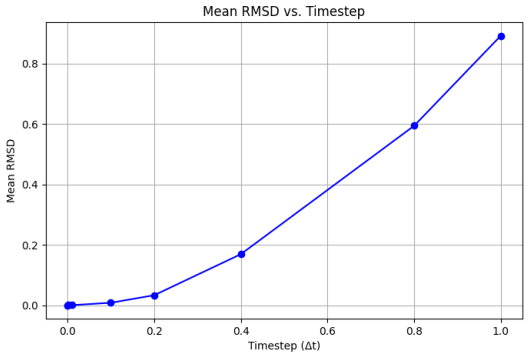
(b) RMSD between analytical and simulated positions.

Figure 6: Simulation results with $\Delta t = 0.4$ for task one.

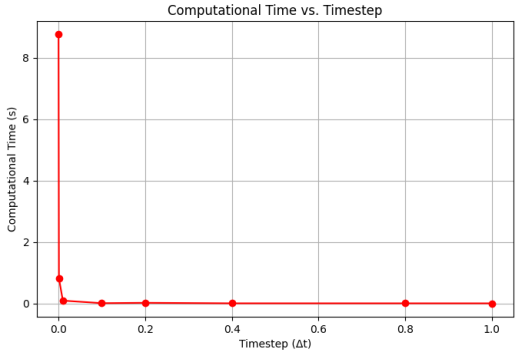
The leapfrog algorithm is only accurate when the timestep is small enough. A smaller timestep increases the accuracy of the simulation by reducing the error between the numerical and analytical solutions, but it also requires more timesteps, leading to increased computational time. Conversely, a larger timestep reduces the computational cost by requiring fewer timesteps, but it compromises accuracy, leading to a higher RMSD.

We plotted mean RMSD vs. Δt (figure 7a) as well as computational time vs. Δt (figure 7b). In 7a, it can be observed that until $\Delta t = 0.2$, RMSD remains low, but higher Δt values make the RMSD increase faster. In 7b, the computational time increases significantly when $\Delta t = 0.1$ is decreased further. Thus, $\Delta t = 0.1$ offers a good balance between accuracy and computational cost, making it the optimal time step for the simulation.

If there aren't enough timesteps, the simulation loses accuracy because the system's state is updated in large increments, leading to a loss of important information about its behavior.



(a) RMSD vs. Δt



(b) Computational time vs. Δt

Figure 7: These plots provide insights into the optimal simulation timestep.

As shown in figure 8, linear momentum was conserved throughout the simulation when $\Delta t = 0.1$, demonstrating that the system's behavior remained physically consistent. Although our model was not able to keep the momentum constant, the deviations were usually between $10^{-16} - 10^{-15}$. More action was taken regarding it in task two. However, using longer timestep like $\Delta t = 1.5$, the momentum won't be conserved throughout the simulation (figure 9).

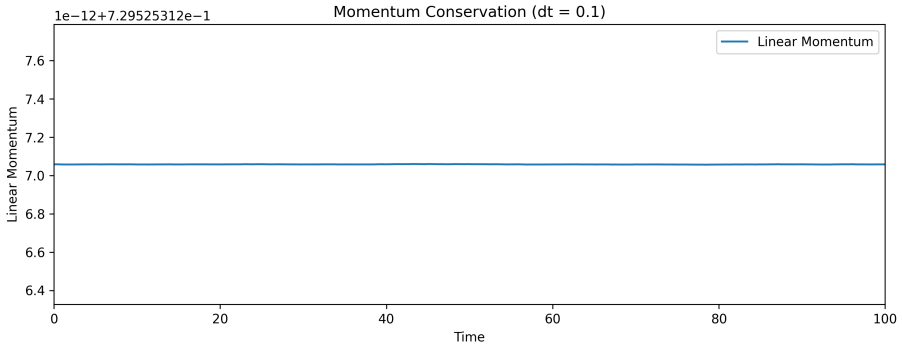


Figure 8: Linear momentum was conserved throughout the simulation when $\Delta t = 0.1$.

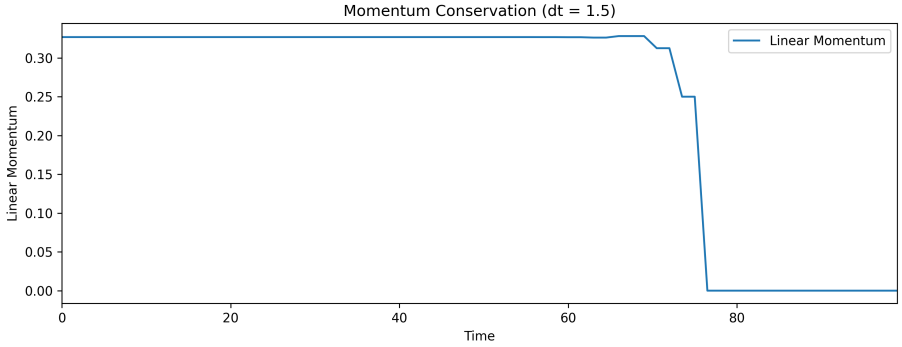
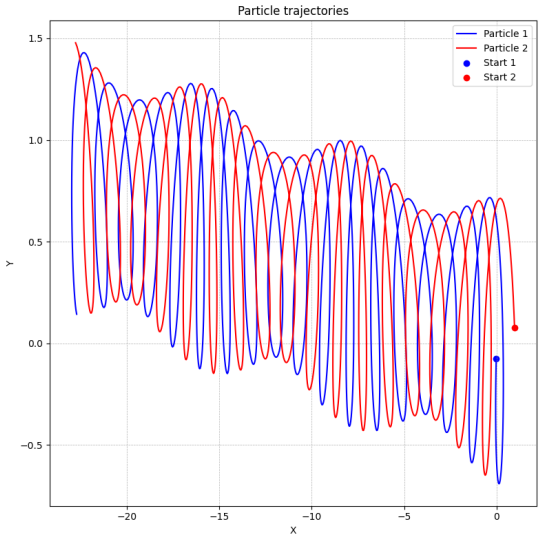


Figure 9: Linear momentum was not conserved throughout the simulation when $\Delta t = 1.5$.

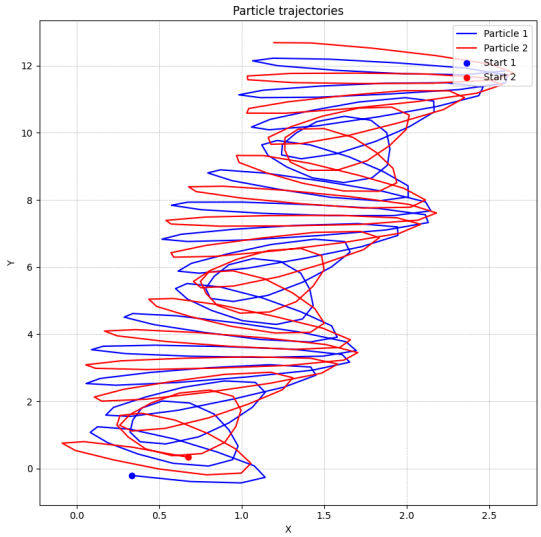
4.2 Two-Particle Harmonic System in 2D

The aim of the second task was to simulate two particles interacting through harmonic potential using a leap-frog algorithm in two dimensions. To evaluate the accuracy of the leap-frog solution, we checked the conservation of linear and angular momentum as well as total energy conservation throughout the simulation time. Please find the code for a two-body harmonic system in 2D on the `GitHub` repository: `harmonic_oscillator_2D.py`.

Two possible trajectories with different Δt values, 0.1 and 0.5, are visible in the figures 10a and 10b. When Δt becomes larger, the trajectory appears less smooth because the curves are not captured accurately.



(a) Trajectories of two particles when $\Delta t = 0.1$. Randomly generated initial velocities were $v_1x = -0.219$, $v_1y = -0.751$, $v_2x = -0.246$, $v_2y = 0.7677$.

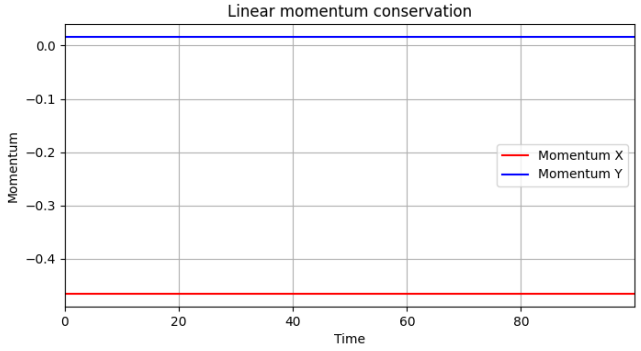


(b) Trajectories of two particles when $\Delta t = 0.5$. Randomly generated initial velocities were $v_1x = 0.5466$, $v_1y = -0.418$, $v_2x = -0.518$, $v_2y = 0.663$.

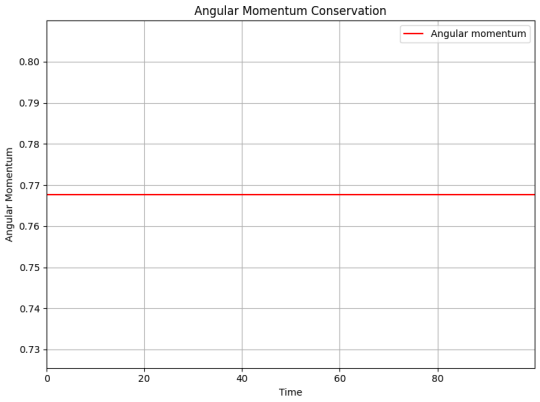
Figure 10: Simulation results with $\Delta t = 0.1$ (a) and $\Delta t = 0.5$ (b) for task two.

To further study the effect of Δt on simulation accuracy, we calculated linear

momentum, angular momentum, and total energy, and plotted them to check their conservations. The linear and angular momentum conservation plots for $\Delta t = 0.1$ are shown in figure 11. The conversion results are good. Similarly to the 1D simulator, our model was not able to keep momentum constant. Therefore, we experimented with `mpmath` library and set the precision of calculations to 50 decimal places. The fluctuations still exist but are now extremely small values, usually around $10^{-47} - 10^{-50}$.



(a) Linear momentum conservation for x-component and y-component. Linear momentum change (final value − initial value) was for x-component 1.27^{-50} and y-component 6.01^{-51}.

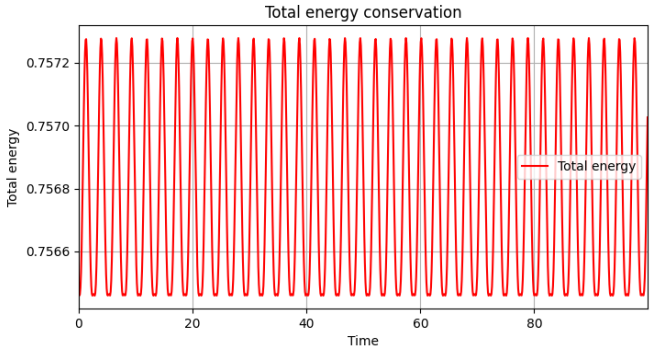


(b) Angular momentum conservation. Angular momentum change (final value − initial value) was -1.52^{-49}.

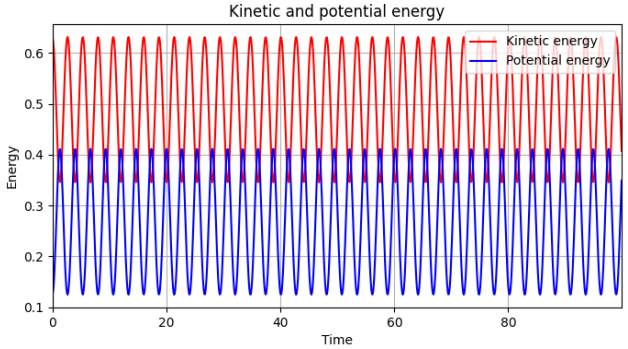
Figure 11: Linear (a) and angular (b) conservation plots of two particles when $\Delta t = 0.1$. Randomly generated initial velocities were $v_1x = -0.219$, $v_1y = -0.751$, $v_2x = -0.246$, $v_2y = 0.7677$.

Figure 12 illustrates the total energy change in the system as well as its

components, kinetic and potential energy. Here, the total energy value doesn't remain constant; however, the oscillation conserves its frequency and amplitude.



(a) Total energy conservation.

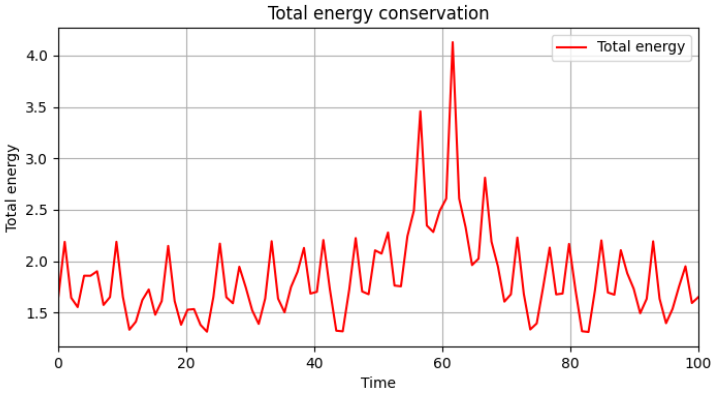


(b) Kinetic and potential energy.

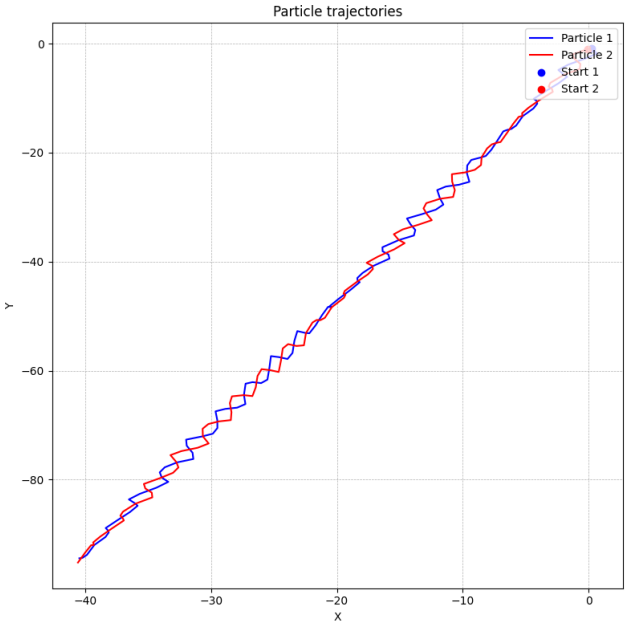
Figure 12: Total energy (a) conservation and its components (b) kinetic and potential energy, plots when $\Delta t = 0.1$. Randomly generated initial velocities were $v_1x = -0.219$, $v_1y = -0.751$, $v_2x = -0.246$, $v_2y = 0.7677$.

Similar to task one, the Δt increase reduces the quality of the simulation. If there aren't enough time steps, the simulation will have lower accuracy as the particle system configuration is updated in large increments, causing information loss of the system's behavior. In addition to a less smooth trajectory plot (10b) or completely off trajectory plot, also momentum and total energy might not be conserved in an expected way. We brought a few examples with $\Delta t = 1.0$ in the figure 13.

For $\Delta t = 1.0$, the linear and angular momentum were still nicely conserved. The eventual momentum drops in the simulation occurred at much longer Δt values ($\Delta t >= 2.0$), but in such cases, the trajectory plots already showed that the model with such timestep is useless.



(a) Total energy fluctuations.



(b) Trajectories of two particles in 2D.

Figure 13: Total energy (a) conservation and trajectory (b) plots at $\Delta t = 1.0$ show low accuracy.

4.3 Lennard-Jones Interactions

From the exercise, it's clear that our aim is to check and verify 3 points from the Lennard-Jones simulation.

1. Show how they scatter without the PBC condition
2. Add periodic condition and test the timestep needed
3. Test conservation of energy

To get confirmation on these points, we did the following:

- Particle positions are visualized at each timestep.
- The total energy is plotted over time to verify conservation.
- Observed the density effect in the PBC condition and optimized the timestep and density value from the trend.

4.3.1 Without Periodic Boundary Conditions (PBCs)

In the **Figure 14** , even though there is no PBC in the simulation, the observation for the plot is set for a certain dimension of 5x5. In this area, we can see that the particles scatter away from each other and leave the box almost immediately after 28 steps. For a detailed inspection, run the lennard_jones.py as per instruction and look at the trajectory named `lj_trajectory_no_pbc.xyz` from the output file.

4.3.2 With Periodic Boundary Conditions (PBCs)

Energy Conservation

In the periodic boundary condition, particles remain within the box, and the system behaves like an infinite periodic system. The total energy remains approximately constant, demonstrating energy conservation within fluctuating within a limited range. A simulation video can be seen from this **lj_sim.mp4**. The simulation programm can be found **Github** repository.

Effect of Timestep

In the PBC condition, we have a small timestep ($\Delta t \approx 0.0005$) required to maintain stability and energy conservation. Larger timesteps lead to huge energy drift and instability in the simulation box. This indicates we need to adjust the density of a simulation, keeping the simulation time the same. In our simulation, we took a total simulation time of 10s and then tried to reduce the density starting from 1.

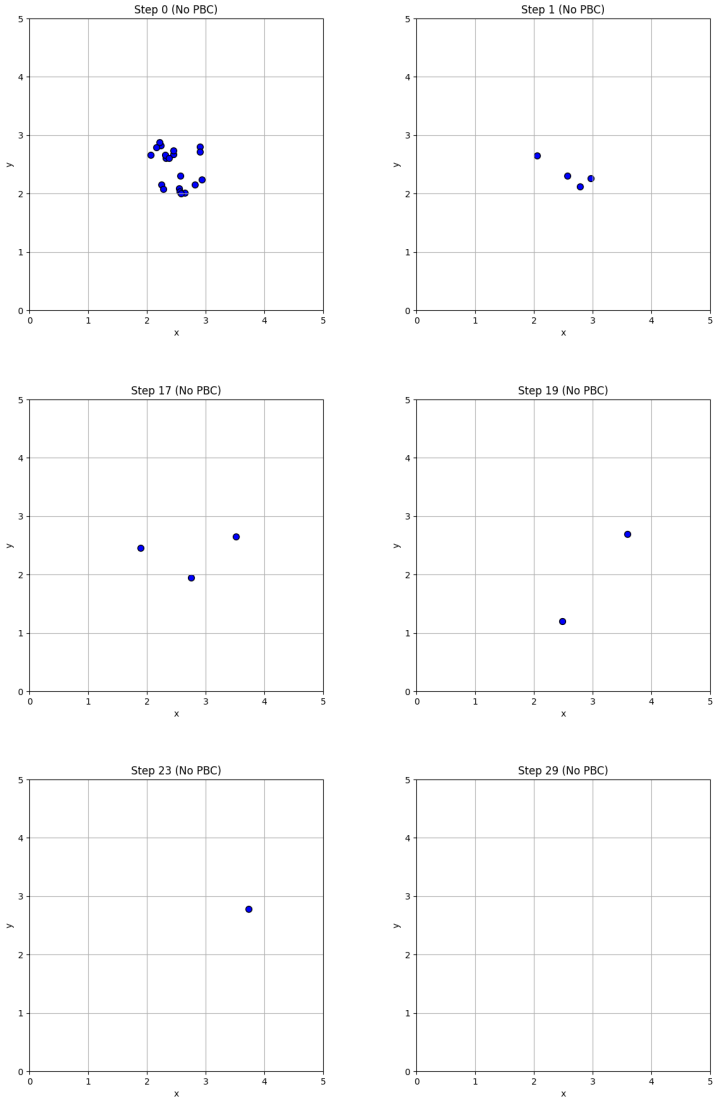


Figure 14: Particles without PBC. dt=$2 * 10^{-7}$sec. They scatter almost immediately (after 28 step=$5.8 * 10^{-7}$sec) after starting the simulation.

Effect of Density

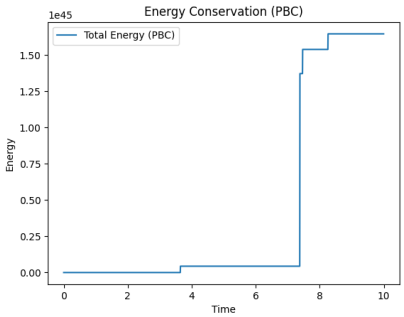
We could have run a loop of density [0.1,0.10001,...] with a pretty low-density change (0.00001) per simulation run in order to check the stability of the simulation. However, it would be really time-consuming for us, and we would have had to check manually a lot of simulations in order to check the simulation's stability. Thus, we made an educated guess and first reduced the density by 10. First, we try with density 0.1, then 0.01, 0.001, and 0.0001 until we see a reasonably stable total energy.

Reasonable lower energy means significantly lower total energy from the higher-density simulation, like the (a) and (b) of figure 15. Low density led to fewer collisions and, thus, less energy. However, at 0.0001 density, we are getting a slow velocity. After that, we slowly increase the density up to the point where the simulation gives a reasonable total energy yet higher velocity, which shows at least the particle is crossing the Periodic Boundary and collision in plain eyes. That gives us a **density of 0.0006**. We can see from (c) in figure 15 that the energy in PBC is reasonably small enough for the 10s simulation time.

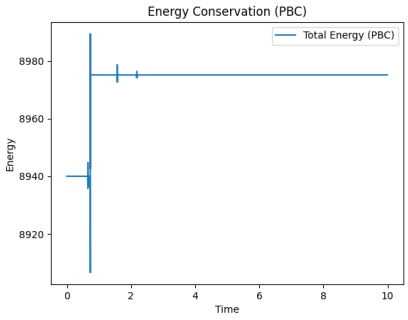
In Lennard-Jones simulations, numerical integration algorithms, in our case, Leapfrog, introduce small errors in energy conservation. These errors can manifest as fluctuations in total energy. Also, the choice of time step affects energy conservation. A larger time step can lead to more significant fluctuations, even in low-density systems.

Discussion on the need for thermostat

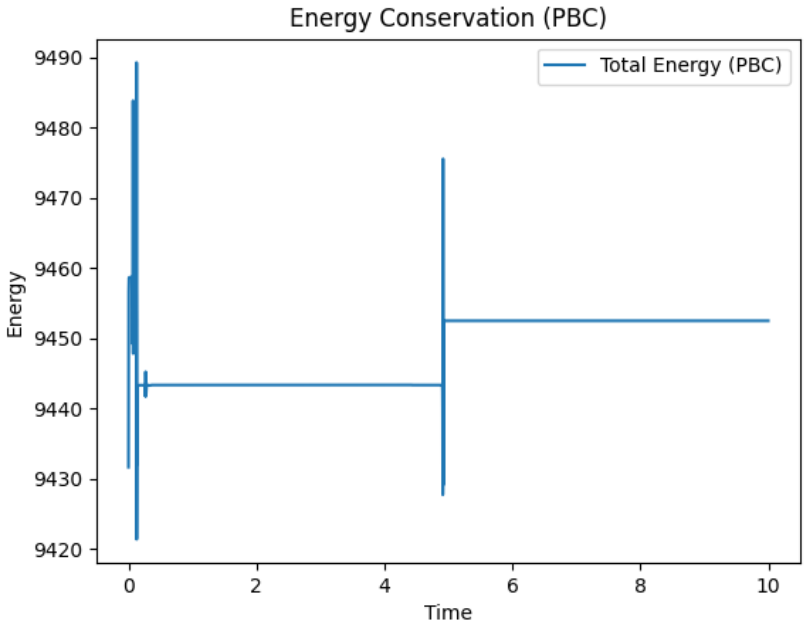
Even after all these, the simulation becomes unstable sometimes depending on the initial position velocity, leading to a huge total energy. This suggests that we need some more physics or mechanics to get more control over the simulation. In a typical simulation, we use a thermostat to do that[1]



(a) Density=0.1



(b) Density=0.0001



(c) Density=0.0006

Figure 15: (a) High density gives high energy at the end of the simulation (b) Low density gives reasonably low energy (c) Reasonably optimized density in 10s total simulation time

References

[1] Thomas Joseph Barrett and Marilyn L. Minus. Thermostat, barostat, and damping parameter impact on the tensile behavior of graphene. *SSRN*, 2022.

[2] Alex Bunker. Molecular modelling. *Lecture at University of Helsinki (KEM 342)*, 2025.

[3] Alex Bunker. Programming projects in molecular modelling. *Lecture at University of Helsinki (KEM 381)*, 2025.

5 Supplementary Information and Code availability

All code and supplementary information can be found in **Github**.

6 Appendix

A List of Lennard Jones simulation's Functions and Descriptions

The following is a list of all the functions used in the provided Python code, along with their descriptions:

1. `lj_potential(r, sigma=1.0, epsilon=1.0, rcutoff=2.5)`
   Description: Computes the Lennard-Jones potential for a given distance r. The potential is cut off at $r = \text{rcutoff}$.
   Parameters:
   - r: Distance between two particles.
   - sigma: Distance at which the potential is zero.
   - epsilon: Depth of the potential well.
   - rcutoff: Cutoff distance for the potential.
2. `lj_force(r, sigma=1.0, epsilon=1.0, rcutoff=2.5)`
   Description: Computes the Lennard-Jones force for a given distance r. The force is cut off at $r = \text{rcutoff}$.
   Parameters:
   - r: Distance between two particles.
   - sigma: Distance at which the potential is zero.
   - epsilon: Depth of the potential well.
   - rcutoff: Cutoff distance for the force.
3. `initialize_particles(N, density=1.0, desired_temperature=298.0)`
   Description: Initializes the positions and velocities of N particles in a 2D box. The velocities are scaled to match the desired temperature.
   Parameters:
   - N: Number of particles.
   - density: Density of the system.
   - desired_temperature: Desired temperature of the system.

4. `compute_forces(positions, L, sigma=1.0, epsilon=1.0, rcutoff=2.5)`
   Description: Computes the forces and potential energy for all particles using the Lennard-Jones potential.
   Parameters:
   - positions: Array of particle positions.
   - L: Size of the simulation box.
   - sigma: Distance at which the potential is zero.
   - epsilon: Depth of the potential well.
   - rcutoff: Cutoff distance for the potential.

5. `update_positions_velocities(positions, velocities, forces, dt, L, use_pbc=True)`
   Description: Updates the positions and velocities of particles using the Leapfrog algorithm. Periodic boundary conditions (PBC) can be applied if enabled.
   Parameters:
   - positions: Array of particle positions.
   - velocities: Array of particle velocities.
   - forces: Array of forces acting on particles.
   - dt: Time step.
   - L: Size of the simulation box.
   - use_pbc: Whether to apply periodic boundary conditions.

6. `save_xyz(positions, step, filename="lj_trajectory.xyz")`
   Description: Saves the particle positions to an .xyz file for visualization.
   Parameters:
   - positions: Array of particle positions.
   - step: Current time step.
   - filename: Name of the output .xyz file.

7. `plot_positions(positions, L, step, use_pbc, save_path)`
   Description: Plots the particle positions and saves the plot as an image.
   Parameters:
   - positions: Array of particle positions.
   - L: Size of the simulation box.
   - step: Current time step.
   - use_pbc: Whether periodic boundary conditions are used.
   - save_path: Path to save the plot image.

8. `calculate_total_energy(positions, velocities, L)`
   Description: Calculates the total energy (kinetic + potential) of the system.
   Parameters:
   - positions: Array of particle positions.
   - velocities: Array of particle velocities.
   - L: Size of the simulation box.

9. `simulate(N=20, density=0.8, dt=0.001, steps=5000, use_pbc=True, desired_temperature=298.0, filename="lj_trajectory.xyz")`
   Description: Runs the molecular dynamics simulation.
   Parameters:
   - N: Number of particles.
   - density: Density of the system.
   - dt: Time step.
   - steps: Number of time steps.
   - use_pbc: Whether to apply periodic boundary conditions.
   - desired_temperature: Desired temperature of the system.
   - filename: Name of the output .xyz file.

B Details of the `simulate()` function

The `simulate()` function is the main driver of the Lennard Jones simulation. Below is a detailed description of its parameters:

1. `N=20`
   Type: `int`
   Description: The number of particles in the simulation.
   Default Value: `20`
   Usage: Determines how many particles are initialized in the system. Increasing `N` will increase the computational cost of the simulation.

2. `density=0.8`
   Type: `float`
   Description: The density of the system, defined as the number of particles per unit area (in 2D).
   Default Value: `0.8`
   Usage: Controls the size of the simulation box. The box length L is calculated as $L = \sqrt{N/\text{density}}$. Higher density means particles are packed more closely together.

3. `dt=0.001`
   Type: `float`
   Description: The time step used in the simulation.
   Default Value: `0.001`
   Usage: Determines the size of the time step for integrating the equations of motion. Smaller `dt` values improve accuracy but increase the number of steps required to simulate a given time interval.

4. `steps=5000`
   Type: `int`
   Description: The total number of time steps to simulate.
   Default Value: `5000`
   Usage: Controls the duration of the simulation. The total simulation time is steps $\times$ dt.

5. `use_pbc=True`
   Type: `bool`
   Description: Whether to use periodic boundary conditions (PBC).
   Default Value: `True`
   Usage: If `True`, particles that move outside the simulation box are wrapped around to the other side using the minimum image convention. If `False`, particles can move freely outside the box.

6. `desired_temperature=298.0`
   Type: `float`
   Description: The desired temperature of the system (in arbitrary units where Boltzmann constant $k_B = 1$).
   Default Value: `298.0`
   Usage: Used to scale the initial velocities of the particles so that the system starts at the desired temperature. The temperature is related to the kinetic energy of the particles.

7. `filename="lj_trajectory.xyz"`
   Type: `str`
   Description: The name of the file where the particle positions are saved in `.xyz` format.
   Default Value: `"lj_trajectory.xyz"`
   Usage: The `.xyz` file stores the trajectory of the particles over time, which can be used for visualization or further analysis.

C List of `harmonic_oscillator_1D.py` program and Descriptions

1. `parse_arguments()`

- **Description**: Read in and validate the user input from the command line

- **Parameters**: None.
- **Returns**:
  - `time`: Total simulation time.
  - `dt`: Time step length.

2. `calculate_force(x1, x2, spring_constant, x_equilibrium)`

- **Description**: Computes the force acting between the two particles due to the harmonic spring.
- **Parameters**:
  - `x1`: Position of particle 1.
  - `x2`: Position of particle 2.
  - `spring_constant`: Spring constant (k).
  - `x_equilibrium`: Equilibrium position of the spring.
- **Returns**:
  - Force acting between the two particles.

3. `calculate_momentum(mass, v1, v2)`

- **Description**: Computes the total linear momentum of the system.
- **Parameters**:
  - `mass`: Mass of the particles.
  - `v1`: Velocity of particle 1.
  - `v2`: Velocity of particle 2.
- **Returns**:
  - Total linear momentum of the system.

4. `calculate_velocity(v1_old, v2_old, dt, force, mass)`

- **Description**: Updates the velocities of the particles using the Leapfrog method.
- **Parameters**:
  - `v1_old`: Current velocity of particle 1.
  - `v2_old`: Current velocity of particle 2.

- `dt`: Time step length.
- `force`: Force acting between the particles.
- `mass`: Mass of the particles.

- **Returns**:
  - Updated velocities of particles 1 and 2.

5. `calculate_position(x1_old, x2_old, dt, v1_halfstep, v2_halfstep)`

- **Description**: Updates the positions of the particles using the Leapfrog method.
- **Parameters**:
  - `x1_old`: Current position of particle 1.
  - `x2_old`: Current position of particle 2.
  - `dt`: Time step length.
  - `v1_halfstep`: Velocity of particle 1 at the half-step.
  - `v2_halfstep`: Velocity of particle 2 at the half-step.
- **Returns**:
  - Updated positions of particles 1 and 2.

6. `analytical_solution(t, x1_0, x2_0, v1_0, v2_0, mass, spring_constant, x0=0.0)`

- **Description**: Computes the analytical solution for the positions of two identical masses connected by a spring.
- **Parameters**:
  - `t`: Time at which to compute the solution.
  - `x1_0`: Initial position of particle 1.
  - `x2_0`: Initial position of particle 2.
  - `v1_0`: Initial velocity of particle 1.
  - `v2_0`: Initial velocity of particle 2.
  - `mass`: Mass of the particles.
  - `spring_constant`: Spring constant (k).
  - `x0`: Equilibrium position of the spring (default: 0.0).
- **Returns**:
  - Analytical positions of particles 1 and 2 at time t.

7. `calculate_rmsd(x1_sim, x2_sim, x1_analytical, x2_analytical)`

- **Description**: Computes the Root Mean Square Deviation (RMSD) between simulated and analytical positions.
- **Parameters**:
  - `x1_sim`: Simulated position of particle 1.
  - `x2_sim`: Simulated position of particle 2.
  - `x1_analytical`: Analytical position of particle 1.
  - `x2_analytical`: Analytical position of particle 2.
- **Returns**:
  - RMSD between simulated and analytical positions.

8. `leapfrog(timesteps, dt, mass, spring_constant, x_equilibrium, x1, x2, v1, v2)`

- **Description**: Runs the Leapfrog simulation for two particles connected by a harmonic spring.
- **Parameters**:
  - `timesteps`: Total number of time steps.
  - `dt`: Time step length.
  - `mass`: Mass of the particles.
  - `spring_constant`: Spring constant (k).
  - `x_equilibrium`: Equilibrium position of the spring.
  - `x1`: Initial position of particle 1.
  - `x2`: Initial position of particle 2.
  - `v1`: Initial velocity of particle 1.
  - `v2`: Initial velocity of particle 2.
- **Returns**:
  - Lists of positions, velocities, momentum, RMSD, and analytical positions for both particles.

9. `plot_results(timesteps, x1_list, x2_list, momentum_list, rmsd_list, x1_analytic_list, x2_analytic_list, dt)`

- **Description**: Plots the results of the simulation, including particle positions, momentum conservation, and RMSD.
- **Parameters**:
  - `timesteps`: Total number of time steps.
  - `x1_list`: List of positions for particle 1.
  - `x2_list`: List of positions for particle 2.
  - `momentum_list`: List of total momentum values.
  - `rmsd_list`: List of RMSD values.
  - `x1_analytic_list`: List of analytical positions for particle 1.
  - `x2_analytic_list`: List of analytical positions for particle 2.
  - `dt`: Time step length.
- **Returns**: None.

10. `save_xyz(filename, x1_list, x2_list)`

- **Description**: Saves the trajectory of the two particles in .xyz format.
- **Parameters**:
  - `filename`: Name of the output file.
  - `x1_list`: List of positions for particle 1.
  - `x2_list`: List of positions for particle 2.
- **Returns**: None.

D List of `harmonic_oscillator_2D.py` program and Descriptions

1. `parse_arguments()`

- **Description**: Read in and validate the user input from the command line.
- **Parameters**: None.
- **Returns**:
  - `time`: Total simulation time.
  - `dt`: Time step length.

2. `calculate_force(x1, x2, y1, y2, spring_constant, C)`

- **Description**: Computes the force acting between the two particles due to the harmonic spring in 2D.
- **Parameters**:
  - `x1`, `y1`: Position of particle 1.
  - `x2`, `y2`: Position of particle 2.
  - `spring_constant`: Spring constant (k).
  - `C`: Equilibrium length of the spring.
- **Returns**:
  - Force components (F_x, F_y) acting between the two particles.

3. `calculate_lin_momentum(mass, v1_x, v2_x, v1_y, v2_y)`

- **Description**: Computes the total linear momentum of the system in 2D.
- **Parameters**:
  - `mass`: Mass of the particles.
  - `v1_x`, `v1_y`: Velocity components of particle 1.
  - `v2_x`, `v2_y`: Velocity components of particle 2.
- **Returns**:
  - Total linear momentum components (p_x, p_y).

4. `calculate_ang_momentum(mass, x1, x2, y1, y2, v1_x, v2_x, v1_y, v2_y)`

- **Description**: Computes the total angular momentum of the system.
- **Parameters**:
  - `mass`: Mass of the particles.
  - `x1`, `y1`: Position of particle 1.
  - `x2`, `y2`: Position of particle 2.
  - `v1_x`, `v1_y`: Velocity components of particle 1.
  - `v2_x`, `v2_y`: Velocity components of particle 2.
- **Returns**:
  - Total angular momentum.

5. `calculate_kinetic_energy(mass, v1_x, v2_x, v1_y, v2_y)`

- **Description**: Computes the total kinetic energy of the system.
- **Parameters**:
  - `mass`: Mass of the particles.
  - `v1_x`, `v1_y`: Velocity components of particle 1.
  - `v2_x`, `v2_y`: Velocity components of particle 2.
- **Returns**:
  - Total kinetic energy.

6. `calculate_potential_energy(spring_constant, x1, x2, y1, y2, C)`

- **Description**: Computes the potential energy of the spring system.
- **Parameters**:
  - `spring_constant`: Spring constant (k).
  - `x1`, `y1`: Position of particle 1.
  - `x2`, `y2`: Position of particle 2.
  - `C`: Equilibrium length of the spring.
- **Returns**:
  - Potential energy of the system.

7. `calculate_velocity(v1_x, v2_x, v1_y, v2_y, dt, force_x, force_y, mass)`

- **Description**: Updates the velocities of the particles using the Leapfrog method.
- **Parameters**:
  - `v1_x`, `v1_y`: Current velocity components of particle 1.
  - `v2_x`, `v2_y`: Current velocity components of particle 2.
  - `dt`: Time step length.
  - `force_x`, `force_y`: Force components acting on the particles.
  - `mass`: Mass of the particles.
- **Returns**:
  - Updated velocity components for both particles.

8. `calculate_position(x1, x2, y1, y2, dt, v1_x, v2_x, v1_y, v2_y)`

- **Description**: Updates the positions of the particles using the Leapfrog method.
- **Parameters**:
  - `x1`, `y1`: Current position of particle 1.
  - `x2`, `y2`: Current position of particle 2.
  - `dt`: Time step length.
  - `v1_x`, `v1_y`: Velocity components of particle 1.
  - `v2_x`, `v2_y`: Velocity components of particle 2.
- **Returns**:
  - Updated positions of both particles.

9. `leapfrog(timesteps, dt, mass, spring_constant, C, x1, x2, y1, y2, v1_x, v2_x, v1_y, v2_y)`

- **Description**: Runs the Leapfrog simulation for two particles connected by a harmonic spring in 2D.
- **Parameters**:
  - `timesteps`: Total number of time steps.
  - `dt`: Time step length.
  - `mass`: Mass of the particles.
  - `spring_constant`: Spring constant (k).
  - `C`: Equilibrium length of the spring.
  - `x1`, `y1`: Initial position of particle 1.
  - `x2`, `y2`: Initial position of particle 2.
  - `v1_x`, `v1_y`: Initial velocity components of particle 1.
  - `v2_x`, `v2_y`: Initial velocity components of particle 2.
- **Returns**:
  - Lists of positions, linear momentum, angular momentum, total energy, kinetic energy, and potential energy for both particles.

10. `plot_results(timesteps, x1_list, x2_list, y1_list, y2_list, lin_momentum_list, ang_momentum_list, total_energy_list, kin_energy_list, pot_energy_list)`

- **Description**: Plots the results of the simulation, including particle trajectories, linear and angular momentum conservation, and energy conservation.
- **Parameters**:
  - `timesteps`: Total number of time steps.
  - `x1_list`, `y1_list`: Positions of particle 1 over time.
  - `x2_list`, `y2_list`: Positions of particle 2 over time.
  - `lin_momentum_list`: Linear momentum over time.
  - `ang_momentum_list`: Angular momentum over time.
  - `total_energy_list`: Total energy over time.
  - `kin_energy_list`: Kinetic energy over time.
  - `pot_energy_list`: Potential energy over time.
- **Returns**: None.

11. `save_xyz(filename, x1_list, x2_list, y1_list, y2_list)`

- **Description**: Saves the trajectory of the two particles in .xyz format.
- **Parameters**:
  - `filename`: Name of the output file.
  - `x1_list`, `y1_list`: Positions of particle 1 over time.
  - `x2_list`, `y2_list`: Positions of particle 2 over time.
- **Returns**: None.