Indian Institute Of Technology,
Hyderabad

Fundamental of Scientific Computing Project

Molecule Dynamics Simulator

Abhinav Choudhary (co25btech11001)
Aniruddha Kabra (co25btech11005)
Rudra Thummar (co25btech11024)
Srijan Maity (co25btech11026)
Sriramnarayanan Mohan (co25btech11027)
Tilak Asodariya (co25btech11028)

September, 2025.

1 Setting up the environment

1.1 Importing All The Essential Libraries

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
```

Numpy:

- NumPy is Python's core library for fast numerical computing, built around powerful multi-dimensional arrays.
- NumPy's broadcasting and matrix manipulation features make it easy to compute pairwise distances, directions, and forces compactly.
- NumPy's optimized linear algebra functions help update positions and velocities accurately at each time step.

Matplotlib:

- Matplotlib is a Python library for data visualization. It acts as your digital sketchpad for turning numbers into graphs and plots.
- It provides functions to create everything from simple line charts and bar graphs to more complex scatter plots and histograms. It helps you to customize colors, labels, scales, and even add mathematical equations.
- It's widely used in data science, machine learning, and research because it helps translate abstract data into patterns humans can actually see.

HTML:

- It allows you to directly embed HTML inside Jupyter notebooks (or IPython environments). That means you can display formatted text, links, tables, videos, or even interactive widgets.
- Instead of plain text output, you can make results visually appealing—like wrapping results in styled HTML, highlighting key outputs, or structuring information cleanly.
- It lets you pull in snippets of HTML or web-based resources (like iframes for YouTube, maps, or dashboards) directly into your workflow.

1.2 Define Parameters

```
# Simulation parameters
N = 12               # number of particles
L = 10.0             # box size (0..L in both x and y)
dt = 0.1             # time step
T = 1000             # number of steps (for offline runs)
```

1.3 Generating Initial State

```
# Initial state
pos=np.random.rand(N,2)*L
vel=np.random.uniform(-0.3,0.3,size=(N,2))
mass=1
plt.rcParams['animation.embed_limit'] = 100

k = 1.0       #Spring Constant
r0 = 0.8      #Preferred spacing
rc = 2.0      #Cutoff distance
```

The above code initializes a system of N particles in a two-dimensional box of size $L \times L$ with random positions and velocities. Each particle is assigned a mass of 1, and parameters are set for spring-like interactions, including a spring constant, a preferred spacing, and a cutoff distance beyond which particles do not interact. For the sake of simplicity, we assume `r0 = 0.8` and `rc = 2.0` as given in the problem statement. We will later discuss what happens if we change `r0` and `rc`.

2 Wall Reflections (bounce off walls)

2.1 Original Code

```
def walls_reflect ( pos , vel , L ):

    # Left wall (x < 0)
    left_hits = pos [: , 0] < 0
    vel [ left_hits , 0] *= -1 # flip x velocity
    pos [ left_hits , 0] = 0.0 # push back inside

    # Right wall (x > L)
    right_hits = pos [: , 0] > L
    vel [ right_hits , 0] *= -1
    pos [ right_hits , 0] = L

    # Bottom wall (y < 0)
    bottom_hits = pos [: , 1] < 0
    vel [ bottom_hits , 1] *= -1
    pos [ bottom_hits , 1] = 0.0

    # Top wall (y > L)
    top_hits = pos [: , 1] > L
    vel [ top_hits , 1] *= -1
    pos [ top_hits , 1] = L

    return pos , vel
```

2.2 Vectorized Implementation

```
def walls_reflect_other(pos, vel, L, x, y):
    vel[x, 0] *= -1.0
    pos[x, 0] = np.where(pos[x, 0] < 0, 0.0, np.where(pos[x, 0] >
        L, L, pos[x, 0]))

    vel[y, 1] *= -1.0
    pos[y, 1] = np.where(pos[y, 1] < 0, 0.0, np.where(pos[y, 1] >
        L, L, pos[y, 1]))
```

The two functions, `walls_reflect_other` and `walls_reflect`, both implement reflections of particles off the walls of a square box, but they do so in different ways. `walls_reflect_other` only updates the velocities and positions of two specific particles, identified by indices x and y, reflecting them in the x and y directions respectively, using np.where to clamp their positions within the bounds [0, L]. In contrast, `walls_reflect` operates on the entire set of particles at once, detecting all particles that have crossed any of the four walls and flipping their corresponding velocity components while resetting their positions to exactly lie on the wall. The first function is particle-specific and uses conditional assignment per particle, while the second is vectorized for all particles and checks each wall independently. Additionally, `walls_reflect_other` updates the positions and velocities directly in place, avoiding unnecessary data copying and improving performance in larger simulations, whereas `walls_reflect` returns new arrays instead. Essentially, the first is a targeted, conditional update for a few particles, while the second is a comprehensive, fully vectorized boundary handling for a full particle system.

3 Pair forces: springs between particles

3.1 Original Code

```
def pair_forces(pos):
    N = pos.shape[0]
    F = np.zeros_like(pos)
    for i in range(N):
        for j in range(i + 1, N): #implementing a nested for loop
            d = pos[j] - pos[i]
            r = np.linalg.norm(d)
            if r < rc and r > 1e-12:  #apply cutoff
                u = d / r
                fmag = k * (r - r0)
                F[i] += fmag * u
                F[j] -= fmag * u
    return F
```

3.2 Vectorized Implementation

```
def pair_forces(pos):
    d = pos[None, :, :] - pos[:, None, :]
    r = np.linalg.norm(d, axis=2)
    mask = (r < rc) and (r > 1e-12)
    u = d / (r[:, :, None] + 1e-12)
    fmag = -1 * k * (r - r0) * mask
    F = np.sum(fmag[:, :, None]*u, axis=1)
    return F
```

This function `pair_forces` computes the net force on each particle due to all other particles based on a spring-like interaction. It calculates the pairwise displacement vectors and distances between particles, applies a cutoff so that only pairs within a certain range interact (here `rc`), and normalizes the displacement vectors to get directions. The force magnitude is calculated using a linear spring formula with preferred spacing `r0` and spring constant `k`, and the net force on each particle is obtained by summing the contributions from all interacting pairs.

3.3 Let's verify Newton's Third Law

By Newton's Third Law — the law of action and reaction — every pairwise interaction produces equal and opposite forces. Consequently, when all pair forces in the system are summed, they must cancel exactly, resulting in a net total force of zero on the entire collection of particles. The below Python code implements the idea.

```
def compute_pairwise_forces(pos, k=1.0, r0=0.8, rc=2.0):
    N = pos.shape[0]
    d = pos[None, :, :] - pos[:, None, :]
    r = np.linalg.norm(d, axis=2)
    mask = (r < rc) & (r > 1e-12)
    u = d / (r[:, :, None] + 1e-12)
    fmag = -1 * k * (r - r0) * mask
    F_ij = fmag[:, :, None] * u
    idx = np.arange(N); F_ij[idx, idx, :] = 0
    F_net = np.sum(F_ij, axis=1)
    return F_ij, F_net

F_ij, F_net = compute_pairwise_forces(pos, k=k, r0=r0, rc=rc)
sum_fx = np.sum(F_net[:, 0])
sum_fy = np.sum(F_net[:, 1])
sum_vector = np.sum(F_net, axis=0)
```

The code computes displacement vectors, distances, and unit directions between particles, then applies Hooke's law (with a rest length and cutoff) to find pairwise forces. These forces are summed to get each particle's net force, and the total x and y components are checked to nearly cancel, confirming equal and opposite reactions.

This code won't give an exact total force of zero due to floating-point precision limits. Operations like subtraction, normalization, and multiplication introduce tiny rounding errors that accumulate when many forces are summed, producing residuals around 10^{-14}. These small deviations are purely numerical, not physical.
The results for a test run are:

```
Sum of x-components (np.sum): 7.105427357601002e-15
Sum of y-components (np.sum): 4.440892098500626e-15
Vector sum of net forces: [7.54951657e-15 5.30131494e-15]
```

4 Updating positions and velocities

4.1 Brute force Implementation (Not accurate enough)

```
def step_once(pos,vel,mass,dt,L):
    F = pair_forces(pos)
    acc = F / mass
    vel = vel + acc * dt
    pos = pos + vel * dt
    x = (pos[:, 0] < 0) | (pos[:, 0] > L)
    y = (pos[:, 1] < 0) | (pos[:, 1] > L)
    walls_reflect_other(pos, vel, L, x, y)
    return pos, vel
```

This function `step_once` advances the particle system by one time step. It computes forces on all particles, updates velocities and positions, and then reflects any particles that cross the boundaries of a box of size L. Essentially, it performs one iteration of the simulation, handling both particle motion and wall collisions.

However, a better and smoother implementation exists. The below implementation, `step_smooth` is more accurate than the simple `step_once`. Instead of updating the velocity fully after computing the new acceleration, it updates it in two half-steps, using the previous and new accelerations to better approximate the continuous motion. This reduces numerical errors, especially in systems with oscillatory or spring-like forces, and conserves energy more effectively over time. In contrast, `step_once` uses a straightforward update method that can lead to significant errors in energy and particle trajectories over longer simulations. Essentially, `step_smooth` provides more stable and physically accurate updates per time step.

4.2 Better and smoother implementation

```
def step_smooth(pos,vel,mass,dt,L,F_prev):
    if F_prev is None:
        F_prev = pair_forces(pos)
    a_prev = F_prev / mass
    vel = vel + 0.5 * a_prev * dt
    pos = pos + vel * dt
    x = (pos[:, 0] < 0) | (pos[:, 0] > L)
    y = (pos[:, 1] < 0) | (pos[:, 1] > L)
    walls_reflect_other(pos, vel, L, x, y)
    # new forces
    F_new = pair_forces(pos)
    a_new = F_new / mass
    vel = vel + 0.5 * a_new * dt
    x = (pos[:, 0] < 0) | (pos[:, 0] > L)
    y = (pos[:, 1] < 0) | (pos[:, 1] > L)
    return pos, vel, F_new
```

5 Energy check

Let's define functions for kinetic energy, potential energy, and total energy, which will allow us to better analyze the system's behavior.
Kinetic Energy:

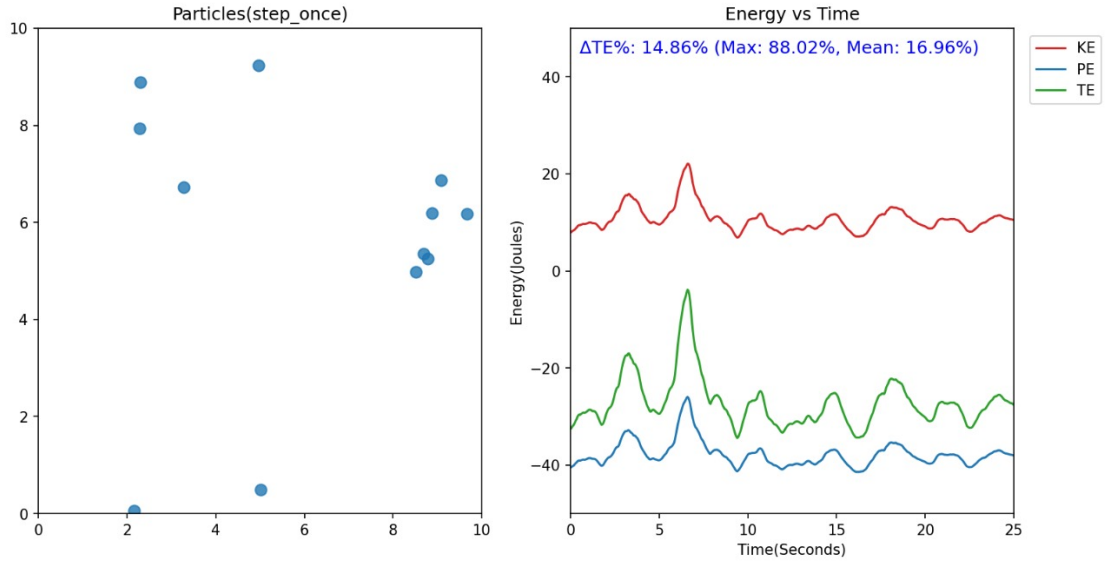
```
def kinetic_energy(vel, mass):
     KE = 0.5 * mass * (np.sum(vel** 2))
     return KE
```

We can also use `np.linalg.norm` to calculate kinetic energy.
Potential Energy:

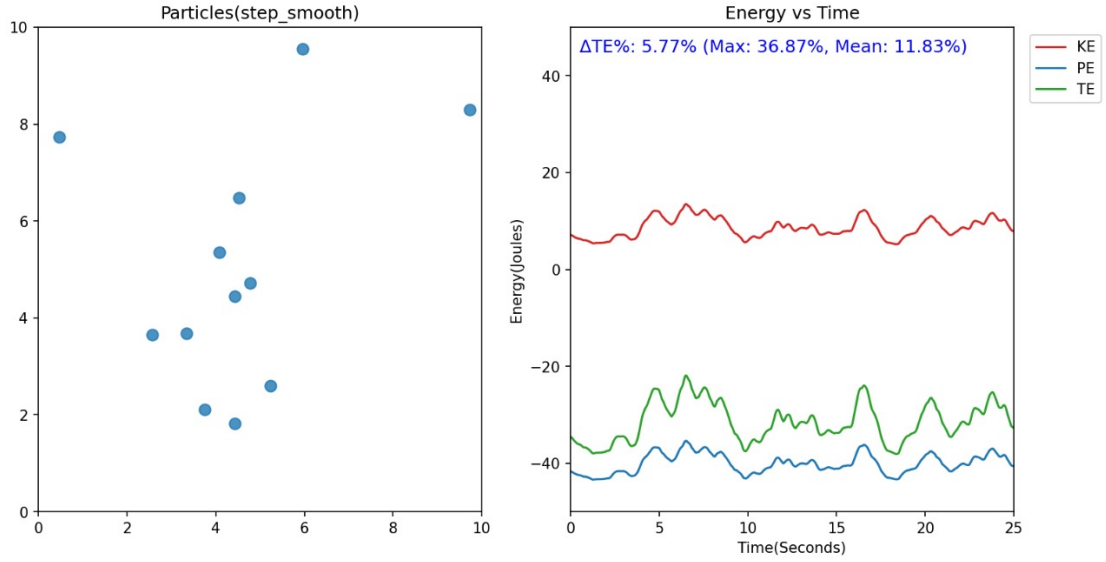
```
def potential_energy(pos, k=1.0, r0=0.8, rc=2.0):
    d = pos[None, :, :] - pos[:, None, :]
    r = np.linalg.norm(d, axis=2)
    mask_valid = (r > 1e-12)
    mask_inside = (r < rc) & mask_valid
    U = -0.5 * k * (r - r0)**2 * mask_inside
    U += (-0.5 * k * (rc - r0)**2) * ((r >= rc) & mask_valid)
    # Divide by 2 to avoid double-counting pairs
    PE = np.sum(U) / 2.0
    return PE
```

Note that the potential energy computed here will be negative. The total energy of the system is the sum of kinetic and potential energies, as expected.
We will now draw the graphs related to `step_once` and `step_smooth`.

5.1 Graph of 2D step_once for 12 particles



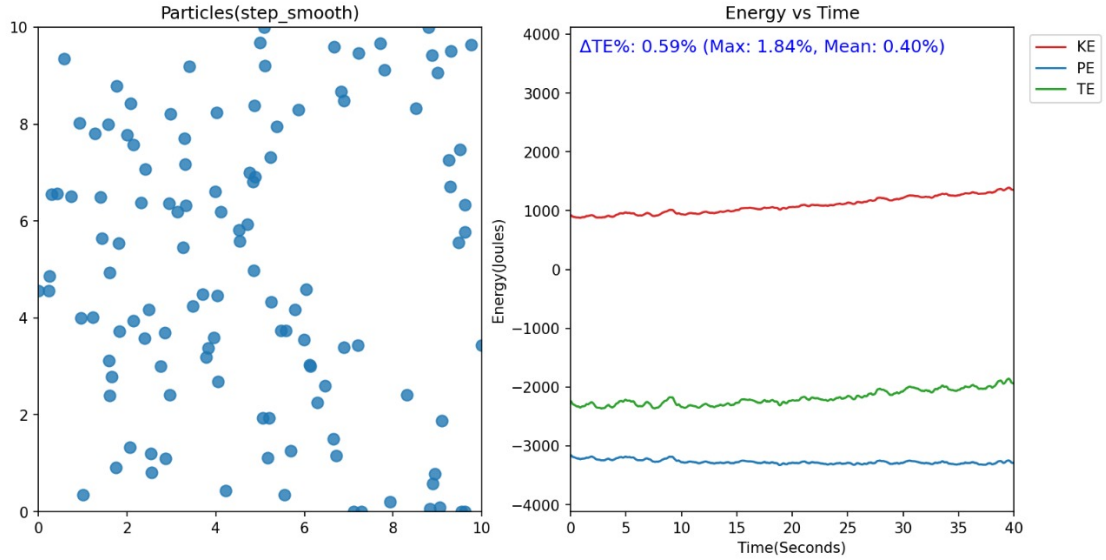
5.2 Graph of 2D step_smooth for 12 particles



It is clearly evident from the graphs that the `step_smooth` method significantly outperforms `step_once`. The smoother integration scheme exhibits far greater numerical stability, as reflected by the remarkably smaller fluctuations in both the maximum and mean total energy. This indicates that `step_smooth` preserves the system's physical energy more accurately over time, leading to a simulation that is not only smoother but also more faithful to the underlying dynamics.

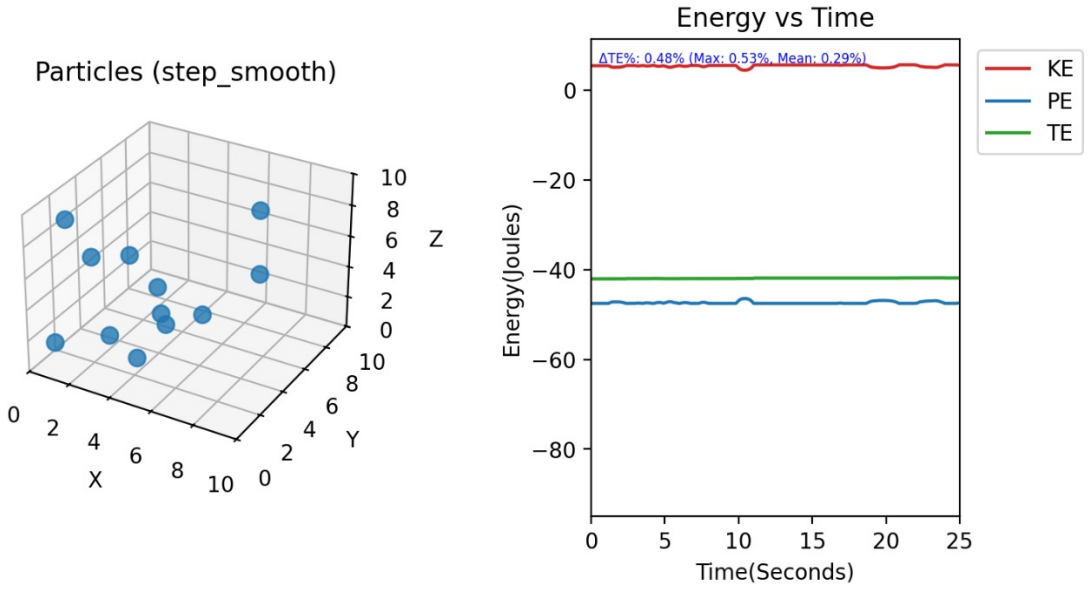
6 For 10 times the Number of Particles

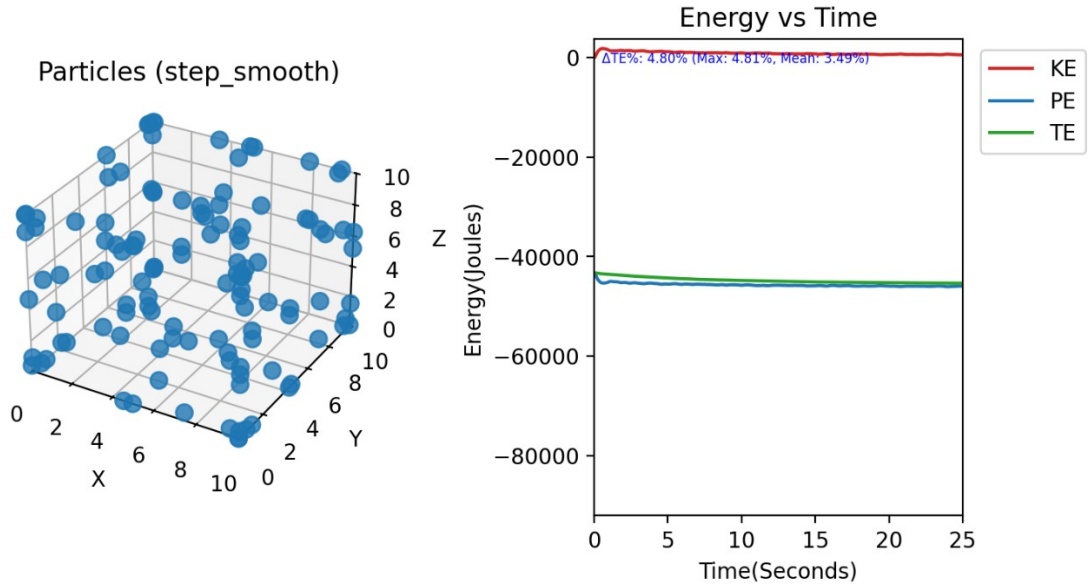
Increasing the system size to `N=120` - tenfold our earlier case - invites a new level of complexity. The calm gives way to chaos; patterns dissolve, and emergent behavior takes the stage.



7 Expanding to Other Dimensions

Instead of a box, what if we consider a room full of such particles. What happens?

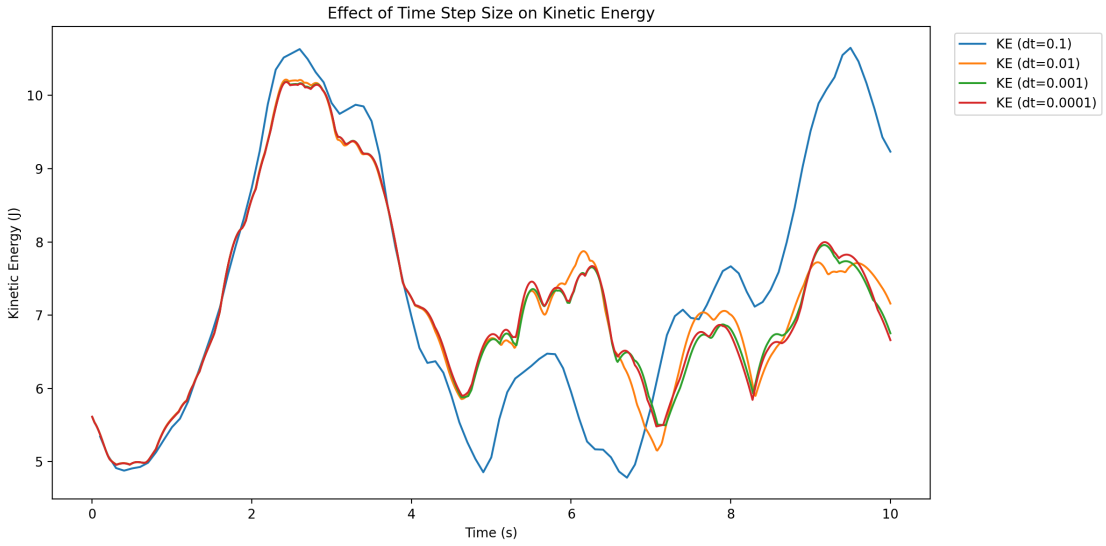




8 Toying with the model

8.1 What if we change our Timestep?

We might suspect that the observed inconsistency in kinetic energy arises because the timestep used in the integration is not infinitesimally small. To verify this, we can run simulations with different timestep sizes and plot the kinetic energy versus time. Comparing these plots will help us see how the integration accuracy—and thus energy conservation—depends on the timestep size.



In an ideal, continuous system, energy should be conserved. However, since we approxi-

mate motion using finite timestep dt, numerical errors creep in. These can cause small oscillations or even systematic drifts in kinetic or total energy. If dt is too large:

- The numerical integration becomes less accurate.
- Energy conservation deteriorates.
- The trajectory can deviate significantly from the true physical behavior.