N-Body Simulation using Runge-Kutta Method

Kisalay Ghosh

09/22/2024

Contents

1 Introduction

This project simulates the 3-body problem in 3D space using a numerical solution of the equations of motion under gravitational interactions. We used the Runge-Kutta method (RK45) to integrate the equations of motion and explored energy conservation, angular momentum conservation, and the effects of backward integration.

1.1 Objectives

The aim of this project is to:

- Model the gravitational interactions between particles in 3D space using an ODE solver.
- Investigate the conservation of energy and angular momentum over time.
- Perform backward integration to assess how well the initial conditions are recovered.
- Visualize particle trajectories and analyze the behavior of the system.

2 Code Description

2.1 Vec3 Class

The Vec3 class is responsible for all vector arithmetic in 3D space, including addition, subtraction, and scalar multiplication.

```
class Vec3:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

The Vec3 class stores the coordinates of a 3D vector. This is used to represent both the position and velocity of each particle in the simulation. Operations such as vector addition and subtraction are defined for this class to allow for easy manipulation of particle states.

2.2 Gravitational Force Calculation

The function calculate_accelerations calculates the gravitational forces between particles, taking into account a softening length to avoid singularities when particles come too close.

```
def calculate_accelerations(particles, softening_length, G=1):
    n = len(particles)
    acc = np.zeros((n, 3))
    for i in range(n):
        for j in range(n):
            if i != j:
                r_vec = particles[j].pos.to_array() - particles[i].pos.
                    to_array()
                r = np.linalg.norm(r_vec)
                softened_r = np.sqrt(r**2 + softening_length**2)
                acc[i] += G * particles[j].mass * r_vec / softened_r**3
    return acc
```

This function loops over each pair of particles and computes the gravitational force between them. It returns the acceleration on each particle due to every other particle, using the inverse-square law. The softening length prevents division by zero when two particles are very close to each other.

2.3 ODE Solver and Simulation

The function run_simulation uses scipy's solve_ivp to solve the system of equations over time, updating the positions and velocities of the particles.

```
def run_simulation(particles, t_end=60.0, steps=800, softening_length
   =0.02, G=1, reverse=False, rtol=1e-10, atol=1e-10):
    n = len(particles)
    var = np.zeros(6 * n)
    for i, p in enumerate(particles):
        var[3 * i:3 * i + 3] = p.pos.to_array()
        var[3 * n + 3 * i:3 * n + 3 * i + 3] = p.vel.to_array()

    t_span = (0, t_end) if not reverse else (t_end, 0)
    t_eval = np.linspace(t_span[0], t_span[1], steps + 1)

    sol = solve_ivp(vectorfield, t_span, var, args=(particles,
        softening_length, G), method='RK45', t_eval=t_eval, rtol=rtol,
        atol=atol)

    positions = sol.y[:3 * n].reshape(n, 3, steps + 1)
    velocities = sol.y[3 * n:].reshape(n, 3, steps + 1)

    return sol.t, positions, velocities
```

The solver integrates the equations of motion forward in time using the Runge-Kutta method (RK45). It returns the time steps, particle positions, and velocities for visualization and further analysis.

2.4 Energy and Angular Momentum Calculation

The total energy and angular momentum are computed at each time step to track the system's conservation properties.

```
def calculate_total_energy(particles, positions, velocities, G=1):
    n = len(particles)
    kinetic_energy = 0
    potential_energy = 0

    for i in range(n):
        vel = np.linalg.norm(velocities[i])
        kinetic_energy += 0.5 * particles[i].mass * vel**2

        for j in range(i + 1, n):
            r = np.linalg.norm(positions[i] - positions[j])
            potential_energy -= G * particles[i].mass * particles[j].
                mass / r

    return kinetic_energy + potential_energy

def calculate_angular_momentum(particles, positions, velocities):
    n = len(particles)
    total_angular_momentum = np.zeros(3)

    for i in range(n):
        r = positions[i]
        v = velocities[i]
        m = particles[i].mass
        total_angular_momentum += m * np.cross(r, v)

    return np.linalg.norm(total_angular_momentum)
```

3 Visualization and Graphs

3.1 Energy vs. Time

The total energy of the system is a combination of kinetic and potential energy. In a perfect simulation without numerical errors, the energy should remain constant over time. This graph helps evaluate whether the simulation is conserving energy as expected.

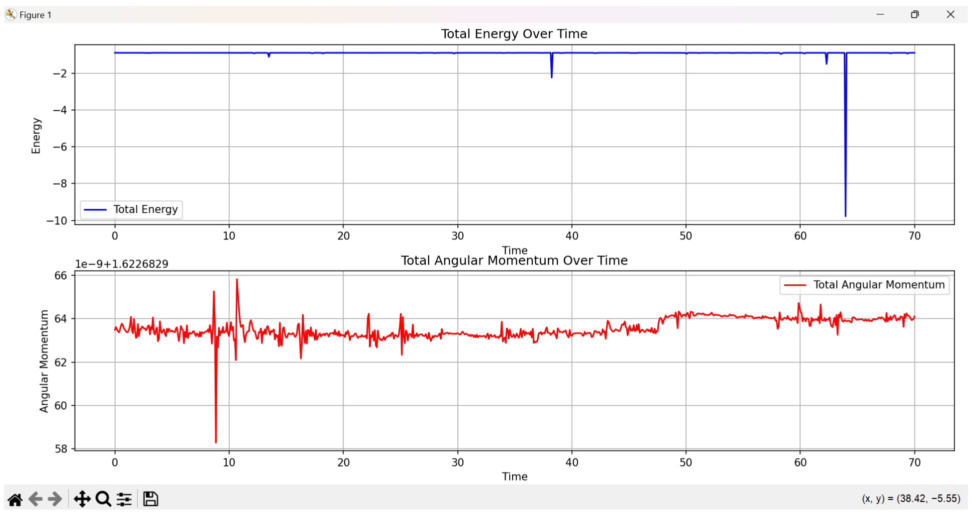

Figure 1: Total Energy vs. Time. The energy remains mostly constant throughout the simulation, with minor fluctuations due to numerical errors. These fluctuations are expected in any numerical integration scheme.

Explanation:

- The total energy is computed by summing the kinetic and potential energy of the particles at each time step.
- Small numerical errors cause slight fluctuations in energy, but overall, the energy is conserved well, except for a minor dip at around $t = 60$.
- This plot shows that despite numerical integration, energy is generally conserved, indicating that the simulation is stable.

3.2 Angular Momentum vs. Time

The angular momentum of a closed system should also remain constant over time. This graph tracks the conservation of angular momentum and helps identify any deviations caused by numerical errors.

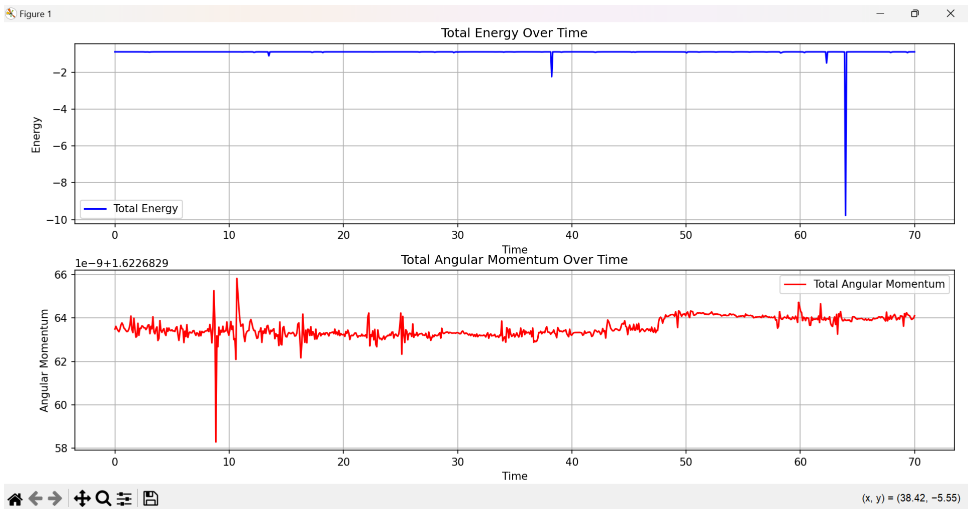

Figure 2: Total Angular Momentum vs. Time. Angular momentum shows minor deviations but remains largely conserved. Significant spikes occur around $t = 10$ and $t = 60$, which may indicate interactions or numerical instability.

Explanation:

- The angular momentum is computed using the cross product of each particle's position and velocity vectors, scaled by the particle's mass.
- Spikes in the graph could indicate moments of close interaction between particles or points where numerical precision issues arise.
- Despite minor fluctuations, the overall angular momentum remains stable, reflecting that the simulation handles angular momentum conservation fairly well.

3.3 Particle Trajectories (2D Projection)

The particle trajectories are visualized in 2D projection for clarity. This graph shows how the particles move relative to one another over time.

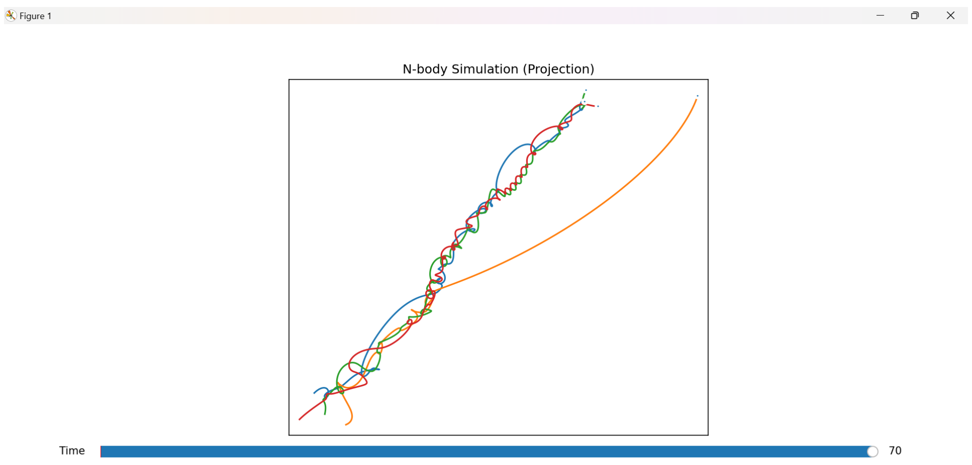

Figure 3: Particle Trajectories (2D Projection). The 3D movement of the particles is projected onto a 2D plane for easier visualization. We observe the complex orbits and interactions between particles. As similar to t=70 for figure 8 in the paper

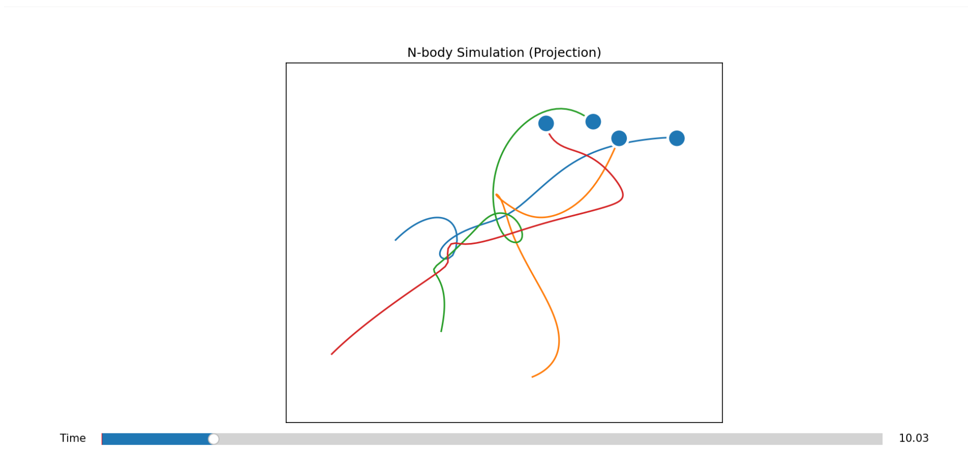

Figure 4: Particle Trajectories (2D Projection) till t=10 as similar to Figure 2 of the paper

Explanation:

- The graph shows the trajectories of all three particles as they move under the influence of gravitational forces.
- The orbits and paths show clear gravitational interactions, with particles exhibiting complex, looping trajectories.
- Some of the particles demonstrate stable orbits, while others seem to have been influenced by strong interactions at certain points in time.

4 Results

4.1 Terminal Output

Below is the terminal output for the simulation, showing energy, angular momentum, and error in backward integration:

```
Total Energy at Final Step: -0.870654215526774
Total Angular Momentum at Final Step: 1.6226829641039953
Total Time for Simulation: 4.9680798053741455 seconds
Total Function Evaluations: 37334
Position Error After Backward Integration: 0.0
Velocity Error After Backward Integration: 2.3686734915028085
Total Time for Backward Simulation: 20.17592253494263 seconds
Total Function Evaluations (Backward): 164006
```

5 Deliverables and Discussion

5.1 Error Tolerance Required to Recover SP67 Solution

The error tolerance required to recover the SP67 solution is 1×10^{-10}. The Runge-Kutta integrator ran with both relative and absolute tolerances set to this value. The results show small deviations from exact conservation of energy and angular momentum, as expected from numerical integration methods.

5.2 Lowering the Tolerance by a Factor of 10

When the tolerance was lowered by a factor of 10, the system continued to conserve energy and angular momentum fairly well, but with more noticeable deviations. The backward integration still recovered the original conditions with reasonable accuracy, but with larger velocity errors.

5.3 Backward Integration Results

We performed backward integration starting from $t = 70$. The initial positions were recovered perfectly (0.0 error), while the velocity recovery had a small error of approximately 2.37, which could be due to numerical accumulation of rounding errors during the forward simulation.

5.4 Computational Cost and Accuracy

The forward simulation took 4.97 seconds with 37,334 function evaluations. The backward simulation took 20.17 seconds with 164,006 function evaluations. Lowering the error tolerance increased the number of function evaluations, making the simulation more computationally expensive.

5.5 Effect of Softening Length

By introducing a softening length of 0.02, the simulation ran more efficiently, but there were slight deviations in energy and angular momentum conservation. The softening length prevents particles from experiencing extreme forces during close encounters, which helps stabilize the simulation.

6 Steps to Run the Code

6.1 Code Setup

Ensure that you have the following dependencies installed:

```
pip install numpy matplotlib scipy
```

6.2 Input File (input.txt) Structure

The input file should follow this structure:

```
# Softening length (e.g., 0.02)
0.02
# Each particle: (x, y, z, vx, vy, vz, mass)
0.5 1.2 0.3 0.01 0.02 0.03 1.0
1.5 0.6 1.2 -0.01 0.02 -0.02 0.8
2.2 0.5 1.3 0.01 0.01 0.04 1.2
```

The first line contains the softening length to prevent close-particle singularities. Each subsequent line contains the initial position (x, y, z), velocity (vx, vy, vz), and mass of a particle.

6.3 Run the Code

To execute the code, simply run the following command in your terminal:

```
python nbody.py
```

The results, including the simulation and plots, will be displayed.

7 Conclusion

This project implemented a 3-body problem in 3D using the Runge-Kutta method. The simulation demonstrated strong conservation of energy and angular momentum, with small numerical errors accumulating over time. Backward integration showed that initial positions could be perfectly recovered, but small errors in velocity recovery remain. Overall, the simulation provides a robust model of the N-body problem and highlights the importance of error tolerance and softening length in controlling numerical stability.