N-Body Simulation in C++

Goal: Implement an N-body simulation in C++ to compute the interactions between particles based on gravitational forces and simulate their motion.

Before starting, note that the problem looks more complicated than it actually is.

1 Overview of the N-Body Problem

The N-body problem involves predicting the individual motions of a group of particles that interact with each other through gravitational forces. This problem is foundational in astrophysics, computational physics, and numerical simulations.

Key aspects of the N-body problem:

- **Gravitational Force**: Each particle in the system exerts a gravitational force on every other particle. The force between two particles is computed using Newton's law of gravitation:

$$F = G\frac{m_1 m_2}{r^2}, \tag{1}$$

  where $G = 6.674 * 10^{-11}$ is the gravitational constant, m_1 and m_2 are the masses of the two particles, and r is the distance between them.

  The direction of the force is from the particle that applies the force and towards the particle that it is applied on. That is to say, for particle 1 and 2 at location l_1, l_2, the force applied by particle 2 on particle 1 is $\vec{F} = \frac{l_1 - l_2}{||l_1 - l_2||} G \frac{m_1 m_2}{||l_1 - l_2||^2}$

- **Softening Factor**: A small value is added to r^2 (or $||l_1 - l_2||^2$) to prevent numerical instability when two particles are very close. (In order to avoid a division by 0.)

- **Equations of Motion**: The motion of each particle is updated based on the net gravitational force acting on it. The equations of motion are:

$$\vec{a} = \frac{\vec{F}}{m}, \quad \vec{v}_{new} = \vec{v}_{old} + \vec{a}\Delta t, \quad \vec{x}_{new} = \vec{x}_{old} + \vec{v}_{new}\Delta t, \tag{2}$$

  where $\vec{a}$ is the acceleration, $\vec{v}$ is the velocity, $\vec{x}$ is the position, and Δt is the time step.

- **Applications**: The N-body problem is used to model planetary systems, galaxy formation, and interactions between celestial bodies.

2 Programming Requirements

TODO. Implement the N-body simulation in C++ by completing the following tasks:

1. **Define the Simulation Structure**: Create a structure or class to represent the state of the simulation. This should include:

    - Particle properties: masses, positions, velocities, and forces (e.g., arrays or vectors to store these values for all particles).
    - Initialization functions:
        - Random initialization of particle properties (masses, positions, velocities).
        - Predefined configurations such as a simple 2 or 3 particle setup (e.g., Sun, Earth, Moon).
        - Load from file. Check recommended format.

2. **Force Calculation**: Write a function to compute the gravitational force between every pair of particles and sum the force vectors for each particle. Ensure that forces are reset to zero at the start of each time step.

3. **Integration of Motion**: Implement functions to:
    - Update particle velocities based on the computed forces and their masses.
    - Update particle positions based on their velocities and the time step.

4. **Output State**: Outputs the state of the simulation (e.g., positions, velocities, forces) to a log file or standard output. If you use the recommended format for input and output, you can use the python script to visualize the output.

5. **Simulation Loop**: Write a main simulation loop that:
    - Iterates over a fixed number of time steps.
    - Updates forces, velocities, and positions at each step.
    - At regular intervals, output the state to a log file.

Output Requirements. Your program should output the state of the simulation (positions, velocities, and forces of all particles) at regular intervals to a log file or the console.

Suggestions.

- Use double precision for all calculations to ensure numerical stability.
- Start with a small number of particles (e.g., 3 for a solar system model) to debug and validate your implementation.
- You can do the steps highlighted one at a time and check the values manually (check the force is in the right direction for instance)
- **Command-Line Interface**: Ensure the program accepts the following inputs as command-line arguments:
    - Number of particles: a number indicate to run a random model with that number of particle. A string parses the format from a file.
    - Time step size (Δt).
    - Number of iterations (time steps).
    - How often to dump the state.
- **Output format:** We recommend to use the same input and output format to follow the convention of `solar.tsv`. That is to say
    - one state per line
    - each line starts with a number that indicates how many particles there are

  - then each particle is encoded in that line by giving first mass, then position (x, y, and z), then velocity (vx,vy,vz), then force (fx,fy,fz)
  - each value on the line is separated by a tab. (this is the tsv format, spreadsheet can open them)

- If you use the output format you can use the plotting script to visualize your output: `python3 plot.py solar.tsv solar.pdf 10000`. You can pass a multi line file instead of solar.tsv to have one page per state. The last parameter is to display an arrow

3 Benchmark

TODO: On centaurus. How long does it take to simulate the solar system with dt = 200 and for 5000000 steps? How long does it take to simulate 100 particles dt=1 for 10000 steps? How long does it take to simulate 1000 particles dt=1 for 10000 steps?

4 Submission Instructions

TODO. Submit a single archive containing the following:

- Source code files implementing the simulation.
- A Makefile to compile the project.
- A README file with detailed instructions on how to compile and run your simulation.
- Log files containing the simulation output for at least three test cases with different configurations.

Note. Ensure your code is well-documented and follows clean coding practices. Use version control (e.g., git) to manage your project and track your progress.