CSE305 Concurrent Programming: N-Body simulation project

Martin Lau, Oscar Peyron, Ziyue Qiu

May 20, 2025

1 Introduction

This document outlines the N-Body simulation project for CSE305, which includes how we approached the problem, our project structure, code breakdown and strategies, and encountered difficulties.

To get started quickly, try typing `make nbody` in the project directory, and generate a basic gif file. For information on how to execute the code, check our README file.

For this project, our aim is to be able to simulate systems of bodies with forces interacting with one another in 2D (such as orbiting planets around the solar system with gravitational forces, or with particles interacting with each other with Coulomb forces). This therefore includes two sections:

1. Simulation of the N bodies and their evolving states.
2. Visualization of recorded telemetries into an animated output.

Below is a general view of our project repository file structure.

Project directory summary

```
/
├── output ..................................... Output gif and csv files
│   ├── telemetry.csv
│   └── out.gif
├── src ..................................................... Core files
│   ├── core.cpp/hpp ............................ Used throughout the entire proj
│   ├── simplesimulation.cpp/hpp
│   ├── barneshutt.cpp/hpp
│   ├── particlemesh.cpp/hpp
│   └── particlemeshcuda.cpp/hpp
├── py ..................................................... Python testing code
│   └── verifier.py
├── makefile ..................................................... Project builder
├── main.cpp ..................................................... Main execution file
├── mainparticlemesh.cpp ........................... Aux particle mesh run file
├── test_telemetry.cpp ..................................... Telemetry test file
└── report (PDF, TeX)
```

The work has been split as follows. The arrangement is not defining, as we like to help each other out in different parts. Our files have still been mostly separated to properly separate who worked on what functionalities.

1. Martin handles general project structure, core classes, visualization, and the naive and its optimized algorithm implementation.
2. Ziyue handles the Barnes Hutt algorithm.
3. Oscar handles the Particle Mesh algorithm implementation.

2 Core components

2.1 Main elements

There are 3 primary classes defined used throughout our project: `Vector, Body, System`. These are defined in `core.cpp/hpp`.

1. `Vector` holds information on a pair of numbers, and also allows for operations with other vectors/scalars (as opposed to using `std::pair`).
2. `Body` holds information on a given particle or body, including its mass, coordinates, velocity, and acceleration. It also contains an update method which updates its position and velocity based on acceleration.
3. `System` stores a collection of bodies and its recorded telemetry from the simulations we are going to do. It also contains the visualization function, which takes its recorded telemetry and outputs an animated file.

Note: This organization is heavily inspired from one assignment from CSE306 Computer Graphics.

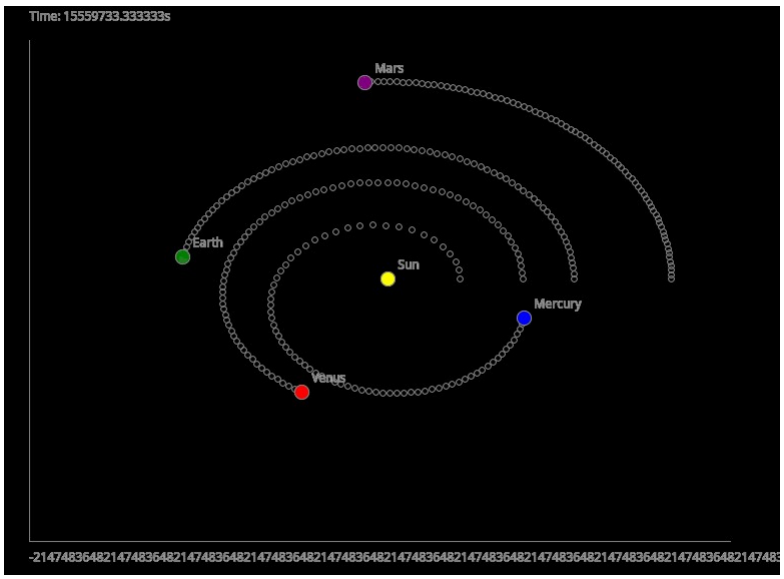
Elaborating more on the telemetry stored within the `System` class, this is stored as a vector of vector of `Vector`. What a mouthful! Our simulations creates different steps/frames. Each step/frame contains the positions of all bodies inside the system (this is a vector of `Vector`). To have the entire telemetry, we have a vector of frames, or the aforementioned data structure for the telemetry.

2.2 Visualization code

To visualize the code, we opt to create a gif animation after the telemetry is recorded, using the `ImageMagick/Magick++` libraries. The bulk of the visualization code is located in `core.cpp` as a method for the `System` class. As of present, the visualization function works in two steps. First, it creates the frames for the animation, then writes the frames to a gif file. The frame creation is relatively quick, and most of the visualization time is actually spent in one line (`writeImages(frames.begin(), frames.end(), name)`).

We will look into further solutions to try and decrease the time taken to create the visualization gifs, like reducing image quality, or the number of frames.

For testing purposes, there is also a visualizer in `visualizer.py` which creates a animation using Python's `matplotlib` library, working significantly faster, allowing us to test the correctness of our telemetries.



3 Naive algorithm

So far, we have a basic implementation of the naive algorithm without multi threading (most of my time was taken bugfixing core functionalities and getting visualization to work).

Algorithm 1 Naive simulation outline

Require: System of bodies with masses m_i, initial positions $\vec{r}_i$, velocities $\vec{v}_i$
Require: Time step Δt, number of steps N
  telemetry $\leftarrow \emptyset$
  telemetry.append(initial positions)
  for step $\leftarrow 0$ to $N-1$ **do**
    for each body i **do**
      $\vec{a}_i \leftarrow \vec{0}$ ▷ Reset accelerations
    end for
    for $i \leftarrow 0$ to $n-1$ **do**
      for $j \leftarrow i+1$ to $n-1$ **do**
        $\vec{F}_{ij} \leftarrow$ ComputeGravitationalForce($body_i$, $body_j$)
        $\vec{a}_i \leftarrow \vec{a}_i + \vec{F}_{ij}/m_i$
        $\vec{a}_j \leftarrow \vec{a}_j - \vec{F}_{ij}/m_j$
      end for
    end for
    for each body i **do**
      $\vec{v}_i \leftarrow \vec{v}_i + \vec{a}_i \Delta t$ ▷ Update velocity
      $\vec{r}_i \leftarrow \vec{r}_i + \vec{v}_i \Delta t$ ▷ Update position
    end for
    telemetry.append(current positions)
  end for
Ensure: Position history for all bodies stored in telemetry

Other aspects (parallelizing the update steps, parallelizing forces computations, avoiding race conditions) will be implemented later on.

4 Particle Mesh based algorithm

5 Barnes Hutt algorithm