N-Body Simulator - Documentation

Ryan Kwong

November 2024
rev. 1

1 Introduction

The three-body problem is a famous unsolvable physical system in classical mechanics. Because each body is influenced by both surrounding bodies, slight changes in the initial conditions (i.e., initial position, initial velocity) can lead to very different trajectories. Thus, there is no equation that can represent the solution to every possible initial condition.

However, it *is* possible to numerically simulate the 3-body problem. This simple Python program visualizes the effects of changing the initial conditions. In addition, it can be scaled up to support an arbitrary number of bodies n. This documentation will go over the mathematical and programming considerations that went into this simulation.

2 The Mathematics

This simulation relies on the kinematic equations, Newton's 2nd Law of Motion, and Newton's Law of Gravitation. Formulaically, they can be expressed as:

$$\vec{a} = \frac{d\vec{v}}{dt}$$

$$\vec{v} = \frac{d\vec{r}}{dt}$$

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

$$\vec{F_G} = G\frac{m_1 m_2}{|\vec{r_2} - \vec{r_1}|^3}(\vec{r_2} - \vec{r_1})$$

where $G \approx 6.67 \times 10^-11\mathtt{m}^3/\mathtt{kg} \cdot \mathtt{s}^2$

Applying these equations to each body results in a large number of equations. This can be reduced by using a transition matrix equation:

$$\vec{s}[t + \Delta t] = \mathbf{T}\vec{s}[t] + \vec{a}_{aug} \tag{1}$$

where

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\vec{s}[t] = \begin{bmatrix} r_x[t] \\ r_y[t] \\ v_x[t] \\ v_y[t] \end{bmatrix}$$

$$\vec{a}_{aug} = \begin{bmatrix} a_x[t] \\ a_y[t] \\ a_x[t] \\ a_y[t] \end{bmatrix} \cdot \frac{\Delta t^2}{2}$$

Notice that the matrix transformed the equations from the continuous time domain to the discrete time domain. This makes the programming of the simulation possible. Also notice that the constant acceleration kinematic equation $x(t) = \frac{1}{2}at^2 + v_0 t + x_0$ was used. This is usually a sufficient approximation in the discrete time domain for sufficiently small Δt. More information about this approximation can be found in Section 5.1.1. Finally, in order to account for the forces between the bodies, Newton's Second Law of Motion can be combined with Newton's Law of Gravitation to find the acceleration of the ith body as such:

$$\vec{a}_i = \sum_{j \neq i}^{n} G \frac{m_j}{|\vec{r}_j - \vec{r}_i|^3}(\vec{r}_j - \vec{r}_i) \tag{2}$$

These equations are applied at each time step of the simulation.

3 The Program

3.1 The Simulator

The simulation utilizes a `Body` class to represent the physical properties of each individual body. The following are the data members of the class:

Class Members		
Member Name	Type	Description
position	NDArray	Position vector
prev	NDArray	Previous position vector(will be used in future calculations for Verlet integrations)
velocity	NDArray	Velocity vector
acceleration	NDArray	Acceleration vector
mass	float	Body mass
color	tuple	Tuple for RGB color
radius	int	Radius for pygame to draw
tracer	list	List of points for pygame to draw a polyline for tracers

Figure 1: Table of data members

The simulation has two main functions that drive the mechanics, `calculate` and `update`.

`calculate` takes in a list of `Body` objects, `bodies`, and an integer ,`g`, and calculates the new accelerations of each body in `bodies` using equation (2). The `g` parameter can be used to adjust the gravitational constant in Newton's Law of Gravitation. This is especially useful for stabilizing the simulation when working with large numbers, which happens often with realistic planet masses and distances.

`update` takes in a list of `Body` objects, `bodies`, and a time step parameter, `dt`, and applies the state transition matrix expressed in equation (1). The `dt` parameter can be adjusted to speed up or slow down the simulation. Note that a large `dt` will decrease the accuracy of the simulation, as explained in the kinematic equations described in Section 2, while a smaller `dt` can be more resource intensive.

3.2 The GUI

The GUI uses the `pygame` module to draw out the simulation graphics. It uses the `position`, `color`, `radius`, and `tracer` members of the `Body` class to draw out the bodies and a tracer path. See Section 5.2.1 for more information about future improvements to the GUI.

4 Using the Simulation

4.1 2 Body Orbit

As a sanity check to make sure the physics of the simulation is working, the default configuration is a numerically accurate representation of the Moon orbiting the Earth. If the resulting orbit is elliptical, the simulation is working as intended. The result should look like:

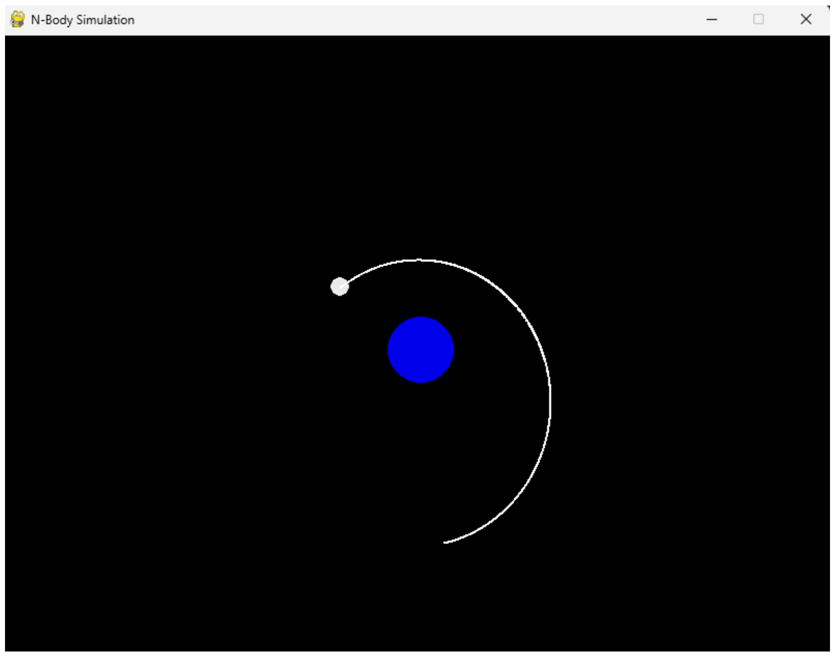

Figure 2: 2-Body Elliptical Orbit

4.2 3 Body Orbit

Consider the same Moon-Earth system. If another moon was placed on the opposite side of the earth with an opposite orbital path, it is expected that the acceleration due to the forces between the moons will cause both moons to deviate from their orbital path. The result should look like:

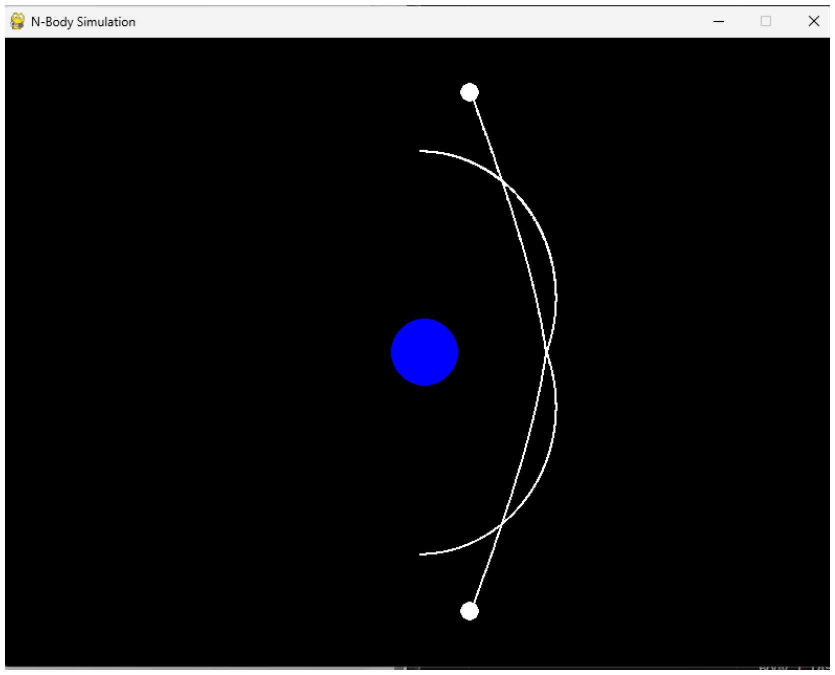

Figure 3: 3 Body Orbit

Of course, the two moons should realistically collide. See Section 5.1.2 for more information about future iterations of the simulation and how it will incorporate collisions.

5 Considerations

This section will cover the shortcomings and possible future improvements for the simulation.

5.1 Physical and Mathematical Considerations

5.1.1 Degenerate Cases

One shortcoming of the simulation is its approximation using constant acceleration. While this works for scenarios with generally ”smooth” accelerations, the approximation breaks down in scenarios with large fluctuations in acceleration. To address this, there are a couple possible methods:

1. Runge-Kutta(RK4): It is possible to use the 4th-order Runge-Kutta method to better numerically evaluate the integrals required to compute position and velocity from acceleration. However, this comes at the tradeoff of the program being computationally more intensive at each step.

2. Verlet Integration: This is a common integration method in simulation software. This involves using the previous timestep to calculate position and velocity. By doing so, large jerk can be considered, which results in better simulation results.

In future iterations of the simulation, there will be an option for the user to choose which numeric method (constant acceleration, Verlet, or RK4) should be used.

5.1.2 Collisions

Another shortcoming of the simulation is its inability to account for collisions. In future iterations of the simulation, collisions will be handled using `pygame`'s collision detection. The body will also have momentum attributes, which, on collision, will calculate the transfer of momentum between the colliding objects.

5.2 Program Considerations

5.2.1 GUI

At its current iteration, all parameters must be directly edited in the code. In future iterations of the simulation, the GUI should have sliders and buttons to provide easy access to the parameters. Future iterations may also display important vectors such as force, acceleration, and velocity to further visualize how each body affects the physical properties of the surrounding bodies.

5.2.2 Self-Stabilization

At its current iteration, the user must manually adjust stability parameters(`mass_stability_scale`, `distance_stability_scale`, `dt`) to account for extremely large numbers when simulating realistic planet masses and distances. In future iterations, the program will automatically calculate the required stability scalings before running the simulation to ensure a smooth animation.