**Overview**

This is a program that will simulate the solar system with N-body objects interacting through Newtonian gravity.  
This document describes the code format, modules, and classes written for the program, with implemented methods outlined. The code is written in Python.

Module layouts and internal and external dependencies follow:

**Trajectory File**

This output file uses VMD to visualize the simulation.

**Properties:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Notes** |
| simulation parameters | float | number of steps, time step, etc. |
| details of the particle | vector table | number of particles, labels, masses, starting positions, starting velocities |

**Format:**

The format of Trajectory File for VMD is the following,

where the header line is the specific number of points (e.g. 12), m is the number of steps in the trajectory, and s is the name of each point with (x,y,z) Cartesian coordinates.

12

Point = 1

s1 x11 y11 z11

s2 x21 y21 z21

.

.

.

s12 x121 y121 z121

12

Point = m

s1 x1m y1m z1m

s2 x2m y2m z2m

.

.

.

s12 x12m y12m z12m

. . .

**Particle3D Class**

A class that describes a particle-like object moving in 3D space.

**Properties:**

Properties to hold the object’s mass, label, position (pos), and velocity (vel).

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Notes** |
| m | float | mass of object |
| label | string | name of object |
| pos | numpy array | position of object |
| vel | numpy array | velocity of object |

**Initialisation:**

|  |  |
| --- | --- |
| **Arguments** | **Notes** |
| float m, string label, numpy pos, numpy vel | creates a dynamic particle with position and velocity, with initial values from the inputs in ParticleManyBody class |

**Methods:**

**def \_\_str\_\_(self)**

Return a string output format.

label + particle\_pos + particle\_vel + particle mass.

**def kinetic\_energy(self)**

Return kinetic energy as .

**def leap\_velocity(self, dt, force)**

First-order velocity update,

.

**def leap\_pos1st(self, dt)**

First-order position update,

.

**def leap\_pos2nd(self, dt, force)**

Second-order position update,

.

**Static Methods:**

**def create\_particle(file\_handle)**

Create a particle from file entry. The method read through the line of file

Return Particle3D(label, pos, vel, mass).

**def separation(particle1, particle2)**

Static method to return relative vector separation of two particles.

**ParticleManyBody Class**

This module contains the main program that simulates N-body systems for the solar system, with given initial conditions.

**Properties:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Notes** |
| n steps | int | number of steps |
| t steps | float | time of each step |
| T | float | total simulation time |
| force\_sun | numpy array | gravitational force acting on each object relative to sun |
| force\_moon | numpy array | gravitational force acting on moon relative to earth |
| distance\_sun | float | distance of orbit around sun |
| distance\_moon | float | distance of orbit around earth |

**Initialisation:**

|  |  |
| --- | --- |
| **Arguments** | **Notes** |
| int ns, float ts, float T, numpy force\_sun, numpy force\_moon, float distance\_sun, float distance\_moon | creates orbit for each object using input files and total time of the simulation |

**Methods:**

**def force\_grav(particle1, particle2, m1, m2)**

Method to return the gravitational force between two objects separated by center-to-center distance r, given by .

**def distance(particle 1, particle 2)**

Method to return distance between two objects.

**Static Methods:**

**def main(str[ ] argv)**

The main method goes through the two input files and one output trajectory file.

The simulation based on using velocity Verlet time integration algorithm which is a numerical method to integrate the equations of motion in Particle3D and here in order to find the position of particles at each time during the simulation.

Before running the simulation, we make sure the main method reads in an arbitrary number of objects (planets) and their initial positions and velocities from the first input file, also, reads the simulation parameters from the second input file. Then we set up a list of Particle3D objects to hold the planets.

As we get our initial data from an external database we can see that the initial conditions of a N-body system usually have a non-zero linear momentum, this might lead to drift of centre of mass, to avoid it we need to subtract the centre of mass velocity from all the initial velocities of the particles inside the simulation and change it the list of Particle3D objects:

We then use direct methods to solve the equations of motion for each of the 12 particles and store its result as a numpy arrays, these equations are given by Newton’s law of gravity. Using Newton’s third law we can make the program faster ( ). In our solar system simulation, the force acting on the *i*-th particle is given by:

Here is the external potential. For this simulation, we use Astronomical system of units. Therefore,.

Thus, what we are actually solving is a set of non-linear second order differential equations using the velocity Verlet algorithm.

In the main loop, we iterate the force vector acting on each particle and update the velocity and position vectors for the bodies at each time step. Also, at each time step we write the value of position vectors in the trajectory output file that can be visualised using VMD. We could see that the overleap of position vectors and they finish the full period of the orbits and run the simulation for approximate total time of 250 years (orbital period of Pluto is 248 years) and time steps 1/50000 of the total simulation time.

**References:**

[1] Project A – Astronomical N-body simulation, Computer Modelling course

[2] Exercise 2 Instructions, Computer Modelling course

**Team members:**

Austin Morris (s1728541)

Mohammadreza Aboutalebi (s1664598)

Fergus Davidson (1351641)