Design Document: Project A

Benjamin Cox & Ricardo Velasco

Due 2018-01-25

Contents

1 Overview

This program will simulate the Newtonian Solar System using the Velocity Verlet algorithm applied to classical mechanics.

This document contains descriptions of the modules and classes that are to be implemented in this program and gives descriptions of the algorithms and how they interact with each other.

All written/submitted code is in Python (Version 3.5+ with numpy). Error checking is to be carried out using Perl for inputs and comparisons.

Unless explicitly mentioned all units are standard SI units (kilograms, metres, joules etc.)

1.1 Internal Dependencies

Dependencies	Name	Dependants
`Python 3.5+, numpy`	`Particle3D.py`	`SolarSimulator.py, GravUtils.py`
`Particle3D.py`	`GravUtils.py`	`SolarSimulator.py`
`GravUtils.py, Particle3D.py`	`SolarSimulator.py`	-

2 `Particle3D.py` (Class)

Each instance of this class represents a point-mass particle in 3D space. We assume that no collisions occur between the particles and that they are not travelling at relativistic speeds.

2.1 Properties

Name	**Type**	**Notes**
label	string	Name of object
`mass`	float	Mass of object (kg)
`position`	3x1 numpy array (floats)	Position of object (m)
`velocity`	3x1 numpy array (floats)	Velocity of object (ms^{-2})
`prev_force`	3x1 numpy array (floats)	Previous force applied to object (N)
`current_force`	3x1 numpy array (floats)	Current force being applied to object (N)

2.2 Initialisation

Arguments	Notes
`string label, float mass, array position, array velocity`	Creates a particle with these properties as above.

2.3 Methods

2.3.1 `__str__`

Arguments : `(self)`

Returns a string containing the particles label and its current position in space in the format `label pos_x pos_y pos_z`.

Return: string `pos_str`

2.3.2 `kinetic_energy`

Arguments: `(self)`

Calculates and returns the classical kinetic energy of the particle using

$$E_k = \frac{1}{2}mv^2,$$

where E_k is kinetic energy of the particle, m is the particle mass and v is the particle velocity.

Return: float `E_k`

2.3.3 `step_velocity`

Arguments: `(self, 3x1 numpy array vector force, float timestep)`

This methods updates the velocity of the particle given a force and the interval it is acting over. This is achieved using

$$\mathbf{v}_{t+\delta t} = \mathbf{v}_t + \mathbf{F}_t \cdot \frac{\delta t}{m},$$

with $\mathbf{v}_t$ representing the velocity at time t, $\mathbf{F}_t$ representing the force applied, δt representing the timestep and m representing the particle mass.

Return: `void`

2.3.4 `first_order_posint`

Arguments: `(self, float timestep)`

This method performs first order time integration on the particle. This is done using the following formula:

$$\mathbf{p}_{t+\delta t} = \mathbf{p}_t + \mathbf{v}_t \cdot \delta t,$$

where $\mathbf{p}_t$ represents position at time t, $\mathbf{v}_t$ represents velocity at time t, and δt represents the timestep.

Return: `void`

2.3.5 `second_order_posint`

Arguments: `(self, 3x1 numpy array force, float timestep)`

This method performs second order time integration upon the particle. This is done using the following formula:

$$\mathbf{p}_{t+\delta t} = \mathbf{p}_t + \mathbf{v}_t \cdot \delta t + \delta t^2 \cdot \frac{\mathbf{F}_t}{2m},$$

where $\mathbf{p}_t$ is position at time t, $\mathbf{v}_t$ is velocity at time t, $\mathbf{F}_t$ is force at time t, m is the particle mass, and δt is the timestep.

Return: `void`

2.4 Static Methods

2.4.1 `make_from_file`

Arguments: `(filehandle filehandle)`

This method takes an open filehandle and parses the data in the associated file, creating a Particle3D out of it. The file is expected to be in the following format:

```
label
mass
position_x,position_y,position_z
velocity_x,velocity_y,velocity_z
```

Notice how the commas are not spaced from the data.

Return: Particle3D `Particle`

2.4.2 `separation`

Arguments: `Particle3D p1, Particle3D p2`

This method calculates the vector separation of `p1` and `p2` as follows. Let r_1 denote the position of `p1` and r_2 the position of `p2`. Then

$$\text{Separation} = r_1 - r_2.$$

It is important that the order that the positions are in is known as subtraction is not commutative.

Return: array `vec_sep`

3 `P3DGravUtils.py` (Module)

This code will contain all of the repeatedly called gravity related functions/methods. It exists to tidy up the program file, as it is neater and more 'pythonic' to have a file containing all the large methods. It also makes debugging a lot easier.

3.1 Methods

3.1.1 `grav_force`

Arguments: (`Particle3D p1, Particle3D p2, float G`)

This method returns the force due to gravity between two instances of `Particle3D`. This is done as follows. Let m_1 be the mass of `p1` and m_2 be the mass of `p2`. Let the separation between them be the vector **r**, and the magnitude of **r** be r. Then the force between them is as follows:

$$\mathbf{F} = \frac{G m_1 m_2}{r^3}\mathbf{r},$$

where **F** is the force due to gravity, G is the Gravitational Constant, and the other variables are defined as above.

Return: array `grav_force`

3.1.2 `grav_pot`

Arguments: (`Particle3D p1, Particle3D p2, float G`)

This method returns the potential due to gravity between two instances of `Particle3D`. This is done as follows. Let m_1 be the mass of `p1` and m_2 be the mass of `p2`. Let the separation between them be the vector **r**, and the magnitude of **r** be r. Then the energy potential is as follows:

$$E = -\frac{G m_1 m_2}{r},$$

where **E** is the potential due to gravity, G is the Gravitational Constant, and the other variables are defined as above.

Return: array `grav_pot`

3.1.3 drift_correct

Arguments: (list particle_list)

This method subtracts the centre of mass velocity from the velocity of all particles in the system. This means that we get a nicer looking simulation with less drift.

This is done by first calculating the momentum of the system using the formula

$$\mathbf{P} = \sum_i m_i \mathbf{v}_i,$$

with $\mathbf{P}$ being the system momentum, m_i being the mass of the i-th particle, and $\mathbf{v}_t$ being the velocity of the i-th particle.

We then calculate the velocity of the centre of mass using the formula

$$\mathbf{v}_{\text{CoM}} = \frac{1}{\sum_i m_i}\mathbf{P},$$

with $\mathbf{v}_{\text{CoM}}$ being the velocity of the centre of mass, and all other terms defined as above.

We adjust the velocity of each particle in the list by the negative of this to avoid drift.

Return: void

3.1.4 total_energy

Arguments: (list particle_list, G)

This method calculates the total energy of the system.

First it sums the kinetic energies of the particles.

Then it performs two nested for loops over the list of particles to calculate the gravitational potential between them.

Algorithm:

```
t_energy = 0
for i in particle_list:
    t_energy = t_energy + i.kinetic_energy()
for i in range(len(particle_list)):
    for j in range(i, len(particle_list)):
        t_energy = t_energy + grav_pot(particle_list[i], particle_list[j], G)
return t_energy
```

Return: float t_energy

3.1.5 `get_particles`

Arguments: (`particle_file`)

This method reads an arbitrary number of particles in from a file.

The file should be formatted as follows:

```
#lines that begin with # are comments
#particle 1 params
#label
string label
#mass
float mass
#position
(float p_x), (float p_y), (float p_z)
#velocity
(float v_x), (float v_y), (float v_z)
#particle 2 params
...
...
#arbitrarily many particles
```

The algorithm that fetches these is complex and is heavily commented in the code.

Return: list `particle_list`

3.1.6 `get_params`

Arguments: (`params_file`)

This method parses an input file that contains the simulation parameters (G, dt, numsteps, init_time).

Return: list `param_list`

3.1.7 `step_time`

Arguments: (`particle_list, dt, time, output_file`)

This method steps the simulation forwards in time using the Velocity Verlet time integration algorithm.

First it calculates the new forces on each particle due to every other particle. It then calculates the next position of each particle in accordance with the algorithm. It then steps the particles to that position.

Finally it writes a block to the output file using the `__str__` method in `Particle3D.py`.

It does not step the time forwards, that is done in the main program.

Return: `void`

3.1.8 `conic_fitter`

Arguments: (`particle_list, moon_index, prev_pos`)

This method fits a plane to a sample of five points on each bodies orbit. It then projects these points onto the plane and fits an ellipse to them.

From this ellipse the orbital parameters are calculated (periapsis, apoapsis, period.)

We will use `scipy.optimize` for the fitting and optimization (way better than I could write.)

Return: `void`

3.1.9 `conic_to_min`

Arguments: (`theta, f`)

This method takes in these arguments and gives the point on the ellipse corresponding to the equation:

$$r(\theta; a, e) = \frac{a(1-e^2)}{1-e\cdot\cos\theta},$$

Where θ is the angle, a is the semi-major axis, and e is the eccentricity.

`f` contains a at index 0, and e at index 1.

Return: point on ellipse in polar co-ordinates.

3.1.10 `residuals`

Arguments: (`f, p, theta`)

This method calculates the residuals of the ellipse to aid in fitting.

Return: residual of fitted ellipse compared to true data.

4 `SolarSimulator.py`

This contains the main body of simulator code. It mostly just brings together methods from the above modules.

4.1 Methods

4.1.1 `main`

Arguments: (`void`)

This is the only method in `SolarSimulator`.

It first parses the command line arguments (and tells the user how to do it if they don't know.)

It opens the output files for writing.

Then it sets up the lists and begins the time integration. At each step of the integration the position and number of each particle is written to the output in xyz format.

Every 20 steps it will updates the approximation for the ellipse and thus the apoapsis, periapsis, and period (with considerations for the moon).

However if this is unable to be implemented for whatever reason we will use a climbing algorithm to determine the periapsis, apoapsis, and period.

After all the simulation is done it will close the output and input files, perform cleanup, and close.

Maybe we will implement a progress bar.