***Solar Report***

# Introduction

This project aims to simulate the Solar System in 2D representation. Solar System represents a many body problem meaning that the system involves interactions between more than two bodies therefore it is necessary to use approximation techniques or model systems. I’ll be using Beeman algorithm of Numerical Integration to calculate the position and velocity of a body/planet after certain time step. Once finished writing the whole code I’ll be doing some experiments with the simulations. There are six experiments in total and in some way they all are aimed to check how correct and reliable the simulation is, does your simulation follows the laws of physics and finally how big is the uncertainty in measurements. This projects also aims to predict the launch conditions for satellites to successfully reach planets and if possible, come back to the planet from where it was launched.

# Diagram Description automatically generatedMethods

This is the UML class diagram of my project. SolarSystem, SolarBody, Planet and Sun classes are contained in one module called solar\_system.

### SolarBody:

SolarBody is class that represents any object in a Solar System. It has all the attributes and functions an object in a Solar System would have such as name, mass, position, velocity, orbital radius, etc. and functions such as get kinetic energy, get potential energy, has year passed, etc. “has\_year\_passed()” function checks whether the body/planet has completed its orbit.

update function updated the values using Beeman Scheme.

### Planet:

Planet class is sub-class of SolarBody because a planet is a body in space but with some extra information. Planet has an extra attribute “time\_passes\_in\_year” (not a good name for the purpose), this basically stores how much time has passed before that planet complete its full year. So, if the planet has completed half of its orbit, time\_passes\_in\_year should be 0.5(this is not guaranteed value. It could be less or more than 5) and because it takes 6 out of 12 months to complete half orbit. Planet class also has two functions calculate\_initial\_vel and calculate\_initial\_acc; Initial velocity is calculated using the Beeman method and Initial acceleration is calculated using the Newton Law of Gravitation.

### Sun:

I made Sun a sub-class of SolarBody even though it does not have any function of its own because having Sun class with some pre-defined values made easier to initialise Sun. I only need to call the give it the Solar System Sun is in and mass of sun and its done; a Sun is created. Its still arguable that it’s less confusing or just better to have sun as SolarBody object rather than Sun object but I before starting the actual project I was creating kind of a prototype and I was searching on web and saw a example where they had sun as Sun object so I stuck with that.

Now I will start explaining my classes and why I choose to design my project this way.

## SolarSystem:

### Variables:

* **gravitational\_const**: this is a class variable and stores the value of the Gravitational Constant and I require in further calculations.
* **furthest\_revolving\_body:** Used to store the planet that is furthest from the Sun. Reason why I am string this is because at some places in my code I need the information about the furthest planet/body that is revolving around the Sun. I can’t just use the last body in solar\_system. bodies because when a satellite is added, it’s not necessary that the satellite will be the furthest body from Sun but if it is the furthest “**furthest\_revolving\_body”** will be equal the satellite.
* **bodies:** this variable is being used to store all bodies that are in a solar system. This makes it easier to iterate and access any/all bodies in the solar system.
* **fig, axis:** These variable holds the simulation itself.
* **timestep:** this represents in each iteration how far in time I will calculate the data I need to keep the simulation running.
* **patches:** This holds all circle images and their position on axis for all bodies in bodies.
* **time\_passed:** Keeps track of much time has passed since the simulation started.
* **ke\_values:** Stores list total kinetic energy of all bodies every timestep.
* **pe\_values:** Stores list total potential energy of all bodies every timestep.
* **total\_e\_values:** Stores list total energy of all bodies every timestep.
* **planetary\_alignments:** Stores the list of time whenall the bodies were aligned.

### Functions

* **save\_orbitial\_periods:** This function takes in list of orbital periods for all planets same store them in a variable.
* **create\_patch:** create new circle image for a body.
* **add\_body:** adds a new body in the solar system.
* **calculate\_energies:** calculate and returns the total kinetic energy, potential energy and total energy (kinetic energy + potential energy) of system
* **plot\_energies:** plots the Energy against time graph for Kinetic, Potential and Total Energy
* **has\_sat\_reached\_mars\_earth:** checks whether satellite reached mars and came back to earth.
* **animate, animate\_only\_sub\_g:** calculates the values needed to run the simulation
* **run\_simulation:** handles the simulation such as plotting, showing, updating, etc.
* **compare\_orbital\_periods:** compares the simulated orbital period of all planets with original period and return uncertainty
* **check\_planetery\_alignment:** check whether planets are aligned.