Final Project Report

Finding a missing planet from its gravitational effect on surrounding bodies

McGill University
PHYS 512: Computational Physics with Applications

Juan Diego Uribe Carrillo (260987692), Kamar Razek (261051522)

December 7, 2023

Abstract

Objective To create a model to simulate gravitational dynamics within a stellar system comprising multiple celestial bodies and use numerical methods to find 'missing' bodies within the system.

Methodology By defining a function that calculates the error in the trajectories of the bodies in the system and uses an adaptive step size within a time evolution loop to find the mass of the object minimizing the combined error in the orbits.

Results Our code successfully simulates the interactions between a multi-body stellar system and our function can successfully find the mass of the missing body that minimizes the combined error of the orbits between our simulated data and our new model.

1 Introduction

The canvas of space, adorned with celestial wonders, has always beckoned humanity's curiosity and fascination. Within this vast expanse, the detection and understanding of celestial bodies beyond our known horizons stand as pivotal milestones in our cosmic exploration. One such remarkable achievement, rooted in the mastery of gravitational principles, is the discovery of Neptune — a celestial revelation that underscored the profound impact of gravity in unveiling hidden cosmic entities.

In the early 19th century, astronomers confronted an enigmatic puzzle — discrepancies in the orbit of Uranus defied Newtonian gravitational predictions. These irregularities in Uranus' trajectory hinted at the gravitational tug of an unseen celestial body exerting its influence from the depths of space. Leveraging the predictive power of Newton's law of universal gravitation, mathematicians Urbain Le Verrier and John Couch Adams independently embarked on groundbreaking calculations to deduce the existence and position of an elusive eighth planet in our stellar system.

Employing meticulous mathematical analyses based on perturbations in Uranus' orbit, Le Verrier in France and Adams in England computed the hypothetical location where a massive, yet unseen, celestial body could exert gravitational influences to explain the observed discrepancies. Their predictions, remarkably aligned despite working independently, culminated in the awe-inspiring discovery of Neptune — the eighth planet from the Sun — in 1846, validating the power of gravitational theory in celestial detective work.

The discovery of Neptune stands as a testament to the influential role of gravitational principles in probing the cosmos for hidden celestial bodies. This landmark achievement not only expanded our understanding of our own solar system but also showcased the predictive prowess of gravitational theory, serving as a beacon for astronomers seeking to unravel the mysteries of the universe.

This report aims to delve into the gravitational methods instrumental in the discovery of Neptune, elucidating the intricate interplay between orbital dynamics, mathematical predictions, and observational astronomy. Through the lens of this historic discovery, we explore the application of gravitational principles in detecting celestial bodies, offering insights into the methods that continue to shape our understanding of the cosmic realm.

2 Problem Description

Our code tackles the intricate challenge of simulating and analyzing the gravitational interactions within a stellar system composed of multiple celestial bodies. The dynamic interplay between gravitational forces among these cosmic entities often poses a complex puzzle for astronomers seeking to comprehend the celestial dance unfolding in the vast expanse of space.

In the quest to model and understand the behaviour of celestial bodies, gravitational effects play a pivotal role, dictating the trajectories, interactions, and eventual fates of these cosmic entities. However, simulating such intricate gravitational interactions, especially within multi-body systems, demands sophisticated computational models capable of capturing the nuances of these gravitational influences.

The significance of our code lies in its ability to simulate the three-dimensional gravitational dynamics of a stellar system comprising multiple planets, providing a computational framework to investigate how gravitational interactions shape the orbits and movements of these cosmic bodies. By leveraging fundamental principles of gravitational physics and numerical methods, our code facilitates the exploration of celestial mechanics, shedding light on the intricate interplays that govern the motions of planets, moons, and other celestial objects.

Through its capacity to simulate and elucidate the gravitational complexities of multi-body systems, our code stands as a valuable tool in for astronomical investigations, facilitating the exploration and comprehension of the gravitational interaction of multi-body systems.

3 Code Description

We were originally inspired by the 3D simulation of a stellar system found in *The Python Coding Book*[1]. At its core, the code implements an object-oriented approach, defining classes for celestial bodies such as planets, stars, and other cosmic entities. Each celestial body object encapsulates essential attributes including mass, position, velocity, and methods to calculate gravitational interactions.

Since we are working in three dimensions, the calculations of gravitational interactions require both the magnitude and direction of the objects' velocities as well as their positions in our 3-D space. Meaning we will be performing vector calculations. To make working with vectors easier in the code, we started by defining the `Vector` class exactly as in our source material [1], which is fundamental to our simulation and represents spatial coordinates. This offers functionalities for vector addition, subtraction, normalization, magnitude calculations, and scalar operations, providing the backbone for handling spatial positions and velocities of celestial bodies. The complete code definition for the `Vector` class can be found in Appendix A.

The `SolarSystem` class orchestrates the simulation, managing the collection of celestial bodies and orchestrating their interactions. It oversees the updating of positions and velocities based on gravitational forces, taking care of the progression of the simulated cosmic system through time:

```
class SolarSystem:

...

def calculate_all_body_interactions(self):
    bodies_copy = self.bodies.copy()
```

```
        for idx, first in enumerate(bodies_copy):
            for second in bodies_copy[idx + 1:]:
                first.accelerate_due_to_gravity(second)
```

The `calculate_all_body_interactions()` method goes through all the bodies in the stellar system. Each body interacts with every other body in the stellar system

Furthermore, subclasses such as `Sun`, `Planet`, and others inherit from the `SolarSystemBody` class, defining specific attributes and behaviours for distinct celestial entities. These subclasses allow for the creation of a diverse array of celestial bodies within the simulation, each characterized by unique properties and gravitational influences:

```
    class SolarSystemBody:
        min_display_size = 10
        display_log_base = 1.3

        ...

        def accelerate_due_to_gravity(self, other):
            distance = Vector(*other.position) - Vector(*self.position)
            distance_mag = distance.get_magnitude()
            force_mag = self.mass*other.mass/(distance_mag**2)
            force = distance.normalize()*force_mag
            reverse = 1
            for body in self, other:
                acceleration = force/body.mass
                body.velocity += acceleration*reverse
                reverse = -1
```

We modified the classes as shown in the source material to be able to store the trajectories of each celestial body. The complete definitions of the `SolarSystem` and `SolarSystemBody` classes, as well as all the other subclasses, can be found in Appendix B.

To calculate the error and minimize it, we defined the functions `error_calcualtion` and `find_missing_planet_mass` inside the `orbit_fit.py` file. The error is calculated as the mean square of the difference between the data trajectories and the model trajectories with a ‘guess’ mass:

```
    def error_calculation(data_trajectory, model_trajectory):
        data_planet1_trajectory = np.array([data_trajectory[0]])
        data_planet2_trajectory = np.array([data_trajectory[1]])
        data_planet3_trajectory = np.array([data_trajectory[2]])
        data_planet4_trajectory = np.array([data_trajectory[3]])

        model_planet1_trajectory = np.array([model_trajectory[0]])
        model_planet2_trajectory = np.array([model_trajectory[1]])
        model_planet3_trajectory = np.array([model_trajectory[2]])
        model_planet4_trajectory = np.array([model_trajectory[3]])

        #Calculate diff
        diff_planet1_trajectory = abs(data_planet1_trajectory - model_planet1_trajectory)
        diff_planet2_trajectory = abs(data_planet2_trajectory - model_planet2_trajectory)
        diff_planet3_trajectory = abs(data_planet3_trajectory - model_planet3_trajectory)
        diff_planet4_trajectory = abs(data_planet4_trajectory - model_planet4_trajectory)

        #Calculate square error
        err_planet1_trajectory = np.sqrt(diff_planet1_trajectory[0][:, 0]**2
        + diff_planet1_trajectory[0][:, 1]**2 + diff_planet1_trajectory[0][:, 2]**2)
        err_planet2_trajectory = np.sqrt(diff_planet2_trajectory[0][:, 0]**2
        + diff_planet2_trajectory[0][:, 1]**2 + diff_planet2_trajectory[0][:, 2]**2)
        err_planet3_trajectory = np.sqrt(diff_planet3_trajectory[0][:, 0]**2
        + diff_planet3_trajectory[0][:, 1]**2 + diff_planet3_trajectory[0][:, 2]**2)
        err_planet4_trajectory = np.sqrt(diff_planet4_trajectory[0][:, 0]**2
        + diff_planet4_trajectory[0][:, 1]**2 + diff_planet4_trajectory[0][:, 2]**2)

        avg_error = np.mean([
        np.mean(err_planet1_trajectory),
        np.mean(err_planet2_trajectory),
        np.mean(err_planet3_trajectory),
        np.mean(err_planet4_trajectory)
        ])
        return avg_error
```

Then, the error is minimized by varying the 'guess' mass and checking if the error increases or decreases. Somewhat similar to the adaptive Runge-Kutta integrator discussed in lectures:

```
def find_missing_planet_mass(accuracy, initial_mass_guess, data_trajectory):
    errors = []
    count = 0 #initialize count variable
    Mass = initial_mass_guess
    error = accuracy+1
    dM = 0.5

    while error>=accuracy:

        ...

        if errors == []:
            Mass += dM
        else:
            # Scale dM based on the distance of the error from zero
            dM *= max(0.1, min(1.0, abs(new_error / accuracy)))
            # Update Mass based on the direction of the error change
            if new_error < error:
                Mass += dM
            else:
                Mass -= dM

        error = new_error
        errors.append(error)

return Mass, errors, count
```

Overall, our code serves as a tool enabling the simulation and analysis of gravitational interactions within multi-body celestial systems. Its modular structure, combined with Python's flexibility, empowers users to explore and understand the intricate gravitational dynamics governing celestial bodies in the cosmos.

4 Code Tests

We tested the `vector` class by performing some simple vector calculations:

```
test1 = Vector(3, 5, 9)
print(test1)
print(repr(test1))
Out:
3i + 5j + 9k
Vector(3, 5, 9)


test2 = Vector(3, 5, 9) + Vector(1, -3, 2)
print(test2)
Out:
4i + 2j + 11k

test3 = Vector(3, 5, 9)*Vector(1, -3, 2)
print(test3)
Out: 6
```

To test that the time evolution of the multi-body system and the gravitational interactions between them work properly, we tested two types of systems: a simple system containing a single star and two planets (`simple_solar_system.py`) and a binary star system containing two planets as well (`binary_star_system.py`).

To test our problem as stated in the previous section, we compared the mean squared difference between a strictly 4-planet system, the expectation, and a 5-planet system with a "hidden" planet, the data. Since the initial velocities and positions of the first four planets remained the same in both models, we expected the error to increase as time progressed in the simulations. Fig.1 shows

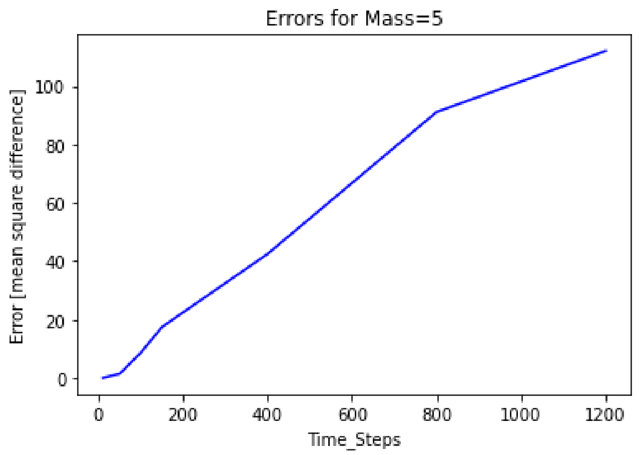

Figure 1: Code test displaying increasing mean square difference for increasing time steps in simulated orbits

the expected behaviour, confirming our code works.

5 Results and Analysis

For our exploration, we set the mass of the star at $m_* = 10,000$ and the first four planets at $m_1 = 10$, $m_2 = 20$, $m_3 = 30$, $m_4 = 15$. Furthermore, the mass of the 'missing' planet was $m_x = 10$.

Using an adaptive step size (dM), we calculated the mean square difference between data and model trajectories for different masses. Fig. 2 illustrates how the error is at a minimum around the correct mass for the missing planet, with some other local minima at around 6 and 15.

Fig. 3 shows how long it took for the code to find the correct mass, that is, how many time step iterations it took to minimize the error below the specified accuracy threshold of 10^{-6}.

Given the nature of the problem, there existed a local minimum of error values that skewed many attempted runs. For example, at Mass=7, Error=0.234, but at Mass=8, Error=0.349, this would inhibit movement towards the expected Mass value of 10. Due to this, the success of the final code is unfortunately very dependent on the initial conditions of the mass and mass step size. Still, with an initial mass guess of 8, and dM=0.5, the code determines that the missing planet's mass: is 10.05 in 5 iterations.

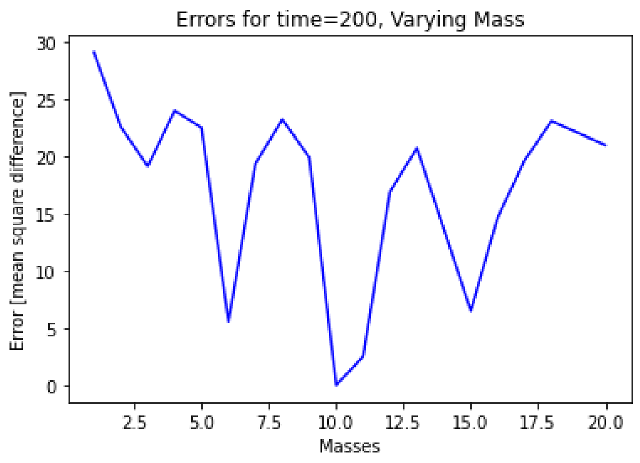

Figure 2: Mean square difference varying mass+=1 with orbit simulation time=200

6 Discussion and Further Improvements

Given the nature of our experiment, despite having obtained a correct result -the code was able to find the mass of the 'missing planet' by minimizing the combined error of the orbits to within the desired accuracy threshold- it could have been interesting to analyze the error in the trajectories for each planet. This could provide a more accurate representation of the orbit deviation caused by the missing planet and allow for a reduced error in the found mass. Also, the star and planet masses, as well as the initial positions and velocities of the planets were carefully chosen to prevent orbital instabilities (i.e. planet's being expelled from our field of view due to a "gravitational slingshot" effect.) over the time period of the simulation. A longer timeframe could have still revealed these instabilities. This code could have also been used to study the factors that affect orbital stability.

However, to further enhance the code's capabilities and refine its simulations, several avenues for improvement and expansion present themselves. For example, incorporating additional factors such as collision effects, relativistic effects, gravitational waves, or the influence of external celestial bodies could expand the code's scope. Considering these factors would provide a more comprehensive simulation, particularly in scenarios involving extreme gravitational environments or interactions with distant cosmic objects. Also, introducing advanced visualization techniques or analysis tools would augment the code's utility. Implementing functionalities to generate plots, histograms, or phase-space diagrams of celestial trajectories could facilitate deeper insights into the dynamics of the simulated stellar system. Furthermore, optimizing the code for performance by leveraging parallel computing techniques or optimizing computational algorithms could significantly improve simulation efficiency. This

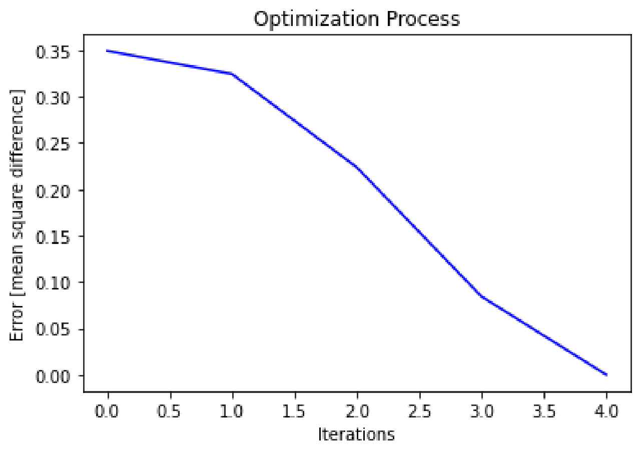

Figure 3: Optimization process t=100

enhancement would enable the handling of larger-scale simulations or longer timeframes without compromising computational speed.

7 Conclusion

In conclusion, our code successfully simulates the interactions between a multi-body stellar system and our function can successfully find the mass of the missing body that minimizes the combined error of the orbits between our simulated data and our new model. Although, the code can be further expanded upon for a more in-depth analysis.

8 Contribution Statement

Juan Diego contributed by finding the reference material to implement the definition of the different classes and modifying them to record and extract the trajectories and other properties. Juan Diego also contributed to the redaction of almost all sections of this report.

Kamar contributed to the development of the numerical integrator function that minimized the error and found the mass of the missing planet. Kamar also contributed to the redaction of the results section of this report and the visualization of the results by creating all plots found in this report.

References

[1] S. Gruppetta, "The python coding book," https://thepythoncodingbook.com/2021/12/11/simulating-3d-solar-system-python-matplotlib/, December 2021.

A Defining the `Vector` class

```python
import math

class Vector:
    def __init__(self, x:float=0, y:float=0, z:float=0):
        self.x = x
        self.y = y
        self.z = z

    def __repr__(self):
        return f"Vector({self.x}, {self.y}, {self.z})"

    def __str__(self):
        return f"{self.x}i + {self.y}j + {self.z}k"

    def __getitem__(self, item):
        if item == 0:
            return self.x
        elif item == 1:
            return self.y
        elif item == 2:
            return self.z
        else:
            raise IndexError("There are only three elements in the vector")

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y, self.z + other.z)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y, self.z - other.z)

    def __mul__(self, other):
        if isinstance(other, Vector):  # Vector dot product
            return (self.x*other.x + self.y*other.y + self.z*other.z)
        elif isinstance(other, (int, float)):  # Scalar multiplication
            return Vector(self.x*other, self.y*other, self.z*other)
        else:
            raise TypeError("operand must be Vector, int, or float")

    def __truediv__(self, other):
        if isinstance(other, (int, float)):
            return Vector(self.x/other, self.y/other, self.z/other)
        else:
            raise TypeError("operand must be int or float")
    def get_magnitude(self):
        return math.sqrt(self.x**2 + self.y**2 + self.z**2)

    def normalize(self):
        magnitude = self.get_magnitude()
        return Vector(self.x/magnitude, self.y/magnitude, self.z/magnitude)
```

B Defining the object classes

```
import math
import matplotlib.pyplot as plt
from matplotlib import colormaps
from vectors import Vector

class SolarSystem:
    def __init__(self, size, projection_2d=False):
        self.size = size
        self.projection_2d = projection_2d #2d projection on the xy plane
        self.bodies = []

        self.fig, self.ax = plt.subplots(1, 1, subplot_kw={"projection": "3d"},
            figsize=(self.size / 50, self.size / 50))
        self.fig.tight_layout()
        # self.ax.view_init(0, 0) #This changes the perspective of the graph
        if self.projection_2d:
            self.ax.view_init(10, 0)
        else:
            self.ax.view_init(0, 0)

    def add_body(self, body):
        self.bodies.append(body)

    def update_all(self):
        self.bodies.sort(key=lambda item: item.position[0])
        for body in self.bodies:
            body.move()
            body.draw()

    def draw_all(self):
        self.ax.set_xlim((-self.size / 2, self.size / 2))
        self.ax.set_ylim((-self.size / 2, self.size / 2))
        self.ax.set_zlim((-self.size / 2, self.size / 2))
        if self.projection_2d:
            self.ax.xaxis.set_ticklabels([])
            self.ax.yaxis.set_ticklabels([])
            self.ax.zaxis.set_ticklabels([])
        else:
            self.ax.axis(False) #Removes grid and axis if 2d projection is off
        plt.pause(0.001)
        self.ax.clear()

    def calculate_all_body_interactions(self):
        bodies_copy = self.bodies.copy()
        for idx, first in enumerate(bodies_copy):
            for second in bodies_copy[idx + 1:]:
                first.accelerate_due_to_gravity(second)


class SolarSystemBody:
    min_display_size = 10
    display_log_base = 1.3

    def __init__(self, solar_system, mass, position=(0, 0, 0), velocity=(0, 0, 0)):
        self.solar_system = solar_system
        self.mass = mass
        self.position = position
        self.velocity = Vector(*velocity)
        self.display_size = max(
        math.log(self.mass, self.display_log_base),
        self.min_display_size
        )
        self.colour = "black"
        self.solar_system.add_body(self)

    def move(self):
        self.position = (
            self.position[0] + self.velocity[0],
            self.position[1] + self.velocity[1],
            self.position[2] + self.velocity[2],
        )

    def draw(self):
        self.solar_system.ax.plot(
            *self.position,
            marker="o",
            markersize=self.display_size + self.position[0] / 30,
            color=self.colour
        )
        if self.solar_system.projection_2d: #second plot is added when projection_2d is
            self.solar_system.ax.plot(
                self.position[0],
                self.position[1],
                -self.solar_system.size / 2,
                marker="o",
                markersize=self.display_size / 2,
                color=(.5, .5, .5),
            )

    def accelerate_due_to_gravity(self, other):
        distance = Vector(*other.position) - Vector(*self.position)
        distance_mag = distance.get_magnitude()
        force_mag = self.mass * other.mass / (distance_mag ** 2)
        force = distance.normalize() * force_mag
        reverse = 1
        for body in self, other:
            acceleration = force / body.mass
            body.velocity += acceleration * reverse
            reverse = -1


class Sun(SolarSystemBody):
    def __init__(
            self,
            solar_system,
            mass=10_000,
            position=(0, 0, 0),
            velocity=(0, 0, 0),
    ):
        super(Sun, self).__init__(solar_system, mass, position, velocity)
        self.colour = "yellow"


class Planet(SolarSystemBody):
    color_map = colormaps.get_cmap('tab20')  # Choose a colormap: 'vidiris', 'plasma'

    def __init__(
        self,
        solar_system,
        mass=10,
        position=(0, 0, 0),
        velocity=(0, 0, 0),
    ):
        super(Planet, self).__init__(solar_system, mass, position, velocity)
        self.colour = self.color_map(len(solar_system.bodies))  # Assign a color
        self.hidden = False  #Flag indicates if hidden
        self.trajectory = []  #Empty list to store trajectory
```

```
def move(self):
    super().move()  # Call the parent class's move method
    self.trajectory.append(self.position)  # Store the current position in the traje

def draw(self):
    if not self.hidden:  # Check if the planet is hidden
        self.solar_system.ax.plot(
            *self.position,
            marker="o",
            markersize=self.display_size + self.position[0] / 30,
            color=self.colour
        )
        if self.solar_system.projection_2d:
            self.solar_system.ax.plot(
                self.position[0],
                self.position[1],
                -self.solar_system.size / 2,
                marker="o",
                markersize=self.display_size / 2,
                color=(.5, .5, .5),
            )
```