BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

VISUALIZATION AND SIMULATION METHODS OF SOLAR SYSTEM BODIES

METODY VIZUALIZACE A SIMULACE TĚLES SLUNEČNÍ SOUSTAVY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ANDREJ HÝROŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ NOVÁK**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148248

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Hýroš Andrej**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Visualization and simulation methods of Solar System bodies**
Category:	Modelling and Simulation
Academic year:	2022/23

Assignment:

1. Study the basic principles of orbital mechanics (two-body problem, N-body problem, Kepler's laws of planetary motion).
2. Extract highly accurate ephemeridal data of chosen objects in the Solar System.
3. Create a tool with Graphical User Interface (GUI) enabling visualization of objects orbital motion.
4. Perform a motion simulation of chosen objects modelled as two-body problem and N-body problem.
5. Evaluate the achieved results and discuss future development of the project.

Literature:

- CURTIS, Howard D. *Orbital mechanics for engineering students*. Amsterdam ; Boston: Elsevier Butterworth-Heinemann, 2005, xv, 673 s. : il. ; 26 cm. ISBN 0-7506-6169-0.

Requirements for the semestral defence:
1. and 2. assignment points.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Novák Jiří, Ing.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	23.1.2023

Abstract

The aim of this thesis is to create program for simulating and visualizing the motion of n bodies according to Newton's laws. The input consists of a list of bodies and their initial conditions. After completing the simulation, the motion of individual objects of the entire system is animated in application with graphical user interface, and a file containing positions of simulated system is saved. During the simulation, first-order differential equations are solved. Various numerical integration algorithms were implemented in the program, which allows the user to choose a suitable algorithm for their simulation needs and achieve optimal simulation results. The program has been validated by comparing its output with data from NASA.

Abstrakt

Táto práca sa zaoberá vývojom programu na simuláciu a vizualizáciu pohybu n telies podľa Newtonových zákonov. Vstupom je zoznam telies a ich počiatočné podmienky. Po dokončení simulácie je vzájomný pohyb telies celého systému animovaný v programe s grafickým užívateľským rozhraním a je uložený súbor s priebehom simulácie. Počas simulácie sú riešené diferenciálne rovnice prvého rádu. V práci boli implementované viaceré algoritmy numerickej integrácie, ktoré umožňujú používateľovi zvoliť si vhodný algoritmus pre potreby jeho simulácie a dosiahnuť tak optimálny priebeh simulácie. Program bol prehlásený ako validný po porovnaní výstupu programu s dátami od NASA.

Keywords

simulation, visualization, n-body problem, orbital mechanics, Newton's Laws, Kepler's Laws, Solar system, numerical methods

Kľúčové slová

simulácia, vizualizácia, problém n telies, orbitálna mechanika, Newtonove pohybové zákony, Keplerove zákony, Slnečná sústava, numerické metódy

Reference

HÝROŠ, Andrej. *Visualization and simulation methods of Solar System bodies.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Novák

Rozšírený abstrakt

Táto práca sa zaoberá vývojom programu na simuláciu a vizualizáciu pohybu n telies podľa Newtonových zákonov. Je dôležité vedieť predpovedať pohyby planét v našej slnečnej sústave, pretože to umožňuje lepšie porozumieť prírode a vývoju vesmíru. Navyše, poznanie týchto pohybov nám umožňuje plánovať vesmírne misie a navigáciu vo vesmíre. Schopnosť predpovedať pohyby planét nám pomáha v oblasti astronómie a astrofyziky, napríklad pri objavovaní nových exoplanét a lepšom porozumení týmto svetom. Okrem toho, skúmanie pohybov planét a ostatných telies v slnečnej sústave pomáha aj v pochopení histórie nášho slnečného systému a jeho vzniku. Pohyby planét a nebeských telies sa všeobecne riadia Newtonovým zákonom univerzálnej gravitácie. Tento zákon hovorí, že každé hmotné teleso v priestore gravitačne ovplyvňuje pohyby iných telies. Sila pôsobiaca medzi dvoma telesami je priamo úmerná ich hmotnostiam a nepriamo úmerná druhej mocnine vzdialenosti medzi nimi. V dôsledku týchto zákonov sa planéty pohybujú okolo Slnka po eliptických dráhach. Pohyby planét možno predpovedať pomocou matematických modelov založených na zákonoch fyziky a matematiky. Základom pre predpovedanie pohybu planét je znalosť ich hmotností a počiatočných podmienok, teda ich pozície a rýchlosti v určitom čase. Použitím Newtonových pohybových zákonov a zákonu gravitácie sa potom vypočíta sila pôsobiaca na planétu a jej dráha v určitom čase. Tieto výpočty môžu byť veľmi zložité a výpočetne náročné, pretože zahrňujú mnoho výpočtov interakcií medzi mnohými telesami. Problém predpovedania pohybu telies radíme do dvoch kategórií: problém dvoch telies a problém n telies. Problém dvoch telies sa týka pohybu dvoch teles v priestore, ktoré gravitčne interagujú. Pri riešení tohto problému sa zanedbávajú akékoľvek iné sily a predpokladá sa, že vzájomná príťažlivosť týchto dvoch telies je jedinou silou, ktorá na ne pôsobí. Je to ale idealizovaný model a pre praktické aplikácie nie veľmi použitelný. Toto rieši problém n telies, ktorý berie do úvahy vplyvy všetkých telies v modelovanom systéme. Riešenie tohto problému je zložitejšie a vyžaduje si použitie numerických metód na riešenie diferenciálnych rovníc, ktoré opisujú pohyb všetkých telies v modelovanom systéme. V rámci tejto práce bol vypracovaný program, ktorý je schopný riešiť obidva problémy a predpovedať pohyby jednotlivých objektov modelovaného systému v čase. Pre riešenie diferenciálnych rovníc boli implementované tri integračné metódy, a to Eulerova, Runge-Kutta a Adams-Bashfort. Užívateľ si tak môže zvoliť vhodnú metódu pre jeho potreby. Ak potrebuje simulovať priblíženie asteroidu s presnosťou na stovky kilometrov, použije pre-snejšiu, ale výpočtovo omnoho náročnejšiu metódu. V prípade simulácie, ktorá nevyžaduje vysokú presnosť môže siahnuť po rýchlejšom algoritme. Súčasťou práce bola aj extrakcia vysoko presných efemeridov (tabuľky pozícií a rýchlostí nebeských telies) od americkej agentúry NASA, ktoré slúžia na validáciu implementovaného modelu. Ďalej bol vytvorený program s grafickým užívateľským rozhraním, ktorý umožní zadávať počiatočné podmienky pre simuláciu a následne zobrazí animáciu vykonanej simulácie. Systém a jeho počiatočné podmienky sú definované v súboroch typu JSON, ktoré je možné importovať do aplikácie. Výstupom aplikácie je mimo animácie aj textový súbor typu csv, ktorý je štruktúrou podobný efemeridom od NASA. Počas testovania boli výstupné dáta validované s pomocou existujúcich efemeridov a otestované boli aj doby trvania simulácie pri rozličných numerických integračných metódach a rozličnom počte telies v simulovanom systéme. V rámci experimenotvania s hotovým programom bolo simulovaných niekoľko scenárov, ako napríklad blízke priblíženie asteroidu Apophis k Zemi v roku 2029, alebo hypotetický prelet červeného trpaslíka našou sústavou.

Visualization and simulation methods of Solar System bodies

Declaration

Prehlasujem, že som bakalársku prácu vypracoval samostatne pod vedením pána Ing. Jiřího Nováka. Uviedol som všetky zdroje a literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

. . . .
Andrej Hýroš
May 9, 2023

Acknowledgements

I would like to thank my supervisor Ing. Novák for assistance, patience and always pointing me in the right direction.

Contents

Chapter 1

Introduction

Aim of this thesis is to provide a tool for simulation and visualization of various celestial bodies in our Solar system, including planets, comets, asteroids and other objects. The problem of predicting motion of individual objects in a group that are gravitationally influencing each other is known as the n-body problem. This problem has been known since the ages of Newton, when he realized that not only initial velocity and position of object must be known to predict its motion, but also the gravitational influence of nearby bodies must be taken into account. Despite this problem being known for hundreds of years, no practical analytical solution was ever found for more than two bodies. However, numerical integration methods can be used to get approximate trajectories of objects over time. The simulation of such systems has been a fundamental aspect of scientific research for many years. These simulations allow scientists to study and predict the behavior of complex systems that are difficult or impossible to observe in real life. The simulation algorithms used in created application include Euler, Runge-Kutta and Adams-Bashforth. The application is written in Python and takes a file specifying the simulated objects as input. The output of the application is a document containing simulated data and an animation of the system in a 3D plot. The application provides the user with interactive features, including the ability to zoom inside the animated system and rotate the plot to view the system from different angles. Chapter 2 introduces the basics of orbital mechanics to the reader, which are necessary to understand the n-body problem. Chapter 3 takes a closer look on n-body problem and how it can be solved. Specifics of created application are described in chapter 4.

Chapter 2

Fundamentals of orbital mechanics

Orbital mechanics is a field of study that deals with the motion of objects in space under the influence of gravity. It is a fundamental concept in space exploration and is used to predict the motion of planets, satellites, and other celestial objects. It is also necessary for planning of any space mission. This chapter will provide an overview of some of the key concepts and equations used in orbital mechanics. We will start by introducing the basic principles of Newtonian mechanics and the laws of motion. The chapter will then explore Kepler's laws of planetary motion. Kinematics is also briefly explained. Finally, we will discuss some of the challenges that arise when attempting to model and predict the motion of objects in space. Following text is not intended to be overly detailed as this chapter on orbital mechanics presents only the most important concepts extracted from the great amount of available literature such as [1], [16], [5] and [2]. Overall, this chapter aims to provide a foundation for understanding the principles of orbital mechanics and the equations that govern the motion of objects in space. It will be a useful reference for anyone interested in space or other related fields.

2.1 Newton's Laws of Motion

Newtonian mechanics is a fundamental branch of physics that describes the behavior of objects in motion. The principles of Newtonian mechanics are based on three laws of motion, which were first described by Sir Isaac Newton in the late 17th century. These laws form the basis for understanding how objects move and interact with each other. Laws citation from this section and much more details about Newton's laws can be found at [16].
The first law of motion, also known as the law of inertia, states:

„A Body remains at rest or in uniform motion unless acted upon by a force.“

Mathematically, this can be expressed as equation

$$F_{net} = \sum F = 0 \tag{2.1}$$

where F_{net} is the net force acting on the object, or in other words, sum of all forces acting on the object. If $F_{net} = 0$, then the object will remain at rest or continue to move at a constant velocity. If $F_{net} \neq 0$, then the object will accelerate. Simpler interpretation of this law is that the only way to alter the motion of a body is to exert a force upon it.

The degree to which the motion is altered depends on how great the exerted force is. This is stated by Newton's second law:

„A body acted upon by a force moves in such a manner that the time rate of change of momentum equals the force“

This law is also known as the law of acceleration. It relates the net force acting on an object to its acceleration, expressed by equation as

$$F = am \tag{2.2}$$

where m is the mass of the object, a is its acceleration, and F is the net force acting on the object. This law shows that the acceleration of an object is directly proportional to the net force acting on it and inversely proportional to its mass.

The third law of motion, also known as the law of action and reaction, states:

„If two bodies exert forces on each other, these forces are equal in magnitude and opposite in direction.“

This means that if an object exerts a force on another object, the second object will exert an equal and opposite force on the first object. Mathematically, this can be expressed as equation

$$F_1 = -F_2 \tag{2.3}$$

where F_1 is the force exerted by one object on another, and F_2 is the force exerted by the second object on the first.

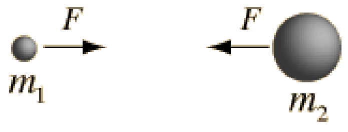


Figure 2.1: Visualization of Newton's third law. Image source at [11].

These three laws of motion can be used to derive the equations of motion for a wide range of systems, including those involving orbital mechanics. For example, the motion of a satellite in orbit around a planet can be described by Newton's second law of motion, taking into account the gravitational force between the satellite and the planet. The resulting equation of motion is known as the two-body problem and can be solved analytically. This problem is described in greater detail in chapter 3.

2.2 Newton's Law of Universal Gravitation

Newton's law of universal gravitation is one of the fundamental laws of physics and plays a crucial role in understanding the behavior of celestial bodies. The law, derived from observations by Isaac Newton [12], states that any two particles in the universe attract each other with a force that is proportional to the product of their masses and inversely proportional to the square of the distance between them. Mathematically, it can be expressed as

$$F = \frac{Gm_1m_2}{r^2} \tag{2.4}$$

where G is universal constant named gravitational constant. It is fundamental constant of universe that determines the strength of gravitational force between two objects. The value of G is known to four significant digits as $6.674 \times 10^{11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$.
This law is also applicable to all objects across the universe. This law not only explains the motion of celestial objects like planets and stars, but it also governs the motion of objects on Earth. For example, the gravitational attraction between the Earth and the Moon causes the Moon to orbit around the Earth and also causes the tides on Earth. Together with Newton's laws of motion, these laws enable us to model the motion of objects in space and on Earth, and to make predictions about their behavior. It is worth noting that the law of universal gravitation is only accurate for relatively small distances and speeds. At high speeds and in strong gravitational fields, the effects of general relativity become important and must be taken into account. Nonetheless, Newton's law of universal gravitation remains a crucial tool in understanding the behavior of celestial bodies.

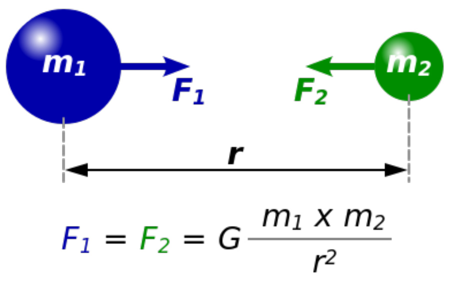


Figure 2.2: Visualization of Law of Universal Gravitation. Image source at [13].

2.3 Kepler's Laws of planetary motion

Kepler's laws describe the motion of planets in the solar system and are fundamental to understanding orbital mechanics. These laws were formulated by Johannes Kepler in the early 17th century, based on the observations of the planets made by Tycho Brahe.

Kepler's first law, also known as the law of orbits, states that planets move around the sun in elliptical orbits, with the sun at one of the foci of the ellipse [11]. The shape of the ellipse is determined by the eccentricity of the orbit, which is a measure of how elongated the ellipse is. The distance between the sun and the planet varies throughout the orbit, with the closest point called the periapsis and the furthest point called the apoapsis. Depending on the body being orbited, these points can have different names like aphelion/perihelion for Sun, or apogee/perigee for the Earth. This law can be expressed mathematically as:

$$r = \frac{p}{1 + e\cos\theta} \tag{2.5}$$

where r is the distance between the planet and the sun, p is the semi-latus rectum of the ellipse, e is the eccentricity of the orbit, and θ is the angle between the perihelion and the current position of the planet.

Kepler's second law, also known as the law of areas, states that a planet moves faster when it is closer to the sun and slower when it is farther away [11]. This law can be expressed mathematically as:

$$\frac{dA}{dt} = \frac{1}{2} r^2 \frac{d\theta}{dt} \tag{2.6}$$

where A is the area swept out by the line connecting the planet to the sun in a given time interval, and $\frac{dA}{dt}$ is the rate at which this area is swept out [11]. This law implies that the planet moves with equal areas in equal times.

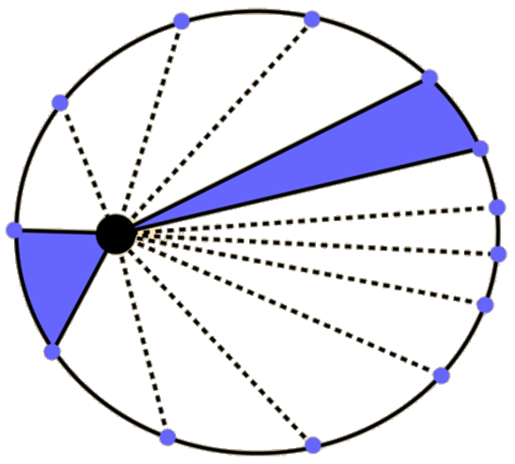


Figure 2.3: Visualization of Kepler's second law - blue areas are equal and swept in equal time. Image source at [11].

Kepler's third law, also known as the law of periods, states that the square of the orbital period of a planet is proportional to the cube of the semi-major axis of its orbit. This law can be expressed mathematically as:

$$T^2 = \frac{4\pi^2}{G(M+m)} a^3 \tag{2.7}$$

where T is the orbital period of the planet, a is the semi-major axis of the orbit, M is the mass of the sun, m is the mass of the planet, and G is the gravitational constant.

Kepler's laws provide a fundamental framework for understanding the motion of planets in the solar system, and have been used extensively in the field of orbital mechanics. They played a key role in the development of the laws of gravitation by Isaac Newton, which describe the motion of objects under the influence of gravity.

2.4 Kinematics

In this section, basics of kinematics are explored. Kinematics is the study of motion without considering its causes [1]. Specifically, the concepts of position, velocity, and acceleration over time are described as these will be important part of solving the n-body problem.

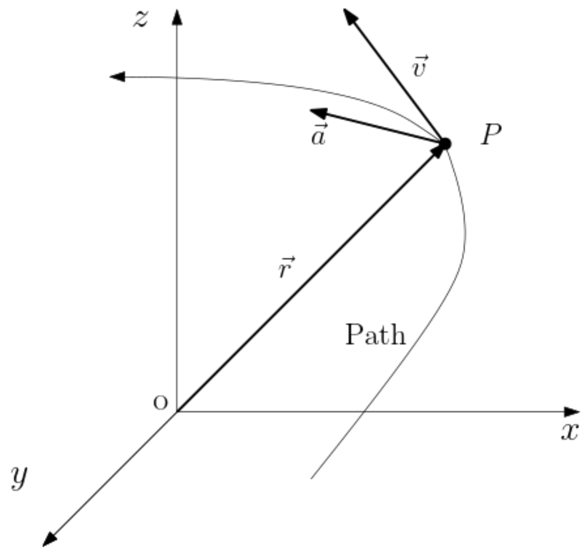


Figure 2.4: Object P traveling along path in reference frame. Position, velocity and accelerations vectors are visible.

Position refers to the location of an object relative to a reference point. It is usually represented by a vector that describes the distance and direction between the object and the reference point. In one-dimensional motion, the position of an object is typically measured along a straight line, while in two-dimensional or three-dimensional motion, the position can be described by two or three coordinates, respectively.

To keep track of motion of any object over time, frame of reference is needed [5]. It consists of three dimensional cartesian coordinate system and clock that keeps track of time.

Rephrased from [8] and [2], velocity is a vector quantity that describes the rate at which an object changes its position. It is defined as the change in position divided by the change in time. Note that vectors in following equations are shown in bold. Mathematically, velocity is expressed as:

$$\mathbf{v} = \frac{\Delta \mathbf{r}}{\Delta t} \tag{2.8}$$

where $\mathbf{v}$ is the velocity vector, $\Delta\mathbf{r}$ is the change in position vector, and Δt is the change in time. The unit of velocity is meters per second $(\mathrm{m\,s^{-1}})$ in the SI system. The velocity vector has both magnitude and direction. The magnitude of the velocity vector is called speed, which is the distance traveled per unit of time.

Acceleration is the rate at which an object changes its velocity. It is defined as the change in velocity divided by the change in time. Mathematically, acceleration is expressed as:

$$\mathbf{a} = \frac{\Delta \mathbf{v}}{\Delta t} \tag{2.9}$$

where **a** is the acceleration vector, and $\Delta\mathbf{v}$ is the change in velocity vector. The unit of acceleration is meters per second squared ($\mathrm{m\,s^{-2}}$) in the SI system.

Following two equations are taken from [5]. As stated above, velocity **v** and acceleration **a** of the object travelling through time frame are first and second derivatives by time of the position vector r:

$$\mathbf{v}(t) = \frac{dx(t)}{dt}\hat{\mathbf{i}} + \frac{dy(t)}{dt}\hat{\mathbf{j}} + \frac{dz(t)}{dt}\hat{\mathbf{k}} = v_x(t)\hat{\mathbf{i}} + v_y(t)\hat{\mathbf{j}} + v_z(t)\hat{\mathbf{k}} \tag{2.10}$$

$$\mathbf{a}(t) = \frac{dv_x(t)}{dt}\hat{\mathbf{i}} + \frac{dv_y(t)}{dt}\hat{\mathbf{j}} + \frac{dv_z(t)}{dt}\hat{\mathbf{k}} = a_x(t)\hat{\mathbf{i}} + a_y(t)\hat{\mathbf{j}} + a_z(t)\hat{\mathbf{k}} \tag{2.11}$$

Vectors $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ are all unit vectors pointing from origin in the positive direction of x, y and z axes. For convenience, time derivatives have shorthand notation where overhead dot is used, so:

$$\mathbf{v} = \dot{\mathbf{r}} \tag{2.12}$$

$$\mathbf{a} = \dot{\mathbf{v}} = \ddot{\mathbf{r}} \tag{2.13}$$

Chapter 3

Predicting motions of celestial bodies

Predicting motions of celestial bodies is very important ability for any civilization and is absolutely necessary for sending spacecrafts to Moon and other planets. As we discover more and more asteroids in the vicinity of Earth (known as near-Earth asteroids), we can also predict their motion. For example, NASA plans to launch telescope carrying spacecraft[1] into low Earth orbit, which is expected to find 90% of near-Earth asteroids over 140 meters in length. Motions of these asteroids can then be simulated into the future. In case of discovering high probability of collision with the Earth with enough time in advance, action can be taken.

Different approaches to predicting these motions are used based of number of objects in simulated system. With two body systems, like Earth-Moon system, analytical approach can be used. With systems where 3 or more bodies are involved, equations became more complex and can only be solved numerically [2]. In classical mechanics, these problems are known as the two-body problem and the n-body problem, respectively. I will describe them in greater detail in following sections.

3.1 Two-body problem

As stated above, two-body problem is problem of predicting motions of two bodies that influence each other by their gravity. It is assumed that no other forces exists in the system or are ignored. Unlike n-body problem, 2-body problem can be solved analytically by calculating gravitational force between the two objects by using Newton's Law of Universal Gravitation equation (2.4). The two-body problem is an idealized model, as it assumes that the two masses are point masses that are not affected by gravitational influence from any other objects in the universe. While this is not true in real-world situations, the two-body problem provides a useful approximation for analyzing the motion of celestial bodies over short timescales.

3.2 N-body problem

N-body problem is a problem in classical mechanics of predicting motions of three or more bodies that are gravitationally interacting with each other and is much more complex

[1]NEO Surveyor - https://www.jpl.nasa.gov/missions/near-earth-object-surveyor

then two-body problem [2]. The n-body problem assumes n point masses (from now on referenced as objects) where each object is defined by its mass m_i, position vector $\mathbf{r_i}$ and velocity vector $\mathbf{v_i}$ within inertial reference frame. Also $i = 1, 2, ..., n$.

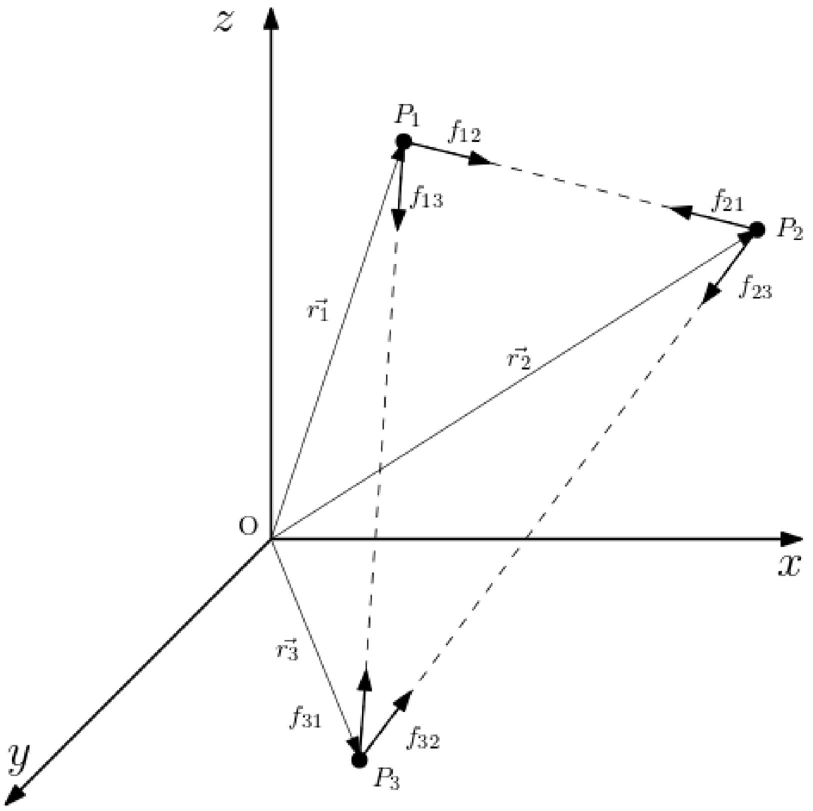


Figure 3.1: Visualization of three body problem. Each body is exerting gravitational force on every other body.

Solving this problem involves solving equations of motions for all objects in the system. These equations are the Newtonian equations of motion, which are a set of second-order Ordinary Differential Equations (ODE). They describe the motion of each object in the system as a function of time. Governing equations of n-body problem are Newton's second law (2.2) [2] and Newton's law of Universal Gravitation (2.4), which is the only source of force. Therefore, no other force than gravity is taken into account. From (2.2) [1], the force exerted on object i by object j is

$$\mathbf{F_{ij}} = -G\frac{m_i m_j}{|r_{ij}|^2}\hat{\mathbf{r}}_{\mathbf{ij}} \tag{3.1}$$

where vector $\mathbf{r_i}$ is position vector of object i in reference frame. Vector $\mathbf{r_{ij}}$ is pointing from object i to object j and $|r_{ij}|$ is its magnitude. Symbol $\hat{\mathbf{r}}_{\mathbf{ij}}$ is a unit vector pointing in the same direction as $\mathbf{r_{ij}}$, but has length equal to one. Using $\hat{\mathbf{r}}_{\mathbf{ij}} = \frac{\mathbf{r_{ij}}}{|r_{ij}|}$ to manipulate the equation we get

$$\mathbf{F_{ij}} = -G\frac{m_i m_j}{|r_{ij}|^2}\frac{\mathbf{r_{ij}}}{|r_{ij}|} = -G\frac{m_i m_j}{|r_{ij}|^3}\mathbf{r_{ij}} \tag{3.2}$$

Let F_i be the total force exerted upon object i by all the other objects. Thus the total force F_i is sum of all the forces from other objects

$$\mathbf{F_i} = \sum_{j,j\neq i}^{n} \mathbf{F_{ij}} = \sum_{j,j\neq i}^{n} -G\frac{m_i m_j}{|r_{ij}|^3}\mathbf{r_{ij}} \tag{3.3}$$

At any given point in time t, position and velocity vectors, $\mathbf{r_i}$ and $\dot{\mathbf{r_i}}$ respectively, of any given object, are known. Thus:

$$\dot{\mathbf{v}}_i = \mathbf{a}_i = \frac{\mathbf{F}}{m_i} = \frac{1}{m_i}\sum_{j,j\neq i}^{n} -Gm_i m_j\frac{\mathbf{r_{ij}}}{|r_{ij}|^3} = -G\sum_{j,j\neq i}^{n}\frac{\mathbf{r_i}-\mathbf{r_j}}{|r_i - r_j|^3} \tag{3.4}$$

is set of $6N$ first-order ODEs [4]. By knowing initial condition it is possible to calculate position and velocity in any given time t.

3.3 Simulating N-body problem

To predict motions of celestial bodies or objects, model of system containing such objects must be created first. By definition, model is imitation of one system by other system [14], in this case computer program is used to model Solar System from the real world. This modeled system can be greatly simplified compared to the real world one, as not all information about real world system is relevant to the purpose of model. Experimenting with such model to gain new information about modeled system is called simulation. For this thesis specifically, new information about Solar System means predicting motions of its elements, that is celestial bodies. Important to note is that data from simulation can not be perfectly accurate compared to the reality. This is because reality is continuous, while computer trying to model it is strictly discrete. Therefore, some amount of error will always be present in results of simulation. Though this error can be reduced by choosing the right simulation methods and its parameters. To validate created model, i. e. to declare that it models reality with enough precision to be considered accurate or usable, data from NASA will be used. Several terms regarding simulation are used many times in this thesis. This listing explains them briefly:

- Simulation time - Total duration of simulated process.
- Simulation step - Single iteration of simulation.
- Time step - Amount of simulated time that elapses between two following simulation steps.
- Step count - Number of simulation steps that have been taken during simulation. Its value is calculated by dividing simulation time with time step.

Note that the unit of time throughout the simulation are seconds. Since first-order ODE are part of created model, numerical methods are used for integrating them. Numerical methods involve dividing continuous time into many small discrete time steps, calculating forces between objects at each time step and then using calculated force to determine the acceleration of each object. Object is then moved in this direction for chosen time step. Main control loop of simulation simplified and written into code as:

```python
simulation_time = 60 * 60 * 24 # one day of simulation
elapsed_time = 0
time_step = 1
while elapsed_time < simulation_time:
    for body in bodies:
        acc = get_acceleration(body)
        integrate(body, acc, time_step)
    elapsed_time += time_step
```

Listing 3.1: Simulation loop

In this code snippet, `while` loops represents passing of simulation time, where each iteration is single simulation step. Of course, every object in simulated system must be taken into account and thus `for` loop that iterates over all of the objects. For each object, force exerted upon it is calculated by equation (3.4). Written into code as:

```python
def get_acceleration(current_body):
    acceleration = [0, 0, 0] # 3D vector
    for body in bodies:
        if body is current_body:
            continue
        r = current_body.position - body.position
        acceleration += body.mass * r / (magnitude(r) ** 3)
    acceleration *= -G_CONSTANT
    return acceleration
```

Listing 3.2: Acceleration calculation

Because acceleration of every objects needs to be computed in respect to every other object, two nested `for` loops are required to create every possible pair of objects. This makes the time complexity of single step $O(n^2)$ where n is number of objects in simulated system. When acceleration is known, its integration can be performed. Three different integration methods were implemented. These methods vary in computation cost and accuracy. Most simple and straightforward numerical integration methods is Euler method. Written into code as:

```python
def integration(body, acceleration, time_step):
    body.position += body.velocity * time_step
    body.velocity += acceleration * time_step
```

Listing 3.3: Euler integration step

Implemented algorithms for solving numerical integration are described in following subsections 3.3.1, 3.3.2, 3.3.3 and were studied from [3], [14] and [17].

3.3.1 Euler method

Euler method, is a numerical method for solving ODEs with a given initial value. The method is based on approximating the solution curve with small line segments, using the slope of the tangent line at each point to determine the direction and length of the segment. It is also the simplest, as it calculates result using only initial value [14]. General formula for calculating value in next step is

$$y(t+h) = y(t) + h \cdot f(t, y(t)) \tag{3.5}$$

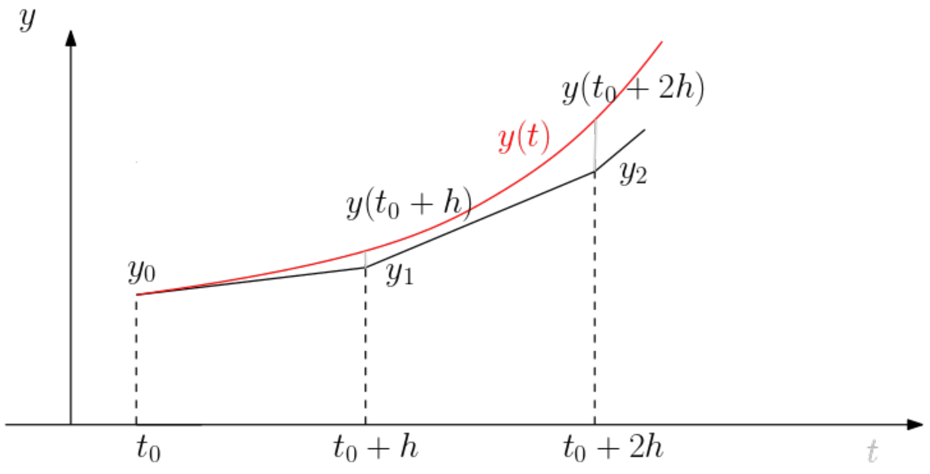


Figure 3.2: Visualization of how Euler method works. Small step in the direction of the derivative of the solution at each point is taken.

In case of this thesis, two equations are being solved, one for position and one for velocity of simulated object. From (3.4) these are

$$r(t+h) = r(t) + h \cdot V(t, r(t), v(t)) \tag{3.6}$$

$$v(t+h) = v(t) + h \cdot A(t, r(t), v(t)) \tag{3.7}$$

where r and v are position and velocity at time t, respectively. h is selected time step and functions V and A returns velocity and acceleration at time t, respectively. These functions are defined as

$$V(t, r(t), v(t)) = \dot{r} \tag{3.8}$$

$$A(t, r(t), v(t)) = \dot{v} \tag{3.9}$$

Initial position and velocity must be known to start integration method. Eueler method is fairly simple and its computation cost is smaller compared to other methods, but it is way less precise.

3.3.2 Runge-Kutta

Runge-Kutta methods perform other computations within single step to achieve more precise results while keeping same step size [14]. There are many variants of this method, but most widely used one is known as Runge-Kutta 4. Runge-Kutta 4 is method of fourth order and calculates four intermediate values that are then used to approximate the solution of given differential equation. They are typically denoted as $k_1, ..., k_n$, where n is the order of method. These intermediate values are then used to update the dependent variable and advance the solution to the next step. To approximate the value of $y(t+h)$, we use formula

$$y(t+h) = y(t) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \tag{3.10}$$

where intermediate values are calculated as

$$k_1 = h \cdot f(t, y(t)) \tag{3.11}$$

$$k_2 = h \cdot f(t + \frac{h}{2}, y(t) + \frac{k_1}{2}) \tag{3.12}$$

$$k_3 = h \cdot f(t + \frac{h}{2}, y(t) + \frac{k_2}{2}) \tag{3.13}$$

$$k_4 = h \cdot f(t + h, y(t) + k_3) \tag{3.14}$$

Based on these formulas, it can be noted that the function f is evaluated four times during single step. While Runge-Kutta 4 method is much more precise then Euler method, it is computationally more expensive, because it has to evaluate function A multiple times during single step. This is even more pronounced in this application where two ODEs are being solved. As stated in 3.3.1, velocity and acceleration are both calculated in single step, therefore two sets of intermediate values are calculated:

$$k_1 = h \cdot V(t, r(t), v(t)) \tag{3.15}$$

$$l_1 = h \cdot A(t, r(t), v(t)) \tag{3.16}$$

$$\ldots \tag{3.17}$$

$$k_4 = h \cdot V(t, r(t) + k_3, v(t) + l_3) \tag{3.18}$$

$$l_4 = h \cdot A(t, r(t) + k_3, v(t) + l_3) \tag{3.19}$$

Finally calculating values at the end of the step:

$$r(t+h) = r(t) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \tag{3.20}$$

$$v(t+h) = v(t) + \frac{l_1}{6} + \frac{l_2}{3} + \frac{l_3}{3} + \frac{l_4}{6} \tag{3.21}$$

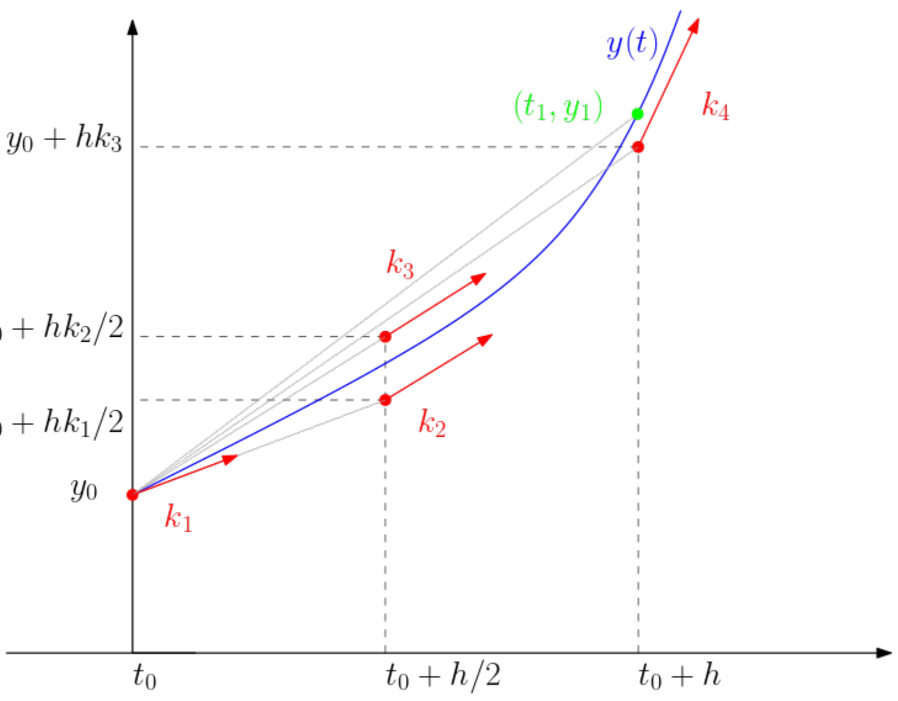


Figure 3.3: Visualization of how Runge-Kutta method works. Runge-Kutta method calculates a weighted average of several estimates of the derivative of the solution at each point to compute a more accurate approximation of the solution.

3.3.3 Adams-Bashfort

Adams-Bashfort method is an example of multi-step method. These methods use values from previous simulation steps to compute solution at the next step. Specifically, if the method uses n previous solutions then this method is of n-th order. The problem with multi-step methods is starting them while only one previous solution, the initial value, is provided. This can be solved by using other single step method, like Euler method, to compute first n solutions and then continue with multi-step method. General formula for n-th order of Adams-Bashfort method is:

$$y(t+h) = y(t) + h \cdot (c_1 f(t) + c_2 f(t-h) + c_3 f(t-2h) + ... + c_n f(t-h(n+1))) \tag{3.22}$$

where c_n are coefficients that can be determined by integrating the differential equation using a Taylor series expansion [3]. The idea is to use the known values of the function and its derivatives at previous time steps to approximate the value of the function at the next time step. For solving n-body problem, 4-th order method was implemented. Formulas are

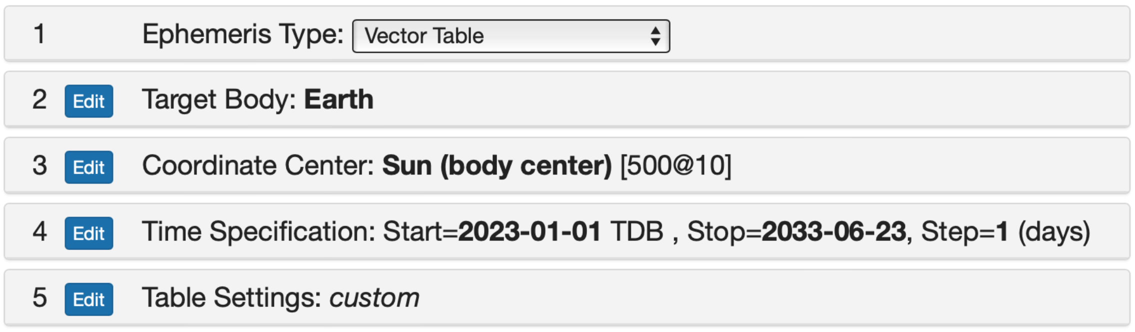
$$r(t+h) = r(t) + \frac{h}{24}(55V(t) + 59V(t-h) + 37V(t-2h) + 9V(t-3h)) \tag{3.23}$$

$$v(t+h) = v(t) + \frac{h}{24}(55A(t) + 59A(t-h) + 37A(t-2h) + 9A(t-3h)) \tag{3.24}$$

Compared to other numerical methods for solving ODEs, such as the Runge-Kutta methods, this method can be faster for large systems of equations, since it only requires information from a few previous time steps to compute the solution at the next time step. However, it can be less accurate than some other methods. More about Adams-Bashfort can be found at [3].

3.4 Data source

For modeling real planetary system, precise initial conditions are needed. These data are available at NASA's Horizons system at [7]. It is an online Solar system data and ephemeris[2] computation service that provides information about the positions and motions of objects in our Solar system. The Horizons system can be quite complex and may require some knowledge of astronomy and celestial mechanics to use effectively. However for needs of this thesis, simple textual vector data is sufficient.


1 Ephemeris Type: Vector Table

2 Edit Target Body: **Earth**

3 Edit Coordinate Center: **Sun (body center)** [500@10]

4 Edit Time Specification: Start=**2023-01-01** TDB , Stop=**2033-06-23**, Step=**1** (days)

5 Edit Table Settings: *custom*

Figure 3.4: Interface of Horizons system web application with all the relevant settings of ephemeris data.

Listing 3.4 show example of data[3] retrieved from interface.

```
**********************************************
JDTDB, Calendar Date, X, Y, Z, VX, VY, VZ,
***********************************************
$$SOE
2459945.5,-2.546E+07,1.448E+08,-7.309E+03,-2.981E+01,-5.280E+00,-6.389E-04,
2459946.5,-2.804E+07,1.444E+08,-7.364E+03,-2.971E+01,-5.799E+00,-6.203E-04,
2459947.5,-2.984E+07,1.440E+08,-7.402E+03,-2.9610+01,-5.988E+00,-6.078E-04,
$$EOE
************************************************
```

Listing 3.4: Example data from Horizons

[2]Table, chart, or other document that provides information about the positions and movements of celestial objects

[3]Decimal places were trimmed for better readability

These table settings were used when extracting data for simulating Solar system and other experiments:

- Ephemeris type - State vector table containing position and velocity vectors.
- Calendar type - Mixed.
- Units - Kilometers and seconds. These were converted to meters before importing them to solver application.
- Table format - Comma separated values (csv) file type.

These data are used for initial conditions in simulations and in experiments from section 4.5.2 to validate implemented simulator.

Chapter 4

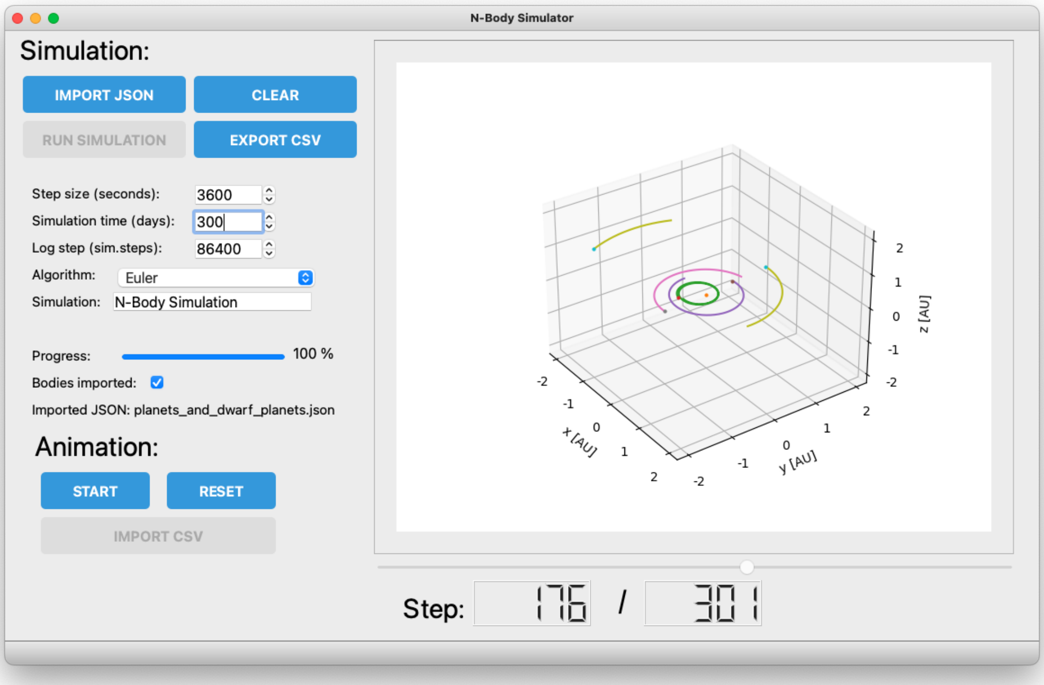
Program design and implementation

Main purpose of implemented application is to provide general n-body problem solver with multiple integration methods to choose from. This also included implementing Graphical User Interface (GUI). The interface is intuitive and simple to navigate, making it accessible to users of all skill levels. The interface includes options for importing systems, selecting numerical integration algorithms, adjusting simulation parameters, and exporting the results. The user can pause and restart the simulation, and observe the motion of the simulated objects. The ability to export the simulation results in csv format is another important feature of the n-body solver application. This enables the user to analyze and manipulate the data in external programs, such as Excel, MATLAB or python notebooks, and further explore the behavior of the simulated system. The csv format provides a flexible and widely supported file type, making it easy to share the results with others. The n-body

problem solver also allows the user to import and simulate any system by importing a specific JavaScript Object Notation (JSON) file. This provides flexibility and convenience, as the user can easily switch between different systems without having to manually enter the initial conditions for each object. The ability to import systems also enables the user to collaborate and share simulations with others, as they can easily share the JSON files. This also ensures ability to create custom systems, so that user is not constrained to presets.

4.1 Graphical user interface

Graphical user interface was implemented using Qt framework, specifically its Python binding PyQt [15]. Qt is cross-platform framework, which makes this application usable on most platforms.



Step size (seconds): 3600
Simulation time (days): 300
Log step (sim.steps): 86400
Algorithm: Euler
Simulation: N-Body Simulation
Progress: 100 %
Bodies imported: ✓
Imported JSON: planets_and_dwarf_planets.json
Step: 176 / 301

Figure 4.1: Main window of the application in MacOS enviroment.

Layout of GUI itself was designed using QtDesigner tool. Back-end code was written for it in python language. As can be seen on figure 4.2, GUI consists of simulation and animation control section, animation widget and import/export buttons. This section will explain how various elements of GUI works. First step to start simulating any system is to give the application data about the system itself. To do this, user must use `IMPORT` button, which will open file explorer dialog window. User now should locate desired JSON file (JSON input format is described in 4.2.1) and select it. Application will parse bodies from selected JSON file and store them in memory. This is indicated by checked checkbox labeled by text „Bodies imported:“. Name of imported file is also visible in GUI. Next, simulation settings can be adjusted. These include:

- `Simulation time` - Total duration of simulated process.
- `Step size` - Length of single simulation iteration.
- `Log step` - Determines, after how many time steps should data about position and velocity be logged. This option is set by default to single day.
- `Algorithm` - Sets the numerical integration algorithm.

By setting `Log Step` option to one, there will as many logs in output file as there are simulation steps. `Algorithm` As already stated in 3.3, three methods are implemented:

- `Euler method` - Is selected by default. Described in 3.3.1.
- `Runge-Kutta 4` - Described in 3.3.2.
- `Adams-Bashfort` - Described in 3.3.3.

Also possibility to simplify n-body problem to a set of two-body problems was implemented. Purpose of this is purely educational, so that users can see the difference. Whether system is to be solved as n-body or two-body problem is determined by imported JSON file in the “`Type`” field. This is again explained in more detail in 4.2.1. To start simulation after all settings were set, `RUN SIMULATION` button must be pressed. These simulations can take a long time, so they are performed on thread that is separate from GUI thread. This makes sure that GUI will not freeze during longer simulations. When simulation starts, dialog box containing progress bar and percentage appears. This dialog box can only be closed after simulation is done.

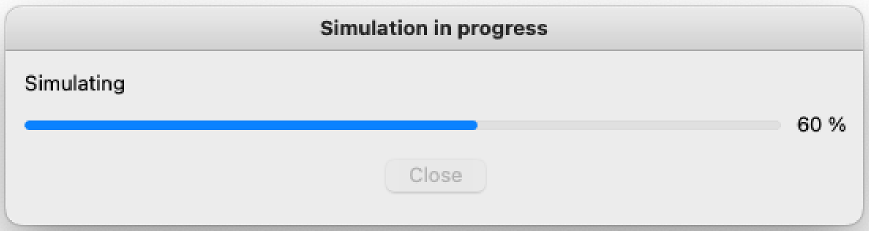


Figure 4.2: Dialog box during simulation.

Note that the animation output is not running alongside the simulation, but rather waits until simulation is done. When simulation is ready, button `START` in animation controls will be enabled, and after clicking it, animation will start. Animation can be brought to the beginning by pressing `RESET` button. In animation, simulated trajectories of objects are traced by colored lines from start of the simulation to current frame. Position and names of simulated objects indicated by round mark leading the trajectory are always displayed at simulation time displayed on top left of animation widget. Animation is 3D scene and can be rotated or zoomed. Distances are indicated by x, y and z axes, which displays distances in AU[1] units. Slider underneath the animation widget indicates progress of the animation.

To clear completed animation and import new system, user should click `CLEAR` button in simulation control section. This will free all previously imported bodies from memory, as well as animation and simulation objects.

4.2 Program data interface

This section describes input/output specification of n-body problem solver.

4.2.1 Input

For any simulation using numerical integration methods, the required input are the initial conditions. For simulation of the n-body problem, each body in simulated system must be described by pair of initial conditions: position and velocity. Additionally, mass of object

[1]Astronomical units - mean distance from the centre of the earth to the centre of the sun ($1.496e+8\,\mathrm{km}$)

is required for force calculations. *N*-body solver takes JSON document describing single system as an input. Two different JSON file configurations exist, one is for the *n*-body problem, while the other one describes two-body problem, which except for list of bodies must include common attractor body for these objects. Both types of JSON documents must contain following fields:

- `SystemName` - String containing the name of the system.
- `Type` - String determining simulation type. Two possible values are "`nbody`" and "`2body`".
- `DataGregorian` - String in format "2000-Jan-01 00:00:00.0000". This string specifies initial date of simulation.
- `DateJulian` - Julian calendar version of previous field.
- `CoordinateCenter` - String specifying coordinate center of imported system. However, value of this field does not have any impact on program as support for different types of coordinate centers was not implemented and therefore is purely infromational.
- `System` - This is the list of objects to be simulated.

Objects have these fields:

- `BodyName` - String containing the name of this object. It is displayed in animation and output file is named by it.
- `Mass` - Integer value representing mass of the object in kilograms.
- `Position` - Vector of three floats. It is the position of an object in reference frame. Unit used is meter.
- `Velocity` - Vector of three floats. It is the velocity of an object. Unit used is meters per second.

Example of such JSON:

```

{
  "SystemName": "Solar System",
  "Type": "nbody",
  "DateGregorian": "2000-Jan-01 00:00:00.0000",
  "DateJulian": 2451544.500000000,
  "CoordinateCenter": "Sun (body center)",
  "System": [
    {
      "BodyName": "Sun",
      "Mass": 1.989e30,
      "Position": [0, 0, 0],
      "Velocity": [0, 0, 0]
    },
    {
      "BodyName": "Mercury",
      "Mass": 0.330e24,
      "Position": [-2.105E+10, -6.640E+10, -3.492E+09],
      "Velocity": [ 3.665E+04, -1.228E+03, -4.368E+03]
    },
        // ... rest of planets is here
    {
      "BodyName": "Apophis",
      "Mass": 2.1e10,
      "Position": [-1.555+11,  -2.068E+10, -2.683E+09],
      "Velocity": [ 7.088E+03, -2.600E+04,  1.547E+03]
    }
  ]
}
```

Listing 4.1: JSON input document example

Additionally, if "`Type`" field is set to "`2body`", another `Attractor` field is required. This field describes main attractor body for two-body simulation and it looks like this:

```
{
  ...
  "Type": "2body",
  ...
  "CoordinateCenter": "Sun (body center)",
  "Attractor": {
      "BodyName": "Sun",
      "Mass": 1.989e30,
      "Position": [0, 0, 0],
      "Velocity": [0, 0, 0]
    },
  "System": [ ... ]
```

Listing 4.2: Attractor field for two-body input file

It is also possible to import csv file that was previously produced by the application and replay its animation. This can be done by pressing `IMPORT CSV` button and selecting folder with csv files.

4.2.2 Output

The application generates two types of output. One is of course the animated simulation which can be viewed in graphical user interface and is described in section 4.4.

Other type is csv document which stores the position and velocity of objects throughout simulation. Example of such document[2] created for Mars:

```
% Coordinate vector over time for "Mars"
date,x,y,z
2000-01-01 00:00:00,207995054983.6,-3143009713.9,-5178781243.4
2000-01-01 00:00:00,207996349987.1,-3116715293.2,-5178262233.7
2000-01-01 11:06:40,208045637615.3,-2064904387.7,-5157439373.0
2000-01-01 22:13:20,208090023716.3,-1013043736.4,-5136494992.1
2000-01-02 09:20:00,208129509790.4, 3884186415.9,-5115429647.3
```

Listing 4.3: CSV output document example

Export is only possible after simulation was performed. Otherwise, `EXPORT` button can not be clicked. After clicking it, file explorer dialog window appears. User should choose folder where he wants to export simulation results. After selecting folder, another sub-folder named by system name from JSON input will be created. If folder with such name already exists in selected folder, numbers will be appended to the name until it is unique within selected folder. Also any whitespaces in the name will be replaced by underscores. Inside this sub-folder, csv files for each body will be written. If JSON from listing 4.1 was imported, created subfolder would be named `Solar_System` and csv files inside `Apophis.csv` such as:

```
selected_folder/
├── Solar_System/
│   ├── Mercury.csv
│   ├── Venus.csv
│   ├── Earth.csv
│   ├── Mars.csv
│   ├── Jupiter.csv
│   ├── Saturn.csv
│   ├── Uranus.csv
│   ├── Neptune.csv
│   └── Apophis.csv
└── Solar_System2/
```

Figure 4.3: Directory structure of exported simulation. There is one csv file for each object in simulated system.

[2]Decimal places were trimmed for vectors to fit in single line

4.3 Simulator implementation

The simulation is implemented in Python using an object-oriented approach. Simulation and integration methods are all implemented without any integration or simulation libraries. The simulation logic is defined in a base class that all simulation types and algorithms inherit from. This allows for easy extension of the simulation with new simulation types or algorithms. Base class 4.3.1 implements main simulation control loop (simplified code at 3.1) and acceleration calculation (see 3.2), as these are mostly same among all methods. Each subclass then defines its `do_step()` function, which is place where logic of each integration method is implemented. Some subclasses however do override the simulation loop method from base class because they require some extra or modified logic.

For the 2-body simulation, the simulation class calculates the gravitational force between two objects based on their masses and distance, and then updates their positions and velocities based on the force and the time step. In two-body simulation, there are two types of objects:

- Attractor - Main body of simulated system. When simulating moon system, this should be planet and when solar systems, it should be star.
- Orbiter - All the other simulated objects in imported system.

Attractor should be always set as the most massive body in imported system. While there can be always exactly one attractor, there can be multiple orbiters. Pair with attractor is created for each orbiter. These pairs are then treated as a set of two-body problems which is solved for duration of selected simulation time. This also means, that while every orbiter is updated once every simulation step, attractor is updated as many times as the count of orbiters in a single step. This process is repeated until the desired simulation time is reached.

For the n-body simulation, the base class uses the same gravitational force calculation, but applies it to all pairs of objects in the simulation. This can quickly become computationally intensive for large numbers of objects, so three algorithms are implemented which have different efficiency to precision ratio: Euler, Runge-Kutta and Adams-Bashforth.

As described in detail in section 3.3.1, the Euler algorithm is the simplest and fastest, but it can be less accurate than the other algorithms. It uses a first-order approximation to update the positions and velocities of the objects at each time step.

The Runge-Kutta algorithm is a more accurate but slower algorithm that uses a 4th order approximation to update the positions and velocities of the objects at each time step. This means, that acceleration calculation must be performed 4 times for every body in each step, making it slowe. For more details, see section 3.3.2.

The Adams-Bashforth algorithm is the accurate and also quite fast. It is a multistep method which uses not only the current state, but also states from past steps. Again, this method is described in section 3.3.3.

Results from simulation are stored into separate lists. The frequency of logging is determined by selected log step[3].

4.3.1 Relevant classes

Some classes and their methods relevant to simulation are listed in this section and explained briefly.

class SimulatorBase

This is base class for all simulations and algorithms. It implements common methods and stores simulation settings and parameters like list of simulated objects, step size, simulation time and log step. This class inherits from `QThread` class, which comes from PyQt framework. This enables moving simulation to its own thread and still be able to communicate progress to GUI using PyQt's slots and signals mechanics. Important methods and signals are:

- `run()` - Overloaded method from base `QThread` class. This method must be implemented for threading to work.
- `start_simulation()` - Starts simulation control loop. Also progress tracking is implemented here.
- `get_acc()` - Calculates acceleration between all pairs of objects.
- `sig_simulation_done` - Signal that is sent to controller after simulation finished.
- `sig_progress_made` - Signal sent after every 5% of progress was made.

class Simulator2Body

Inherits from `SimulatorBase`. Implements simulation of two-body problem.

- `start_simulation()` - Starts simulation control loop. Consists of calling base method from `SimulatorBase` and then appending attractor to list of bodies for further manipulation.
- `get_acc()` - Calculates acceleration between all pairs of objects. Overloaded from base class calculation of acceleration beacuse it is different for two-body problem.
- `do_step()` - Logic of single simulation step

class SimulatorNBodyEuler

[3]Log step is explained in 4.1

Inherits from `SimulatorBase`. Implements simulation of Euler method for n-body problem. Simulation control is inherited from base class. Simulation step is implemented in `do_step()` method.

class SimulatorNBodyRK4

Inherits from `SimulatorBase`. Implements simulation of Runge-Kutta 4 method for n-body problem. Simulation control is inherited from base class. Simulation step is implemented in `do_step()` method.

class SimulatorNBodyAdamsBashfort

Inherits from `SimulatorBase`. Implements simulation of Euler method for n-body problem. Simulation step is implemented in `do_step()` method.

- `start_simulation()` - Starts simulation control loop. Must be overloaded due to specifics of this method. First four steps are calculated using Euler method, followed by normal simulation control loop from base method.
- `do_step_euler()` Euler method step used for starting the method.

class Body

This class describes single simulated object. Its attributes are:

- `pos` - Stores current position of body as 3D vector in meters.
- `vel` - Stores current velocity of body as 3D vector in meters per second.
- `mass` - Stores mass of object in kilograms. Value of this attribute never changes.
- `name` - Name of objects. Is used for export and in animation.
- velocity log - Log for velocity vectors over time. Individual logs for x, y and z elements exist. This log is driven by log step.
- position log - Log for position vectors over time. Individual logs for x, y and z elements exist. This log is driven by log step.
- `t_log` - Log for passed simulation time. Is used during export.
- `acc_log` - Acceleration log for Adams-Bashfort method. Logs acceleration every single step.
- `vel_log` - Velocity log for Adams-Bashfort method. Logs velocity every single step.

Method `log_data()` is used to store current postion and velocity into logs.

4.4 Visualizing the results

The animation of the simulated system is implemented using Python and the `matplotlib` [9] library's `FuncAnimation` class. The plot is presented in 3D, where each axis has astronomical unit (AU) units. The animation can be resumed, stopped, and reset from the graphical user interface (GUI). To create the animation, the `FuncAnimation` class is used to generate a sequence of frames that represent the position of the simulated objects at each log step. Each frame is then plotted using `matplotlib`'s `Axes3D` module to create the 3D plot.

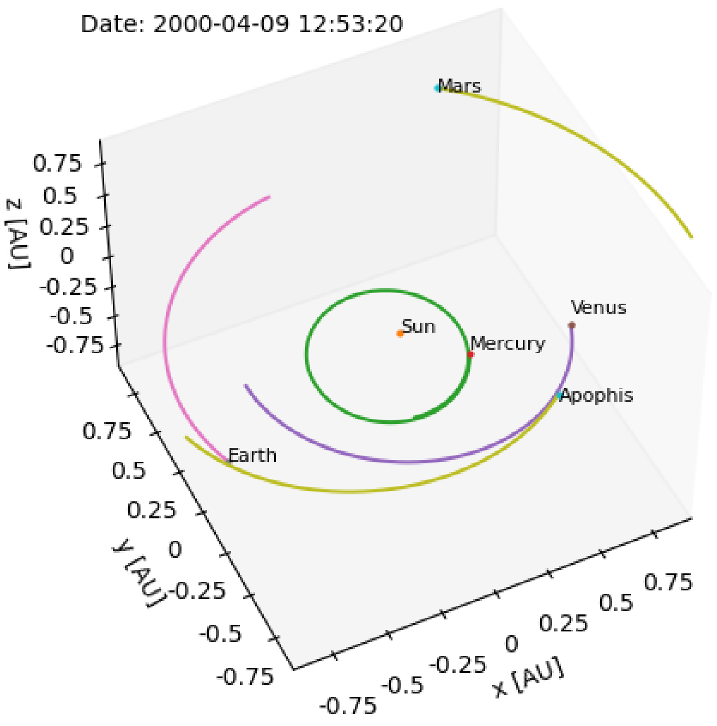


Figure 4.4: Animation widget. It shows simulated objects in the reference frame traveling along their simulated trajectories.

The `FuncAnimation` class takes several arguments, including a function that generates the frames, the number of frames to generate, the time interval between frames, and a callback function to be called at the end of each frame. In this implementation, the function that generates the frames takes the simulated data as input and returns the positions of the objects at each time step. Interval between frames is always set to 20 milliseconds. The GUI allows the user to control the animation using buttons for resuming, stopping, and resetting the animation. These buttons are linked to callback functions that modify the `FuncAnimation` object accordingly. Application also provides the user with the ability to zoom inside the animated system and rotate the whole plot. These functionalities are implemented using matplotlib's interactive features. To enable zooming, the user can use the scroll wheel on their mouse to zoom in or out of the plot. The zooming is implemented using `matplotlib`'s `ax.set_xlim()` and `ax.set_ylim()` functions, which adjust the limits of the x and y axes to zoom in or out. User can also click and drag the plot to rotate it in 3D space. By providing these interactive features, the application allows the user to explore the simulated system in greater detail and gain a better understanding of its behavior. Overall, the animation is an important aspect of the application as it provides a visual representation of the simulated system's behavior over time. By allowing the user to control the animation, the application provides an interactive and informative experience.

4.4.1 Relevant classes

Class responsible for animation is described in this section.

class NBodyAnimation

This class is always created for specific simulation and the instance is cleared from memory before another animation starts. Important methods are:

- `setup_animation()` - Sets up plot to be ready for animation. Labels, limits and ticks of Axes are set accordingly in this method.
- `animate()` - Initialises `FuncAnimation` object. Creates artist objects for it. These are line plots for trajectory, markers for current position of objects, and labels for names of objects. Also single label for displaying time is created here.
- `update_animation()` - Draws current frame. This function is called for every frame update.
- `adjust_ticks()` - Dynamically adjusts tick spacing after zoom events. This ensures that axes of animation plot are clean and readable.

4.5 Testing and validation

In this section, we will discuss the testing methodology used for the simulation application and the obtained results.

Three algorithms, namely Euler, Runge-Kutta, and Adams-Bashforth, were implemented and tested on two different systems of planets: the Solar system planets including asteroid Apophis[4] and the Solar System planets plus four dwarf planets Ceres, Pluto-Charon system, Makemake and Sedna. These systems have 10 and 13 objects, respectively. The time taken to complete the simulation and the accuracy of the results obtained were measured and compared for the three algorithms. Initial

[4]Apophis is famous near Earth asteroid. More at 5.2

conditions for these tests and masses were retrieved from [7] and [10]. Simulation settings were following:

- `Simulation time` - 10 years
- `Time step` - 3600 seconds = 1 hour

Based on these settings, number of steps can be calculated as `Simulation time / Time step` and it is equal to 87600 steps. Then another set of tests was performed, with smaller time step of 1 minute.

4.5.1 Time

Six testing runs were recorded. Different combinations of algorithms and simulated system were tried. Duration results are shown in following table:

Table 4.1: Table shows time it took to simulate each combination of algorithm and count of bodies in simulated system.

Algorithm	**10-body system**	**13-body system**
Euler	91 s (1.51 min)	268 s (4.46 min)
Runge-Kutta	508 s (8.46 min)	1095 s (18.25 min)
Adams-Bashfort	256 s (4.26 min)	281 s (4.68 min)

These results confirm that Runge-Kutta 4 is indeed slowest of the implemented methods as stated in 3.3.2. Of course, these numbers are highly depended on specifications of machine, that run the simulation, but they can be used reliably to compare integration methods in terms of performance. Another interesting observation is the difference between Euler and Adams-Bashfort methods. When simulating 10-body systems, difference between these methods was 195 seconds, while with 13-body system it was only 25 seconds.

4.5.2 Precision

Results of following tests were exported and analyzed using `Jupyter notebook` and `matplotlib` tools. The accuracy of each algorithm was evaluated based on the deviation of the simulated results from the actual data obtained from NASA. The results showed that the Runge-Kutta algorithm was the most accurate among the three algorithms. After 10 years of simulation, the simulated position and velocity of Earth were almost identical to the actual data from NASA. On the other hand, the Euler algorithm performed the worst, with the simulated position of Earth being completely opposite to the actual data after nine years of simulation. The Adams-Bashforth algorithm also showed some deviation from the actual data, but the difference was not as significant as that of the Euler algorithm. It is worth noting that smaller time step would of course lead to smaller errors, but the computational cost would be higher. Following image shows differences between results (blue) and data from NASA (orange). The Runge-Kutta plots closely track real data such that the resulting plot appears to show only single curve. Based on these results, implemented solver was declared as valid.

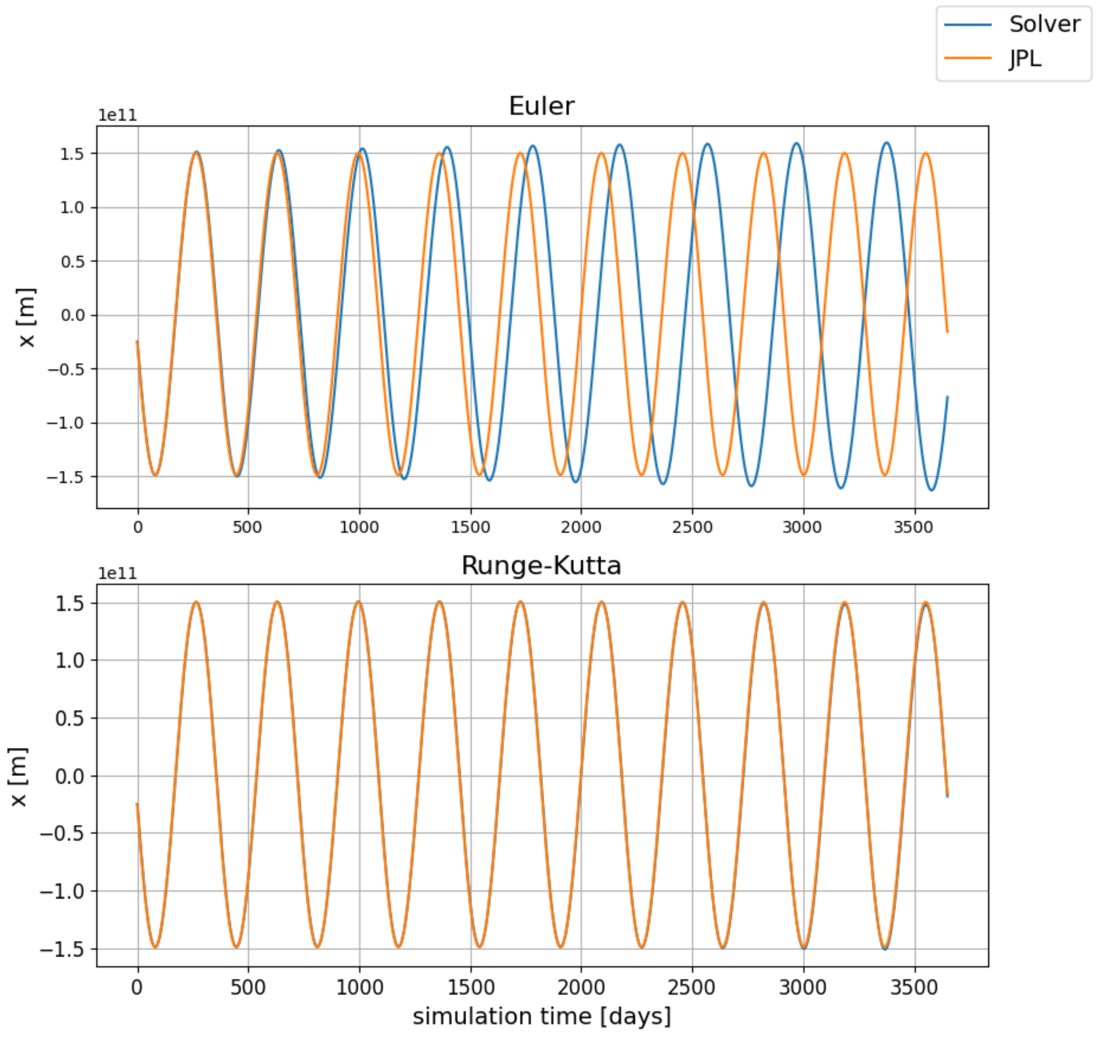


Figure 4.5: Results from Euler and Runge-Kutta simulations compared to data from NASA's Horizons interface. Simulation time is displyed on x axis in days and y axis shows first element of position vector of simulated body. While Euler shows serious deviations after only few years, Runge-Kutta is perfectly alligned with real data.

Figure 4.6 shows a plot of simulated orbits and further demonstrates inaccuracy of Euler method with selected time step, where Mercury was receding from Sun significantly and after only six orbits (in blue) reached orbit of Venus (in orange).

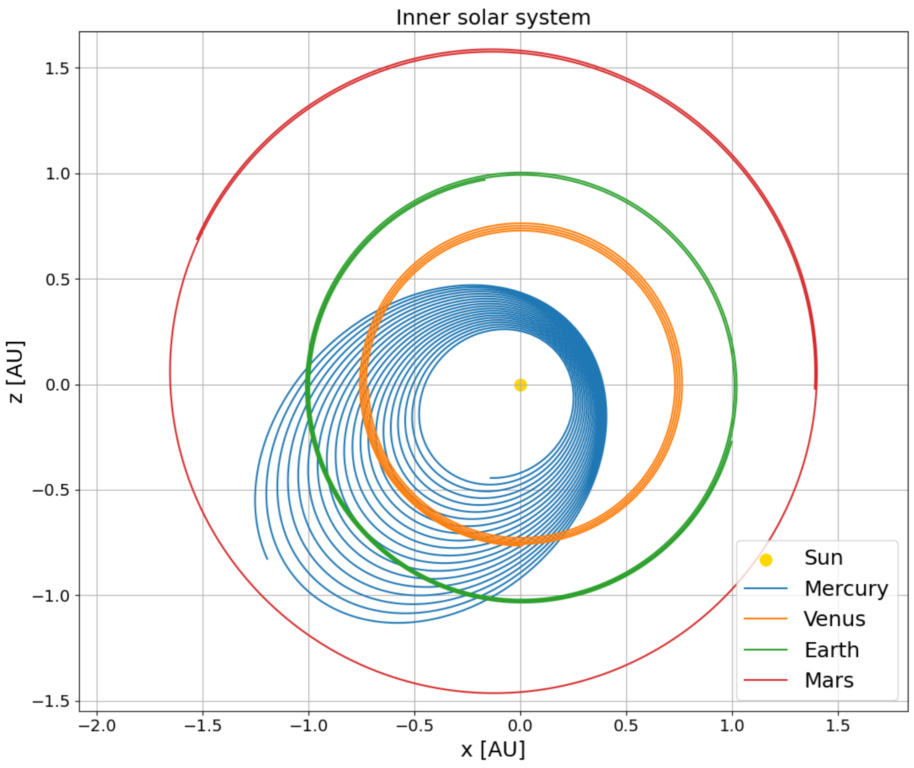


Figure 4.6: Mercury's unstable orbit in blue during simulation using Euler method. Orbits of Earth and Venus also show significant errors as can be seen by thickness of their plots.

Chapter 5

Experiments

This chapter discusses performed experiments with implemented solver. Some experiments are then compared to data from NASA.

5.1 Experiment 1: Red dwarf flyby

The aim of this experiment was to observe how the planets of the Solar system would react to a fictive star that was placed "above" Solar system between orbits of Jupiter and Mars and sent through the Solar system. The simulation was run to determine the effect of the star's gravitational force on the planets' orbits.

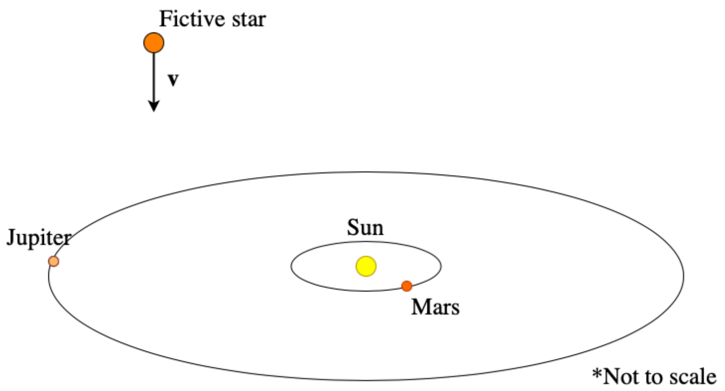


Figure 5.1: Approximate placement of fictive red dwarf star with its velocity vector shown.

Two experiments were performed with two different masses of fictive star. These are two extremes of known red dwarf masses and they are 0.6 and 0.08 solar masses, respectively. Velocity is that of Proxima Centaury, closest star to our Sun, and it is equal to 22.2 km s^{-1}. Exact position vector can be found in file `massive_red_dwarf.json` from A. Position of other planets are from 1st of January 2000 and were retrieved from [7].

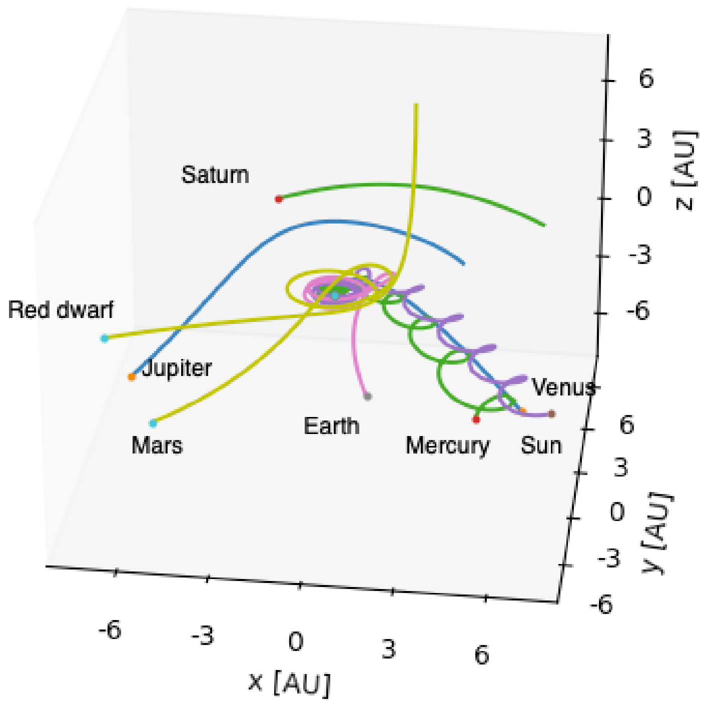


Figure 5.2: Disrupted Solar system after flyby.

As expected, Solar system was completely disrupted and broken by flyby of such massive star. While Mercury and Venus stayed on similar orbits around the Sun, Earth's orbit was significantly changed. Mars, Jupiter and Saturn were "stolen" by passing star and started orbiting around it. Neptune and Uranus were catapulted into deep space and became so-called rogue planets. A rogue planet, also known as a free-floating planet, is a planetary-mass object that orbits through space without being gravitationally bound to a star.

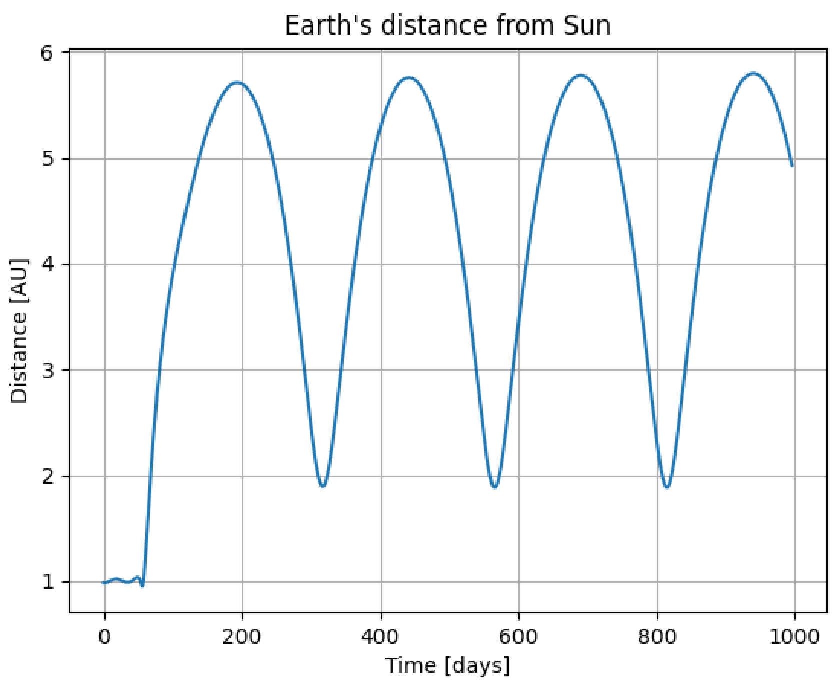


Figure 5.3: New orbit of Earth shown as a distance from the Sun. Each period represents single orbit. Aphelion and perihelion can be seen as maximum and minimum in single period.

As can be seen on figure 5.3, Earth's new orbit has high eccentricity, with perihelion and aphelion at ~2 AU and 6.7 AU, respectively. Period of orbit was also shortened to 249 days. Event like this would of course end our civilization. It is worth mentioning that results of such event could be completely different depending on position of planets during the event.

Second part of experiment involved lighter red dwarf star of 0.08 solar masses. This flyby was much less dramatic and all planets remained on orbit around the Sun with slightly changed orbits. Figure 5.4 show how orbits of inner planets have changed after flyby. While Mercury was not affected at all, Earth's eccentricity was slightly increased. Most significant changes happened to Mars - its aphelion was increased by almost 50 %.

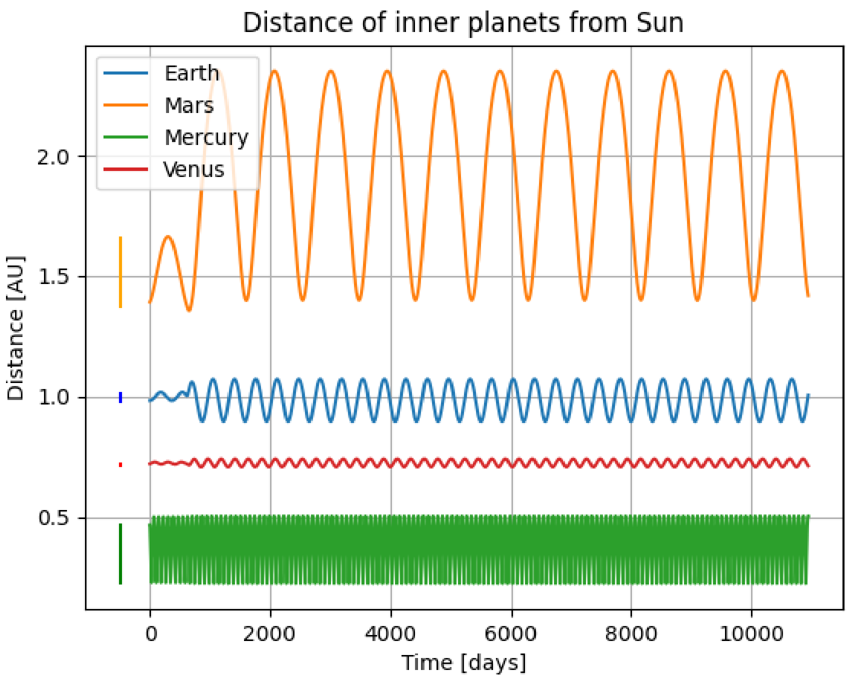


Figure 5.4: Vertical bars to the left of curves show present-day aphelion-perihelion interval of inner planets. Flyby event happened around day 1000 and its effects on planets can be seen as a change of aphelion and perihelion.

Paremeters of this simulation are following:

- Simulation time - 30 years
- Time step - 55 minutes
- Initial date - 1st of January, 2000
- Integration method - Runge-Kutta

5.2 Experiment 2: Close encounter with asteroid Apophis

In this experiment, Runge-Kutta simulator was used to simulate the solar system, including the asteroid Apophis, for a period of 6.7 years from 2023 to 2029. The time step used was 60 seconds, which provided a reasonable balance between simulation accuracy and computational time required.

Apophis is a near-Earth asteroid discovered in 2004. Initially it caused concern in the science community due to calculations showing a possibility of impact with planet Earth in 2029 or 2036. Apophis is approximately 370 meters in diameter and is classified as a potentially hazardous asteroid due to its size and orbit that brings it close to Earth periodically. Scientific observations and simulations of Apophis are important for understanding the potential risks of asteroid impacts and developing strategies for mitigating them. If it was to collide with Earth, then it could severely disrupt our civilization. More about asteroid Apophis can be found at [6].

During this simulation, it was found out that Apophis had a close approach to Earth in 2029. This simulation was also used to again test the accuracy of implemented simulation. I compared our simulated data with data provided by NASA and found that the simulation was off by about three million kilometers in predicting the closest approach distance of Apophis to Earth. Although this might sound like a significant error, it is relatively small when considering the scale of astronomical distances. Predicted closest approach distance was 3.2 million kilometers, while NASA's prediction was only 65,000 kilometers. This difference can be attributed to various factors, such as different model used by NASA's simulator, small inaccuracies in the input data. Smaller time step could be also used to increase precision.

To visualize and compare our simulated data with NASA's data, plots were created plots of the predicted positions of Apophis for each day during the period of the close approach. These plots showed the differences between predicted positions and NASA's predicted positions. Despite the simulation's inaccuracy, this experiment shows the usefulness of numerical simulation methods in predicting the behavior of complex systems such as the solar system. Simulation provided a reasonably accurate prediction of Apophis's position during its close approach to Earth. Furthermore, this experiment highlights the importance of verifying simulation results with real-world data to ensure their accuracy and reliability.

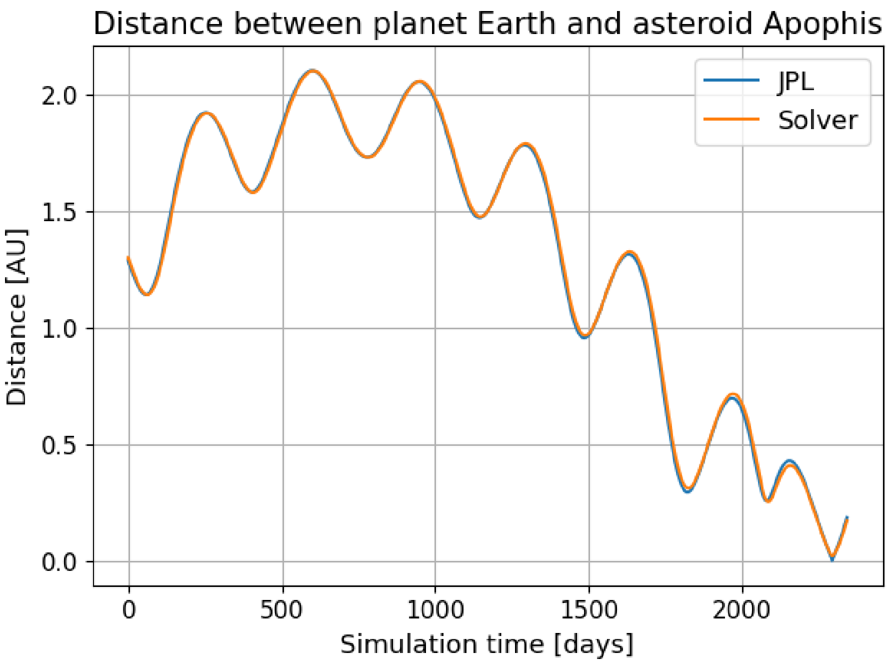


Figure 5.5: Distance between planet Earth and asteroid Apophis accoring to implemented simulator and NASA.

Predictions shown on figure 5.5 overlay nicely and it is hard to see the difference. Zoomed part of plot showing the close encounter event is shown on figure 5.6.

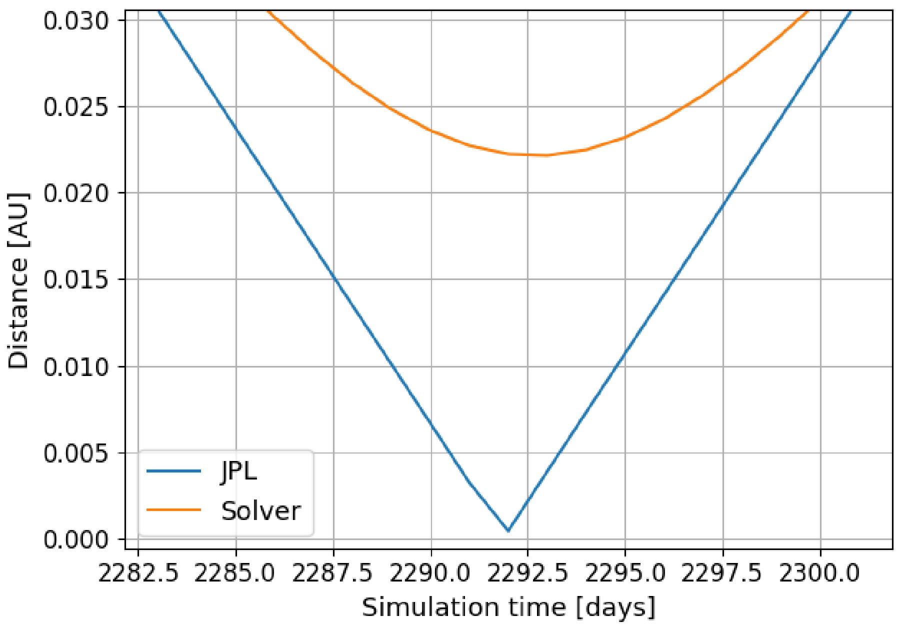


Figure 5.6: Difference can be clearly seen when zoomed into plot 5.5

Exact distance predicted by simulation is 0.022 AU, which is 329 100 kilometers while NASA's figure is 0.00043 AU which is 65 075 kilometers. This distance is smaller distance then orbits of some satellites. Experiment was repeated with 6 times smaller time step of 10 seconds, but results were not significantly different.

5.3 Experiment 3: n-body vs 2-body simulation

The purpose of this experiment was to compare the results obtained from a n-body simulation with those from a 2-body simulation. The Solar system was used as the system to be simulated. The 2-body simulation involved only the sun and the simulated object, while the n-body simulation included all the planets of the Solar system. Simulations were run for the same amount of time, and the results were compared.

Simulations were done with these settings:

- Simulation time - 2 years
- Time step - 60 seconds
- Initial date - 1st of January, 2000
- Integration method n-body - Runge-Kutta 4

First simulated object was planet Earth. On figure 5.8, distance between Earth in n-body and two-body simulation can be seen. Maximum deviation during simulated two years was 0.0042 AU, which is ~628311.057 km, about twice the distance between Earth and Moon. In astronomical scale, this is relatively small distance.

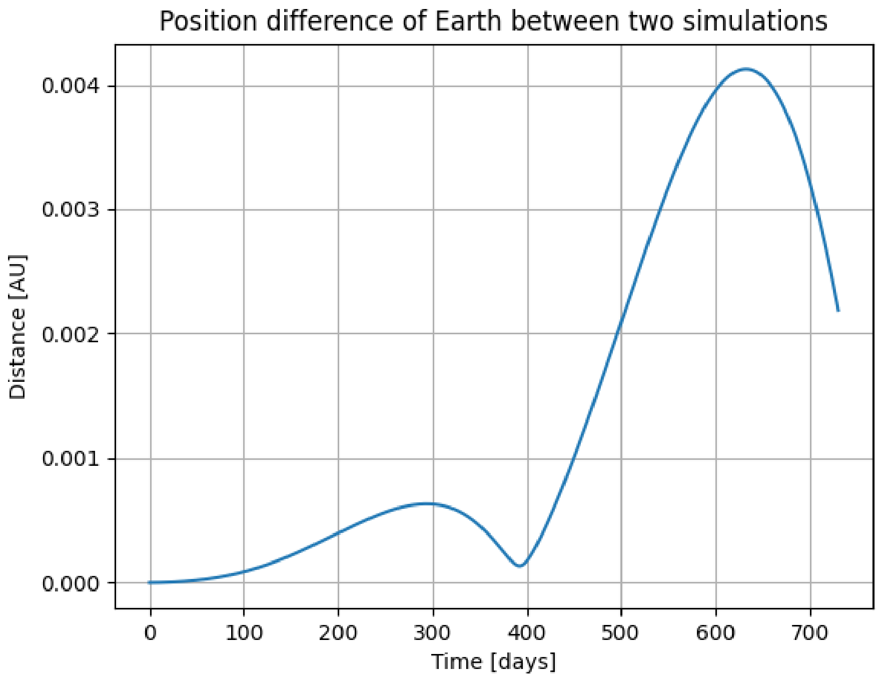


Figure 5.7: Position differences of Earth between 2 an n-body simulations.

Second simulated body was planet Mercury. It was chosen because of its close distance to the Sun, therefore differences should be more notable. Same figure as with Earth:

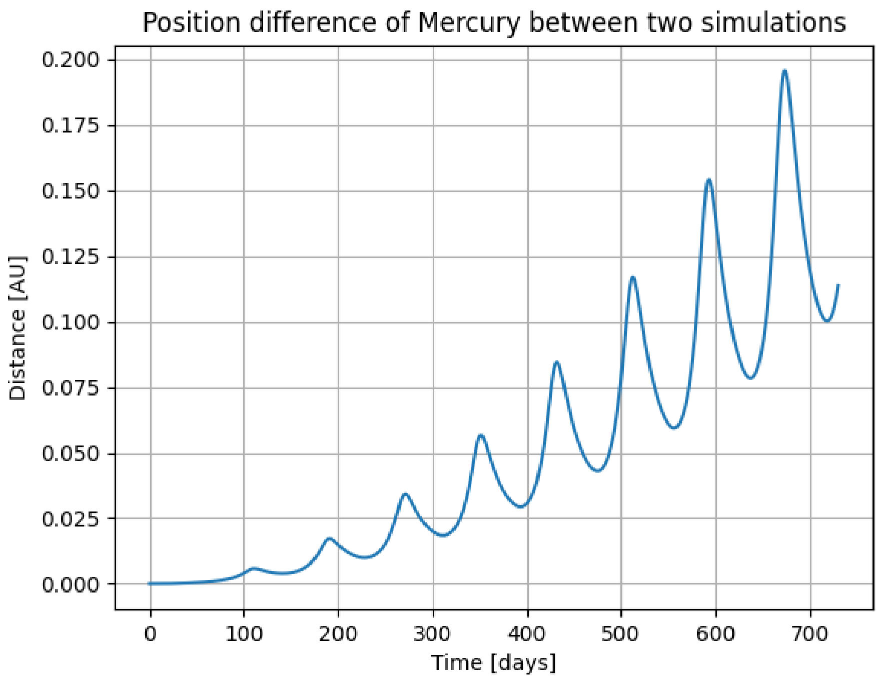


Figure 5.8: Position differences of Mercury between 2 an n-body simulations.

In simulated two years, Mercury made 8 orbits around the Sun. These corresponds to spikes that can be seen on the plot. With increasing time of simulation, the error is increasingly growing. It seems like that time step of 60 seconds makes the Mercury-Sun system unstable, and therefore smaller time step should be chosen for this algorithm.

5.4 Experiment 4 - Resonance of Jupiter's moons

In this experiment, simulation of Jupiter and its three closest moons Ganymede, Europa and Io was performed to confirm that the three moons are indeed in 4:2:1 resonance. This means, that while Ganymede makes single orbit, Europa makes exactly two and Io exactly four orbits. Integration method used was Runge-Kutta for high precision. Time step was set to sixty seconds and simulation time was just over 7 days. Results can be seen on following figure:

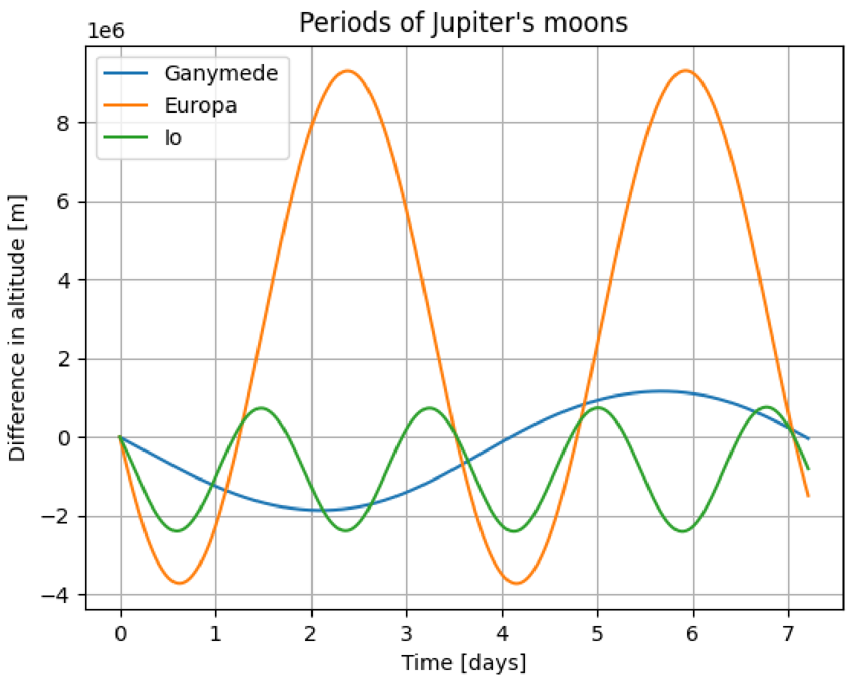


Figure 5.9: Differences in altitude of Jupiter's moon over simulated time. Individual orbits and their 4:2:1 resonance can be seen as periods of graphed curves.

Orbital resonance can be visually confirmed from results in figure 5.9. Resonance was also confirmed numerically from simulated data in `jupyter notebook` tool.

Chapter 6

Future work

While the current simulation program provides satisfactory results, there are still improvements that could be made to the code to increase its capabilities. One potential area of future work is rewriting the simulation program in a faster language, such as C++. While Python is a powerful and versatile language, it can be computationally expensive for large-scale simulations. By rewriting the program in a faster language, the simulations could be run more efficiently and effectively, allowing for more complex and large-scale simulations. Also vectorization could be utilized, instead of using object-oriented approach to access object data. Another area of future work could be to allow for more simulations to run in parallel or at least queue them. Currently, the simulation program can only run one simulation at a time. By enabling the program to run multiple simulations simultaneously, time of users could be saved. Furthermore, the ability to customize the algorithms used in the simulation could be added. For example, the order of the Runge-Kutta or Adams-Bashforth methods could be selected, allowing for greater control over the accuracy and efficiency of the simulation. This could be particularly useful for simulations that require a high degree of accuracy or are computationally expensive. Lastly, another improvement could be the ability to rewind the animation of the simulation by manipulating the animation slider. Currently, the animation can only be played forward, making it difficult to analyze specific moments in the simulation. By allowing for the ability to rewind the animation, users could easily go back to specific moments and analyze the motion of the celestial objects in greater detail. Also GUI tool for creating input JSON files could be created, as writing them manually is tedious and error-prone.

In conclusion, while the current n-body problem simulation program is functional, there are still improvements that can be made to increase its accuracy, efficiency, and capabilities. By rewriting the program in a faster language, allowing for more simulations to run in parallel, enabling greater customization of algorithms, and allowing for the ability to rewind the animation, the program could become an even more powerful tool for understanding the motion of celestial objects.

Chapter 7

Conclusion

The objective of this thesis was to explore the concepts of orbital mechanics and n-body problem, and develop a program that enables users to simulate general n-body problem and visualize it through an animation. Users can define the simulation by choosing an algorithm, step size, and simulation time. Simulating two-body problems is also possible. Simulated systems are defined by JSON files, which contain initial conditions needed for simulation. Results from simulator can be exported as csv file and further manipulated or analyzed. In addition to the numerical output in the form of CSV files, the application also includes an animation feature that displays the simulated system throughout the simulation time. This feature provides a visual way for the user to verify the output of the simulator and to observe expected events. To validate created simulator, highly accurate ephemeris data were extracted from NASA website to use as benchmark data. After the validation process, multiple experiments were conducted. These included a comparison between two-body and n-body problems, a simulation of the asteroid Apophis close approach to Earth in 2029 and finally flyby of a fictive star through the Solar system. While the application produces correct data, there is room for improvements, both in terms of features and performance. Additional algorithms could be incorporated, and more animation controls could be added to enhance the user experience.

Bibliography

[1] Chobotov, V. A. *Orbital Mechanics.* 3rd ed. American Institute of Aeronautics and Astronautics, Inc., 2002.

[2] Goldstein, H., Poole, C. and Safko, J. *Classical Mechanics.* 3rd ed. Addison Wesley, 2008. ISBN 978-3-540-5670-0.

[3] Hairer, E., Norsett, S. P. and Wanner, G. *Solving Ordinary Differentail Equation I.* Springer, 2008. ISBN 978-3-540-5670-0.

[4] Hayes, W. Available at: `https://www.cs.toronto.edu/~wayne/research/thesis/msc/node24.html`.

[5] Howard, C. D. *Orbital mechanics for engineering students.* Amsterdam: Elsevier Butterworth-Heinemann, 2005. ISBN 0-7506-6169-0.

[6] Jet Propulsion Laboratory. *PSmall-Body Database.* Available at: `https://ssd.jpl.nasa.gov/tools/sbdb_lookup.html#/?sstr=Apophis&view=OPC`.

[7] Jet Propulsion Laboratory. *Horizons Web-Interface* [Online resource]. 2022. Available at: `https://ssd.jpl.nasa.gov/horizons.cgi`.

[8] Kleppner, D. and Kolenkow, R. *An introduction to mechanics.* 2nd ed. Cambridge University Press, 2014. ISBN 978-0-521-19811-0.

[9] Matplotlib Development Team. *Matplotlib 3.7.1 documentation* [Online documentation]. 2023. Available at: `https://matplotlib.org/stable/index.html#`.

[10] NASA. *Planetary fact sheet.* 2023. Available at: `https://nssdc.gsfc.nasa.gov/planetary/factsheet/index.html`.

[11] Nave, C. R. *HyperPhysics.* 2005–2023. Available at: `http://hyperphysics.phy-astr.gsu.edu/hbase/kepler.html`.

[12] Newton, I. *The Principia.* University of California Press, 1999.

[13] Nilsson, D. *Law of Universal gravitation visualisation.* Available at: `https://upload.wikimedia.org/wikipedia/commons/0/0e/NewtonsLawOfUniversalGravitation.svg`.

[14] Peringer, P. *Modelování a simulace - IMS Studijní opora* [Online resource]. 2021. Available at: `https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FIMS-IT%2Ftexts%2Fopora-ims.pdf&cid=11453`.

[15] The Qt Company Ltd. *Qt for Python* [Online documentation]. 2023. Available at: `https://doc.qt.io/qtforpython/`.

[16] Thornton, S. T. and Marion, J. B. *Classical Dynamics of Particles and Systems.* 5th ed. Brooks/Cole, 2014. ISBN 0-534-40896-6.

[17] Vitásek, E. *Numerické metody.* Nakladatelství technické leteratury, 1987.

Appendix A

SD card content hierarchy

```
root/
 ├─src/
 │  ├─animation.py
 │  ├─body.py
 │  ├─controller.py
 │  ├─csv_reader.py
 │  ├─csv_writer.py
 │  ├─figure_widget.py
 │  ├─main_window.py
 │  ├─main.py
 │  ├─simulation_dialog.py
 │  ├─main.py
 │  ├─system_importer.py
 │  └─utils.py
 ├─forms/
 │  ├─mainWindow.ui
 │  └─simulationDialog.ui
 ├─simulation/
 │  ├─simulator_2body.py
 │  ├─simulator_base.py
 │  ├─simulator_nbody_adams_bashfort.py
 │  ├─simulator_nbody_euler.py
 │  └─simulator_nbody_rk4.py
 ├─ui/
 │  ├─ui_mainWindow.py
 │  └─ui_simulationDialog.py
 └─presets/
    ├─2body/
    │  ├─2body_solar_system.json
    │  └─earth_and_sun.json
    └─nbody/
       ├─apophis_2023.json
       ├─jupiter_and_moons.json
       ├─massive_red_dwarf.json
       └─solar_system.json
```