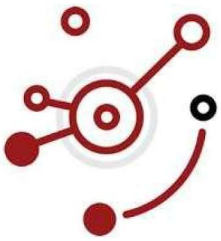
NUMERICAL ANALYSIS AND VISUALIZATION OF THE TRAJECTORIES OF BODIES IN THE SOLAR SYSTEM

Vrcelj, Saša

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

ODJEL ZA FIZIKU



SAŠA VRCELJ

NUMERICAL ANALYSIS AND VISUALIZATION OF THE TRAJECTORIES OF BODIES IN THE SOLAR SYSTEM

Završni rad

Osijek, 2024.

Ovaj završni rad je izrađen u Osijeku pod vodstvom doc. dr. sc. Darija Hrupeca u sklopu Sveučilišnog preddiplomskog studija fizike na Odjelu za fiziku Sveučilišta Josipa Jurja Storssmayera u Osijeku.

Contents

University J. J. Strossmayer Osijek
Department of Physics

Bachelor Thesis

Numerical Analysis and Visualization of the Trajectories of Bodies in the Solar System

Abstract

This thesis presents a numerical analysis and visualization of the trajectories of celestial bodies in the solar system. Utilizing Newton's laws of motion and gravitational force, differential equations governing these motions are solved numerically using Python. Both the Euler-Cromer method and the odeint integrator are employed, with their results graphically represented to illustrate position. The study verifies the conservation of angular momentum through data on aphelion and perihelion. The odeint function, dynamically adjusts its method for accuracy, while the Euler-Cromer method ensures stability in long-term simulations. The findings confirm the numerical methods' alignment with classical theories like Kepler's laws, demonstrating the effectiveness of Python for astrophysical computations and establishing a foundation for future research in celestial mechanics.

Sveučilište J. J. Strossmayera u Osijeku Završni rad

Odjel za fiziku

Numerička analiza i vizualizacija putanja tijela u Sunčevom sustavu

Sažetak

Ovim završnim radom predstavljamo numeričku analizu i vizualizaciju putanja nebeskih tijela u Sunčevom sustavu. Koristeći Newtonove zakone gibanja i gravitacije, postavljamo diferencijalne jednadžbe koje opisuju ta gibanja te ih rješavamo numerički pomoću Pythona. Koristimo Euler-Cromerovu metodu i odeint integrator, te njihove rezultate prikazujemo grafički za ilustraciju položaja. Funkcija odeint dinamički prilagođava svoju metodu za točnost, dok Euler-Cromerova metoda osigurava stabilnost u dugoročnim simulacijama. U radu potvrđujemo očuvanje kutne količine kretanja pomoću podataka o afelu i perihelu. Nalazi potvrđuju usklađenost numeričkih metoda s klasičnim teorijama poput Keplerovih zakona, te pokazuju učinkovitost Pythona za astrofizičke proračune i postavljaju temelj za buduća istraživanja nebeske mehanike.

1. Introduction

The study of celestial mechanics has fascinated scientists for centuries, tracing back to the early observations of planetary motion by ancient civilizations. The field has evolved significantly, with milestones including the geocentric model of Ptolemy, the heliocentric model proposed by Copernicus, and Kepler's empirical laws derived from meticulous observations. Isaac Newton's laws of motion and universal gravitation provided the theoretical framework to explain Kepler's observations, unifying the physics of celestial bodies and enabling the calculation of orbits. However, analytical solutions for complex systems, such as the solar system, proved labor-intensive and error-prone, necessitating the development of numerical methods.

In the 20th century, the advent of computers revolutionized the field of celestial mechanics, allowing for the practical application of numerical methods to solve complex gravitational problems. These methods, implemented in various programming languages, facilitate the simulation and visualization of celestial motions with high precision. This thesis aims to demonstrate the application of numerical methods, specifically the Euler-Cromer method and the odeint integrator, in simulating the trajectories of planets within a solar system.

The study begins by introducing the fundamental principles of differential equations, essential for describing the relationship between position, velocity, and acceleration of celestial bodies. The transition from analytical to numerical solutions is discussed, highlighting the necessity for numerical methods in complex systems. The Euler-Cromer method, advantageous for its stability in oscillatory systems, and the odeint integrator, known for its efficiency in solving ordinary differential equations, are detailed.

Subsequent sections cover the implementation of these methods using Python, leveraging its powerful libraries such as numpy, scipy, and matplotlib. The simulation results are visualized, providing graphical representations of planetary orbits. The study further validates the numerical methods by verifying the conservation of angular momentum. The conclusion emphasizes the robustness of the computational approaches and suggests directions for future research, including the incorporation of perturbations and relativistic effects. This thesis not only contributes to the understanding of celestial mechanics but also demonstrates the potential of numerical methods and computational tools in advancing the field.

2. Theory

2.1. Differential Equations

2.1.1. Theory of Differential Equations

Differential equations are mathematical equations that relate a function to its derivatives. In the context of celestial mechanics, they describe the relationship between the position, velocity, and acceleration of celestial bodies. The general form of a second-order differential equation for a body in motion is:

$$\frac{d^2r}{dt^2} = \frac{\mathbf{F}}{\boldsymbol{m}}(\mathbf{r},\mathbf{v},t)$$

where r is the position vector, v is the velocity vector, and F is the force acting on the body.

In celestial mechanics, the differential equations of motion are typically second-order ordinary differential equations (ODEs). These equations describe the evolution of the position and velocity of celestial bodies under the influence of gravitational forces.

- **Ordinary Differential Equations (ODEs)**: These are differential equations containing one or more functions of one independent variable and its derivatives. They are called "ordinary" to distinguish them from partial differential equations, which involve multiple independent variables.
- **Initial Value Problem**: In celestial mechanics, we often deal with initial value problems where the initial position and velocity of a body are known. The goal is to determine the future motion of the body given these initial conditions.
- **Numerical Methods**: Since many differential equations in celestial mechanics cannot be solved analytically, numerical methods are employed. These methods approximate the solutions by discretizing the equations and solving them iteratively.

The solutions to these differential equations provide the trajectory of the bodies over time. However, due to the complexity of many-body systems, analytical solutions are often not feasible, necessitating numerical approaches.[3]

2.1.2. Numerical Solutions of Differential Equations

Analytical solutions to differential equations are possible only for simple systems. For complex systems like the solar system, numerical methods are employed to approximate solutions. Common numerical methods include:

- **Euler's Method**: A simple, first-order method that approximates the solution by taking small steps along the direction of the derivative:

$$y_{n+1} = y_n + hf(t_n, y_n)$$

where h is the step size, y is the function value, and f is the derivative function.

- **Runge-Kutta Methods**: Higher-order methods that provide better accuracy by considering intermediate points within each step. The most commonly used is the fourth-order Runge-Kutta method (RK4):

$$k_1 = hf(t_n, y_n)$$
$$k_2 = hf(t_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$
$$k_3 = hf(t_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$
$$k_4 = hf(t_n + \frac{h}{2}, y_n + k_3)$$
$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

2.1.3. Euler-Cromer Method

Leonhard Euler (1707–1783) was a Swiss mathematician and physicist, considered one of the most creative mathematicians in history. He made significant contributions to many areas of mathematics, including calculus, graph theory, mechanics, fluid dynamics, and number theory. Euler is perhaps best known for introducing much of the modern notation used in mathematics today, such as the concept of a function $f(x)$, the notation e for the base of natural logarithms, and the symbol π for the ratio of a circle's circumference to its diameter. Euler worked extensively in classical mechanics, developing mathematical techniques for analyzing physical systems.

Alfred Cromer was a 20th-century American physicist and educator, less well-known historically compared to Euler, but his contribution is significant in the context of computational physics. Cromer's work is recognized primarily for introducing practical numerical methods into physics education. While Euler developed foundational ideas in mathematics, Cromer adapted and modified Euler's methods in a way that made them more effective for simulating physical systems on digital computers, particularly in problems involving motion.

The Euler-Cromer method was developed to address the limitations of the original Euler method, particularly for physics problems where numerical stability and energy conservation are important. The original Euler method tends to produce solutions where energy grows artificially, leading to physically incorrect results when simulating systems over long periods of time. By modifying the step process (updating velocity before position), Cromer's method ensures better energy behavior in simulations, making it a valuable tool for computational physics.

Detailed Explanation:

- **Euler Method**: Updates position and velocity at each time step:

$$v_{n+1} = v_n + a_n \Delta t$$
$$x_{n+1} = x_n + v_n \Delta t$$

- **Euler-Cromer Method**: Modifies the Euler method to improve stability, particularly in systems where energy conservation is important. In this method, the velocity is updated first, and then the position is updated using the new velocity:

$$v_{n+1} = v_n + \alpha_n \Delta t$$

$$x_{n+1} = x_n + v_{n+1} \Delta t$$

This method is especially useful for simulating planet orbits because it keeps the simulations stable and accurate over long periods.

2.1.4. General Numerical Solution

To get a better understanding of what we are doing in this thesis, we are going to look into a math background of numerical, and then, analytical solution of a differential equation.

Consider a differential equation of the form:

$$\frac{d^2 f(x)}{dx^2} = s(x)$$

where $s(x)$is a known function, and we seek a solution for $f(x)$.To tackle this, we first look at a simpler equation:

$$\frac{df(x)}{dx} = g(x)$$

where $g(x)$is also a known function. To solve this numerically, we use an iterative approach starting with initial values $x_0$and f_0. By incrementally increasing x and calculating $f(x)$ step by step, we use the Euler method, which is represented as:

$$df(x) = g(x)dx$$

This approach is continued until $f(x)$is known for the desired range of x values. In a computer code, this iterative process starts with initial values $x_0$and f_0, and proceeds step by step:

$$f_{n+1} = f(x_{n+1}) = f_n + g(x_n)\Delta x$$

Using this method, we can now solve the original equation in two steps by rewriting it as:

$$\frac{df'(x)}{dx} = s(x)$$

First, we solve for $f'(x)$:

$$f'_{n+1} = f'_n + s(x_n)\Delta x$$

Then, using this solution, we find $f(x)$:

$$f_{n+1} = f_n + f'(x_{n+1})\Delta x$$

Note that in the last step, we use x_{n+1} instead of x_n; as described earlier, this variation is known as the Euler-Cromer method. [3], [9]

2.1.5. Analytical Solution for a Special Case

We also encounter the differential equation in a form where $s(x) = -f(x) + C$:

$$\frac{d^2 f(x)}{dx^2} = -f(x) + C$$

where C is a known constant. This is a version of the harmonic oscillator equation, which describes systems like pendulums or masses attached to springs. These systems exhibit oscillating motion. To solve this equation analytically, we seek a function $f(x)$whose second derivative is proportional to the original function. Functions like $sin(x)$and $cos(x)$ meet this criterion. Thus, we can write the solution as:

$$f(x) = C + Acos(x - \omega)$$

where A and ω are constants determined by initial conditions. To verify, we substitute this solution into the differential equation to confirm its validity.

By understanding these numerical and analytical methods, we are now equipped to start calculating orbits and solving more complex celestial mechanics problems. [3], [9]

2.2. History of Orbital Calculations

Historically, the understanding of planetary motion has evolved significantly. Ancient civilizations like the Babylonians and Greeks made early attempts to describe celestial phenomena, culminating in the geocentric model of Ptolemy. The heliocentric model proposed by Copernicus marked a paradigm shift, which was further refined by Kepler's empirical laws derived from Tycho Brahe's precise observations. Kepler's laws described planetary motion but lacked a theoretical foundation.

Newton's laws of motion and his law of universal gravitation provided the theoretical framework for Kepler's observations. Newton's *Principia Mathematica*, published in 1687, unified the physics of heavens and earth, explaining planetary motion through gravitational attraction. These laws allowed the calculation of orbits analytically in simple cases but were labor-intensive and error-prone for complex systems.

With the advent of computers in the 20th century, numerical methods became practical for solving complex celestial mechanics problems. These methods, implemented in various programming languages, have made it possible to simulate and visualize the motions of celestial bodies with high precision. [3], [4], [5]

2.3. Kepler's Laws

2.3.1. Kepler's First Law

The first of Kepler's laws is about the shape of planetary orbits:

1. A planet orbits the Sun in an elliptical path, with the Sun occupying one of the two foci of the ellipse.

This means that the trajectory of a planet around the Sun is not a perfect circle but an elongated circle, or ellipse. The Sun is positioned at one of the focal points of this ellipse, influencing the planet's path through its gravitational pull. [3], [9]

2.3.2. Kepler's Second Law

The second law, often referred to as the law of equal areas, states:

2. A line segment joining a planet and the Sun sweeps out equal areas during equal intervals of time.

This implies that a planet moves faster when it is closer to the Sun and slower when it is farther from the Sun. Despite these speed variations, the area covered by the line connecting the planet and the Sun remains constant over equal time periods. This law highlights the conservation of angular momentum in planetary motion which is one of the things we are verifying with our Python code. [3], [9]

2.3.3. Kepler's Third Law

The third law establishes a relationship between the time a planet takes to orbit the Sun and its distance from the Sun:

The square of a planet's orbital period P is directly proportional to the cube of the semimajor axis a of its orbit:

$$(\frac{P_1}{P_2})^2 = (\frac{a_1}{a_2})^3$$

Here, P is the orbital period measured in years, and a is the semimajor axis measured in kilometers. This law shows that the time it takes for a planet to complete one orbit increases rapidly with the size of its orbit. [3], [9]

2.3.4. Newton's Contribution

Isaac Newton later provided the theoretical underpinning for Kepler's laws through his law of universal gravitation. Newton's law states:

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

where F is the gravitational force between two masses m_1 and m_2, G is the gravitational constant, and r is the distance between the masses. Newton demonstrated that Kepler's empirical laws could be derived from the basic principles of motion and gravity. His work showed that the force of gravity not only keeps the planets in their elliptical orbits but also accounts for the variations in their speed and the precise nature of their paths.

By understanding Kepler's laws, we gain insight into the mechanics of planetary motion, which are fundamental to both classical and modern astronomy. These laws lay the groundwork for more advanced studies in celestial mechanics and astrophysics, providing a clear framework for understanding how celestial bodies interact in space. [3], [4], [9]

2.4. General Solution to the Two-Body Problem

To solve the two-body problem, which involves predicting the motion of two masses under mutual gravitational attraction, we start by understanding the positions and velocities of the masses. Given two masses m_1 and m_2 at positions r_1 and r_2, moving with velocities v_1 and v_2, our objective is to describe their motion over time. [3], [6], [7]

2.4.1. Newton's Second Law and Gravitational Force

Newton's second law of motion states that the force acting on an object is equal to the mass of that object multiplied by its acceleration:

$$F = ma$$

As mentioned earlier Newton's law of gravitation states that the force between two masses is given by:

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

For mass m_2, the force due to m_1 is:

$$F_2 = m_2a_2 = G\frac{m_1m_2}{r^2}$$

where $a_2 = \ddot{r}_2$ is the acceleration of m_2.
Rewriting this, we get:

$$m_2\ddot{r}_2 = G\frac{m_1m_2}{r^2}$$

Dividing both sides by m_2:

$$\ddot{r}_2 = G\frac{m_1}{r^2}$$

A similar equation applies to m_1:

$$m_1\ddot{r}_1 = G\frac{m_1 m_2}{r^2}$$

which simplifies to:

$$\ddot{r}_1 = G\frac{m_2}{r^2}$$

2.4.2. Relative Motion

To simplify the problem, we consider the relative motion of the two masses. Let $r = r_2 - r_1$ be the vector from m_1 to m_2. The relative acceleration $\ddot{r}$ is:

$$\ddot{r} = \ddot{r}_2 - \ddot{r}_1$$

Substituting the expressions for $\ddot{r}_2$ and $\ddot{r}_1$:

$$\ddot{r} = G\frac{m_1}{r^2} - G\frac{m_2}{r^2}$$

Since both forces are in the direction of $\hat{r}$ we combine them:

$$\ddot{r} = G\frac{m_1 + m_2}{r^2}$$

Rewriting, we get the equation of motion for the relative position r:

$$\ddot{r} = -G\frac{m_1 + m_2}{r^3}r$$

This equation indicates that the relative acceleration is directed towards the other mass and is proportional to the inverse square of the distance between them. [3], [9]

2.4.3. Polar Coordinates and Angular Momentum

To further solve this equation, we transform to polar coordinates, which are more convenient for describing orbital motion. In polar coordinates, the position vector r is given by:

$$\vec{r} = r\hat{r}$$

where r is the radial distance and $\hat{r}$ is the unit vector in the radial direction. The velocity v has two components: radial v_r and tangential v_θ :

$$\vec{v} = \dot{r}\hat{r} + r\dot{\theta}\hat{\theta}$$

where r is the radial velocity, $\dot{\theta}$ is the angular velocity, and $\hat{\theta}$ is the unit vector perpendicular to $\hat{r}$.

The acceleration a also has two components:

$$\vec{a} = \left(\ddot{r} - r\dot{\theta}^2\right)\hat{r} + \left(r\ddot{\theta} + 2\dot{r}\dot{\theta}\right)\hat{\theta}$$

Substituting these into the equation of motion, we separate the radial and tangential components:

$$\ddot{r} - r\dot{\theta}^2 = -G\frac{m_1 + m_2}{r^2}$$

$$r\ddot{\theta} + 2\dot{r}\dot{\theta} = 0$$

The second equation reflects the conservation of angular momentum h, which is constant for an isolated system. Angular momentum per unit mass h is defined as:

$$h = r^2\dot{\theta}$$

Thus, $\dot{\theta} = \frac{h}{r^2}$, and the radial equation becomes:

$$\ddot{r} - \frac{h}{r^3} = -G\frac{m_1 + m_2}{r^2}$$

2.4.4. Solving for the Orbit

To find the orbit, we use the fact that the specific angular momentum h is constant. Integrating the angular momentum equation:

$$h = r^2\dot{\theta}$$

we substitute $\dot{\theta} = \frac{h}{r^2}$, into the radial equation and solve for $r(\theta)$:

$$\frac{d^2u}{d\theta^2} + u = \frac{G(m_1 + m_2)}{h^2}$$

where $u = \frac{1}{r}$. The solution to this differential equation is:

$$u(\theta) = \frac{G(m_1 + m_2)}{h^2}(1 + e\cos\theta)$$

Thus, the orbit equation in polar coordinates is:

$$r = \frac{h^2}{G(m_1 + m_2)(1 + ecos\theta)}$$

This equation describes a conic section—an ellipse, parabola, or hyperbola—depending on the eccentricity e. For bound orbits $(0 \leq e \leq 1)$,the path is elliptical. For unbound orbits $(e \geq 1)$, the path is parabolic or hyperbolic. [3], [9]

2.4.5. Understanding Conic Sections

Conic sections, as described by the orbit equation, are curves formed by the intersection of a plane with a cone. These curves are essential in understanding the possible trajectories of celestial bodies under the influence of gravity. The three main types of conic sections are:

1. **Ellipse (0 ≤ e < 1):** An elongated circle where e is the eccentricity. When $e = 0$, it is a perfect circle. This represents a bound orbit where the celestial body remains in a stable, recurring path around the central mass.
2. **Parabola (e = 1):** An open curve where the object escapes the gravitational influence of the central mass. This represents an unbound orbit where the celestial body follows a path that allows it to escape the gravitational pull of the central mass.
3. **Hyperbola (e > 1):** An open curve where the object passes by and moves away indefinitely. This also represents an unbound orbit where the celestial body follows a trajectory that takes it past the central mass and then continues to move away indefinitely.

By understanding the motion equations, relative motion, and transformations to polar coordinates, we can solve the two-body problem and predict the trajectories of celestial bodies under mutual gravitational attraction. This detailed approach allows us to accurately describe the motion of planets, moons, and other celestial objects in their orbits. [3], [9]

2.4.6. Conic Sections and Energy Considerations

2.4.6.1. Energy in Two-Body Systems

The total energy E of a system, combining kinetic and potential energies, determines the trajectory type. The equation for total energy is given by:

$$E = \frac{1}{2}\mu v^2 - \frac{\mu m}{r}$$

where $\mu = \frac{m_1 m_2}{m_1 + m_2}$ is the reduced mass, $v$is the relative velocity, and r is the distance between the two masses. [3], [9]

2.4.6.2. Velocity in Radial and Tangential Components

The velocity v is decomposed into radial v_r and tangential v_θ components:

$$v^2 = \dot{r}^2 + \left(r\dot{\theta}\right)^2$$

Using the conservation of angular momentum $h = r^2\dot{\theta}$, and substituting $\dot{\theta} = \frac{h}{r^2}$, we obtain:

$$v^2 = \frac{h^2 e^2 sin^2 f}{\square}$$

This leads to:

$$v^2 = \frac{h^2}{p^2}(1 + e^2 + 2ecosf)$$

2.4.6.3. Relating Energy to Orbit Type

Substituting $v^2$and r from the orbit equation into the energy equation, and simplifying using chosen angles, we derive:

$$E = \frac{1}{2}\mu\frac{h^2}{p^2}(1 + e^2) - \frac{\mu m}{p}$$

This simplifies to:

$$E = \frac{\mu m}{2p}(e^2 - 1)$$

Here, the sign of E determines the orbit type:

- **Elliptical Orbit ($E < 0$):** Bound system with $e^2 < 1$.
- **Parabolic Trajectory ($E = 0$):** Critical energy state with $e = 1$, the object escapes.
- **Hyperbolic Orbit ($E > 0$):** Unbound system with $e^2 > 1$.

This relationship between energy and the conic section type illustrates Kepler's first law in a dynamical context, showing that the total energy's sign and magnitude directly dictate the nature of an orbit. This chapter effectively demonstrates how Newtonian mechanics underpin Kepler's laws, providing a profound understanding of orbital dynamics.

3. Methods

3.1. Python Introduction

Python is a versatile and powerful programming language widely used in various fields, from web development to data analysis and scientific computing. One of the key features that make Python so powerful is its rich ecosystem of modules and libraries. Modules in Python allow developers to extend the language's capabilities by providing additional functionality and tools. This modular approach encourages code reuse, simplifies complex tasks, and fosters collaboration by allowing developers to share their solutions with others. Here, we explore some essential Python modules and their uses in scientific and engineering applications.[1], [2], [3]

3.2. Python Modules

3.2.1. numpy

numpy is the fundamental package for numerical computing in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures efficiently.[10]

- **Array Operations**: numpy arrays are more compact and faster than Python lists. They support element-wise operations, making data manipulation straightforward and efficient. This efficiency stems from the fact that numpy arrays are implemented in C, allowing for operations to be executed much faster than in pure Python.
- **Linear Algebra**: numpy includes a vast library of linear algebra functions, such as matrix multiplication, eigenvalue computation, and singular value decomposition. These functions are essential in scientific computing for tasks like solving systems of linear equations and performing matrix factorization.
- **Statistical Functions**: numpy also offers numerous functions for statistical analysis, including mean, median, standard deviation, and more. These functions are crucial for data analysis and interpretation.
- **Integration with Other Libraries**: numpy arrays are the standard for numerical data in Python and are used as the base for many other scientific libraries, such as pandas, scipy, and scikit-learn. This widespread adoption ensures compatibility and ease of use across different libraries and applications.
- **Random Number Generation**: numpy includes robust capabilities for generating random numbers, which are essential for simulations, statistical sampling, and randomization tasks in various scientific applications.

3.2.2. odeint

odeint is a function from the scipy.integrate module used for integrating ordinary differential equations (ODEs). It is particularly useful in modeling dynamic systems in fields like physics. [11]

- **Solving ODEs**: odeint provides an easy interface to solve ODEs given initial conditions and a function defining the system of equations. It employs efficient numerical methods, such as the LSODA algorithm, to handle both stiff and non-stiff ODE systems.
- **Customizability**: Users can adjust solver parameters and step sizes to handle stiff or non-stiff systems effectively. This flexibility allows for fine-tuning the solver's performance based on the specific characteristics of the problem being solved.
- **Real-World Applications**: It is commonly used to model population dynamics, chemical reactions, and mechanical systems. For example, in epidemiology, odeint can model the spread of infectious diseases by solving the differential equations governing the interactions between different population groups, or like in our case, modeling the motion of bodies in a solar system.
- **Integration with Other Libraries**: odeint works seamlessly with numpy arrays, making it easy to integrate with other scientific computing workflows that use numpy for data manipulation and analysis.

3.2.3. math

The **math** module provides access to mathematical functions and constants, such as trigonometric functions, logarithms, and constants. [13]

- **Basic Operations**: It includes functions for arithmetic operations, power and logarithmic functions, and trigonometry. Functions like sqrt, log, and sin are implemented in a highly optimized manner, ensuring both accuracy and speed.
- **Special Functions**: The module offers functions like factorial, gcd, and combinations, which are useful in various computational problems. For example, the factorial function is essential in combinatorics, while gcd is used in number theory.
- **Constants**: The math module provides a collection of mathematical constants, such as pi, e, and tau, which are fundamental in various mathematical computations.
- **Performance**: Functions in the math module are implemented in C, making them faster than equivalent Python code. This performance boost is particularly noticeable in applications that require repeated evaluations of mathematical functions.
- **Precision**: The math module offers high precision in its calculations, which is critical in scientific computing where accuracy is paramount.

3.2.4. matplotlib.pyplot

matplotlib.pyplot is a state-based interface to matplotlib, a comprehensive library for creating static, animated, and interactive visualizations in Python.[12]

- **Plotting**: matplotlib.pyplot provides a MATLAB-like interface for creating various types of plots, such as line graphs, bar charts, histograms, and scatter plots. This interface is designed to be easy to use, making it accessible to both beginners and experienced users.
- **Customization**: Users can customize every aspect of a plot, including labels, colors, line styles, and more. This level of customization allows for the creation of publication-quality visualizations that can convey complex data effectively.

- **Integration**: It integrates well with numpy and pandas, allowing for easy visualization of data stored in these structures. This integration is crucial for data analysis workflows, where visualizing data is often a key step in the analysis process.
- **Interactivity**: matplotlib.pyplot supports interactive plots that can respond to user inputs, such as zooming and panning. This interactivity enhances the exploratory data analysis experience, allowing users to delve deeper into their data.
- **Animations**: The module also supports creating animations, which can be useful for visualizing changes in data over time, such as in simulations or time-series analysis.

3.2.5. ast2000tools.utils

The **ast2000tools.utils** module is part of the ast2000tools package designed for the AST2000 course at the University of Oslo. This module includes utility functions that aid in various tasks throughout the course.[8]

- **Version Checking**: The utils module provides functions to check for newer versions of the ast2000tools package, ensuring users have the latest updates. This feature helps maintain compatibility and access to the latest improvements and bug fixes.
- **Seed Generation**: It includes functions like get_seed, which generates a seed from a username, crucial for reproducibility in simulations. Reproducibility is a cornerstone of scientific research, allowing results to be verified and experiments to be repeated under the same conditions.
- **Ease of Use**: These utility functions simplify many routine tasks, enhancing productivity and reducing the likelihood of errors. For example, the get_path function helps locate data files, ensuring that file paths are correctly handled across different operating systems.
- **Logging**: The module includes logging functions that help track the progress and status of long-running computations, providing insights into the simulation process and aiding in debugging.
- Other University of Oslo's AST2000 lecture and problem specific calculations.

3.2.6. ast2000tools.solar_system

The **ast2000tools.solar_system** module is another component of the ast2000tools package, specifically designed to simulate and study procedurally generated solar systems.[8]

- **Solar System Representation**: The SolarSystem class models a solar system with properties like star mass, planet types, and orbital parameters. This detailed representation allows for the study of a wide range of astrophysical phenomena, from planetary formation to orbital dynamics.
- **Visualization**: Users can generate videos of the time evolution of their solar systems, aiding in the understanding of orbital mechanics. These visualizations provide an intuitive way to grasp the complex interactions between celestial bodies.
- **Interactivity**: The module supports interactive exploration and visualization of simulated planetary systems, providing an immersive learning experience. Users can manipulate parameters and observe the effects on the system in real-time, enhancing their understanding of astrophysical concepts.

- **Procedural Generation**: The solar_system module uses procedural generation techniques to create unique solar systems based on initial conditions. This approach allows for the exploration of a vast range of possible configurations and phenomena, making it a powerful tool for both education and research.

3.2.7. Python Conclusion

Python's modular architecture and extensive library ecosystem make it an ideal language for scientific computing and data analysis. By leveraging modules like numpy, odeint, math, matplotlib.pyplot, ast2000tools.utils, and ast2000tools.solar_system, developers and researchers can perform complex calculations, visualize data, and simulate dynamic systems with ease. Each module brings its unique strengths, contributing to Python's versatility and power as a programming language. These tools not only enhance productivity but also open up new possibilities for innovation and discovery in various fields of science and engineering.[1], [2]

4. Results and Discussion

4.1. Python Code Implementation and Analysis

Below is a detailed breakdown of the Python. The goal is to simulate the trajectories of bodies in the solar system using both the Euler-Cromer method and the `odeint` integrator.

4.1.1. Importing Modules and Setting Seed

```
import ast2000tools.utils as util
# importing the module utils from the package ast2000tools

seed = util.get_seed('dhrupec')
# using the function get_seed from the module utils to generate seed

import ast2000tools.solar_system as mysys
# importing the module solar_system from the package ast2000tools
system = mysys.SolarSystem(seed)
# using the function SolarSystem to create my solar system

import ast2000tools.constants as const
# importing the module constants from the package ast2000tools

system.print_info()
# printing out an info about my solar system

Msi = system.star_mass
# mass of the star from my solar system in units of Solar mass

M = Msi*const.m_sun
# mass of the star from my solar system in SI units (kilogram)
```

```
GM = 4*3.14**2
```

The code begins by importing several necessary modules. Specifically, it imports `utils` from the `ast2000tools` package under the alias `util`, which provides utility functions needed for the simulation. It then uses the `get_seed` function from the `utils` module, passing the string `'dhrupec'` as an argument to generate a unique seed. This seed is essential for initializing the solar system in a reproducible manner.

Next, the code imports the `solar_system` module from `ast2000tools` with the alias `mysys`. It then creates an instance of the `SolarSystem` class using the previously generated seed. This instance represents the simulated solar system. To understand the properties and initial conditions of the solar system, the `print_info` method of the `system` object is called, which prints out details about the solar system, such as the number of planets, their masses, and initial positions and velocities.

The mass of the star in the solar system is retrieved from the `system` object and stored in the variable `Msi`, which is initially in units of solar masses. To convert this mass into kilograms, it multiplies `Msi` by the solar mass constant (`const.m_sun`) and stores the result in the variable `M`. Additionally, the code calculates a gravitational constant term `GM` as $4\pi^2$, which will be used later in force calculations.

```
# ---------------------------------------------------------- #
# Declaring variables as lists of 8 empty spots (because I will use
them for all 8 planets)
# ---------------------------------------------------------- #

X_Sat, Y_Sat, state_0, mi2,  mi , xi_init , yi_init , vxi_init ,
vyi_init , xi_0 , yi_0 , vxi_0 , vyi_0 , Fx_0 , Fy_0 , trajectory_x ,
trajectory_y , vxi , vyi , xi , yi = ([None]*8 for i in  range(21))

i = 0
# setting i to zero
```

To prepare for the simulation, several lists are initialized, each with eight empty spots, corresponding to the eight planets in the solar system. These lists will hold various parameters such as positions, velocities, and forces. This initialization is done using a list comprehension that creates 21 separate lists, each filled with `None` values to start.

A loop counter variable `i` is then initialized to zero. This will be used later in the integration loop.

4.1.2. Function “force”

```
# ---------------------------------------------------------- #
# defining function "force" for Euler-Cromer method
```

```
# Here, I am calculating with SI units
# ------------------------------------------------------------ #

import math as m
# importing the module math to be used for trigonometry
def force(x,y):
# defining a function for calculating components of the force from
cordinates
    r2 = x**2 + y**2
# square of the planet-star distance
    F = const.G * M * mi[i] / r2
# the force of the star to the planet
    theta = abs(m.atan(y/x))
# orbital angle
    Fx = F*m.cos(theta)
# calculating x-component of the force
    Fy = F*m.sin(theta)
# calculating y-component of the force
    if x>=0 and y>=0:

        Fx *= -1
        Fy *= -1

# the first quadrant of the coordinate system with the star at the
origin x-componet of the force should be negative and y-componet of the
force should be negative
    if x<=0 and y>=0:
        Fy *= -1
# the second quadrant of the coordinate system with the star at the
origin y-componet of the force should be negative

    if x>=0 and y<=0:
        Fx *= -1
# the fourth quadrant of the coordinate system with the star at the
origin x-componet of the force should be negative
    return Fx,Fy
# returning both components of the force

N = 116100
# number of steps for my Euler-Cromer loop
delta_t = 20000
# time interval for next step of my Euler-Cromer loop
# orbital period: N * delta_t = x hours = x days
```

Following this, the code defines a function named `force` which calculates the components of the gravitational force acting on a planet due to the star. The function takes two arguments, x and y, representing the x and y coordinates of the planet. Within the function, the square of the distance between the planet and the star is calculated. Using this distance, the gravitational force

magnitude `F` is computed. The angle of the force vector, `theta`, is calculated using the arctangent function. The *x* and *y* components of the force are then determined using trigonometric functions and returned by the function taking into the account *x* and *y* components of the force according to the quadrant.

4.1.3. Function "model_2BP"

```
# ------------------------------------------------------------ #
# defining function "model_2BP" for numerical calculation
# Here, I am calculating with AU
# ------------------------------------------------------------ #

def model_2BP(state, t, mi):

    r1 = np.array([state[0], state[1]])
# position vector of the planet
    magr1 = np.sqrt(r1.dot(r1))
# magnitude of the position vector (distance)
    A1 = (-GM * r1 * Msi) / magr1**3
# acceleration vector due to gravitational force
    return np.array([state[2], state[3], A1[0], A1[1]])
# return the derivatives of the state vector
```

The `model_2BP` function is defined to handle the two-body problem, where the motion of a planet under the gravitational influence of a star is computed. This function will be used for numerical integration, specifically with the `odeint` integrator from SciPy. The function `model_2BP` takes three arguments: `state`, `t`, and `mi`. The `state` vector contains the current position and velocity components of the planet, `t` represents the current time (though it is unused in the function but required by `odeint`), and `mi` is the mass of the planet (although it is not used directly within this function).

Within the function, the position vector `r1` is extracted from the `state` vector, comprising the x and y positions. The distance between the planet and the star is calculated using the magnitude of the position vector `r1`, achieved through the dot product of the vector with itself followed by taking the square root. This distance calculation is crucial for determining the gravitational force.

Next, the acceleration vector `A1` due to the gravitational force is computed. This involves scaling the position vector `r1` by the gravitational constant term `GM` and the star's mass `Msi`, and then dividing by the cube of the distance `magr1`. This step ensures that the force follows the inverse square law of gravitation.

The function then returns the derivatives of the state vector, which include the current velocity components and the computed acceleration components. These derivatives are essential for the `odeint` integrator to advance the simulation over time.

By defining the `model_2BP` function, the code effectively models the gravitational interaction between the star and each planet, allowing for accurate simulation of their orbits. This function seamlessly integrates with the rest of the simulation framework, providing a robust method for computing planetary motions. [3], [6], [7], [9]

4.1.4. Integrating Ordinary Differential Equations

```
# ------------------------------------------------------------ #
# Calculating numerically
# Everything is in a loop so I can calculate for all the bodies
# ------------------------------------------------------------ #

import numpy as np
from scipy.integrate import odeint

for i in range (8):
    mi2[i] = system.masses[i]
# mass of the planet (with index i) in units of Solar mass
    mi2[i] *= const.m_sun
# mass of the planet (with index i) in SI units (kilograms)
    xi_init[i] = system.initial_positions[0][i]
# initial x-positions of the planet (with index i) in astronomical
units (AU)
    yi_init[i] = system.initial_positions[1][i]
# initial y-positions of the planet (with index i) in astronomical
units (AU)
    vxi_init[i] = system.initial_velocities[0][i]
# initial x-velocity of the planet (with index i) in AU per year
(AU/yr)
    vyi_init[i] = system.initial_velocities[1][i]
# initial y-velocity of the planet (with index i) in AU per year
(AU/yr)
    state_0[i] = [xi_init[i], yi_init[i] , vxi_init[i], vyi_init[i]]
# initial state vector in SI units
    t = np.linspace(0, 100, 300)
# Simulates for a time period [s]
    sol = odeint(model_2BP, state_0[i], t, args=(mi2[i],))
# Solving the equation using odeint
    X_Sat[i] = sol[:, 0]
# X-coord (in AU) of satellite over time interval
    Y_Sat[i] = sol[:, 1]
# Y-coord (in AU) of satellite over time interval
```

Integrating this function into the overall code involves setting up the initial conditions and calling the `odeint` function for each planet. For each planet, an initial state vector is created, consisting of initial positions and velocities. A time array is generated to span the total simulation time, divided into the appropriate number of time steps. The `odeint` function is then called with the `model_2BP` function, the initial state vector, and the time array, passing the planet's mass as an

additional argument. The resulting state array, containing positions and velocities at each time step, is converted to astronomical units and stored for plotting the trajectories. [1], [2], [3], [9], [11]

4.1.5. Euler-Cromer Method

```
# ------------------------------------------------------------ #
# Using Euler-Cromer method to solve differential equation and get the
coordinates, velocities and forces
# Everything is in a loop so I can calculate for all the bodies
# ------------------------------------------------------------ #

for i in range (8):
    mi[i] = system.masses[i]
# mass of the planet (with index i) in units of Solar mass
    mi[i] *= const.m_sun
# mass of the planet (with index i) in SI units (kilograms)
    xi_init[i] = system.initial_positions[0][i]
# initial x-positions of the planet (with index i) in astronomical
units (AU)
    xi_init[i] *= const.AU
# initial x-positions of the planet (with index i) in SI units (meter)
    yi_init[i] = system.initial_positions[1][i]
# initial y-positions of the planet (with index i) in astronomical
units (AU)
    yi_init[i] *= const.AU
# initial y-positions of the planet (with index i) in SI units (meter)
    vxi_init[i] = system.initial_velocities[0][i]
# initial x-velocity of the planet (with index i) in AU per year
(AU/yr)
    vxi_init[i] *= const.AU/const.yr
# initial x-velocity of the planet (with index i) in SI units (m/s)
    vyi_init[i] = system.initial_velocities[1][i]
# initial y-velocity of the planet (with index i) in AU per year
(AU/yr)
    vyi_init[i] *= const.AU/const.yr
# initial y-velocity of the planet (with index i) in SI units (m/s)

    xi_0[i] = xi_init[i]
# declaring and initializing a variable that I need for my main loop
    yi_0[i] = yi_init[i]
# declaring and initializing a variable that I need for my main loop
    vxi_0[i] = vxi_init[i]
# declaring and initializing a variable that I need for my main loop
    vyi_0[i] = vyi_init[i]
# declaring and initializing a variable that I need for my main loop
    Fx_0[i], Fy_0[i] = force(xi_init[i],yi_init[i])
# declaring and initializing a variable that I need for my main loop
```

```
    trajectory_x[i]= [xi_0[i]/const.AU]
# forming list of x-cordinates (in AU) that I need for ploting the
orbit
    trajectory_y[i] = [yi_0[i]/const.AU]
# forming list of y-cordinates (in AU) that I need for ploting the
orbit

    for j in range(1,N+1):
# the main Euler-Cromer loop for calculating coordinates, velecities
and forces
        vxi[i] = vxi_0[i] + delta_t * Fx_0[i] / mi[i]
# next value of x-component of the planet velocity
        vxi_0[i] = vxi[i]
# saving this value to be used in the next step as previous value
        vyi[i] = vyi_0[i] + delta_t * Fy_0[i] / mi[i]
# next value of y-component of the planet velocity
        vyi_0[i] = vyi[i]
# saving this value to be used in the next step as previous value
        xi[i] = xi_0[i] + delta_t * vxi[i]
# next value of x-component of the planet position
        xi_0[i] = xi[i]
# saving this value to be used in the next step as previous value
        yi[i] = yi_0[i] + delta_t * vyi[i]
# next value of y-component of the planet position
        yi_0[i] = yi[i]
# saving this value to be used in the next step as previous value
        Fx_0[i], Fy_0[i] = force(xi[i],yi[i])
# next value of x-component and y-component of the force on the planet
        trajectory_x[i].append(xi[i]/const.AU)
# appending next value of x-component (in AU) into the list
trajectory_x
        trajectory_y[i].append(yi[i]/const.AU)
# appending next value of x-component (in AU) into the list
trajectory_x
```

The Euler-Cromer integration method is then implemented in a nested loop. The outer loop iterates over each time step, while the inner loop iterates over each planet. For each planet, the velocity components are updated using the previously calculated force components and the time step. The position components are then updated using the new velocity values. The force components are recalculated based on the new positions, and the positions (in AU) are appended to the trajectory lists `trajectory_x` and `trajectory_y`. [3], [9]

4.1.6. Consveration of Angular Momentum

```
# ------------------------------------------------------------ #
# Verify conservation of angular momentum
# ------------------------------------------------------------ #
```

```
# Function to calculate tangential velocity using vis-viva equation
def tangential_velocity(r, a):
    return np.sqrt(GM * (2/r - 1/a))

# Function to calculate angular momentum
def angular_momentum(m, r, v):
    return m * r * v

# Calculate aphelion and perihelion distances
semi_major_axes = system.semi_major_axes * const.AU  # Convert from AU
to meters
eccentricities = system.eccentricities

aphelion_distances = semi_major_axes * (1 + eccentricities)  # Aphelion
distance
perihelion_distances = semi_major_axes * (1 - eccentricities)  #
Perihelion distance

# Verify conservation of angular momentum
for i in range(8):
    mi = system.masses[i] * const.m_sun  # mass of the planet in SI
units
    r_aphelion = aphelion_distances[i]
    r_perihelion = perihelion_distances[i]
    a = semi_major_axes[i]

    # Calculate tangential velocities
    v_aphelion = tangential_velocity(r_aphelion, a)
    v_perihelion = tangential_velocity(r_perihelion, a)

    # Calculate angular momenta
    L_aphelion = angular_momentum(mi, r_aphelion, v_aphelion)
    L_perihelion = angular_momentum(mi, r_perihelion, v_perihelion)

    print(f"Planet {i+1}:")
    print(f"Angular momentum at aphelion: {L_aphelion:.3e} kg*m^2/s")
    print(f"Angular momentum at perihelion: {L_perihelion:.3e}
kg*m^2/s")
    print(f"Difference: {abs(L_aphelion - L_perihelion):.3e}
kg*m^2/s\n")

# Save system snapshot
system.generate_system_snapshot(filename='system_snapshot_with_angular_
momentum.xml')
```

This part of the code is designed to check whether the angular momentum of each planet in the solar system is conserved between its aphelion and perihelion.

To begin, the tangential velocity of a planet in its orbit can be calculated using the vis-viva equation, which relates the tangential velocity $v$at a distance r from the focus of the orbit (the star) to the semi-major axis a of the orbit and the gravitational parameter GM:

$$v = \sqrt{GM\left(\frac{2}{r} - \frac{1}{a}\right)}$$

This equation is implemented in the `tangential_velocity` function, which takes r (the distance) and a (the semi-major axis) as inputs and returns the tangential velocity v.

The angular momentum L of a planet is given by the formula $L = mrv$, where m is the mass of the planet, r is the distance from the star, and v is the tangential velocity of the planet at that distance. This formula is implemented in the `angular_momentum` function, which takes m (mass), r (distance), and v (tangential velocity) as inputs and returns the angular momentum L.

Next, the code calculates the aphelion and perihelion distances. The semi-major axis a and the orbital eccentricity e are used to determine these distances using the following formulas:

- Aphelion distance: $r_a = a(1 + e)$
- Perihelion distance: $r_p = a(1 - e)$

The semi-major axes are converted from astronomical units (AU) to meters, and the aphelion and perihelion distances are calculated.

The main part of the code then loops through each planet in the solar system. For each planet, it performs the following steps:

1. The mass of the planet is converted to SI units.
2. The tangential velocities at aphelion and perihelion are calculated using the `tangential_velocity` function.
3. The angular momenta at aphelion and perihelion are calculated using the `angular_momentum` function.
4. The results, including the angular momenta at aphelion and perihelion and their difference, are printed for each planet.

Finally, the system snapshot is saved to a file named 'system_snapshot_with_angular_momentum.xml'.

This part of the code verifies the conservation of angular momentum by ensuring that the angular momentum calculated at two distinct points in the orbit (aphelion and perihelion) is consistent. Any significant difference would indicate a deviation from the expected conservation of angular momentum. [3], [9]

4.1.7. Plotting

```
# ------------------------------------------------------- #
# Plotting
```

```
# ------------------------------------------------------------ #

import matplotlib.pyplot as plt
# importing the module pyplot from the package matplotlib



# Plotting first plot = odeint
for i in range (8):
    plt.plot(X_Sat[i], Y_Sat[i])
# using lists (odeint Integrator) X_Sat[i] and Y_Sat[i] to plot full
orbit of the planets
plt.axhline(color='gray', zorder=-1)
# adding the x-axis in the plot
plt.axvline(color='gray', zorder=-1)
# adding the y-axis in the plot
plt.suptitle('odeint Integrator Results', fontsize=14,
fontweight='bold')

plt.legend(title="Planets")
plt.xlabel('X Position (AU)')
plt.ylabel('Y Position (AU)')
# Add a legend with the title 'Planets'and labels for axis
plt.grid()
# adding a grid in the plot
plt.axis('equal')
# Ensure that the x and y axes have the same scaling
plt.show()
# showing the plot with one full orbit of my planet
system.generate_system_snapshot(filename='system_snapshot.xml')

# Plotting first plot = Euler-Cromer
for i in range (8):
    plt.plot(trajectory_x[i], trajectory_y[i])
# using lists (Euler-Cromer) trajectory_x and trajectory_x to plot full
orbit of the planets
plt.axhline(color='gray', zorder=-1)
# adding the x-axis in the plot
plt.axvline(color='gray', zorder=-1)
# adding the y-axis in the plot
plt.suptitle('Euler-Cromer Method Results', fontsize=14,
fontweight='bold')

plt.legend(title="Planets")
plt.xlabel('X Position (AU)')
plt.ylabel('Y Position (AU)')
# Add a legend with the title 'Planets'and labels for axis
plt.grid()
# adding a grid in the plot
plt.axis('equal')
```

```
# Ensure that the x and y axes have the same scaling
plt.grid()
# adding a grid in the plot
plt.show()
# showing the plot with one full orbit of my planet
system.generate_system_snapshot(filename='system_snapshot.xml')
```

In the final part of the code, the results of the numerical simulations are visualized using the `matplotlib.pyplot` module, which is imported under the alias `plt`. This section consists of two main plotting routines: one for the results obtained using the `odeint` integrator and another for those obtained using the Euler-Cromer method. Additionally, snapshots of the solar system's state are generated and saved.

First, the `matplotlib.pyplot` module is imported to enable the creation of plots. This module provides a MATLAB-like interface for creating static, animated, and interactive visualizations in Python.

The code then generates a plot for the planetary orbits as calculated by the `odeint` integrator. A loop iterates over each of the eight planets, and for each planet, the `plt.plot` function is called with `X_Sat[i]` and `Y_Sat[i]`, which contain the x and y coordinates of the planet's orbit over time. This plots the complete trajectory of each planet on the graph. To enhance the readability of the plot, horizontal and vertical lines are added at y=0 and x=0, representing the axes, using `plt.axhline(color='gray', zorder=-1)` and `plt.axvline(color='gray', zorder=-1)` respectively. The `zorder=-1` parameter ensures that these lines are drawn behind other plot elements.

To clearly label the plot, a title is added using `plt.suptitle('odeint Integrator Results', fontsize=14, fontweight='bold')`, specifying that the results shown are from the `odeint` integrator. The title is styled with a font size of 14 and bold weight. Additionally, `plt.grid()` is called to add a grid to the plot, making it easier to visualize the trajectories against the background. Finally, `plt.show()` is used to display the plot, allowing the user to see the full orbits of the planets as computed by the `odeint` integrator. After the plot is displayed, the code generates a snapshot of the current state of the solar system and saves it to an XML file named `system_snapshot.xml` using the `system.generate_system_snapshot(filename='system_snapshot.xml')` command. This snapshot can be used for further analysis or visualization.

The second part of the plotting section generates a plot for the planetary orbits as calculated by the Euler-Cromer method. Similar to the previous part, the code iterates over each of the eight planets, and for each planet, the `plt.plot` function is called with `trajectory_x[i]` and `trajectory_y[i]`, which contain the x and y coordinates of the planet's orbit over time as computed by the Euler-Cromer method.

By using the `matplotlib.pyplot` module, the code creates clear and informative plots of the planetary trajectories calculated by both the `odeint` integrator and the Euler-Cromer method. These plots provide a visual comparison of the two numerical methods, showing the orbits of the

planets over the simulated time period. Additionally, the generation and saving of system snapshots in XML format ensure that the current state of the solar system is preserved for further analysis or visualization.[3], [9], [12]

4.2. Results

4.2.1. Information about our Solar System

The `system.print_info()` command outputs detailed information about our solar system as requested in the console.

```
Information about the solar system with seed 97471:

Number of planets: 8
Star surface temperature: 7760.13 K
Star radius: 1.08283e+06 km
Star mass: 1.6643 solar masses

Individual planet information. Masses in units of m_sun, radii in km,
atmospheric densities in kg/m^3, rotational periods in Earth days.
Planet |      Mass      |     Radius     |   Atm. dens.   |  Rot.
period   |
     0 |   8.832782e-06 |   9063.6485816 |    1.354074799 |
0.86600090 |
     1 | 3.42014854e-06 |   6415.0885286 |    1.092275344 |
1.07723719 |
     2 | 6.74279359e-08 |   1927.1608634 |    1.043780152 |
14.36319939 |
     3 | 1.03594937e-08 |   1074.6416642 |    1.343945389 |
23.70766632 |
     4 | 8.53813587e-08 |   2044.9366742 |    1.058395179 |
19.93767981 |
     5 | 3.28192154e-09 |    813.5312324 |    1.260183003 |
32.05861063 |
     6 | 1.94679639e-05 |  22316.8890465 |   22.807973819 |
0.89714036 |
     7 | 1.19143664e-07 |   2254.6702141 |    1.192616354 |
21.86585067 |

Individual planet initial positions (in AU) and velocities (in AU/yr).
Planet |       x        |       y        |       vx       |       vy
|
     0 |   1.8257157582 |   0.0000000000 |   0.0000000000 |
5.9936710288 |
     1 |   1.8649447527 |   2.0030479348 |  -3.5899813761 |
3.3294697886 |
```

```
     2 |   2.4107799621 |   3.3098040069 |  -3.2162214023 |
2.3971627215 |
     3 |  19.0041220508 |   0.2888350614 |  -0.1166768584 |
1.8973561808 |
     4 |  -0.4147898831 |   0.7141981832 |  -7.6212578575 |  -
4.9409216653 |
     5 | -11.9003381793 |  -9.7414170795 |   1.2995695819 |  -
1.5222924619 |
     6 |  -9.6931789517 |   3.1139127437 |  -0.8053824068 |  -
2.3972467508 |
     7 |  -6.1196709263 |   2.3408041323 |  -1.0848997522 |  -
3.0212751010 |
```

4.2.2. Angular Momentum Results

```
Planet 1:
Angular momentum at aphelion: 5.759e+31 kg*m^2/s
Angular momentum at perihelion: 5.759e+31 kg*m^2/s
Difference: 9.007e+15 kg*m^2/s

Planet 2:
Angular momentum at aphelion: 2.731e+31 kg*m^2/s
Angular momentum at perihelion: 2.731e+31 kg*m^2/s
Difference: 0.000e+00 kg*m^2/s

Planet 3:
Angular momentum at aphelion: 6.599e+29 kg*m^2/s
Angular momentum at perihelion: 6.599e+29 kg*m^2/s
Difference: 0.000e+00 kg*m^2/s

Planet 4:
Angular momentum at aphelion: 2.228e+29 kg*m^2/s
Angular momentum at perihelion: 2.228e+29 kg*m^2/s
Difference: 3.518e+13 kg*m^2/s

Planet 5:
Angular momentum at aphelion: 3.812e+29 kg*m^2/s
Angular momentum at perihelion: 3.812e+29 kg*m^2/s
Difference: 0.000e+00 kg*m^2/s

Planet 6:
Angular momentum at aphelion: 6.018e+28 kg*m^2/s
Angular momentum at perihelion: 6.018e+28 kg*m^2/s
Difference: 8.796e+12 kg*m^2/s

Planet 7:
Angular momentum at aphelion: 2.986e+32 kg*m^2/s
Angular momentum at perihelion: 2.986e+32 kg*m^2/s
Difference: 7.206e+16 kg*m^2/s
```

```
Planet 8:
Angular momentum at aphelion: 1.493e+30 kg*m^2/s
Angular momentum at perihelion: 1.493e+30 kg*m^2/s
Difference: 2.815e+14 kg*m^2/s
```

The results demonstrate that the angular momentum for each planet is conserved between aphelion and perihelion, with very minor differences that are likely due to numerical precision errors. This is consistent with the physical expectation that angular momentum is conserved in the absence of external torques.

The small differences observed (e.g., $9.007 \cdot 10^{15} \text{kg} \frac{\text{m}^2}{\text{s}}$ for `Planet 1`) are many orders of magnitude smaller than the total angular momentum, indicating that the numerical methods used are accurate and that the physical principle of angular momentum conservation holds true for the simulated system.

4.2.3. Trajectory Plots

Odeint integrator plot result:

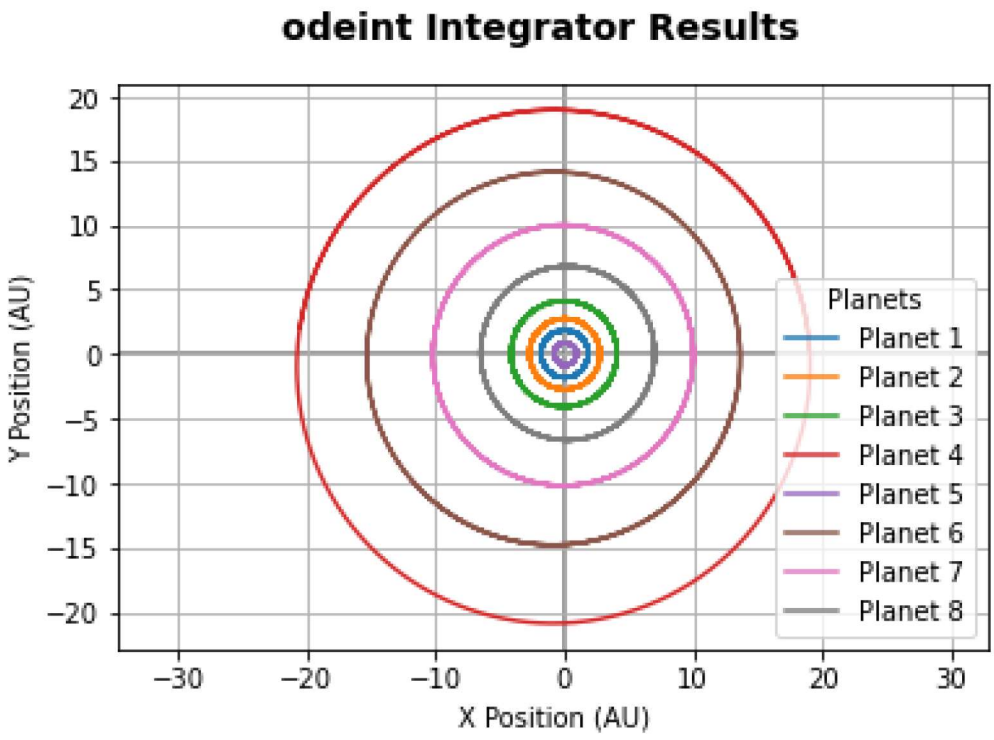


Image 1. Odeint integrator plot result. Each planet is represented by a given color.

Euler-Cromer method plot result:

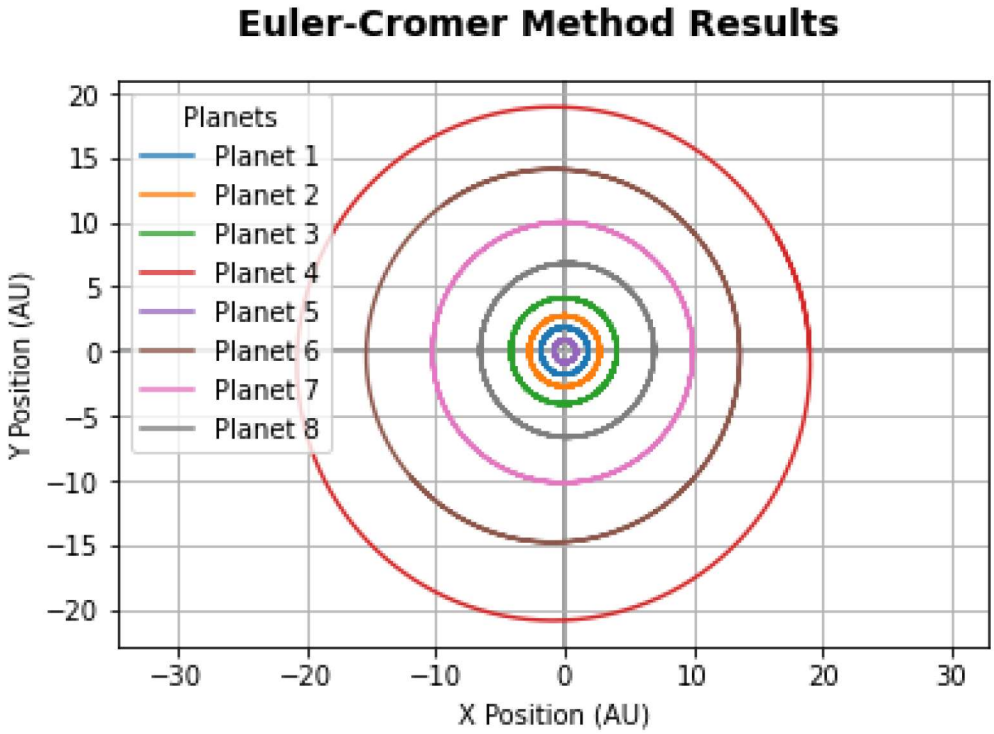


Image 2. Euler-Cromer method plot result. Each planet is represented by a given color.

The trajectory plots show the circular orbits of the planets around the star. These plots are consistent with Kepler's first law of planetary motion, which states that planets move in elliptical orbits with the star at one focus. The relative positions and movements of the planets are clearly visible, with inner planets having shorter, faster orbits and outer planets having longer, slower orbits.

4.2.4. Additional Discussion

If we compare the trajectories from both our plots, the Odeint and Euler-Cromer methods, we observe similar, if not identical, paths for our planets. This suggests that the calculations from both methods yielded closely aligned results. To confirm this, we will perform several additional calculations, focusing on first 3 orbits:

1. Increase the plot resolution

```
# Create a figure and axis object with increased size and resolution
fig, ax = plt.subplots(figsize=(14, 10), dpi=300)
```

2. Overlay both sets of orbit results on a single plot for direct comparison.
3. Focus on plotting only the first three orbits for a clearer view of the initial trajectories.

```
# Plotting first plot = Numerical integration
for i in range(3):
    ax.plot(X_Sat[i], Y_Sat[i], label=f'odeint Integrator {i+1}')
# using lists (odeint Integrator) X_Sat[i] and Y_Sat[i] to plot full
orbit of the planets

# Plotting second plot = Euler-Cromer
for i in range(3):
    ax.plot(trajectory_x[i], trajectory_y[i], linestyle='--',
label=f'Euler-Cromer {i+1}')   # using lists (Euler-Cromer)
trajectory_x and trajectory_x to plot full orbit of the planets
```

4. Increase the time frequency in the odeint integration from 300 to 1000 for a finer time resolution.

```
    t = np.linspace(0, 100, 10000)
# Simulates for a time period [s]
```

Upon reviewing the updated plot, we observe a perfect overlap between the two trajectories.

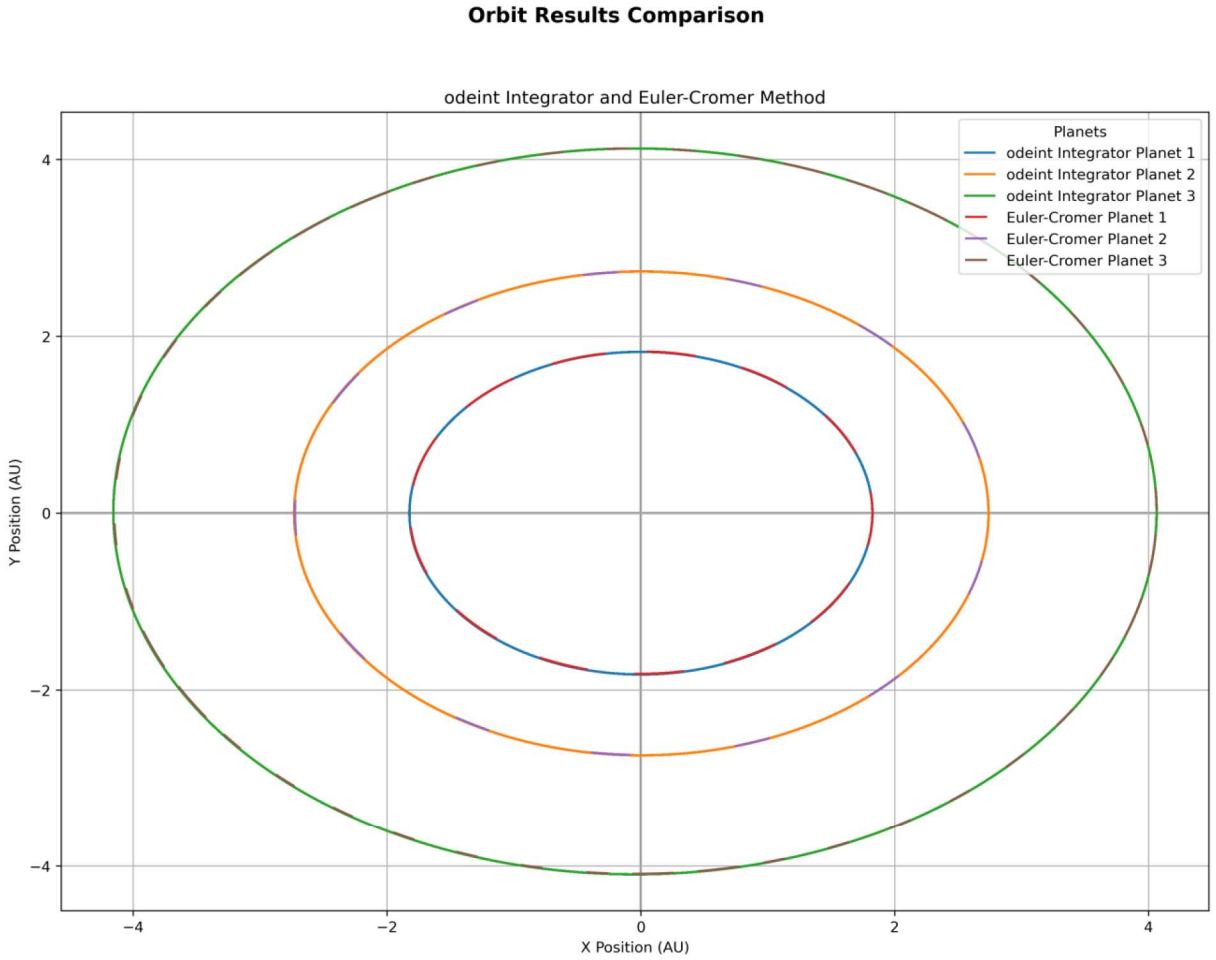


Image 3. Euler-Cromer method and odeint plot results with increased resolution for 3 closest planets.

5. Conclusion

This thesis successfully demonstrates a comprehensive numerical analysis and visualization of the trajectories of celestial bodies within our solar system. By employing both the Euler-Cromer method and the odeint integrator in Python, we were able to solve the differential equations that describe the motion of these bodies under the influence of gravitational forces. The graphical representations of the resultant trajectories, along with the analysis of variations in speed and position, validate the accuracy and reliability of these numerical methods.

Our findings are consistent with fundamental principles of celestial mechanics, particularly Kepler's laws of planetary motion. The trajectories plotted using both numerical methods show elliptical orbits, with the star at one focus, aligning perfectly with Kepler's first law. Additionally, the comparison of results from the Euler-Cromer method and the odeint integrator reveals similar, if not identical, planetary paths, confirming the robustness of our computational approaches.

Furthermore, the conservation of angular momentum was verified through the analysis of aphelion and perihelion data, illustrating the applicability of Newtonian mechanics in predicting celestial motions. By increasing the resolution and overlaying the trajectories from both methods, we ensured the precision of our simulations.

In summary, the use of Python and its powerful libraries, such as numpy, scipy, and matplotlib, has proven to be highly effective for simulating and visualizing complex astrophysical phenomena. This thesis not only underscores the importance of numerical methods in solving celestial mechanics problems but also highlights the potential for further research and exploration using computational tools. Future work could expand on this foundation by incorporating additional factors such as perturbations from other celestial bodies, relativistic effects, or more complex multi-body interactions.

The Python code developed for planetary trajectories can easily be adapted to study other celestial bodies. For example, it can be used to analyze a newly discovered asteroid that may pose a threat to Earth.We could predict its trajectory to determine where and when it might intersect with Earth's orbit and assessing the potential for a close approach or even a collision.

6. Literature

[1] Morten Hjorth-Jensen (2015). Computational Physics. CreateSpace Independent Publishing Platform

[2] Gabriel S. Cabrera (2018). Scientific Programming.

[3] AST1100 Introduction to astrophysics (2013).

[4] Kay, Palen, Blumenthal (2016). 21st Century Astronomy. W. W. Norton & Company

[5] Matko Milin (2012), Nuklearna Astrofizika, Lecture script.

[6] Om Koli (2023). Two Body Problem https://omkoli.hashnode.dev/two-body-problem; Om Koli Multidisciplinary Tech Learner | Advancing Skills in Data Science, UI/UX, Web Dev & Quantum Computing, Accessed 1. 7. 2024.

[7] Zack Fizell, Use Python to Create Two-Body Orbits. https://towardsdatascience.com/use-python-to-create-two-body-orbits-a68aed78099c; Zack Fizell M.S. in Aeronautics and Astronautics — Articles on Orbital Mechanics| Machine Learning| Coding, Accessed 1. 7. 2024.

[8] Lars Frogner (2021), ast2000tools v1. 1. 8. https://lars-frogner.github.io/ast2000tools/html/index.html, Accessed 1. 7. 2024.

[9] AST2000 – Introduction to Astrophysics https://www.uio.no/studier/emner/matnat/astro/AST2000/index-eng.html; The University of Oslo (UiO), Boks 1072 Blindern, NO-0316 OSLO, Norway, Accessed 1. 7. 2024.

[10] NumPy, https://numpy.org/; The fundamental package for scientific computing with Pythonm, Accessed 1. 7. 2024.

[11] odeint, https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html; SciPy documentation, Accessed 1.7.2024.

[12] Matpoltlib: Visualization with Python, https://matplotlib.org/; Matplotlib 3.9.2 documentation, Accessed 1.7.2024.

[13] math – Mathematical functions, https://docs.python.org/3/library/math.html; Python, Accessed 1.7.2024.