UNIVERSITY OF LEEDS

MATH3001: Computational Applied Mathematics

Jai Trehan

Submitted in accordance with the requirements for the degree of BSc. Mathematics and Statistics

The University of Leeds
Faculty of Engineering and Physical Sciences
School of Maths

March 2023

Abstract

In this report, I delve into the fascinating world of computational applied mathematics, specifically focusing on numerical methods employed to tackle problems that would otherwise be unsolvable. My attention is drawn to three well-researched topics: the two-body problem of celestial orbits, employing random numbers for integration, and analysing the period of a pendulum. While some may question the relevance of such problems, the use of computers enables mathematicians to push the boundaries of accuracy in approximating their solutions. These methods hold significant importance across various fields, including engineering, science, finance, and healthcare, and they have a profound impact on our daily lives. Exploring these problems has been an engaging journey, allowing me to study the same challenges that Kepler faced centuries ago, and to discover how randomness can be utilised to solve deterministic problems. Through this investigation, I have learned that the Leapfrog method is the optimal integration scheme for our formulation of the two-body problem, Monte Carlo methods thrive in higher dimensions, and the period of a simple pendulum is more complex than initially anticipated.

I have gained many valuable skills throughout this project, a standout being Python competence. I hope to carry the skills forward into the world of work, contributing to exciting and innovative projects. I have been able to demonstrate my independent research and academic writing ability whilst exploring well known facts and developing some of my own conceptions. If I had more time I would certainly explore the Hamiltonion structures of the two-body problem in order to build on my current understanding of the two-body problem. Should I do the project again, I would use a more efficient Python filing system and not have separated my work into different scripts.

Ethics

Monte Carlo methods, while powerful and versatile, can be used unethically in certain situations. One example would be to manipulate financial markets. Traders can use Monte Carlo simulations to create misleading models or predictions that are biased and favor their investments. They can also compromise decision making by making predictions others are not privy to creating an unfair advantage to benefit oneself. It is crucial to employ Monte Carlo methods responsibly, transparently and ethically.

Intellectual Property

The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others.

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Contents

Chapter 1

How can we accurately evolve planetary orbits using geometric integrators?

1.1 Introduction

In the first part of my overall MATH3001 project I am going to explore the gravitational two-body problem for a planet in orbit about a star of mass M. My focus is on four integration schemes which give an updated position and velocity of the planet around a star, with the aim to determine which of the methods is most suitable for predicting long term orbits.

1.1.1 The two-body Problem

For the two-body problem we first need to make some assumptions. All that exists in the problem is one body of small mass and one body with a large mass, where the only force impacting each object arise from each-other. A famous example of the two-body system is given by Newton's Law of Gravitation, which describes how two particles interact through potential energy.

Newton's third law, the universal law of gravitation was motivated by Kepler's observations of planetary orbits. Newton questioned the forces acting on the planets in order for them to orbit following the rules of his second law, $F{=}ma$. The Universal law of Gravitation states that every particle attracts every other particle in the universe with a force that is proportional to the product of their masses and inversely proportional to the square of the distance between their centres and takes the form:

$$F = G\frac{m_1 m_2}{r^2}, \tag{1.1}$$

where F is the magnitude of the gravitational force acting between the two objects, m_1 and m_2 are the masses of the objects and r is the distance between the centres of their masses. Now we can develop the gravitational problem, which is given by the following equation:

By accelerating we can equate F=ma to (1.1) with a $\frac{d^2\mathbf{r}}{dt^2}$

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{F}(\mathbf{r}) = -\frac{GM}{r^3}\mathbf{r}. \tag{1.2}$$

The system describes a planet in orbit about a star of mass M where G is the gravitational constant, and $\mathbf{r}$ is the position vector of the planet relative to the star. It is important to note, the equation is only capable of describing the orbit of a single planet with a mass much smaller than that of the star it is orbiting. For example, the earth around the sun. Equation 1.2 is

derived from Newton's laws of motion, which states the acceleration of one body is proportional to the force acting on it, and inversely proportional to its mass. By combining the universal gravitation equation with the two-body problem equation, it is possible to describe the motion of two objects under the influence of their mutual gravitational attraction. Thus, the universal gravitation equation provides the force acting on the two bodies, while the two-body problem equation describes how this force influences their motion and trajectories.

We can write (1.2) in vector form:

using $\frac{dr}{dt}$ / differentiating wrt time

$$\dot{\mathbf{r}} = \mathbf{v}, \dot{\mathbf{v}} = \mathbf{F}(\mathbf{r}) = -GM\frac{\hat{\mathbf{r}}}{r^2}, \tag{1.3}$$

where $\mathbf{v}$ is the orbital velocity and $\hat{\mathbf{r}}$ is the corresponding unit vector for the position of the planet.

Each integration scheme, with timestep h, is used to provide an updated position ($\mathbf{r}_{n+1}$) and velocity ($\mathbf{v}_{n+1}$) of the planet. Using this information we can continually update the position of the planet relative to the star for a number of orbits. To do this I will focus on four methods: The Forward Euler, Modified Forward Euler, Leapfrog and Runge-Kutta 4 integration schemes. I am going to analyse their effectiveness to model the long term behaviour of planets in our solar system by considering simulated plots, energy conservation, momentum conservation and my own readings to explore the subject. To start, I will plot the numerical simulation for each scheme. I will use the following parameters when introducing each scheme: $x = a(1+e)$, $y = 0$, $v_x = 0$, $v_y = \sqrt{\frac{GM(1-e)}{a(1+e)}}$, $GM = 1$, $a = 1$, $e = 0.5$, where e is the eccentricity of the orbit and period, $P = 2\pi\sqrt{\frac{a^3}{GM}}$ for 10 orbits. Consider these conditions the baseline parameters for my investigation throughout this chapter. Also, the analytical prediction for the orbit is given by

$$x = a(\cos E + e),\ y = b\sin E,\ b = a\sqrt{1-e^2},\ E \in [0, 2\pi].$$

GM = 1 = a non-dimensionalised to simplify

1.2 The Forward Euler

Starting with a first order method, it is the simplest scheme I am going to investigate. Using the following formulae it computes solutions to the differential equation 1.3

$$\mathbf{r}_{n+1} = \mathbf{r}_n + h\mathbf{v}_n,\ \mathbf{v}_{n+1} = \mathbf{v}_n + h\mathbf{F}(r_n), \tag{1.4}$$

by discretising the time into small intervals h, giving the updated position vector, r_{n+1} and updated velocity vector, v_{n+1}. The order of an integration scheme refers to the accuracy of the method in approximating the solution of a differential equation. A higher order scheme is more accurate, meaning it provides a closer approximation to the true solution, but generally these methods are more computationally expensive. The Forward Euler method is first order because it approximates the solution of a differential equation with a global error that grows linearly with the size of the time step and thus of order $O(h)$. As a result, we achieve a plot which spirals out. This can be seen clearly in figure 1.1 below. The accuracy can be improved using smaller time steps, but this also increases computational time needed to compute said results. Generally, it is more useful to use a method like the *Runge-Kutta fourth order* to achieve more accurate plots, a method I will investigate later on in this chapter.

Global Error → Total Error, total at truncation

Truncation Error → After one step

Order of the integration → how accurate the model is

Higher order ⇒ more accurate.

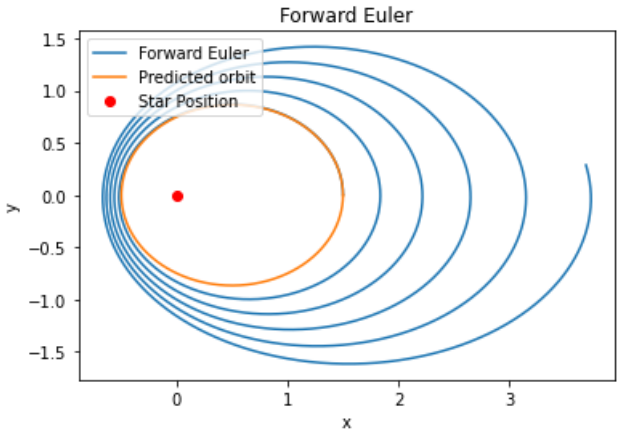


Figure 1.1: Plot of the forward Euler - blue, predicted orbit - orange. 1000 force evaluations per orbit and $e = 0.5$.

In order to better understand why this integration scheme spirals out, the concept of local errors and global errors needs to be introduced. In numerical analysis, the local error refers to the errors incurred at each step of the analysis. The local error is a result of truncation errors, which arise when a finite number of terms are used to approximate an infinite series and quantifies the difference between the exact solution and the approximated solution at a specific point. The global error on the other hand quantifies the total error accumulated over a specific interval, quantifying the difference between the exact solution and the approximation obtained from the numerical method over a set interval. Global errors are the accumulation of truncation errors as well as round off errors; round off errors are a consequence of computational limits which increases as the number of iteration increases. Further down in chapter 2 we will be able to show this. The goal of any numerical method is to minimise the differences between the true value and approximate solution, so it is vital the local and global errors are balanced by selecting a suitable integration scheme and timestep.

As figure 1.1 illustrates the analytical prediction for the orbit spirals out, it is unrealistic to expect a body in orbit to follow a path like this. The Forward Euler method calculates updated positions and velocities of the two bodies based on their current positions and velocities as well as assuming the gravitational forces acting on each body is constant. In reality this is not appropriate since the forces are not constant in the system. Newton's law of gravitation 1.1 shows the gravitational forces change continuously as the bodies move. When the two bodies are closer together, the gravitational forces are greater compared to when they are further apart. Consequently, the the velocity of the orbiting body is highest when it is closest to the central body and slowest when further away; a result of energy conservation principles. The periodic nature of the orbit means changes in velocity of the orbiting body are oscillatory which does not play into the strengths of the Forward Euler method. It is known to be unstable for certain types of problems, including those with oscillatory solutions which can become chaotic, thus failing to model the trajectory of an orbit appropriately. Reducing the time step to be sufficiently small can allow for more accurate results. However, in the case of the two-body problem, this would need to be small enough so the motion and gravitational force change is less than $\frac{1}{10}th$ of the shortest orbital period for each iteration. common guideline

In general, the Forward Euler method is a simple and quick method appropriate for when a rough answer is sufficient. For the purpose of the two-body problem, it is able to provide a good starting point for more complex methods.

numerical methods as one source.

1.3 Modified Forward Euler

The Modified Forward Euler method is a variation of the Forward Euler method, however it computes the updated velocity using the new position it just computed instead of the old position. Like the Euler method discussed in section 1.2 this is also a first order method and follows a very similar formulation to (1.4). To get the updated position and velocity we use the following:

$$\mathbf{r}_{n+1} = \mathbf{r}_n + h\mathbf{v}_n, \;\; \mathbf{v}_{n+1} = \mathbf{v}_n + h\mathbf{F}(\mathbf{r}_{n+1}). \tag{1.5}$$

Updating the velocity then the position, has a significant impact on the simulated orbit as seen in figure 1.2.

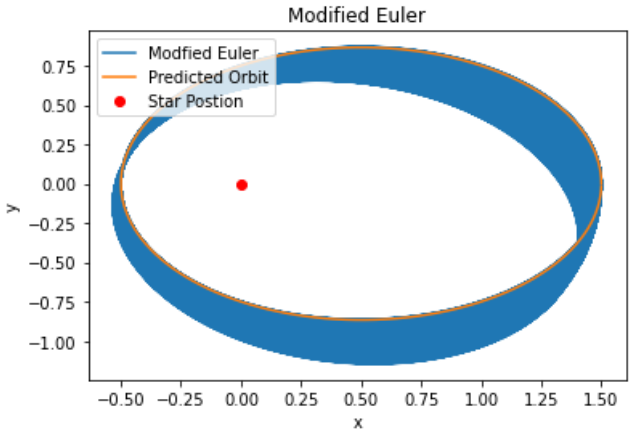


Figure 1.2: Numerical solution using Modified Euler method - Blue, predicted orbit - Orange.

We have used identical parameters to those in section 1.2 in order to maintain a level playing field. Evidently (1.5) more accurately models the path of an orbit since it accounts for the updated velocity of the body when calculating the updated position vector. For a first order method, (1.5) is effective at providing insightful simulations of orbits. By updating the velocity first based on the current position and gravitational forces acting on the orbital body, the system is able to take changes that happen during the timestep into an account when computing the updated position vector. The local error incurred at each time step for this method is $O(h^2)$ and has a global error of $0(h)$. This is the same as the Forward Euler method but, the Modified Euler method exhibits properties of energy conservation making it able to deal with the oscillatory behaviour of the velocity for the orbiting body.

Overall, the Modified Euler method is simple to implement and produces fairly accurate estimates for the two-body problem. It illustrates properties of energy conservation and is symplectic in nature to an extent. Both of these features, I go into detail later in this chapter.

1.4 Leapfrog

Moving on from Euler's methods this system is a second order scheme. The Leapfrog method gets its name from the nature of how we calculate the updated positions, i.e they Leapfrog over each-other. It provides us with an elegant algorithm to integrate (1.2) which isn't too complex. The algorithm works as follows:

1. Initialise the position and the velocity at time t.
2. Choose a timestep size h.

3. Compute the velocity at time $t + \frac{\Delta t}{2}$ using the current position at time t and the forces acting on the system.
4. Compute the new position at time $t + \Delta t$ using the current velocity at time $t + \frac{\Delta t}{2}$.
5. Repeat for a specified length of the simulation.

So, the system is described using the following:

$$\mathbf{r}' = \mathbf{r}_n + \frac{h}{2}\mathbf{v}_n, \ \mathbf{v}_{n+1} = \mathbf{v}_n + h\mathbf{F}(\mathbf{r}'), \ \mathbf{r}_{n+1} = \mathbf{r}' + \frac{h}{2}\mathbf{v}_{n+1}. \tag{1.6}$$

By looking at the derivation of this method, the improvement in stability is a result of using an average of the initial conditions and updated positions. This is explored in more detail in (*The Improved Euler Method and Related Methods* 2020). Using half steps improves the accuracy of the simulation; the errors for the updated velocities are smaller as it is closer in time to the true velocity of the system. This means the system is able to deal with the oscillatory behaviour of the velocity much better than a method like the Forward Euler. However, the position estimates are computed using integer multiples of the full timestep size, this can lead to an increase in the magnitude of the error for the position estimates as it is not synchronised with the velocity calculations. In the case of my investigation this does not appear to be a problem.

As mentioned above, the velocity and position of the orbiting body are directly linked through energy conservation. The second order nature of the Leapfrog method means it is already more accurate compared to those introduced above The local error is $O(h^2)$ while the global error is slightly different to those introduced previously. The global error is proportional to the local error multiplied by the square root of the timestep

$$\text{global error} \approx C\sqrt{\Delta t},$$

where C is the local error. That is to say, as the number of timesteps increases the global error decreases (Press et al. 2007).

As I did in sections 1.2 and 1.3 I am going to plot the trajectory of an orbit using the same conditions. Below, the numerical solution is nearly identical to the predicted orbit.

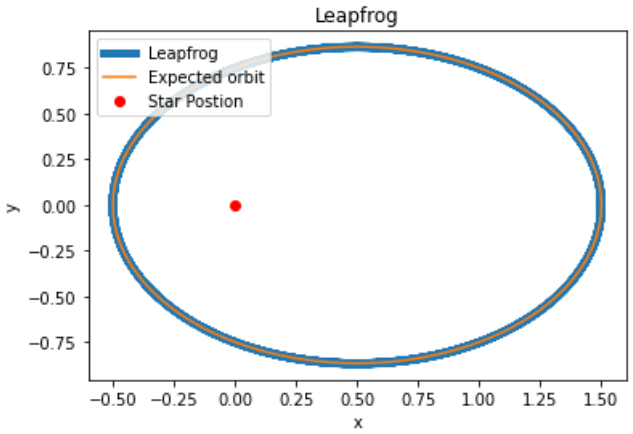


Figure 1.3: Leapfrog plot for 1000 force evaluations.

Consequently, the Leapfrog method is able to accurately model the trajectory of an orbiting body effectively. To add, like all numerical methods, the Leapfrog integration scheme is not perfect. It is important the simulation parameters are chosen appropriately to avoid numerical instability.

1.5 Runge-Kutta Fourth Order

Relative to the schemes I have investigated in the previous sections, this is the most complex. The Runge-Kutta fourth order is derived in a similar fashion to Euler methods but now we take the derivative at the beginning, middle and end of the time step in order to improve accuracy. The algorithm is:

Weighted Average of points across the timestep

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{1.7}$$

where,

$$\mathbf{k}_1 = h\mathbf{f}(\mathbf{w}_n, t_n),$$
$$\mathbf{k}_2 = h\mathbf{f}(\mathbf{w}_n + \frac{1}{2}\mathbf{k}_1, t_n + \frac{1}{2}h),$$
$$\mathbf{k}_3 = h\mathbf{f}(\mathbf{w}_n + \frac{1}{2}\mathbf{k}_2, t_n + \frac{1}{2}h),$$
$$\mathbf{k}_4 = h\mathbf{f}(\mathbf{w}_n + \mathbf{k}_3, t_n + h).$$

In words, the algorithm works as follows:

1. Initialise the velocity and position vectors.
2. Calculate the gravitational force acting on each body based on the current position and masses.
3. Use the current positions and velocities of the two bodies to estimate their positions and velocity at the subsequent time step, using a weighted average of four intermediate estimates.
4. Update the positions and velocities of the two bodies using the weighted average estimates.
5. Repeat the steps 2-4 for each timestep for a simulation of specified length.

By implementing this into a time loop within Python we get the following plot for the numerical prediction of the orbit.

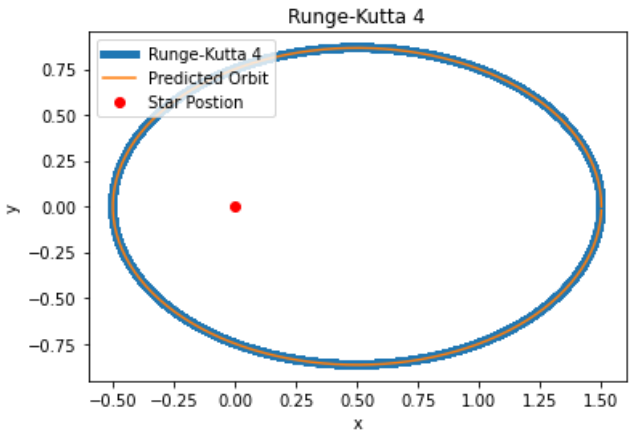


Figure 1.4: RK4 plot for 10 orbits. Note P = h/250

The accuracy of this model makes the RK4 method attractive to use and it is clearly demonstrated in figure 1.4. By taking the derivative at the start, middle and end of the time step we are able to average differences in the system over course of the time step. This is important because in the process of stepping from t_n to t_{n+1} the derivative changes. The weights used

in the fourth-order Runge-Kutta method are chosen such that the global error is proportional to the fourth power of the time step, i.e $O(h^4)$ making it the scheme with the smallest error proportion yet (*Euler Forward Method* 2022). This added accuracy removes the problem we had with methods where propagated errors caused the orbits to spiral out.

Putting everything together, the difference between the final position of the numerical estimate and the predicted orbit is shown in the table below:

Numerical Method	Analytical Difference
Forward Euler	2.21181139
Modified Forward Euler	0.00235205
Leapfrog	0.89972251
Runge-Kutta Fourth Order	0.00854281

Table 1.1: Absolute error between the numerical approximation and the predicted orbit. Parameters remain the same as above

Since we now have a good idea of the behaviour for each scheme, the way they conserve energy and momentum is equally important to decide how effectively they model long term orbits.

1.6 Energy and Momentum

Kepler, during his studies, observed the following three things which formed the basis of his three laws: Planetary orbits around the sun are elliptical, the periods of the orbits are related to their radii, and during orbit it sweeps out equal area in equal times. It is important that the schemes covered in this section conserve energy and angular momentum in order to best model long term planetary orbits.

Energy is given by the following equation:

$$\mathcal{E} = \frac{1}{2}({v^2}_x + {v^2}_y) - \frac{GM}{r}. \tag{1.8}$$

Differentiating (1.8) with respect to time will show Keplerian orbits conserve energy:

$$\begin{aligned}
\dot{\mathcal{E}} &= \frac{d}{dt}\left(\frac{1}{2}({v^2}_x + {v^2}_y) - \frac{GM}{r}\right) \\
\dot{\mathcal{E}} &= (v_x \dot{v_x} + v_y \dot{v_y}) - \frac{d}{dt}(GM(x^2+y^2))^{-\frac{1}{2}} \\
\dot{\mathcal{E}} &= (v_x \dot{v_x} + v_y \dot{v_y}) + \frac{GM}{2}(2x\dot{x} + 2y\dot{y})(x^2+y^2)^{-\frac{3}{2}} \\
\dot{\mathcal{E}} &= \left(v_x \frac{-GMx}{r^3} + v_y \frac{-GMy}{r^3}\right) + GMr^{-3}\left(xv_x + yv_y\right) \\
\dot{\mathcal{E}} &= 0
\end{aligned}$$

Clearly the Keplerian orbit conserves energy. Applying the same method to angular momentum will prove it is conserved for a Keplerian orbit.

$$\mathcal{L} = xv_y - yv_x \tag{1.9}$$

Deriving equation 1.9 is done as follows:

$$\dot{\mathcal{L}} = \dot{r_x} v_y + r_x \dot{v_y} - \dot{r_y} v_x - r_y \dot{vx}$$
$$\dot{\mathcal{L}} = v_x v_y + r_x F(r_y - v_y vx - r_y F(r_x)$$
$$\dot{\mathcal{L}} = -r_x \frac{GMr_y}{r^3} + r_y \frac{GMr_x}{r^3} = 0$$

Energy conservation and conservation of angular momentum are a fundamental physical property of the two-body problem, therefore it is desirable for numerical methods to account for this when calculating trajectories. The conservation of energy dictates the total energy in the system should remain constant as seen above. This means the sum of the kinetic energy and the potential energies must remain constant over the course of the simulation. Similarly, in the absence of external forces, the total momentum of the two-body system should be conserved throughout the simulation. If the numerical method does not exhibit energy or momentum conservation, simulated orbits can be inaccurate or unrealistic.

To understand the energy and momentum error in each scheme, let us first investigate the behaviour of long term orbits. Each plot below considers 300 force evaluations, $e = 0.5$ and 100 orbits. Note, the number of force evaluations is now 300 compared to 1000 I used in the initial plots. Computationally this is less expensive, more efficient and better reflects a real life scenario. If force evaluations are performed too frequently it could result in unrealistic plots.

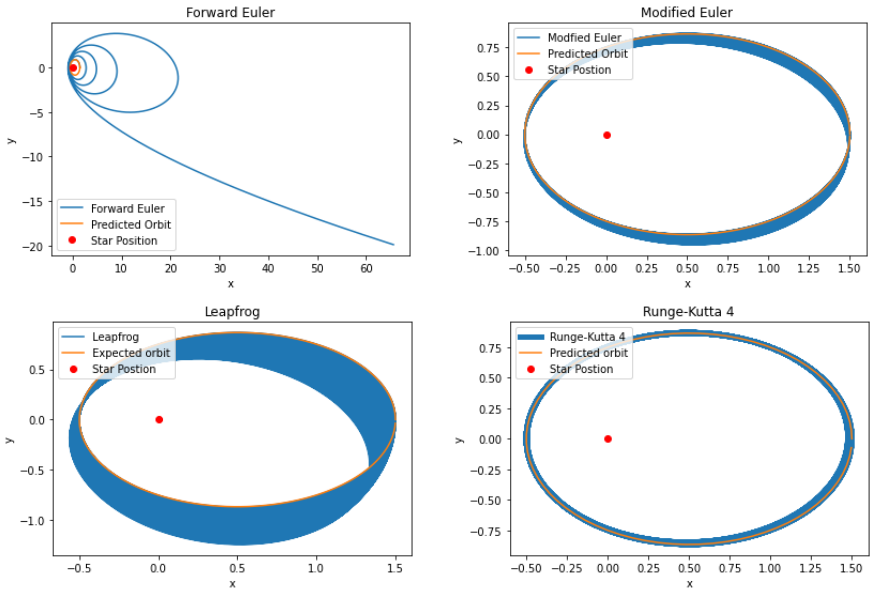


Figure 1.5: Longer term behaviour for each of the schemes, 100 orbits with 300 force evaluations per orbit, e=0.5

As you can see in figure 1.5 RK4 is clearly the most reliable to predict long term orbits, it has the smallest drift from the expected orbit (orange). Forward Euler spirals out even more so than it did in section 1.2. It seems the purpose of the Forward Euler method is to highlight the propagated errors that accumulate. Although Modified Euler was accurate in section 1.3, it can be seen to change orientation whilst maintaining its shape. Similarly, the plot for the Leapfrog method does the same whilst still remaining relatively close to the expected orbit. This behaviour can be explained by the energy and momentum conservation properties of the schemes. In Python it is possible to calculate the fractional error for the energy and momentum of the system in order to see how each of the schemes. Before going into detail about the conservation properties of the schemes, I need to introduce the notion of symplectic integrators.

Symplectic integrators are a type of numerical method, used to solve differential equations that preserve specific geometric properties of the system, such as energy and momentum. This makes them well suited for the problem at hand (Hairer et al. 2006.

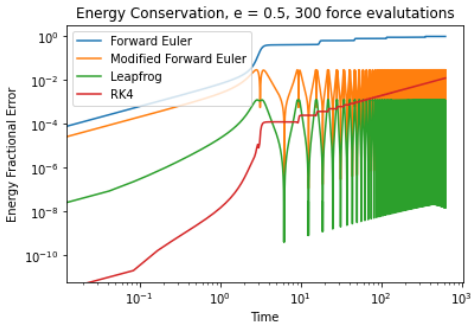


Figure 1.6: Energy Fractional Error: $\frac{|\mathcal{E}-\mathcal{E}_0|}{|\mathcal{E}_0|}$ where $\mathcal{E}_0$ is the initial energy.

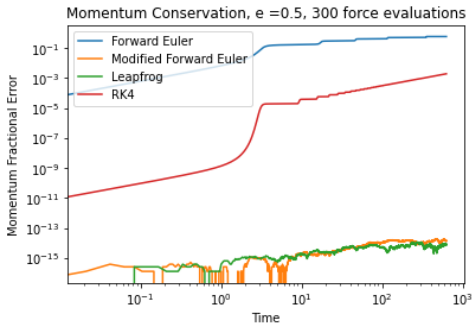


Figure 1.7: Momentum Fractional Error: $\frac{|\mathcal{L}-\mathcal{L}_0|}{|\mathcal{L}_0|}$ where $\mathcal{L}_0$ is the initial momentum.

1.6.1 Forward Euler's Energy Properties

As you can see in figure 1.6 Forward Euler has the largest energy fractional error. It also has the largest fractional error in angular momentum, as seen by the erratic path it takes in figure 1.7. For our problem, which fundamentally relates to gravity, the Forward Euler solution exhibits energy drift over time due to the inherent errors and approximations that propagate over long term orbit simulations. These errors can affect the conservation of angular momentum and energy. The Forward Euler method uses information from the current time step to calculate the solution at the next time step. This can contribute to a loss of accuracy and conservation properties over time. As such, it is not symplectic in nature.

By looking at the Taylor expansion of the position and velocity vector of each of the two masses in the system we can get a better understanding of the truncation errors: $r_i(t+\Delta t) = r_i(t) + v_i(t)\Delta t + O(\Delta t^2)$, $v_i(t+\Delta t) = v_i(t) + a_i(t)\Delta t + O(\Delta t^2)$.

$r_i(t)$ and $v_i(t)$ are the position and velocity vectors of masses i at time t, respectively, and $a_i(t)$ is the acceleration vector of mass i at time t. For example, it is possible the Forward Euler method overestimates the kinetic energy of the system and underestimates the potential energy as a result of the truncation errors or vice versa. For the same reasons, the Forward Euler method is not considered to conserve angular momentum either. As a result over long orbital periods, inaccurate results arise.

1.6.2 Modified Forward Euler's Energy Properties

The Modified Euler scheme has the second largest fractional error for energy for the same reasons as the Forward Euler method. However, it conserves energy better compared to the standard Euler method due to the fact it takes the updated positions into an account when computing updated velocities. This means the numerically predicted velocity is closer to the actual velocity and thus the kinetic energy of the system is more accurately accounted for. Following similar reasoning, the Modified Euler method is better at conserving angular momentum too. This is explored in more depth in (Pang 2006). This specific method is considered to be a symplectic integrator as it does preserve the geometric and Hamiltonian structure of the two-body problem. In particular, the method preserves the structure of the system's phase space, which is a mathematical construct that describes the positions and momentum of the system's constituent parts (*Symplectic Integrators* 2023). Furthermore, because the Modified Euler method exhibits

rotational symmetry it is able to conserve angular momentum too. As a result the method is fairly accurate at providing numerical estimations for the trajectory of the orbiting mass whilst remaining fairly simple.

1.6.3 Leapfrog's Energy Properties

The Leapfrog scheme has the smallest fractional energy error as well as fractional errors in angular momentum. From figure 1.6, it starts to become solid as more orbits are completed. The Leapfrog method is a symplectic integrator and is known to conserve energy for Hamiltonian systems, which includes the two-body problem, making it an attractive method to model long term orbits. As mentioned above, the method calculates the position and velocity at different time steps and is thus able to compute the kinetic energy of the system closer to the true value. Consequently, as the planet evolves over time, the numerically predicted trajectory is able to remain accurate. Additionally, it is also capable of conserving angular momentum due to the rotational symmetry of the system. Rotational symmetry refers to a type of symmetry in which an object or system remains unchanged or invariant under rotation around an axis (MathIsFun 2019). Rotational symmetry is important for mechanical systems as it indicates the system is not dependent on the direction it is observed from. As such, the Leapfrog method is a popular choice to model the two-body problem. Regarding its change in orientation, one reason for this could be the forces acting on the planet are not constant over time. This can cause the particle to rotate as it moves. Additionally, the Leapfrog method can cause rotation if the ODE being solved represents a system with angular momentum, such as a spinning top or a planet orbiting a star.

1.6.4 Runge-Kutta's Energy Properties

The Runge-Kutta 4 starts off with the smallest energy error, which then increases exponentially as time goes on. The effects of a Runge-Kutta method not preserving energy or angular momentum are negligible for short-term orbits. However, in our simulation of 300 orbits, it does not fare very well. Since the scheme relies on numerical approximations of the derivatives of the variables in the system being simulated, these are not exact, and over time errors accumulate. Similarly, it does not conserve angular momentum well for the same reasons, thus it is not a symplectic integrator. It is not explicitly designed to incorporate the underlying symmetries and conservation laws of the two-body problem.

So we have the Modified Forward Euler and Leapfrog method that provide better long-term behaviour in the simulation of mechanical systems. They incorporate the underlying symmetries and conservation laws more accurately.

1.6.5 Why Is Eccentricity Important?

So far, we have considered eccentricity $e = 0.5$ with no explanation as to what it is, nor the impact on our results should it change. The eccentricity parameter determines the shape of the orbit of one body around the other. Using modern day satellite imaging techniques, we know most orbits are elliptical in nature; however, this was not always the case. Before Kepler's observations it was assumed orbits were perfectly circular.

Definition 1.1. *Eccentricity: deviation of a curve or orbit from circularity.*

Eccentricity is measured from 0 to 1, with 0 being perfectly circular and 1 being a parabolic unbound orbit. The eccentricity of the orbit impacts the smoothness and behaviour of the system. When eccentricity is close to zero, the orbit resembles a circle and thus the forces interacting on the two bodies remain relatively constant over time. On the other hand, when the orbital period is not close to zero, the elliptical shape of the orbit means the forces acting

on the two bodies are not constant over time. We can see how changing this parameter impacts the simulated plots. Consider, $e = 0.9$ for $N = 100$ orbital periods and 1000 force evaluations per orbit.

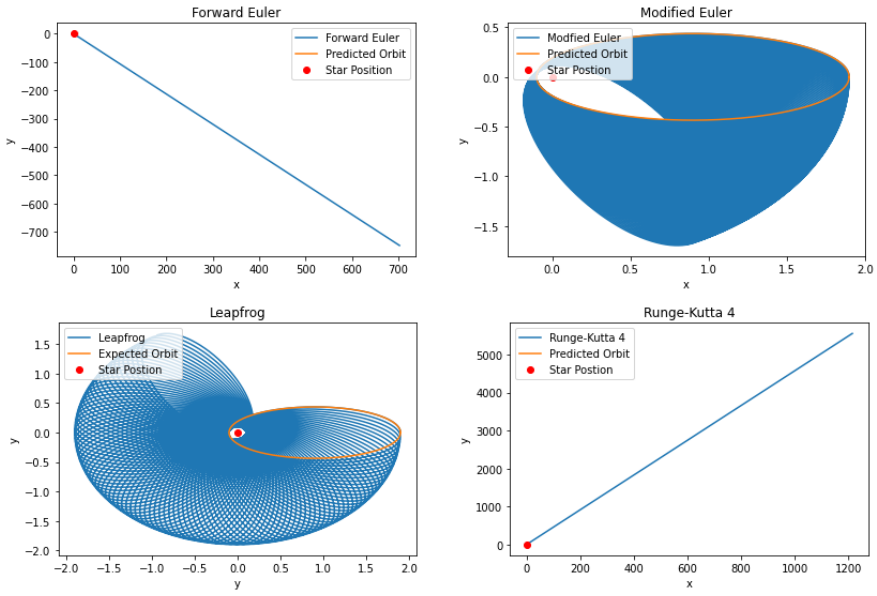


Figure 1.8: High eccentricity orbit simulations.

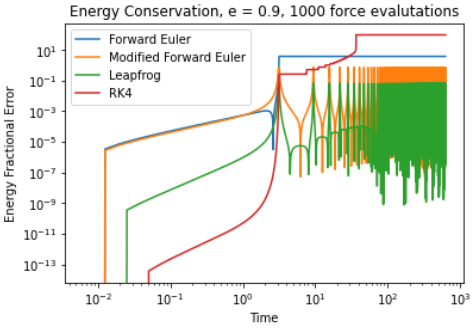


Figure 1.9: Energy Fractional Error: $\frac{|\mathcal{E}-\mathcal{E}_0|}{|\mathcal{E}_0|}$

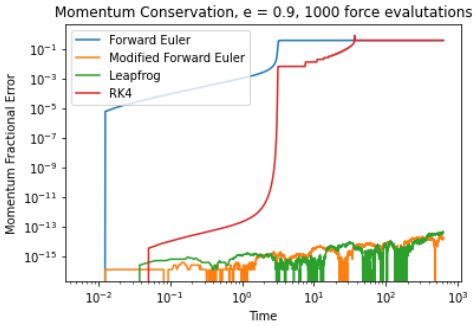


Figure 1.10: Momentum Fractional Error: $\frac{|\mathcal{L}-\mathcal{L}_0|}{|\mathcal{L}_0|}$

In figure 1.8 it becomes apparent the non-symplectic schemes fail to simulate the motion of an orbit accurately. The trajectories of the Forward Euler method and the RK-4 method appear to be straight lines; this highlights how unstable the schemes become, reinforcing the fact that parameter choice is crucial in order to achieve valid results.

Since we are now attempting to model an orbit with high eccentricity, the elliptical shape becomes elongated. Subsequently, the forces acting on the celestial bodies differ significantly over the course of the simulation, the Forward Euler method fails to capture this varying behaviour and thus unable to model the trajectory with precision. A very small time step is needed to accurately capture the behaviour of the system. However, using a infinitely small time step can be computationally expensive, time consuming and produce unrealistic results.

Likewise, the RK-4 method fails for similar reasons. Although, this could be rectified by using higher order versions of the scheme capable of dealing with the rapid rate of changes for the velocities and positions.

The Modified Euler method models the orbit far better, although it is not exact. It is symplectic, as mentioned above, thus able to account for the significant energy changes during an orbit. The Leapfrog method's energy conserving properties are slightly better than the Modified Forward Euler, shown in figure 1.9 and figure 1.10. In turn, this produces the most realistic simulation of high eccentricity orbit. Interestingly, the plots for the Modified Forward Euler and Leapfrog method show orbits that change orientation. I speculate this is a result of their energy and momentum conserving qualities. If the angular momentum of the system is conserved, then any changes in the distance between the two bodies must be accompanied by changes in the velocity of the orbiting body in order to maintain a constant angular momentum. This means that as the distance between the two bodies changes, the eccentricity of the orbit can also change. Over long periods this results in the orientation of the orbit changing but the shape of it does not.

Note: I ran the simulation with a timestep 100 times smaller and more accurate results were achieved. (See appendix for code.)

1.7 Time Reversibility

Time reversibility means that if the system is evolved backwards in time, the dynamics of the process remain well defined (Wikipedia 2021). The concept demonstrates the deterministic nature of computers, if we start with a certain set of inputs and run a program to obtain a set of outputs, we can obtain the original inputs by running the same computation in reverse, starting with the outputs and obtaining the inputs. This proves to be interesting for the two-body problem as it allows us to assess the stability and accuracy of an integration scheme opening up the opportunity to extrapolate for long term simulations. This is illustrated below:

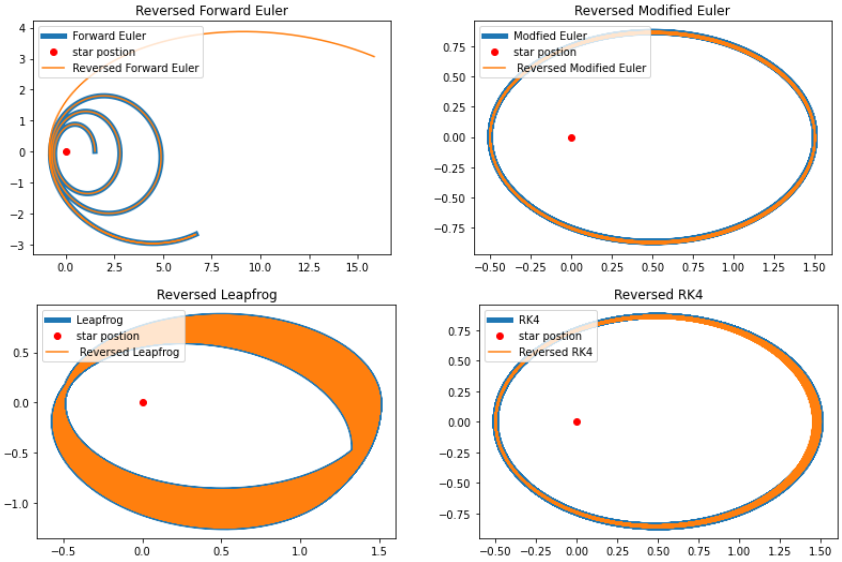


Figure 1.11: Reversed orbital trajectories superimposed onto the forward model. Parameters used: $e = 0.5$, N=10 and 300 force evaluations per orbit.

As expected, the Forward Euler method cannot be reversed accurately, it is not able to preserve the physical properties of the system in the reverse direction. For the other three methods it is less clear cut so we need to specify quantitatively whether each system is time reversible. A system is said to be time reversible if the round off errors $< 10^{-10}$. By calculating the magnitude of the error between the initial and final position of the velocities we can determine which are time reversible for 10 orbital periods. Only Leapfrog is within this threshold as seen in figure 1.12.

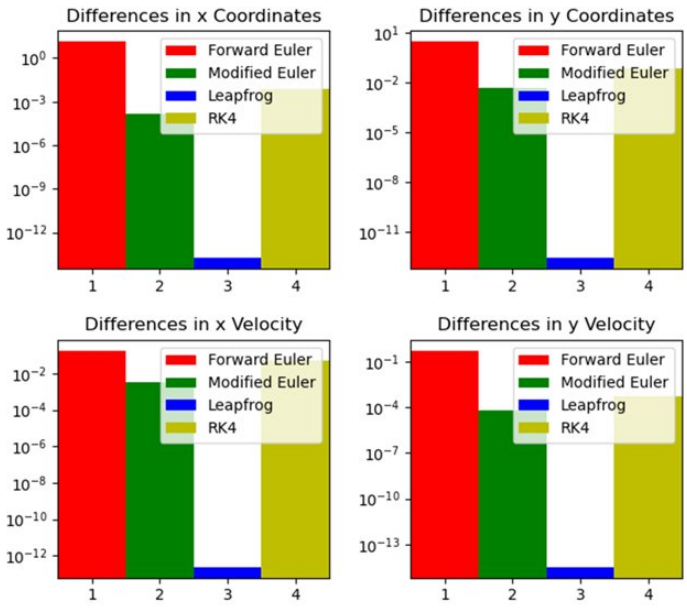


Figure 1.12: Magnitude of errors for reversed simulations.

The Leapfrog method updates the position vectors and velocity vectors at half time steps relative to each other, known as time-centred difference scheme. Specifically, the position update is done at time $t + \Delta t/2$, while the velocity update is done at time $t - \Delta t/2$, where Δt is the time step (Hairer et al. 2006). This symmetry of computing each vector ensures that the energy of the system is conserved. Overall, the reversibility of Leapfrog time integration is an important property that makes it useful for certain applications, such as simulating the motion of celestial bodies or other systems that involve long-term evolution.

1.8 Closing Remarks

Using numerical methods to solve the two-body problem is fascinating. The methods explored in this chapter have their advantages and disadvantages. The Forward Euler method is simple and fast to implement at the expense of accuracy. Although small timesteps provide more accurate results, it is not always feasible due to the added computation required. It is unable to capture the physical behaviour of the problem, and with a little more work, we can produce better results. The Modified Euler method takes all the best features of the Forward Euler method and builds on them. It can capture the long-term behaviour far better thanks to its ability to conserve energy and angular momentum. The Leapfrog method is in the Goldilocks region, with its simplicity, symplectic nature, and time-centred difference scheme making it a popular choice for simulating the two-body problem. The Runge-Kutta fourth-order method is highly accurate, although long-term simulations proved to be its Achilles' heel. It is computationally more expensive; however, the fact that it can be easily adjusted for higher orders to deal with oscillations in velocity and rapid changes makes it attractive for more complex problems.

Chapter 2

Random Numbers to integrate

Monte Carlo methods are random sampling algorithms used to generally obtain numerical results. They use randomness to solve problems that are, in principle, deterministic, such as integration, optimisation, and probability. The invention of Monte Carlo methods is largely attributed to John von Neumann and Stanislaw Ulam with the intention to improve decision making under uncertain conditions. Fittingly named after the famous casino town in Monaco, Monte Carlo methods work as follows:

1. Define a domain of random inputs.
2. Generate inputs randomly from a probability distribution.
3. Perform deterministic computation of the inputs.
4. Aggregate the results.

In this part of my project, I will explore how the Monte Carlo methods can be used to provide us with numerical solutions to integrals.

In order to achieve accurate results for a Monte Carlo simulation the random sequence needs to ensure weak correlation between successive samples. Sawilowsky lists these qualities in detail (*Monte Carlo Method* 2023). The `random` package in python can be used to generate random numbers from a specified statistical distribution. See appendix for code that evaluates the mean and variance of a random sample taken from different distributions.

2.1 Integration in One Dimension

Most integrals we have come across so far during our studies can be performed analytically, however, this is not the case in true applications. Consequently, numerical methods are sufficiently useful to provide approximates to such integrals. Generally, integrals can be interpreted geometrically as the area under the curve of some function $f(x)$. Suppose $f(x) = \frac{1}{2}\int_0^2 cos^2(x)\,dx$, this can be integrated analytically to give $\frac{1}{2}(1 + \frac{1}{4}sin4) \approx 0.40539....$ We will use this function throughout the rest of this section as our case study example and investigate different ways to obtain the solution.

2.1.1 Rectangular Midpoint Method

The most simple way to geometrically interpret the area under a function is to separate the area under the curve into rectangles of regular width and sum the area of each of the rectangles. This is illustrated in figure 2.1 and figure 2.2.

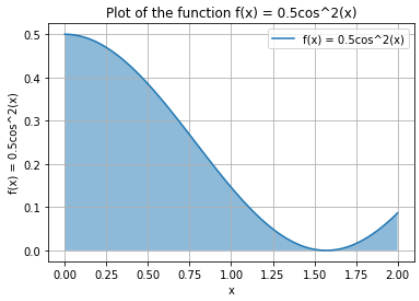


Figure 2.1: Area under the curve of $f(x)$

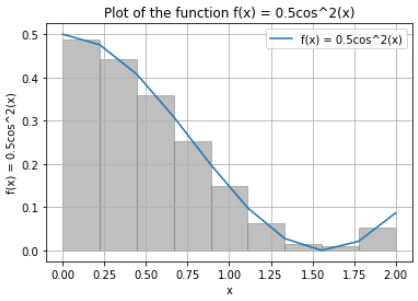


Figure 2.2: Rectangular estimation, $n = 10$

In figure 2.2 the x-axis is divided equally into n rectangles of width $\triangle x$ where $\triangle x = \frac{b-a}{n}$ and $x_n = x_0 + n\triangle x$. Obtaining an estimate for the area is done by calculating the area of each of the rectangles, evaluated at the beginning of the interval and then summing them; given by the following formula:

$$F_n = \sum_{i=0}^{n} f(x_i)\triangle x. \tag{2.1}$$

Applying this method we obtain the following estimate, $f(x) \approx 0.40476606(8dp)$, by taking the difference between the known exact area to obtain an absolute error of $|I - I_n| = 0.00063362(8dp)$, thus we have percentage error ≈ 0.15629621 %. With only 10 divisions we can achieve an approximation with a moderately low percentage error, intuitively, increasing the number of divisions increases the accuracy of the estimate, such changes in parameters I investigate further down, but first I want to introduce the concept of Riemann Sums.

Note: The error bound for the midpoint rule is $|E| \leq \frac{(b-a)^3}{24n^2} * |max f''(c)|$ for some c in the integration interval and where $a \leq x \leq b$.

The method whereby we interpret the area under a curve as a sum of geometrically simple shapes is known as a Riemann sum.

Definition 2.1. *(Riemann Sum) (*Wolfram Alpha *2023) Let a closed interval* $[a, b]$ *be partitioned by points* $a < x_1 < x_2 < \cdots < x_{n-1} < b$*, where the lengths of the resulting intervals between the points are denoted* $\Delta x_1, \Delta x_2, \ldots, \Delta x_n$*. Let* x_k^* *be an arbitrary point in the* k*th subinterval. Then the quantity*

$$S = \sum_{k=1}^{n} f(x_k^*)\Delta x_k$$

is called a Riemann sum for a given function $f(x)$ *and partition, and the value* $\max \Delta x_k$ *is called the mesh size of the partition. If the limit of the Riemann sums exists as* $\max \Delta x_k \to 0$*, this limit is known as the Riemann integral of* $f(x)$ *over the interval* $[a, b]$*.*

The Riemann sum approximates the area between the curve $y = f(x)$ and the x-axis by dividing the area into n strips and approximating each strip with a rectangle whose height is $f(c_i)$ and whose width is Δx_i. As the number of strips increases, the Riemann sum becomes a better approximation for the area under the curve, such that as the number of strips approaches infinity the approximation is equal to the exact definite integral. i.e as $n \to \infty$, $S = \int_a^b f(x)$ (*Numerical Integration and Monte Carlo Methods* 2001). There are different ways to evaluate Riemann sums, we can take the lower Riemann sum, upper Riemann sum or the midpoint, in the case above we took the **midpoint Riemann sum**. Our $f(x)$ line goes through the midpoint of each rectangle. Note that this method incurs the least error out of the three types of Riemann sums

since an average is taken between the two limits of the partition. These methods are explored in detail in (Speight 2021).

Obviously, there is an inherent error when calculating a definite integral using a finite number of divisions, these can be easily visualised in figure 2.2 as the excess area of the rectangle we calculate, hence we have an overestimate.

To reinforce our understanding of quadrature methods I am now going to introduce two more used to approximate definite integrals.

2.1.2 Trapezoidal Rule

Instead of rectangles we now use trapeziums, a trapezium should fit our function, which is parabolic in nature, better compared to a rectangle. This method uses one side equal to $f(x)$ at the start of the interval (x_i) and the other side equal to $f(x)$ at the end of the interval (x_{i+1}). Thus, the total area under the curve is:

$$F_n = [\frac{1}{2}f(x_0) + \sum_{1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)]\triangle x, \tag{2.2}$$

another example of a quadrature method. This can be visualised in figure 2.5

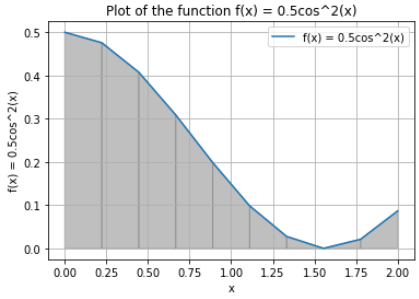


Figure 2.3: $n = 10$

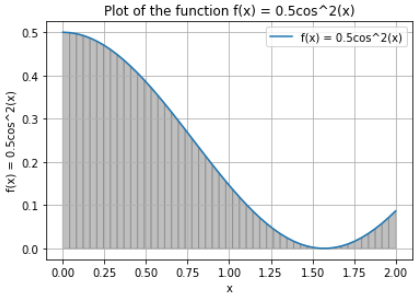


Figure 2.4: $n = 50$

Figure 2.5: Trapezoid estimations of $\frac{1}{2}\int_0^2 cos^2(x)\,dx$ for different n

Evidently, increasing the number of divisions increases the resolution of our curve for $f(x)$. In turn, this produces a better numerical solution for the definite integral. This result is expected, it reinforces the known properties of Riemann sums I introduced in the previous section. Our estimate for $f(x)$ with $n = 10$ and $n = 50$ divisions are $f(x) \approx 0.40666440$, $|I - I_n| = 0.00126471$, with percentage error $= 0.31196717\%$ and $f(x) \approx 0.40545015$, $|I - I_n| = 5.04588822 \times 10^{-5}$, and percentage error 0.01244670% respectively all to 8 decimal places. As you can see this clearly demonstrates having more divisions increases the accuracy of the approximation. How can we quantify the error induced? It would be useful to show how increasing the number of divisions alters our approximation for the definite integral. It is known the trapezoidal error formula is given by $E \leq \frac{(b-a)^3}{12n^3} * |max(f''''(c))|$ where c is within the integration interval and where $a \leq x \leq b$ (Chapra 2022).

2.1.3 Simpson's 1/3 Rule

Simpson's $\frac{1}{3}$ rule, commonly known as Simpson's rule is the final quadrature structure I am going to investigate before we move on to the main part of this section. Unlike the quadrature methods above, quadratic parabolas are fitted to the graph instead over each sub-interval we define. This method was first used by Kepler some 100 years before Thomas Simpson coined

it. The formula is as follows:

$$\frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right]. \tag{2.3}$$

We start by setting the integration value for the lower and upper limit of the function and calculating the step size. For each sub interval, we compute the area using a fitted quadratic parabola and repeat this process for each sub interval until we have an approximation of the entire definite integral. Whilst this is hard to describe verbally, we can better understand this process with the following illustration.

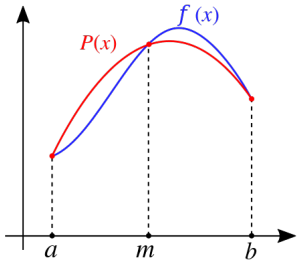


Figure 2.6: The function $f(x)$ (in blue) is approximated by a linear function (in red). Credit: Wikipedia

Using quadratic polynomials results in more accurate answers. For our case study function, $f(x) = \frac{1}{2}\int_0^2 cos^2(x)\,dx \approx 0.40539884$ (8dp) with $n = 10$ divisions, which incurs an absolute error of $8.44912794 \times 10^{-7}$ (8dp) and percentage error of approximately $2.08414762 \times 10^{-6}$ % (8dp).

It's important to note that the number of intervals for Simpson's rule must be even; we require an odd number of quadratic polynomials to approximate a definite integral with two limits. Since Simpson's 1/3 rule uses the endpoints and midpoints of each sub interval, an odd number of sub intervals (n) would result in the last sub interval having an odd number of function evaluations, making it impossible to approximate using the rule. Furthermore, the method may fail to provide accurate results if a function lacks derivatives at certain points or is highly oscillatory.

The error bound in Simpson's $\frac{1}{3}$ rule is $|E| \leq (\frac{(b-a)^5}{180n^4}) * |max(f''''(c))|$ for some c in the interval of integration and where $a \leq x \leq b$. The error is proportional to the fourth derivative of the function being integrated, and is asymptotically proportional to $(b-a)^5$. As a result, one intriguing aspect of Simpson's rule is the approximation becomes exact for functions of up to and including order 3.

To summarise the above, see table 2.1.

Approximations of $\frac{1}{2}\int_0^2 cos^2(x)\,dx$			
Number of Iterations n	Rectangular Method	Trapezoidal Rule	Simpson's Rule
10	0.40476606	0.40666440	0.40539884
50	0.40537446	0.40545015	0.40539969
100	0.40539338	0.40541230	0.40539969
1000	0.40533996	0.40539981	0.40539969

Table 2.1: Approximations for the definite integrals using quadrature methods.

Absolute errors between the exact area and analytical solution of $\frac{1}{2}\int_0^2 \cos^2(x), dx$			
Number of Iterations n	Rectangular Method	Trapezoidal Rule	Simpson's Rule
10	0.00063362	0.00126471	$8.44912794 \times 10^{-7}$
50	$2.52314596 \times 10^{-5}$	$5.04588822 \times 10^{-5}$	$1.34568301 \times 10^{-9}$
100	$6.30698179 \times 10^{-6}$	$1.26137113 \times 10^{-5}$	$8.40931214 \times 10^{-11}$
1000	$6.30669048 \times 10^{-8}$	$1.26133783 \times 10^{-7}$	$7.99360578 \times 10^{-15}$

Table 2.2: Approximation errors using quadrature methods.

It is clear to see that the quadrature methods I have explored provide us with accurate approximations for $f(x)$, this begs the question why even bother with Monte Carlo methods? Quite simply, Monte Carlo methods are far more effective at approximating integrals in higher dimensions. The rest of this chapter will focus on Monte Carlo methods to integrate, highlighting why they're the go to choice for evaluating higher order integrals.

2.1.4 Monte Carlo Methods

Monte Carlo methods are firmly rooted in two statistical theorems, the law of large numbers and the central limit theorem (Dunn and Shultis 2012). Monte Carlo integration in one dimension involves choosing a random number generated from a probability distribution n many times and evaluating the integrand at that point to then calculate the average. For our function $f(x)$ the algorithm is:

1. Pick a number between 0 and 2 randomly generated from a random distribution.
2. Plug that value into the function $f(x) = \frac{1}{2}\int_0^2 cos^2(x)\, dx$.
3. Divide the value of $f(x)$ by the probability of choosing that number to get an estimated area of the function.
4. Repeat this n many times and average the results.

Mathematically, we are expressing our integral $f(x)$ as an expectation of some random variable. Thus, if we draw n numbers generated by a distribution in the integration interval then we estimate the area as:

$$\langle f \rangle = \int_a^b f(x)dx \approx \frac{b-a}{n}\sum_{i=1}^{n} f(x_i) \tag{2.4}$$

Initially, I am going to generate random numbers sampled from a uniform distribution, however different sampling methods include: stratified sampling and importance sampling. Since the use of uniform distribution will be recurrent in this chapter, lets remind ourselves of the definition.

Definition 2.2. *(Uniform Distribution) If* $X \sim (a, b)$ *then* $E[X] = \frac{1}{b-a}$ *and* $Var(X) = \frac{(b-a)^2}{12}$

$$f(x) = \begin{cases} \dfrac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

with a mean squared error of $\frac{(b-a)}{\sqrt{n}}\sigma$ *where* σ *is the standard deviation.*

Like all numerical methods, there is going to be a degree of error. In the specific case of using a uniform distribution, the degree of error is quantified using the mean squared error.

In python we can simulate the algorithm above to obtain the following results in table 2.3 It is evident this method is the worst at evaluating $f(x)$, though is does share one similarity with the quadrature methods. As the number of trials increases, the accuracy get better but at a far slower rate. Using the mean squared error the estimation error decreases as $\frac{1}{\sqrt{n}}$, this is to say

Monte Carlo Integration Estimate for $f(x) = \frac{1}{2}\int_0^2 \cos^2(x), dx$			
Number of Iterations n	Monte Carlo Method	Absolute Error	Mean Squared Error
10	0.37929697	0.02610272	0.32338893
50	0.35913676	0.04626293	0.15705607
100	0.36133610	0.04406360	0.11641569
1000	0.39001614	0.01538354	0.03643476
10000	0.40834740	0.00294771	0.01150594

Table 2.3: Monte Carlo Integration Approximations

doubling the number of samples used, reduced the error by a factor of $\frac{1}{\sqrt{2}}$, nicely illustrated in rows 2 and 3 of table 2.3. ($0.15705607 \times \frac{1}{\sqrt{2}} \approx 0.11641569$).

2.1.5 A deep dive into one dimension errors

To summarise the above, as the number of samples increases we get more accurate results; the Simpson's $\frac{1}{3}$ rule produces the most accurate results and Monte Carlo methods perform the worst. We have also introduced error bounds for each quadrature scheme and the mean squared error for the uniform distribution. What does all of this mean?

Error bounds tell us the maximum possible error in our approximations thus we are able to quantify the accuracy of each method.

1. Midpoint Error Bound: $|E_M| \leq \frac{(b-a)^3}{24n^2} * |max f''(c)|$.
2. Trapezium Rule Error Bound: $|E_T| \leq \frac{(b-a)^3}{12n^3} * |max(f''(c))|$.
3. Simpson's $\frac{1}{3}$ Error Bound: $|E_S| \leq \frac{(b-a)^5}{180n^4} * |max(f''''(c))|$.
4. Monte Carlo Mean Squared Error: $\frac{(b-a)}{\sqrt{n}}\sigma$.

The source of error for the midpoint rule is due to the error in each sub-division, illustrated in figure 2.2. The specific order of the error for the midpoint method is $O(h^2)$ where h is the step size. In table 2.2 increasing the number of steps, n, by a factor of 10 decreases our error by a factor of 100, i.e 10^2. Thus, determine the error $E \propto \frac{1}{n^2}$ and decreases quadratically as the number of divisions increases. Furthermore, this method is exact for polynomials of degree up to and including one (Stewart 2015). This is because the error bound term has a second derivative term in it, such term is zero for polynomials of degree up-to one.

The theory directly above holds for the trapezium rule. However the error bound is slightly larger in this case, meaning it is slower to converge towards the exact area.

The Simpson's $\frac{1}{3}$ rule has the lowest error, with error order $O(h^4)$. We can see this in table 2.2, as the number of steps increases by a factor of 10, the error decreases by a factor of 10^4. That is, $E \propto \frac{1}{n^4}$. Also, this method can achieve exact results for polynomials of degree up-to and including 4. This is because, the error bound includes a fourth derivative term, which in the case of a polynomial of degree 3 is 0. The working below proves this.

$$
\begin{aligned}
f(x) &= ax^3 + bx^2 + cx + d \\
f^{(3)}(x) &= 6a \\
f^{(4)}(x) &= 0 \\
|E_s| &\leq \frac{(b-a)^5}{180n^4} \cdot 0 \\
|E_s| &= 0
\end{aligned}
$$

For Monte Carlo Methods we have to employ the use of the mean squared error to determine the error bound for our approximation. The $\sqrt{n}$ term indicates $E \propto \frac{1}{n^{\frac{1}{2}}}$. We can form probabilistic bounds on the errors applying Chebyshev's inequality: $\Pr\{|E_{mc}| \geq (\frac{\sigma^2}{\delta})^{\frac{1}{2}}\} \leq \delta$, where δ is any positive number. We can achieve smaller bounds on the absolute errors using the central limit theorem, as $n \to \infty$ the distribution of values observed of our randomly generated points follow the normal distribution. More on this can be read in (Kalos and Whitlock 1993).

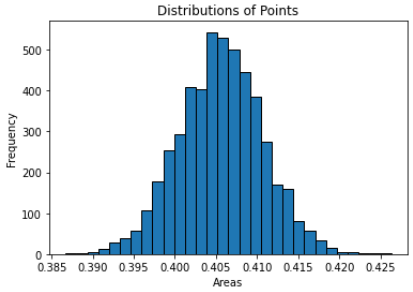


Figure 2.7: Histogram of the randomly generated points for $n = 5000$. It highlights uniformly randomly generated points resemble a normal distribution for the area.

For each of the above schemes, it is possible to present the errors and the convergence towards the true value of $f(x)$ graphically. Suppose 10,000 iterations were taken, figure 2.10 illustrates each algorithms results and errors.

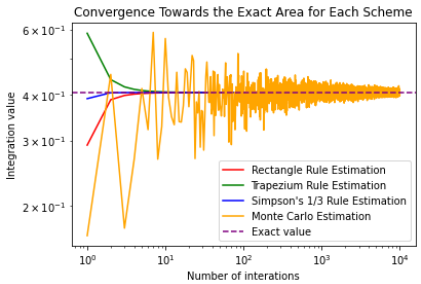


Figure 2.8: Convergence to exact area for each approximation method.

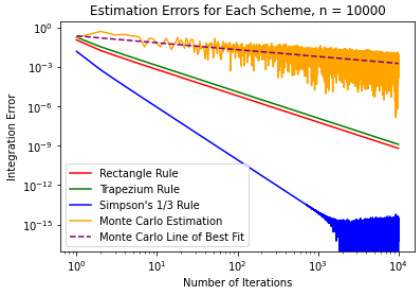


Figure 2.9: Error for each approximation method.

Figure 2.10: Convergence and Error plot on a log-log scale for the 10,000 step approximation to $f(x) = \frac{1}{2}\int_0^2 \cos^2(x)dx$.

Figure 2.9 highlights Simpson's approximations are the most accurate and the Monte Carlo method is the least accurate. The gradient of the lines is reflective of the error proportion

discussed above. The gradients are $-0.5x, -2x$ and $-4x$ for the Monte Carlo, Trapezium rule and Rectangle, and Simpson's rule respectively. To improve the Monte Carlo estimation error, taking the average for the error results in a faster converging result seen in figure 2.11.

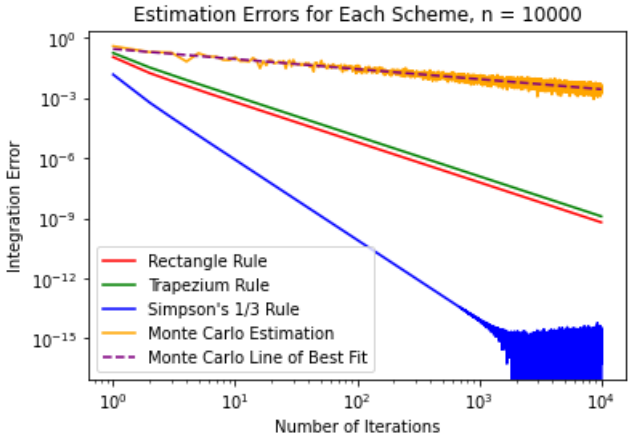


Figure 2.11: Error for each approximation with averaged errors

I have run the code for ten thousand samples and the Simpson's rule breaks down at the five thousand or so mark. This is a result of python not being able to distinguish between the small floating point numbers. Computers store and manipulate numbers using binary representation this can become problematic when very small or large numbers have to be rounded to fit into the available number of bits. Eventually, all the schemes will produce non-deterministic results for the errors after so many steps.

We have a better understanding of the errors now, but arrive at the same conclusion as before: Monte Carlo methods in one dimension are not very accurate and computationally expensive in comparison to the quadrature methods.

2.2 Monte Carlo Integration in Higher Dimensions

Monte Carlo methods prove to be a useful tool to compute integrals in higher dimensions, otherwise intractable by analytical or numerical methods. As the number of dimensions increase, the volume of the space grows exponentially. The number of points needed to accurately discritise the space become significantly large so methods like Simpson's 1/3 rule become inefficient. This is known as the curse of dimensionality. This section explores applications of Monte Carlo methods in higher dimensions in depth.

2.2.1 Estimating Pi (two dimensions)

Laplace famously showed Pi can be estimated using what we now call Monte Carlo methods. He used the analogy of falling needles and a probabilistic approach to obtain accurate approximations for Pi . All he did was count the number of points that land within a unit circle and divide them by the total number of points. Writing this using symbols, $\pi \approx 4\frac{n_{Inner}}{n_{Total}}$, where number n denotes the of points. (Dunn and Shultis 2012). Simulating this in python equates to estimating Pi in order to investigate the efficacy of Monte Carlo sampling in two dimensions.

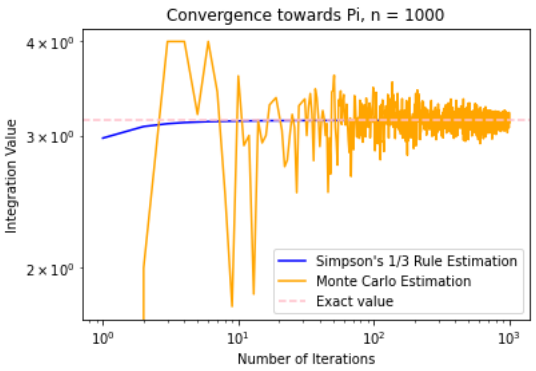


Figure 2.12: Convergence towards Pi.

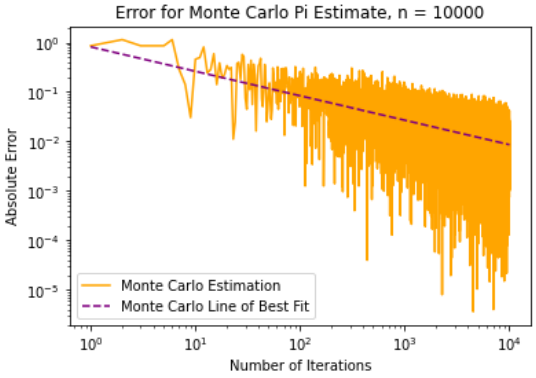


Figure 2.13: Monte Carlo absolute error for Pi.

Figure 2.14: Convergence and Absolute Error plot on a log-log scale for the n-step approximation for Pi.

In figure 2.13 it is evident the Monte-Carlo method performs better for the two-dimensional problem; the absolute error is much smaller in magnitude compared to those in one dimension. In two-dimensions, the ability to sample random values over a larger domain increases the likelihood the points are uniformly distributed (law of large numbers) and land within the unit circle. To play devils advocate, figure 2.12 shows Simpson's rule requires fewer iterations to provide a more accurate estimate for Pi.

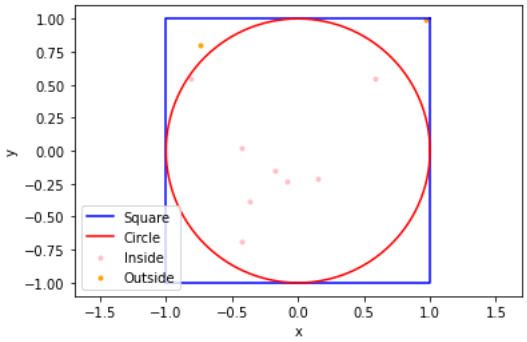


Figure 2.15: $\pi \approx 3.2$ when $n = 10$, 1.8592 % off the true value.

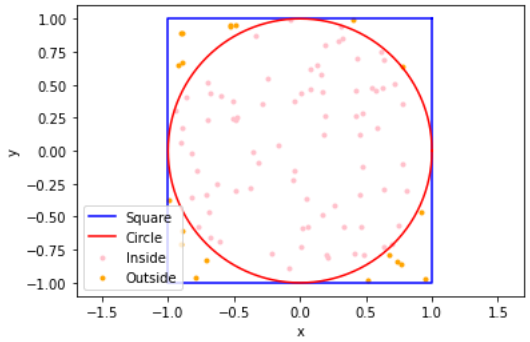


Figure 2.16: $\pi \approx 3.24$ when $n = 100$, 3.1324 % off the true value.

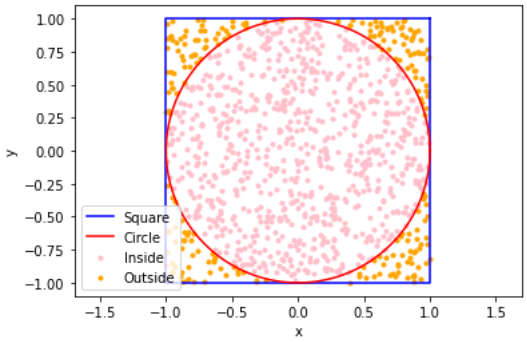


Figure 2.17: $\pi \approx 3.168$ when $n = 1000$, 0.8406 % off the true value.

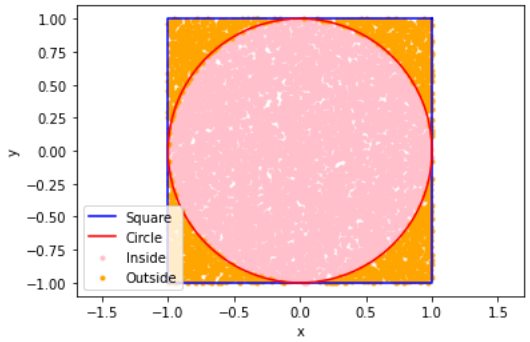


Figure 2.18: $\pi \approx 3.1224$ when $n = 10000$, 0.6109 % off the true value.

Figure 2.19: Estimations for π for different values of n

In figure 2.19 we can visualise how the estimations get better for larger values of n. To achieve an approximation to Pi within 3 decimal places roughly 65,000 iterations are required. Furthermore, in order to achieve an estimation for Pi to 5 decimal places, 3.5 million iterations are required. That result was not reproducible, I got incredibly lucky for one of the runs. Generally, in order to determine how many iterations are required to achieve a level of accuracy above 5 decimal points, extrapolation is needed.

We can reduce the noise in our plot for the absolute errors by averaging them over a set number of runs as above but even so, this is not efficient. Alternatively, another method to reduce the noise in the absolute errors would be to employ stratified random sampling. This method involves randomly sampling for each sub-interval in the domain. Each of the methods are computationally expensive and not worth it as Simpson's $\frac{1}{3}$ is able to produce very convincing results for estimating Pi.

2.2.2 The volume of an N-dimension Hyperspehere

Many applications of integrals involve a domain in more than two dimensions. In this section I explore how Monte Carlo methods to provide estimations for such integrals. In Vector Calculus, methods to evaluate integrals in higher dimensions was explored in depth, more can be read on it in (Evans 2021). The methods previously utilised involved nesting one-dimension integrals as $\int_V f(x)dV = \int_{a_1}^{b_1}\int_{a_2}^{b_2}\cdots\int_{a_N}^{b_N} f(x_1, x_2, \ldots, x_N)dx_N \cdots dx_2 dx_1$, where N is the number of dimensions and V is contained in the hypercube $(\mathbf{a}, \mathbf{b})$. We define f to be zero outside V. It is possible to use the quadrature methods to evaluate integrals in higher dimensions but these become increasingly inefficient and computationally expensive as N gets large.

To further demonstrate how multi-dimensional Monte Carlo estimation works consider the volume of an N-sphere. An N-sphere is the set of points in an (n+1) dimensional Euclidean space where every point is equidistant from the centre. Now consider a unit sphere enclosed in a cube of length two. To obtain a volume estimate of the sphere, determine the ratio between the number of randomly generated points within the sphere and the total number of randomly generated points. This follows directly from the Monte Carlo method used to estimate the value of pi. For even higher dimension spheres, simply consider an N dimension hypercube, still of length two, the N dimension hyper-sphere inscribed within. It is known the exact volume of an N-sphere can be determined by $V_n(r) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}r^n$ where Γ is an extension of the factorial function that can evaluate non-integer arguments. Thus, we can compare the accuracy of Monte Carlo estimates shown in table 2.4. Monte Carlo estimates for larger dimensions is given by:

$$\int_V f(x)dV \approx \frac{V}{n}\sum_{i=1}^{n} f(x_i) \pm V\frac{\sigma}{n}$$

confirming the error scales as $n^{-\frac{1}{2}}$ as well as providing the basis of the function used for the Monte Carlo integration (Adrian Barker 2022).

Although estimations converge towards the theoretical values, the results above surprised me. I was not expecting to observe values decreasing as the volume increases as well as the estimation failing for small values of n.

Dimensions N	Number of Iterations, n	Monte Carlo Estimation	Exact Volume
3	50	3.56	$\frac{4\pi}{3}$
	100	4.32	
	1000	4.248	
5	50	6.4	$\frac{8\pi^2}{15}$
	100	4.48	
	1000	5.568	
7	50	5.12	$\frac{16\pi^3}{105}$
	100	5.12	
	1000	4.544	
9	50	0	$\frac{32\pi^4}{945}$
	100	0	
	1000	3.072	

Table 2.4: Monte Carlo estimates for different dimension hyperspheres

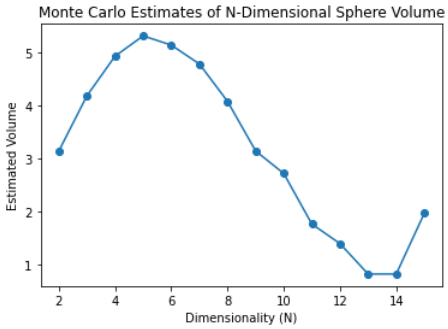


Figure 2.20: Monte Carlo Volume estimation for an N-sphere with n = 100,000.

Higher dimension integrals require many more iterations in order to obtain an estimate. As the number of dimensions increase, the volume of the hypersphere is focused around the boundary layer. As a result, the points randomly generated from the uniform distribution are less likely to 'land' within the volume of the sphere. This means estimations with a lower number of iterations return zero. This is a problem in the sense that the absolute error is 100% but the relative error is infinitely small because the actual volume is infinitely small. This phenomenon is known as the curse of dimensionality. It also affects quadrature methods, the number of divisions needed to estimate the integral grows exponentially, requiring n^N evaluations of the function. This also becomes problematic when $f(x)$ is discontinuous on the domain V.

To obtain more accurate estimates different sampling techniques can be used, these include Quasi-Monte Carlo methods or importance sampling, all of which can be read in depth in (Kalos and Whitlock 1993). Quasi Monte Carlo methods use deterministic sequences of variables that are particularly well distributed throughout the space to minimise errors. This aims to provide results with better convergence compared to traditional Monte Carlo methods. We can implement this method in python with a dedicated module for it, `(scipy.stats.qmc)`, (Virtanen et al. 2021). Importance sampling utilises the use of a probability distribution that is a better fit of the target distribution to provide an estimate with a lower variance. Further down in this chapter I explore how the normal distribution can be used instead. Without access to a quantum computer, my laptop will struggle to generate a very large number of random points in order to efficiently approximate the volume of an N-sphere. The ratio of errors stays the

same for Monte Carlo methods regardless of dimension so it seems the method is only limited by requiring a huge numbers for n.

Instead of drawing random numbers from a uniform distribution we can reduce the variance by choosing a probability distribution $p(x)$ that concentrates the random points in areas of high variance in the integrand to reduce the variance and speed up the convergence towards the exact value of the volume. This is similar to importance sampling, but in importance sampling there is an extra weighting to adjust estimates accordingly. The Monte Carlo estimate is now

$$\langle f \rangle = \int f(x)p(x)dx \approx \frac{(b-a)}{n} \sum_{i=1}^{n} f(x_i).$$

Choosing 10,000 points from a normal distribution, $\mathcal{N}(-1, 1)$, and using it to estimate the volume of a 10 dimension sphere, has less variance for fewer n shown in figure 2.21. However, it is does not produce as good of an estimate compared to the uniform distribution. This suggests the parameters chosen do not fit the function appropriately, using trial and error it is possible to find parameters that produce better results.

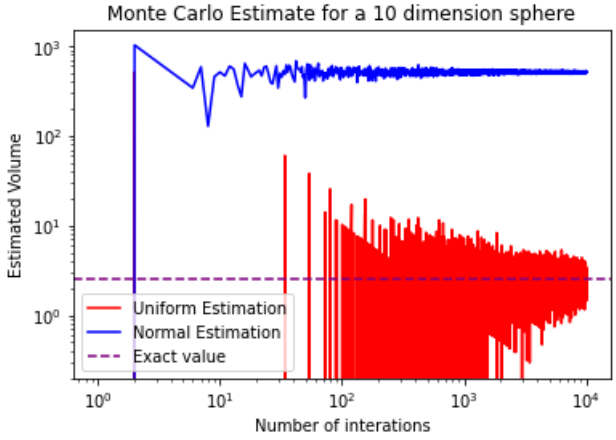
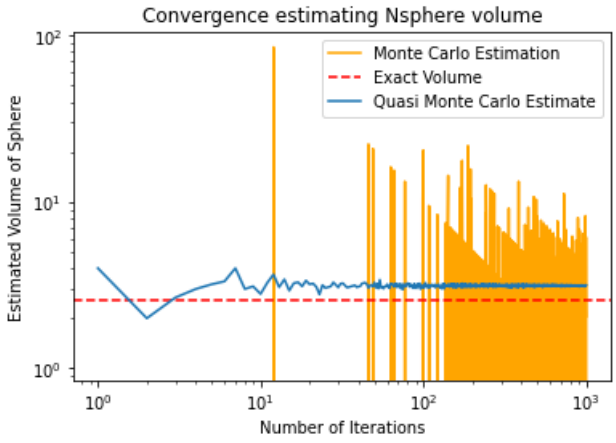


Figure 2.21: Monte Carlo estimation for a 10-sphere with n=10000. Numbers are randomly generated from a uniform distribution and a normal distribution

Instead, using a Quasi-Monte Carlo method for a sphere of 10 dimensions more accurate approximations can be obtained efficiently.



Wrong I did N=2 by accident

Figure 2.22: Quasi-Monte Carlo Sampling, n=1000

.

Even with only 1000 iterations, figure 2.22 demonstrates the effectiveness of the method compared to sampling from a uniform distribution. To an extent, QMC methods are not totally random, the use of low discrepancy sequences like the Sobol sequence mean points are more evenly spaced over the domain of integration resulting in more accurate results. The QMC method does not have the same error proportion as uniform Monte Carlo methods. It can be shown the approximation error for the QMC method is $O\left(\frac{(logN)^s}{N}\right)$. QMC methods do have their own limitations, they can perform badly when singularities exist within a function or when the sequence is chosen badly. Generally speaking however, for large N QMC methods will always outperform random Monte Carlo techniques (Caflisch 1998).

2.2.3 Integration Over Infinite Domains

Many integrals studied in undergraduate mathematics are over infinite domains. In (Mathematical Methods, Griffiths 2022) methods to approximate such integrals were studied in depth. However, Monte Carlo methods are capable of producing precise estimations using appropriate probability distributions. Suppose a problem involves solving $\int_0^\infty exp(-x/\lambda)g(x)dx$. Here the sample distribution best suited is the exponential distribution instead of the uniform distribution. Selecting the correct probability distribution is imperative in order for the Monte Carlo method to work over infinite and semi-infinite domains. The selected distribution should exhibit the following properties:

1. Match the distribution to the integrand: The shape of the distribution should match the shape of the integrand. For example, if the function is a Gaussian function, points should be randomly generated from a normal distribution.
2. Cover the entire domain: For infinite domains, probability distributions with a heavy tail should be used to ensure that the entire domain is covered.
3. Ensure the distribution can be sampled easily: If a distribution is too complex, it can be difficult to sample points from it effectively.
4. Experiment: See what works and what doesn't in order to find the best approach for a specific problem.

Quadrature methods are also capable of working out integrals on infinite domains. In chapter 3, quadrature methods are utilised in order to evaluate functions over an interval that contain singularities.

2.3 Closing Remarks

Integration is a fundamental part of mathematics, forming the basis of many interesting problems. As I have worked through undergraduate mathematics, I have come to realise that functions, especially in real-life applications, cannot be easily evaluated using standard techniques. Quadrature methods and Monte Carlo methods are powerful tools that allow us to approximate such functions. Quadrature methods are simple, easy to implement, and effective ways to estimate functions in lower dimensions, while Monte Carlo methods excel in higher dimensions. Using probability and chance to solve deterministic problems is a marriage between two areas of mathematics I never thought existed. We have seen, increasing iterations generally improves the accuracy of numerically generated estimates. As technology improves, it will be exciting to see how numerical estimation techniques evolve to improve such accuracy. Ultimately, the choice of method depends on the problem at hand and the degree of accuracy one wishes to

obtain. The versatility of Monte Carlo methods make them one of the most useful concepts in maths where they are used in risk analysis, optimisation and financial modelling.

Chapter 3

Period of a Pendulum

The final part of my project will investigate the period of a pendulum. Pendulum's have been used commonly as a classroom tool to demonstrate key physical laws during GCSE and A-level lessons. Despite their simplicity there is no known formula for the period of a pendulum. In this chapter I will use numerical computations to recover known results as well as investigate the issues faced when trying to determine numerical solutions to the problem.

3.1 What is a Pendulum?

A pendulum is defined as a body suspended from a fixed point so that it oscillates back and forth under the influence of gravity.

When a pendulum is displaced sideways from it's equilibrium position, due to the force of gravity, it is subject to a restoring force that will accelerate it back towards the equilibrium position. This restoring force causes the pendulum to oscillate back and forth about the equilibrium position. Pendulums are complex, the motion of a pendulum is non-linear as the force of gravity changes as a pendulum swings.

Friction is difficult to account for when trying to model a pendulum, the bob experiences drag and there is also friction between the pivot and rod. As a result pendulums experience energy loss that is challenging to model.

The motion of a pendulum is very sensitive to its initial conditions, explored later in this chapter, small changes can have a large impact. Length and mass of the rod and bob pose a further challenge when determining the period of a pendulum as well as the tension of the rod. All these factors have to be accounted for when modelling the motion of a pendulum and it becomes quite the challenge.

Pendulums are commonly used in clocks and theme parks rides. Understanding how a pendulum behaves aids the understanding gravity, inertia and centripetal forces.

Figure 3.1: A pendulum at an amusement park. Credit: ThemeParkJames

3.1.1 A Simple Pendulum

A series of assumptions can be made in order to simplify the motion of a pendulum in order to model the behaviour. In the case of a *simple pendulum* the equations of motion can be solved analytically for small angle oscillations explored in section 3.2. A simple pendulum has a mass-less rod and friction-less pivot. The hanging object attached to the rod, of length l, is known as the bob and the maximum value of the angle θ between the rod and the equilibrium is known as the amplitude. In this specific interpretation of a pendulum, only motion in two-dimension is considered. From these assumptions the governing equation for the motion of a pendulum is simply given by

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}sin\theta. \tag{3.1}$$

This is illustrated nicely in the figure below:

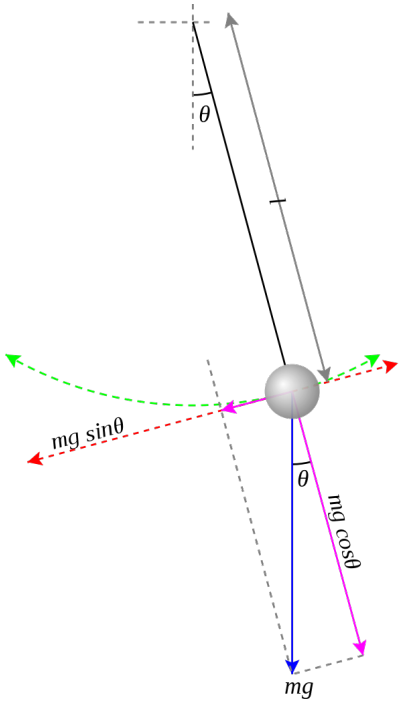


Figure 3.2: A simple pendulum with amplitude θ. θ is measured in radians, the blue arrow is the force of gravity acting on the bob, the direction of the bob's velocity is always considered to be perpendicular to the rod, called the tangential axis in red. Credit: Wikipedia

To arrive at (3.1) first consider Newton's second law $F = ma$. Let θ be the angle between pivot and the rod and the only force acting on the bob is gravity, g. Using trigonometry the magnitude of the force components can be calculated. Thus,

$$F = -mgsin\theta = ma,$$

so

$$a = -gsin\theta. \tag{3.2}$$

Newtons second law is only applied to the tangential component of the force since the system is restricted to motion that follows the path of an arc. The negative sign arises as because θ and a are always in the opposite direction. Intuitively this makes sense, as a pendulum swings further to the left you would expect to to accelerate back toward the equilibrium position on the right. The linear acceleration a along the tangential axis is related to the change in θ and

arc displacement to give:

$s = l\theta$	s denotes the arc length and l is the length of the rod,
$v = \frac{ds}{dt} = l\frac{d\theta}{dt}$	instantaneous velocity of the bob, by differentiating w.r.t t
$a = \frac{d^2s\theta}{dt^2} = l\frac{d^2\theta}{dt^2}$	acceleration of the bob, differentiating w.r.t t again,
$l\frac{d^2\theta}{dt^2} = -gsin\theta$	using Newton's second law,
$\frac{d^2\theta}{dt^2} = -\frac{g}{l}sin\theta$	by rearranging and hence we have recovered 3.1.

To make the equation dimensionless, introduce a re-scaled nondimensional time $t = \tau/k$, where k is a constant with units of time. Differentiate this expression with respect to t to obtain:

$$\frac{d}{dt} = \frac{1}{k}\frac{d}{d\tau}.$$

We want to express the second derivative of θ with respect to t in terms of θ and τ. To do this, we use the chain rule for differentiation, which states that:

$$\frac{d\theta}{dt} = \frac{d\theta}{d\tau} \cdot \frac{d\tau}{dt}.$$

Applying the chain rule again, we can find the second derivative of θ with respect to t:

$$\frac{d^2\theta}{dt^2} = \frac{d}{dt}\left(\frac{d\theta}{dt}\right) = \frac{d}{dt}\left(\frac{d\theta}{d\tau}\right) \cdot \frac{d\tau}{dt} = \frac{d}{d\tau}\left(\frac{d\theta}{d\tau} \cdot \frac{d\tau}{dt}\right) \cdot \frac{d\tau}{dt} = \frac{1}{k^2}\left(\frac{d^2\theta}{d\tau^2}\right).$$

Substituting this expression into the original governing equation, we get:

$$\frac{1}{k^2}\left(\frac{d^2\theta}{d\tau^2}\right) = -\frac{g}{l}\sin\theta.$$

Let $k = \sqrt{\frac{l}{g}}$ to get:

$$\frac{d^2\theta}{d\tau^2} = -\sin\theta,$$

denoted by

$$\ddot{\theta} = -\sin\theta, \tag{3.3}$$

with the initial conditions $\theta(0) = a$, the initial release angle of the pendulum and $\dot{\theta}(0) = 0$ the initial velocity of the pendulum.

Equation (3.3) seems simple but there is no known solution to it. As the problem being explored concerns a simple pendulum, energy losses due to friction or drag do not need to be accounted for. Therefore, the governing equation for the motion of a pendulum can also be derived using the conservation of energy in mind.

3.1.2 Energy Derivation of the Governing Equation

Once the pendulum is released it converts gravitational potential energy into kinetic energy. The change in potential energy and kinetic energy is given by the following equations:

$$\Delta U = mgh \text{ and } \Delta\frac{1}{2}mv^2,$$

where h is the change in height of the bob as it swings. Since energy is conserved; the gain in kinetic energy is equal to the loss in gravitational potential energy, so the system is described by

$$mgh = \frac{1}{2}mv^2.$$

Throughout each oscillation, the value of h changes as is swings, illustrated clearly in the figure below:

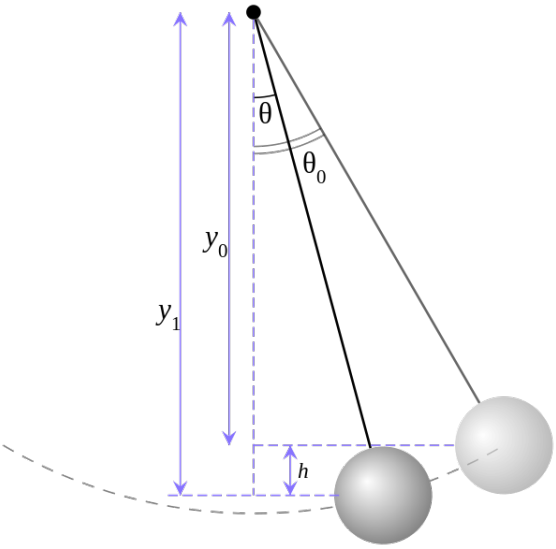


Figure 3.3: The motion of a pendulum with energy conservation components. Credit: CK12.org

The change in velocity for a given change in height can be expressed as $v = \sqrt{2gh}$. The displacement is still the arc length s, so the velocity equation can be expressed as $\frac{d\theta}{dt}$.

$$v = l\frac{d\theta}{dt} = \sqrt{2gh} \rightarrow \frac{d\theta}{dt} = \frac{\sqrt{2gh}}{l},$$

where h is the vertical distance of the pendulum at different points.

Suppose the pendulum starts to swing from the initial release angle a, then y, the vertical distance from the pivot is given by

$$y_0 = lcos(a)$$

and for a given point y_1 along the arc,

$$y_1 = lcos(\theta).$$

Consequently, h is the difference between the two:

$$h = l(cos(\theta) - cos(a))$$

$$\Rightarrow \frac{d\theta}{dt} = \sqrt{\frac{2g}{l}(cos(\theta) - cos(a))}, \text{ known as the first integral of motion.} \tag{3.4}$$

Equation (3.4) describes the velocity in terms of the location of the pendulum. It also contains an integration constant related to the known initial release angle a. Differentiating (3.4) using the chain rule with respect to time t to get the acceleration:

$$\begin{aligned}
\frac{d^2\theta}{dt^2} &= \frac{1}{2}\left(\frac{-sin\theta\frac{2g}{l}}{\sqrt{\frac{2g}{l}(cos\theta - cosa)}}\right)\frac{d\theta}{dt} \\
&= \frac{1}{2}\left(\frac{-sin\theta\frac{2g}{l}}{\sqrt{\frac{2g}{l}(cos\theta - cosa)}}\right)\sqrt{\frac{2g}{l}(cos\theta - cosa)} \\
&= -\frac{g}{l}sin\theta \\
&\Rightarrow \frac{d^2\theta}{dt^2} + \frac{g}{l}sin\theta = 0.
\end{aligned}$$

Thus, arriving at (3.1) as required.

3.2 Small Amplitude Approximation

Trigonometric functions can be approximated for small angles making the governing system easier to evaluate. To apply this to the period of a pendulum, consider a release angle a that is much smaller than 1 radian, $a << 1$. Then trigonometric functions can be expressed as follows: $sin(\theta) \approx \theta$, $cos(\theta) \approx 1 - \frac{\theta^2}{2} \approx 1$ and $\tan(\theta) \approx \theta$. This is nicely illustrated in the figure below:

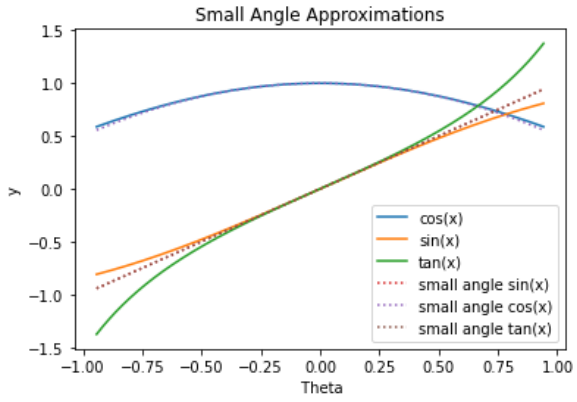


Figure 3.4: Small angle approximations evaluated and compared to the exact trigonometric function.

The small angle approximations are derived from the Taylor series expansions of each trigonometric function. Consider the Taylor expansion for sine:

$$\sin\theta \approx \theta - \frac{\theta^3}{6} + \frac{\theta^5}{120}\ldots, \text{ where the terms of order } \theta^3 \text{ and above are neglected.}$$

Similarly, Taylor's expansion is used to arrive at the small angle approximations for both cosine and tangent functions. Like with any approximation, there is an error. This error is easy to quantify simply as the order of the neglected term. For example, for sine the order of the neglected term is of $O(\theta^3)$ which is equivalent to the order of the error. As for cosine the order of the neglected term is $O(\theta^4)$. To show this error, I have plotted the relative percentage error

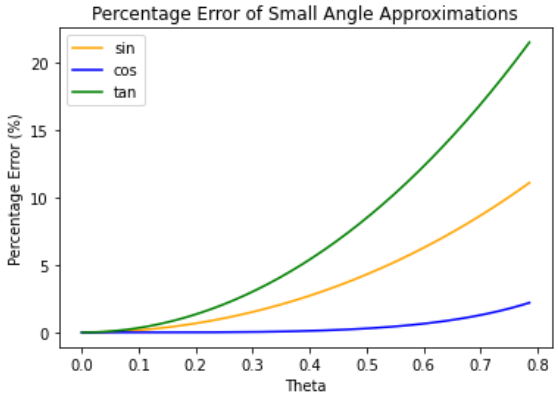


Figure 3.5: Percentage error for different angles when using small angle approximations

in Python. As you can see, the value of the percentage error grows rapidly for larger values of theta. This is because the θ^3 term increases beyond the point of it being negligible.

As figure 3.4 highlights, for release angles $a << 1$, approximations can be made about that value of θ with minimal error. So (3.3) can be re-written as

$$\ddot{\theta} = -\theta, \text{with initial conditions } \theta(0) = a \text{ and } \dot{\theta}(0) = 0. \tag{3.5}$$

The system can be solved using standard techniques for homogeneous second order differential equations.

$$\ddot{\theta} = -\theta$$
$$\frac{d^2\theta}{d\tau^2} + \theta = 0$$

First, writing the auxiliary equation:

$$\lambda^2 + 1 = 0,$$

solving for λ returns

$$\lambda = \pm i$$

Thus, the general solution for the system is

$$A_1 cos(\tau) + A_2 sin(\tau)$$

where A_1 and A_2 are constants to be determined.

Using the initial conditions $\theta(0) = a$ and $\dot{\theta}(0) = 0$ to find the particular solution.

$$\theta(0) = A_1 \cos(0) + A_2 \sin(0) = A_1 = a$$

$$\dot{\theta}(0) = -A_1 \sin(0) + A_2 \cos(0) = A_2 = 0.$$

Therefore, the solution to the differential equation with the given initial conditions is:

$$\theta(t) = a \cos(\tau).$$

The period of a pendulum is the time is takes to complete one full oscillation denoted by T. The solution to the differential equation (3.5) represents the motion of a simple pendulum, which oscillates sinusoidally with a period of 2π and an amplitude of a.

The period can also be found by using an integral method. To do this, first multiply (3.3) by $\dot{\theta}$ then integrate with respect to τ.

$$\dot{\theta}\ddot{\theta} = -\theta\dot{\theta},$$

$$\frac{d}{d\tau}\left(\frac{\dot{\theta}^2}{2}\right) = -\frac{d}{d\tau}\left(\frac{\theta^2}{2}\right),$$

$$\frac{d}{d\tau}\left(\frac{\dot{\theta}^2}{2} + \frac{\theta^2}{2}\right) = 0, \ (\frac{\dot{\theta}^2}{2} = \text{kinetic energy}, \ \frac{\theta^2}{2} = \text{potential energy}).$$

$$\frac{\dot{\theta}^2}{2} + \frac{\theta^2}{2} = constant = \frac{a^2}{2},$$

$$\dot{\theta}^2 = a^2 - \theta^2,$$

$$\frac{d\theta}{d\tau} = -\sqrt{a^2 - \theta^2}, \tag{3.6}$$

$$\frac{d\tau}{d\theta} = -\frac{1}{\sqrt{a^2 - \theta^2}} \text{ by taking the reciprocal.} \tag{3.7}$$

The first equation describes the motion for the pendulum simply relating the angular displacement to the gravitational acceleration and the instantaneous velocity of the pendulum. It also shows the velocity of the pendulum is in the opposite direction of the acceleration.

The second equation is just writing the above in terms of a a derivative w.r.t time in more familiar notation. In words it shows the rate of change of kinetic energy is equal to the opposite to the rate of change in gravitational potential energy of the system.

The third equation follows from the energy derivation of the governing equation, showing the energy in the system is conserved. Recall this is a key assumption of a simple pendulum explained above.

By integrating both sides with respect to τ the sum of energy in the system is equal to some constant, namely $\frac{a}{2}$. This is as expected since the total energy in the system is conserved. Also, for this specific system the potential energy is proportional to the square of the displacement.

Taking the derivative with respect to time τ describes the relationship between the velocity and angular displacement.

Lastly, the final equation relates the time it takes to complete one full oscillation and thus the period of the pendulum. To find the period T, integrating (3.7)over the domain $0 < \tau < T/4$ recovers the known period of 2π for small release angle a.

$$d\tau = -\frac{d\theta}{\sqrt{a^2 - \theta^2}} \quad \text{and integrate both sides,}$$

$$\int_0^{T/4} d\tau = \int_0^a \frac{d\theta}{\sqrt{a^2 - \theta^2}},$$

$$\int_0^{T/4} d\tau = \frac{1}{a}\int_0^a \frac{d\theta}{\sqrt{1-\frac{\theta^2}{a^2}}},$$

$$\frac{T}{4} = \frac{1}{a}\int_0^a \frac{d\theta}{\sqrt{1-\frac{\theta^2}{a^2}}}.$$

Using substitution let $\frac{\theta}{a} = sinu \Rightarrow d\theta = acosu \cdot du$, the integral can be expressed as:

$$\frac{T}{4} = \frac{1}{a}\int_0^a \frac{acosu}{\sqrt{1-sin^2u}}du.$$

By observation the numerator and denominator are equivalent because of the Pythagorean trigonometric identity and so,

$$\frac{T}{4} = \left[arcsin\left(\frac{\theta}{a}\right)\right]_0^a$$

So the period T is calculated as

$$T = 2\pi.$$

The limit of integration is chosen to be $\frac{T}{4}$ because the pendulum is symmetric about the equilibrium point. Since the period measures the time taken for the pendulum to return to the original release angle, the time it takes for the pendulum to swing from its maximum positive angle to its maximum negative angle is one-fourth of the period and hence why the integral bound is chosen to be $\frac{T}{4}$. Many textbooks and literature online denote the period of a simple pendulum for small angle $T = 2\pi\sqrt{\frac{l}{g}}$ which is the same as what is derived above but here I have non-dimensionalised the governing equations (Lumen 2023). Hence, for small release angles, the result of $T \approx 2\pi$ has been recovered and is illustrated in figure 3.6. It is particularly

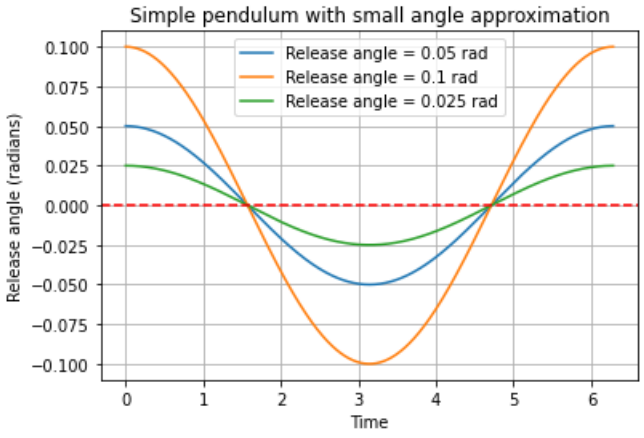


Figure 3.6: Period of one oscillation for different small release angles a. All of the periods are 2π as expected

interesting how gravity is related to time. This means the period of the same pendulum with the same release angle will be different for different places on earth where there are slight variations in gravitational acceleration. Such accuracy proves to be very useful in clocks and other measurement instruments since they do not rely on external influence.

The period T is a perfect example of simple harmonic motion which is explored in depth in many Leeds modules such as *Multivariable Calculus*. Using small angles makes it easier to find the period of the pendulum as investigate the motion for small release angles. To summarise, here are some key properties for the system.

1. The motion is periodic and is governed by a simple harmonic equation.
2. The motion is isochronous meaning the period T is independent of the release angle highlighted in figure 3.6.
3. The maximum displacement (amplitude) is proportional to the displacement.
4. Kinetic energy and gravitational energy are conserved in the system.
5. Kinetic energy changes and changes in gravitational potential energy are directly proportional.
6. The restoring force is proportional to the displacement.

What happens if the release angle is not sufficiently small? Small angle approximations can no longer be employed so return back to the un-approximated system (3.3).

3.3 Arbitrary Amplitudes and Their Consequences

For amplitudes too big for small angle approximations, the motion of the pendulum is no longer described by simple harmonic motion. The period T is much larger as the angle of displacement increases due to the non-linearity of the gravitational restoring forces. For small displacement angles, the restoring force is proportional to the displacement but this is not the case for larger release angles and hence lager displacements.

The force of gravity acting on the bob is no longer constant throughout the motion of the pendulum. As the pendulum swings, the displacement angle changes, and so does the force of gravity acting on the pendulum. As a result, the restoring force is no longer directly proportional to the displacement, and the motion of the pendulum becomes more complex. Mathematically, this is written as $F = -mgsin\theta$, where F is the restoring force.

Energy conservation for the system remains the same, but now potential and kinetic energy are no longer directly proportional to each other and no longer in phase with each other. The potential energy of the pendulum now depends on $sin\theta$ and the kinetic energy depends on the square of the velocity. As the release angle increases, the angle of displacement also increases, causing the potential energy to increase. At the same time, the velocity of the pendulum also increases, causing the kinetic energy to increase. However, the rate at which the potential energy increases is not proportional to the rate at which the kinetic energy increases, due to the non-linear nature of the system. This is explored in depth in (Strogatz 2014) in Chapter 6. This complex behaviour can result in chaos over time and makes it difficult to find the period. To make things even more challenging the periodic behaviour of the pendulum is no longer constant too. Refer back to (3.3), re-writing it as an integral like (3.7) it follows that the period for the pendulum is

$$T = 2\sqrt{2}\int_0^a \frac{d\theta}{\sqrt{cos\theta - cosa.}} \tag{3.8}$$

This integral cannot be solved using elementary methods, thus, to solve it for a given release angle quadrature structures can be employed. When using a quadrature structure, it is important to notice the integrand becomes unbounded as $\theta \to a^-$ (Adrian Barker 2022). To derive (3.8) adopt a similar approach to how (3.7) was determined and multiply the governing equation

through by $\dot{\theta}$:

$$\ddot{\theta}\dot{\theta} = -\dot{\theta}sin\theta$$

We know $\ddot{\theta}\dot{\theta} = \frac{d}{d\tau}\left(\frac{1}{2}\dot{\theta}^2\right)$ and $\dot{\theta}sin\theta = \frac{d}{d\tau}\left(-cos\theta\right)$. Thus, it follows

$$\frac{d}{d\tau}\left(\ddot{\theta}\right) = -\frac{d}{d\tau}sin\theta,$$

$$\frac{d}{d\tau}\left(\frac{\dot{\theta}^2}{2}\right) = -\frac{d}{d\tau}\left(-cos\theta\right),$$

$$\frac{d}{d\tau}\left(\frac{\dot{\theta}^2}{2} - cos\theta\right) = 0,$$

$$\frac{\dot{\theta}^2}{2} + cos\theta = constant = cosa \textit{ from the initial conditions},$$

$$\frac{\dot{\theta}^2}{2} = cos\theta - cosa,$$

$$\dot{\theta}^2 = 2\left(cos\theta - cosa\right),$$

$$\frac{d\theta}{d\tau} = \pm\sqrt{2}\sqrt{cos\theta - cosa} \textit{ (Integral of motion)},$$

$$\frac{d\tau}{d\theta} = \frac{\sqrt{2}}{2}\frac{1}{\sqrt{cos\theta - cosa}},$$

Using separation of variables and integrating between $0 < \tau < T/4$ in order to achieve the quarter period.

$$\int_0^{T/4} d\tau = \frac{\sqrt{2}}{2}\int_0^a \frac{d\theta}{\sqrt{cos\theta - cosa}} \rightarrow \frac{T}{4} = \frac{1}{\sqrt{2}}\int_0^a \frac{d\theta}{\sqrt{cos\theta - cosa}} \tag{3.9}$$

3.3.1 Energy Derivation for the Period

It follows that 3.9 can be derived using energy principles in mind. Recall equation 3.4, the integral of motion which was derived with using conservation of energy principles. Since in this section, I have non-dimensionalised the governing equation so it now reads

$$\frac{d\theta}{d\tau} = \sqrt{2\left(cos\theta - cosa\right)}. \tag{3.10}$$

Using separation of variables it is possible to derive a formula for $\dot{\theta}$ at time τ. Integrating with respect to τ from time 0 to $T/4$ and therefore a and θ gives

$$\frac{1}{\sqrt{2}}\int_0^a \frac{d\theta}{\sqrt{cos\theta - cosa}} = \int_0^{T/4} d\tau.$$

Evaluating the integral on the right hand side gives

$$\frac{1}{\sqrt{2}}\int_0^a \frac{d\theta}{\sqrt{cos\theta - cosa}} = \frac{T}{4}.$$

This describes the time taken τ for the bob to drop to an angle θ. The limits of integration describe the time taken for the pendulum to reach the equilibrium position after being dropped

from an initial angle a and time $\tau = 0$. Since it is assumed the periodicity is symmetrical the period of the oscillation is 4 times the time it takes to reach the lowest point. So we have

$$T = 2\sqrt{2}\int_0^a \frac{d\theta}{\sqrt{cos\theta - cosa}},\ QED \tag{3.11}$$

which considers all displacement angles. Also note, this method is very much intertwined with the force derivation for the period. In line seven of the working out above, the integral of motion is spat out. Interestingly, the same formula (3.11) can be derived using torque and angular acceleration. It is explored in (Wikipedia 2022) and gone into detail in Srvirin 2023. Personally I preferred the latter, if interested you can explore the torque derivation, however it is fairly repetitive to the two above methods. The result for the period T in this source is

$$4\int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - k^2 sin^2\theta}}, \tag{3.12}$$

where $sin\frac{a_0}{2} = k$.

I mentioned above the integral requires quadrature methods to solve it as it becomes unbounded, that isn't to say known values for the period cannot be recovered. One such known value is $T = \frac{(\Gamma(\frac{1}{4}))^2}{\sqrt{\pi}} \approx 7.4163$ when $a = \frac{\pi}{2}$. This result follows nicely from (3.11) substituting $\frac{\pi}{2}$ for a. Under these conditions the equation is simplified to

$$T_{\frac{\pi}{2}} = 2\sqrt{2}\int_0^{\pi/2} \sqrt{sec\theta} = \sqrt{2}K\left(\frac{1}{2}\right) \approx 2.62206 \tag{3.13}$$

according to symbolab and Wolfram Alpha, where k is the complete elliptic integral of the first kind. Elliptic integrals are explored further on in this chapter. It follows from equation (3.13) the period is

$$T \approx 2\sqrt{2} \cdot 2.622206 \approx 7.41631.$$

Also, consider a vertical release angle $a = \frac{\pi}{2}$. Equation (3.11) is now

$$T_\pi = 2\sqrt{2}\int_0^{\pi} \frac{d\theta}{\sqrt{cos\theta + 1}}.$$

This integral becomes unbounded as it approaches π. Using trigonometric identities $cos^2\frac{\theta}{2} = 1 + cos\theta$ and thus the integrand can be expressed as

$$2\int_0^{\pi} sec\frac{\theta}{2} d\theta.$$

When this seemingly simple integral is evaluated with the specified initial conditions the results tend towards infinity since the function does not converge. Whilst this might seem pointless, it tells us that starting the pendulum from a a vertical position will not move and hence the period is infinitely long or conversely a pendulum will never reach the vertical.

3.3.2 Elliptical Integrals and Pendulums

The topic of elliptic integrals is vast, complex and different to anything I have studied in undergraduate mathematics whilst at Leeds. However, they play a vital role in evaluating the time taken for the pendulum to complete an oscillation when small angle approximations cannot be used.

Elliptical integrals provide us with a method to integrate functions that cannot be expressed in elementary functions. In section 3.3.1 the integrals evaluated either imploded on themselves

or were increasingly large; only solvable using elliptic integrals. There are two types of elliptic integrals; complete and incomplete. Complete elliptic integrals are functions with one argument (3.13) and incomplete integrals are functions with two arguments. Incomplete elliptic integrals of the first kind are defined as

$$F(\phi, k) = \int_0^{\phi} \frac{1}{\sqrt{1 - k^2 \sin^2 \theta}} d\theta \tag{3.14}$$

where $0 \leq \phi \leq \frac{\pi}{2}$ is the Jacobi amplitude of the elliptic function and $0 < k^2 < 1$ is the elliptic modulus (*Wolfram Alpha*). Letting $t = sin\theta$ equation (3.14) can be written as $\int_0^{sin\phi} \frac{dt}{(1-k^2t^2)(1-t^2)}$. Elliptic integrals are said to be complete and of the first kind when $\phi = \frac{\pi}{2}$, that is to say

$$K(k) = \int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{1 - k^2 \sin^2 \theta}} d\theta \tag{3.15}$$

where $0 \leq k < 1$ is the elliptic modulus. Notice that this is the same form as (3.12) and consequently used to solve (3.13). Similarly, it can be written as $\int_0^1 \frac{dt}{\sqrt{(1-t^2)(1-k^2t^2)}}$ using the same substitution explained above. There are known values for the period computed using the gamma function. The question then arises - how are elliptic integrals related to the gamma function? The gamma function is defined as

$$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt, \tag{3.16}$$

where z is a complex or real number except for negative integers (Stewart 2015), known as Euler's integral form. Also, note $K(k)$ can be expressed in a power series:

$$K\left(k\right) = \frac{\pi}{2}\left\{1 + \left(\frac{1}{2}\right)^2 k^2 + \left(\frac{1 \cdot 3}{2 \cdot 4}\right)^2 k^4 + \left(\frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6}\right)^2 k^6 + \ldots + \left[\frac{(2n-1)!!}{(2n)!!}\right]^2 k^{2n} + \ldots\right\}. \tag{3.17}$$

This means the elliptic integral of the first kind can be expressed in terms of the gamma function so we have:

$$K(k) = \frac{(\Gamma(\frac{1}{4}))^2}{\sqrt{\pi}}. \tag{3.18}$$

by setting $k = 0$ and $a = \frac{\pi}{2}$. The above relation is explored in more detail in (Jamie 2015). Whilst elliptic integrals are one way to evaluate integrals of this kind, approximation and numerical techniques are preferred. The numerical methods relate heavily to chapter 2 of this report and are generally simpler to implement and produce accurate results.

3.4 Numerical Methods

Just like chapter 2, numerical methods prove to be a powerful tool used to give estimates for the period of a pendulum. Refer back to the trapezium rule in subsection 2.1.2 and the Simpson's rule in subsection 2.1.3. The integral (3.9) when $a = 2$ simplifies to (3.13) which was evaluated using elliptical integrals. However, using the trapezium rule or the Simpson's rule to estimate the integral and discarding the singularity at $\frac{\pi}{2}$ is far easier to do providing acceptable results.

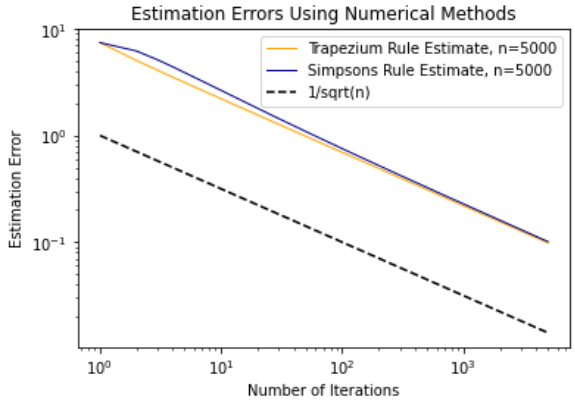


Figure 3.7: Numerical errors Using numerical methods to estimate the period of the pendulum, T.

The graph shows, even for 1000 iterations there is still a substantial error, that is, the rate of convergence is low. Many iterations are required in order to obtain an estimate with negligible error. The actual value for the period, calculated by the gamma function is 7.41629871 (8dp), yet the closest approximation is still some way off. The graph also highlights the rate of convergence is only $\frac{1}{\sqrt{n}}$ for both the numerical methods, given by the gradient of the lines in figure 3.7. The decrease in the rate of convergence is a result of 'chopping off' the singularity at the upper integration limit, as a result, we have less information about how the function behaves at the singularity as thus less accurate results.

Number of Iterations	Trapezium Rule Estimate	Simpson's Rule Estimate
10	6.34083889	4.77321676
100	7.07586536	6.66349090
200	7.17557414	6.89298574
1000	7.30864315	7.19215197
5000	7.31802034	7.31516891

Table 3.1: Numerical method estimates for the period of a pendulum.

Unfortunately, the method above is not able to provide results with high enough accuracy. Instead a more sophisticated technique is to integrate the improper integral associated with the singular endpoint exactly. The remainder can be evaluated using standard numerical methods. Utilising methods familiar to (Griffiths 2022), using a Taylor expansion to express the integrand is a useful way to approach a problem like this.

3.5 Dealing with Singularities

Making use of Taylor series expansions is an elegant way to deal with the singularity at $a = \frac{\pi}{2}$. We have a formula for the period, (3.11)

$$T = 2\sqrt{2}\int_0^a \frac{d\theta}{\sqrt{cos\theta - cosa}},$$

however, as covered in section 3.4 this function is not well defined for the release angle $\frac{\pi}{2}$ and caused the numerical methods to fail. Recall the formula for Taylor series expansions centred

at a, given below:

$$\sum_{n=0}^{\infty} \frac{f^n(a)}{n!}(x-a)^n.$$

Expanding the *cosine* function, a standard result that can be found in lots of mathematical literature.

$$\cos(\theta) = \cos(a) - (\theta - a)\sin(a) - \frac{(\theta-a)^2}{2}\cos(a) + \frac{(\theta-a)^3}{6}\sin(a) + \frac{(\theta-a)^4}{24}\cos(a) - \cdots$$

using this Taylor expansion, centred at $a = \frac{\pi}{2}$, substitute it into the left hand side of the expression to get:

$$\frac{1}{\sqrt{\cos\theta - \cos a}} = \left[(a-\theta)\sin a - \frac{(\theta-a)^2}{2}\cos a + \frac{(\theta-a)^3}{6}\sin a\right]^{-\frac{1}{2}}$$

Rewriting the right hand side to obtain:

$$\frac{1}{\sqrt{\cos\theta - \cos a}} = \frac{1}{\sqrt{(\sin a)(a-\theta)}}\left[1 + \frac{1}{2}(a-\theta)\frac{\cos a}{\sin a} - \frac{1}{6}(a-\theta)^2 + ...\right]^{-\frac{1}{2}} as \quad \theta \to a^-.$$

Using a clever trick the right hand side can be re-written as:

$$\frac{1}{\sqrt{\cos\theta - \cos a}} = \frac{1}{\sqrt{\cos\theta - \cos a}} + \frac{1}{\sqrt{(\sin a)(a-\theta)}} - \frac{1}{\sqrt{(\sin a)(a-\theta)}}. \tag{3.19}$$

The above step is obscure, however it allows the integral to be expressed a sum of two integrals in order to exactly consider the singularity at $a = \frac{\pi}{2}$. We are able to integrate completely over the interval now.

$$\int_0^{\frac{\pi}{2}} \left(\frac{1}{\sqrt{\cos\theta - \cos a}} + \frac{1}{\sqrt{\sin a(\theta - a)}}\right) d\theta - \int_0^{\frac{\pi}{2}} \left(\frac{1}{\sqrt{\sin a(\theta - a)}}\right) d\theta \tag{3.20}$$

The integral the right hand side is regular and comes out to be

$$\int_0^a \frac{1}{\sqrt{(\sin a)(a-\theta)}} d\theta = \frac{2\sqrt{a}}{\sqrt{\sin a}}.$$

Whilst there is still a singularity, the difference between *cosine* and *sine* parts in integrand is zero at the end-point according to the Taylor expansion. Putting everything together, the period T can now be calculated even with a singularity and one of the endpoints of the integral.

$$T = 4\sqrt{\frac{2a}{\sin a}} + 2\sqrt{2}\int_0^a \left(\frac{1}{\sqrt{\cos\theta - \cos a}} - \frac{1}{\sqrt{(\sin a)(a-\theta)}}\right) d\theta.\ QED \tag{3.21}$$

The result is a function that is well-behaved, allowing the period T to be calculated even for challenging release angles. This is because if we omit the endpoint from our integral by simply setting the integrand to zero there, this formula exactly accounts for the end-point because the integrand is actually zero there from the Taylor expansion of cosine. In figure 3.8 the impact of making this change results in a steeper gradient for the errors implying faster convergence. For the trapezium rule the gradient is $-n$ and for the Simpson's rule it is $-2.5n$, which is an improvement compared to 'chopping off' the singularity done previously.

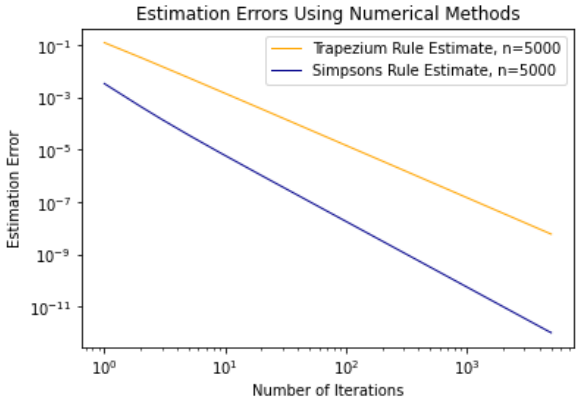


Figure 3.8: Estimation Error using numerical methods for the updated function to better deal with the singularity.

3.6 Closing Remarks

A seemingly simple task to those not familiar with the mathematics used to model the motion of a pendulum, its period is no easy so calculate. It is a subject that has been studied in depth and still there is no known solution. This chapter of my project demonstrates how the problem quickly becomes complex and intertwines with subjects in mathematics I've never come across before. Making use of quadrature structure, Taylor series and asymptotic expansions, the period of a pendulum, regardless of its release conditions can be determined. The practical and historical importance associated with the period of a pendulum makes it a worthwhile problem to investigate. While the period of a pendulum might seem like a specific and narrow concept, its significance extends far beyond the motion of a swinging weight. Its played a vital role in scientific discovery such as the development of classical mechanics all as well as in GCSE physics classrooms.

References

Adrian Barker, Stephen Griffiths (2022). *MATH3001 Computational Applied Mathematics*.

Caflisch, Russel E. (1998). "Monte Carlo and quasi-Monte Carlo methods". In: *Acta Numerica* 7, pp. 1–49. DOI: 10.1017/S0962492900002804.

Chapra, Steven C. (2022). *Applied Numerical Methods With MATLAB for Engineers and Scientists*. McGraw-Hill Education.

Dunn, William L. and Shultis, J. Kenneth (2012). "1 - Introduction". In: *Exploring Monte Carlo Methods*. Ed. by William L. Dunn and J. Kenneth Shultis. Amsterdam: Elsevier, pp. 1–20. ISBN: 978-0-444-51575-9. DOI: https://doi.org/10.1016/B978-0-444-51575-9.00001-4. URL: https://www.sciencedirect.com/science/article/pii/B9780444515759000014.

Euler Forward Method (2022). Accessed: November 12, 2022. Wolfram Web. URL: https://mathworld.wolfram.com/EulerForwardMethod.html (visited on 08/11/2022).

Evans, Dr R. M. L. (2021). *MATH2365 Vector Calculus*. University of Leeds.

Griffiths, Stephen (2022). *Math3365 Mathematical Methods*. University of Leeds.

Hairer, Ernst, Lubich, Christian, and Wanner, Gerhard (2006). *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*. Springer Science & Business Media.

Jamie, Snape (2015). "Applications of Elliptic Functions In Classical and Algebraic Geometry". In: URL: https://www.jamiesnape.io/assets/publications/mmath/dissertation.pdf.

Kalos, Malvin H. and Whitlock, Peter A. (1993). *Monte Carlo Methods*. John Wiley & Sons.

Lumen (2023). *The Simple Pendulum*. https://courses.lumenlearning.com/suny-physics/chapter/16-4-the-simple-pendulum/. Accessed on March 11, 2023.

MathIsFun (2019). *Rotational Symmetry*. https://www.mathsisfun.com/definitions/rotational-symmetry.html. Accessed on: March 22, 2023.

Monte Carlo Method (2023). Accessed: March 8, 2023. URL: https://en.wikipedia.org/wiki/Monte_Carlo_method.

Numerical Integration and Monte Carlo Methods (2001). Accessed: February 9, 2023. URL: https://lcn.people.uic.edu/classes/che205s17/docs/che205s17_reading_01f.pdf.

Pang, Tao (2006). *An Introduction to Computational Physics*. 2nd. Cambridge: Cambridge University Press. ISBN: 978-0-521-82559-0. DOI: 10.1017/CBO9780511813946.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*. 3rd. Cambridge University Press, p. 912.

Speight, JM (2021). *MATH2017 Real Analysis*. University of Leeds.

Srvirin, Alex (2023). *Nonlinear pendulum*. Accessed: March 12, 2023. URL: https://math24.net/nonlinear-pendulum.html.

Stewart, James (2015). *Calculus: Early Transcendentals*. 8th. Cengage Learning.

Strogatz, Steven H (2014). *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering*. 2nd ed. Westview Press.

Symplectic Integrators (2023). Accessed: March 12, 2023. URL: https://en.wikipedia.org/wiki/Symplectic_integrator#Splitting_methods_for_general_nonseparable_Hamiltonians.

The Improved Euler Method and Related Methods (2020). Accessed: November 12, 2022. Libre Texts.

Virtanen, Pauli, Gommers, Ralf, and Oliphant (2021). *SciPy: Quasi-Monte Carlo integration*. https://docs.scipy.org/doc/scipy/reference/stats.qmc.html. Accessed on: February 28, 2023.

Wikipedia (2021). *Time Reversibility*. Accessed: January 30, 2023. URL: https://en.wikipedia.org/wiki/Time_reversibility.

– (2022). *Pendulum (mechanics)*. https://en.wikipedia.org/wiki/Pendulum_(mechanics). Accessed: March 12, 2023.

Wolfram Alpha (2023). Accessed: February 14, 2023. Wolfram Alpha. URL: https://www.wolframalpha.com/input?i=Riemann+sum.

Appendix A

Appendix Chapter 1

A.1 Python Code: Planetary Orbits

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Nov  3 14:01:56 2022

@author: jai
"""

# Week 2 project code rough

import matplotlib.pyplot as plt
import math
import numpy as np

#The code below alters the example code to have the initial conditions and values
# instructed to use from the comp maths sheet emailed to us.
# Fundamental two body assumptions:
# - All that exists bi the universe is one small body and one large body,
# in this case small is a spacecraft and large is earth



#define parameters
GM = 1 # from the sheet
a = 1 # from the sheet
P= 2*math.pi*(math.sqrt(a**3/GM))
e = 0.5 # the ellipicity from the sheet
tstart=0
tend=100*P
h=P/1000 #as instructed in the sheet
b = a * math.sqrt(1-e**2)
E = np.arange(0,2*math.pi, 0.00001)



#set initial conditions
x0= a*(1+e)
```

UNIVERSITY OF LEEDS

You must sign this (digitally signing with your name is acceptable) and include it with each piece of work you submit.

I am aware that the University defines plagiarism as presenting someone else's work, in whole or in part, as your own. Work means any intellectual output, and typically includes text, data, images, sound or performance.

I promise that in the attached submission I have not presented anyone else's work, in whole or in part, as my own and I have not colluded with others in the preparation of this work. Where I have taken advantage of the work of others, I have given full acknowledgement. I have not resubmitted my own work or part thereof without specific written permission to do so from the University staff concerned when any of this work has been or is being submitted for marks or credits even if in a different module or for a different qualification or completed prior to entry to the University. I have read and understood the University's published rules on plagiarism and also any more detailed rules specified at School or module level. I know that if I commit plagiarism I can be expelled from the University and that it is my responsibility to be aware of the University's regulations on plagiarism and their importance.

I re-confirm my consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the UK) to monitor breaches of regulations, to verify whether my work contains plagiarised material, and for quality assurance purposes.

I confirm that I have declared all mitigating circumstances that may be relevant to the assessment of this piece of work and that I wish to have taken into account. I am aware of the University's policy on mitigation and the School's procedures for the submission of statements and evidence of mitigation. I am aware of the penalties imposed for the late submission of coursework.

Student Signature		Date	23.03.2023
Student Name	Jai Trehan	Student Number	201315963

```
y0=0
vx0=0
vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0

#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]


# # Energy and angular momentum arrays

energy_array= []
momentum_array = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn**2 + vyn**2) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vyn-yn*vxn
    return l
time = []


#main time-stepping loop using 1st order Euler
while t<tend:
 rn=math.sqrt(xn*xn+yn*yn);
 Fxn=-xn/rn**3; Fyn=-yn/rn**3
 xnp1=xn+h*vxn
 ynp1=yn+h*vyn
 vxnp1=vxn+h*Fxn
 vynp1=vyn+h*Fyn
 t=t+h
 xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
 x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

 #Append energy and momentum
 energy_array.append(energy(vxn, vyn, rn))
 momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
 time.append(t)



#plot results
plt.figure(1)
plt.plot(x,y, label = "Forward Euler")
plt.xlabel("x")
plt.ylabel("y")
plt.title('Forward Euler')
#Predicted orbit
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Predicted orbit")
plt.title('Forward Euler')
# Star position
plt.plot(0, 0, 'o' , color ='red', label = "Star Position")
plt.legend(loc = "upper left")
plt.show()
```

```
96
97
98
99  #Energy conservation
100 energy_frac = []
101 for i in energy_array:
102     energy_frac.append(abs((i-energy_array[0])/energy_array[0]))
103
104 mom_frac = []
105 for i in momentum_array:
106     mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
107
108
109
110
111 # plot the log log scale of energy and angular momentum
112 plt.figure(5)
113 plt.loglog(time, energy_frac)
114 plt.title("Forward Euler Energy conservation log-log")
115 plt.show()
116 plt.figure(6)
117 plt.loglog(time, mom_frac)
118 plt.title("Forward Euler momentum conservation log-log")
119 plt.show()
120
121 # calculate difference in distance between numerical solution and
        predicted orbit at the end point
122 xn_end = x[-1]
123 yn_end = y[-1]
124 rn_end = math.sqrt(xn_end*xn_end+yn_end*yn_end)
125
126 xp_end = a*(np.cos(E)+e)
127 yp_end = b*np.sin(E)
128 rp_end = np.sqrt(xp_end*xp_end+yp_end*yp_end)
129
130 difference = math.sqrt((xn_end-xp_end[-1])**2 + (yn_end-yp_end[-1])**2)
131 print("Difference in distance at the end point Forward Euler:",
        difference)
132 #!/usr/bin/env python3
133 # -*- coding: utf-8 -*-
134 """
135 Created on Fri Nov  4 12:38:11 2022
136
137 @author: jai
138 """
139
140 import matplotlib.pyplot as plt
141 import math
142 import numpy as np
143
144 #The code below alters the example code to have the initial conditions
        and values
145 # instructed to use from the comp maths sheet emailed to us.
146 # Fundamental two body assumptions:
147 # - All that exists bi the universe is one small body and one large
        body,
148 # in this case small is a spacecraft and large is earth
149
```

```
150
151
152 #define parameters
153 GM = 1 # from the sheet
154 a = 1 # from the sheet
155 P= 2*math.pi*(math.sqrt(a**3/GM))
156 e = 0.5 # the ellipicity from the sheet
157 tstart=0
158 tend=300*P # not sure what this is
159 h=P/300 #as instructed in the sheet, increasing this to p/100000 gets
       identical to predicted plot
160 b = a * math.sqrt(1-e**2)
161 E = np.arange(0,2*math.pi, 0.001)
162
163
164
165 #set initial conditions
166 x0= a*(1+e); y0=0; vx0=0; vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
167 t=tstart
168 xn=x0; yn=y0; vxn=vx0; vyn=vy0
169
170 #define arrays to store data for plotting/analysis
171 x=[]; y=[]; vx=[]; vy=[]
172
173
174 #ii) Modified Euler
175
176 # Energy and angular momentum arrays
177
178 energy_array= []
179 momentum_array = []
180
181 def energy(vxn, vyn, rn):
182     e = 1/2*(vxn**2 + vyn**2) - GM/rn
183     return e
184
185 def angular_momentum(xn, vxn, yn, vyn):
186     l = xn*vyn-yn*vxn
187     return l
188 time = []
189
190
191 #Creating the loop and plugging in the rn+1 value into the Vn+1
192 while t<tend:
193   time.append(t)
194   rn=math.sqrt(xn**2+yn**2);
195   # Calculate F(r')
196   Fxn=-xn/rn**3
197   Fyn=-yn/rn**3
198
199   # Calculate r'
200   xnp1=xn+h*vxn
201   ynp1=yn+h*vyn
202
203   rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)
204
205  # Calculate F(r')
206   Fxnp1=-xnp1/rnp1**3
```

```
  Fynp1=-ynp1/rnp1**3


  # Calculate the new velocities
  vxnp1=vxn+h*Fxnp1
  vynp1=vyn+h*Fynp1


  t=t+h
  xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
  x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

  #Append energy and momentum
  energy_array.append(energy(vxn, vyn, rn))
  momentum_array.append(angular_momentum(xn, vxn, yn, vyn))



plt.figure(2)
plt.plot(x,y, label = "Modfied Euler")
plt.xlabel("x")
plt.ylabel("y")
#Predicted orbit
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label ="Predicted Orbit")
plt.title('Modified Euler')
# Star position
plt.plot(0, 0, 'o' ,color ='Red', label = "Star Postion")
plt.legend(loc = "upper left")
plt.show()




#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))



# plot the log log scale of energy and angular momentum
plt.figure(5)
plt.loglog(time, energy_frac)
plt.title("Modified Euler Energy conservation log-log")
plt.show()
plt.figure(6)
plt.loglog(time, mom_frac)
plt.title("Modified Euler momentum conservation log-log")
plt.show()
```

```
#Reversing
vxn = -vxn
vyn = -vyn
t=0

#Reversing
while t<tend:

  rn=math.sqrt(xn**2+yn**2);
  # Calculate F(r')
  Fxn=-xn/rn**3
  Fyn=-yn/rn**3

  # Calculate r'
  xnp1=xn+h*vxn
  ynp1=yn+h*vyn

  rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)

 # Calculate F(r')
  Fxnp1=-xnp1/rnp1**3
  Fynp1=-ynp1/rnp1**3


  # Calculate the new velocities
  vxnp1=vxn+h*Fxnp1
  vynp1=vyn+h*Fynp1


  t=t+h
  xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
  x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

plt.plot(x,y)
plt.title("Reversed Modified Euler")
plt.show()

xdiff_modified_euler = abs(x[-1]-x0)
ydiff_modified_euler = abs(y[-1]- y0)
vxdiff_modified_euler = abs(-vx[-1]-vx0)
vydiff_modified_euler = abs(-vy[-1]-vy0)
print(xdiff_modified_euler, ydiff_modified_euler, vxdiff_modified_euler
    , vydiff_modified_euler)


# calculate difference in distance between numerical solution and
    predicted orbit at the end point
xn_end = x[-1]
yn_end = y[-1]
rn_end = math.sqrt(xn_end*xn_end+yn_end*yn_end)

xp_end = a*(np.cos(E)+e)
yp_end = b*np.sin(E)
rp_end = np.sqrt(xp_end*xp_end+yp_end*yp_end)

difference = math.sqrt((xn_end-xp_end[-1])**2 + (yn_end-yp_end[-1])**2)
print("Difference in distance at the end point Modified Euler:",
    difference)
```

```
320
321 # #!/usr/bin/env python3
322 # # -*- coding: utf-8 -*-
323 # """
324 # Created on Fri Nov  4 13:25:36 2022
325
326 # @author: jai
327 # """
328
329 import matplotlib.pyplot as plt
330 import math
331 import numpy as np
332
333 #define parameters
334 P=2*math.pi
335 tstart=0
336 tend=1000*P
337 h=P/500
338
339 e = 0.5
340 E = 0
341 b = math.sqrt((1-e*e))
342 a = 1
343 GM = a
344
345 #set initial conditions
346 x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
347 t=tstart
348 xn=x0; yn=y0; vxn=vx0; vyn=vy0
349
350
351 #define arrays to store data for plotting/analysis
352 x=[]; y=[]; vx=[]; vy=[]
353
354
355
356 #Energy and Angular Momentum arrays
357 energy_array = []; momentum_array = []
358
359 def energy(vxn, vyn, rn):
360     e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
361     return e
362
363 def angular_momentum(xn, vxn, yn, vyn):
364     l = xn*vyn-yn*vxn
365     return l
366 time = []
367
368
369
370 #main time-stepping loop
371 while t<tend:
372
373     rn = math.sqrt(xn*xn+yn*yn);
374     vn = math.sqrt(vyn*vyn+vxn*vxn)
375
376     #Calculate r'
377
```

```
    xnd = xn + h*vxn/2
    ynd = yn + h*vyn/2
    rnd = math.sqrt(xnd*xnd + ynd*ynd)

    #Calculate F(r')
    Fxnd = -xnd/rnd**3
    Fynd = -ynd/rnd**3

    #Calculate new velocities
    vxnp1 = vxn + h*Fxnd
    vynp1 = vyn + h*Fynd

    #Calculate new coordinates
    xnp1 = xnd + h*vxnp1/2
    ynp1 = ynd + h*vynp1/2

    t=t+h
    xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

    #Append energy and momentum
    energy_array.append(energy(vxn, vyn, rn))
    momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
    time.append(t)



#plot results


plt.plot(x,y, '-', linewidth = '5', label = "Leapfrog")
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Expected orbit")
plt.title('Leapfrog')
plt.xlabel("x")
plt.ylabel("y")
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "Star Postion")
plt.legend(loc = "upper left")
plt.show()




#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))


# plot the log log scale of energy and angular momentum
plt.loglog(time,energy_frac)
plt.title("Leapfrog Energy conservation log-log")
plt.show()
```

```
plt.loglog(time, mom_frac)
plt.title("Leapfrog momentum conservation log-log")
plt.show()

# calculate difference in distance between numerical solution and
    predicted orbit at the end point
xn_end = x[-1]
yn_end = y[-1]
rn_end = math.sqrt(xn_end*xn_end+yn_end*yn_end)

xp_end = a*(np.cos(E)+e)
yp_end = b*np.sin(E)
rp_end = np.sqrt(xp_end*xp_end+yp_end*yp_end)

difference = math.sqrt((xn_end-xp_end[-1])**2 + (yn_end-yp_end[-1])**2)
print("Difference in distance at the end point Leapfrog:", difference)
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 23 21:44:49 2022

@author: jai
"""

import matplotlib.pyplot as plt
import math
import numpy as np

#define parameters
P=2*math.pi
tstart=0
tend=100*P
h=P/250

e = 0.9
E = 0
b = math.sqrt((1-e*e))
a = 1
GM = a

#set initial conditions
x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0
r0 = math.sqrt(vx0**2 + vy0**2)
initial_energy = 0.5*(vx0**2 + vy0**2)-1/r0


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]



#Energy and Angular Momentum arrays
energy_array = []; momentum_array = []; time = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
```

```
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vyn-yn*vxn
    return l
time = []


#main time-stepping loop using 4th order RK4
while t<tend:
    rn=math.sqrt(xn*xn+yn*yn)
    wn=np.array([xn,yn, vxn, vyn] )
    wdotn=np.array([vxn,vyn,-xn/rn**3,-yn/rn**3])
    k1=h*wdotn
    wn1=wn+1/2*k1
    rn1=math.sqrt(wn1[0]**2+wn1[1]**2)
    k2= h*np.array ([wn1[2], wn1[3],-wn1[0]/rn1**3, -wn1[1]/rn1**3])
    wn2=wn+1/2*k2
    rn2 = math.sqrt(wn2[0]**2+wn2[1]**2)
    k3= h*np.array([wn2[2] , wn2[3],-wn2[0]/rn2**3, -wn2[1]/rn2**3])
    wn3=wn+k3
    rn3=math.sqrt(wn3[0]**2+wn3[1]**2)
    k4=h*np.array([wn3[2],wn3[3], -wn3[0]/rn3**3,-wn3[1]/rn3**3])
    wnp1=wn+1/6*(k1+2*k2+2*k3+k4)
    t=t+h

    xn =wnp1[0]; yn =wnp1[1]; vxn=wnp1[2]; vyn=wnp1[3]
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn)

    #Append energy and momentum
    energy_array.append(energy(vxn, vyn, rn))
    momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
    time.append(t)

#plot results
plt.plot(x,y, linewidth = '5', label = "Runge-Kutta 4")
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Predicted Orbit")
plt.title('Runge-Kutta 4')
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "Star Postion")
plt.legend(loc = "upper left")
plt.xlabel('x')
plt.ylabel('y')
plt.show()


#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))


# plot the log log scale of energy and angular momentum
```

```
plt.loglog(time, energy_frac)
plt.title("Runge Kutta 4 Energy conservation log-log")
plt.show()
plt.loglog(time, mom_frac)
plt.title("Runge-Kutta 4 momentum conservation log-log")
plt.show()

# # calculate difference in distance between numerical solution and predicted orbit at the end point
# xn_end = x[-1]
# yn_end = y[-1]
# rn_end = math.sqrt(xn_end*xn_end+yn_end*yn_end)

# xp_end = a*(np.cos(E)+e)
# yp_end = b*np.sin(E)
# rp_end = np.sqrt(xp_end*xp_end+yp_end*yp_end)

# difference = math.sqrt((xn_end-xp_end[-1])**2 + (yn_end-yp_end[-1])**2)
# print("Difference in distance at the end point RK4:", difference)
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 21 18:34:01 2023

@author: jai
"""

import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.interpolate import interp1d

#
    ================================================================================

# Qauantifying the errors for each scheme for 10 orbits
#
    ================================================================================


#define parameters
GM = 1 # from the sheet
a = 1 # from the sheet
P= 2*math.pi*(math.sqrt(a**3/GM))
e = 0.5 # the ellipicity from the sheet
tstart=0
tend=10*P
h=P/1000 #as instructed in the sheet
b = a * math.sqrt(1-e**2)
E = np.arange(0,2*math.pi, 0.00001)


#set initial conditions
x0= a*(1+e)
y0=0
vx0=0
vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
```

```
603 t=tstart
604 xn=x0; yn=y0; vxn=vx0; vyn=vy0
605
606 #define arrays to store data for plotting/analysis
607 x=[]; y=[]; vx=[]; vy=[]
608
609 time = []
610
611 #
    ================================================================================

612 # FORWARD EULER
613 #
    ================================================================================

614 #main time-stepping loop using 1st order Euler
615 while t<tend:
616  rn=math.sqrt(xn*xn+yn*yn);
617  Fxn=-xn/rn**3; Fyn=-yn/rn**3
618  xnp1=xn+h*vxn
619  ynp1=yn+h*vyn
620  vxnp1=vxn+h*Fxn
621  vynp1=vyn+h*Fyn
622  t=t+h
623  xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
624  x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
625
626 #plot results
627 plt.figure(1)
628 plt.plot(x,y, label = "Forward Euler")
629 plt.xlabel("x")
630 plt.ylabel("y")
631 plt.title('Forward Euler')
632 #Predicted orbit
633 X = np.arange(0, 2*math.pi, h)
634 plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Predicted orbit")
635 plt.title('Forward Euler')
636 # Star position
637 plt.plot(0, 0, 'o' , color ='red', label = "Star Position")
638 plt.legend(loc = "upper left")
639 plt.show()
640
641 # calculate difference in distance between numerical solution and
       predicted orbit at the end point
642 xn_end = x[-1]
643 yn_end = y[-1]
644 rn_end = math.sqrt(xn_end*xn_end+yn_end*yn_end)
645
646 xp_end = a*(np.cos(E)+e)
647 yp_end = b*np.sin(E)
648 rp_end = np.sqrt(xp_end*xp_end+yp_end*yp_end)
649
650 difference = math.sqrt((xn_end-xp_end[-1])**2 + (yn_end-yp_end[-1])**2)
651 print("Difference in distance at the end point FE:", difference)
652
653 #
    ================================================================================
```

```
654 # Modified forward Euler
655 #
    ================================================================================
656 #initialise
657 x0= a*(1+e)
658 y0=0
659 vx0=0
660 vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
661 t=tstart
662 xn=x0; yn=y0; vxn=vx0; vyn=vy0
663 #define arrays to store data for plotting/analysis
664 x=[]; y=[]; vx=[]; vy=[]
665 time = []
666
667 #Main time stepping loop: modified forward euler
668 while t<tend:
669   time.append(t)
670   rn=math.sqrt(xn**2+yn**2);
671   # Calculate F(r')
672   Fxn=-xn/rn**3
673   Fyn=-yn/rn**3
674
675   # Calculate r'
676   xnp1=xn+h*vxn
677   ynp1=yn+h*vyn
678
679   rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)
680
681  # Calculate F(r')
682   Fxnp1=-xnp1/rnp1**3
683   Fynp1=-ynp1/rnp1**3
684
685
686   # Calculate the new velocities
687   vxnp1=vxn+h*Fxnp1
688   vynp1=vyn+h*Fynp1
689
690   t=t+h
691   xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
692   x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
693
694 plt.plot(x,y, label = "Modfied Euler")
695 plt.xlabel("x")
696 plt.ylabel("y")
697 #Predicted orbit
698 X = np.arange(0, 2*math.pi, h)
699 plt.plot(a*(np.cos(X)+e),b*np.sin(X), label ="Predicted Orbit")
700 plt.title('Modified Euler')
701 plt.show()
702
703 # calculate difference in distance between numerical solution and
      predicted orbit at the end point
704 xn_end = x[-1]
705 yn_end = y[-1]
706 rn_end = math.sqrt(xn_end*xn_end+yn_end*yn_end)
707
708 xp_end = a*(np.cos(E)+e)
```

```
yp_end = b*np.sin(E)
rp_end = np.sqrt(xp_end*xp_end+yp_end*yp_end)

difference = math.sqrt((xn_end-xp_end[-1])**2 + (yn_end-yp_end[-1])**2)
print("Difference in distance at the end point MFE:", difference)

#
    ================================================================================

# Leapfrog
#
    ================================================================================

P = 2 * math.pi
tstart = 0
tend = 1000 * P
h = P / 500

e = 0.5
b = math.sqrt((1 - e * e))
a = 1
GM = 1

# initialise
x0 = a * (1 + e)
y0 = 0
vx0 = 0
vy0 = math.sqrt((GM * (1 - e)) / (a * (1 + e)))
t = tstart
xn = x0
yn = y0
vxn = vx0
vyn = vy0

# define arrays to store data for plotting/analysis
x = []
y = []
vx = []
vy = []

time = []

# main time-stepping loop
while t < tend:

    rn = math.sqrt(xn * xn + yn * yn)
    vn = math.sqrt(vyn * vyn + vxn * vxn)

    # Calculate r'

    xnd = xn + h * vxn / 2
    ynd = yn + h * vyn / 2
    rnd = math.sqrt(xnd * xnd + ynd * ynd)

    # Calculate F(r')
    Fxnd = -GM * xnd / rnd ** 3
    Fynd = -GM * ynd / rnd ** 3

```

```
763     # Calculate new velocities
764     vxnp1 = vxn + h * Fxnd
765     vynp1 = vyn + h * Fynd
766 
767     # Calculate new coordinates
768     xnp1 = xnd + h * vxnp1 / 2
769     ynp1 = ynd + h * vynp1 / 2
770 
771     t = t + h
772     xn = xnp1
773     yn = ynp1
774     vxn = vxnp1
775     vyn = vynp1
776     x.append(xn)
777     y.append(yn)
778     vx.append(vxn)
779     vy.append(vyn)
780 
781 plt.plot(x, y, '-', label="Leapfrog")
782 X = np.arange(0, 2 * math.pi, h)
783 plt.plot(a * (np.cos(X) + e), b * np.sin(X), label="Expected orbit")
784 plt.title('Leapfrog')
785 plt.xlabel("x")
786 plt.ylabel("y")
787 
788 # calculate difference in distance between numerical solution and
        predicted orbit at the end point
789 xn_end = x[-1]
790 yn_end = y[-1]
791 rp_end = a * (np.cos(X[-1]) + e)
792 difference = math.sqrt((xn_end - rp_end) ** 2 + yn_end ** 2)
793 print("Difference in distance at the end point Leapfrog:", difference)
794 
795 plt.legend()
796 plt.show()
797 #
        ================================================================================

798 # Runge Kutta
799 #
        ================================================================================

800 P=2*math.pi
801 tstart=0
802 tend=100*P
803 h=P/250
804 e = 0.5
805 b = math.sqrt((1-e*e))
806 a = 1
807 GM = a
808 #initialise
809 x0= a*(1+e)
810 y0=0
811 vx0=0
812 vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
813 t=tstart
814 xn=x0; yn=y0; vxn=vx0; vyn=vy0
815 
```

```
#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]

time = []
#main time-stepping loop using 4th order RK4
while t<tend:
    rn=math.sqrt(xn*xn+yn*yn)
    wn=np.array([xn,yn, vxn, vyn] )
    wdotn=np.array([vxn,vyn,-xn/rn**3,-yn/rn**3])
    k1=h*wdotn
    wn1=wn+1/2*k1
    rn1=math.sqrt(wn1[0]**2+wn1[1]**2)
    k2= h*np.array ([wn1[2], wn1[3],-wn1[0]/rn1**3, -wn1[1]/rn1**3])
    wn2=wn+1/2*k2
    rn2 = math.sqrt(wn2[0]**2+wn2[1]**2)
    k3= h*np.array([wn2[2] , wn2[3],-wn2[0]/rn2**3, -wn2[1]/rn2**3])
    wn3=wn+k3
    rn3=math.sqrt(wn3[0]**2+wn3[1]**2)
    k4=h*np.array([wn3[2],wn3[3], -wn3[0]/rn3**3,-wn3[1]/rn3**3])
    wnp1=wn+1/6*(k1+2*k2+2*k3+k4)
    t=t+h

    xn =wnp1[0]; yn =wnp1[1]; vxn=wnp1[2]; vyn=wnp1[3]
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn)

#plot results
plt.plot(x,y, linewidth = '1', label = "Runge-Kutta 4")
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Predicted Orbit")
plt.title('Runge-Kutta 4')
# calculate difference in distance between numerical solution and
   predicted orbit at the end point
xn_end = x[-1]
yn_end = y[-1]
rn_end = math.sqrt(xn_end*xn_end+yn_end*yn_end)

xp_end = a*(np.cos(E)+e)
yp_end = b*np.sin(E)
rp_end = np.sqrt(xp_end*xp_end+yp_end*yp_end)

difference = math.sqrt((xn_end-xp_end[-1])**2 + (yn_end-yp_end[-1])**2)
print("Difference in distance at the end point RK4:", difference)

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 25 13:53:04 2022

@author: jai
"""

import matplotlib.pyplot as plt
import math
import numpy as np

#Edited forward euler for part B conditions:
# e = 0.5, 300 force evaluations (h/300) and N = 100

```

```
873
874 #define parameters
875 GM = 1 # from the sheet
876 a = 1 # from the sheet
877 P= 2*math.pi*(math.sqrt(a**3/GM))
878 e = 0.5 # the ellipicity from the sheet
879 tstart=0
880 tend=100*P # not sure what this is
881 h=P/300 #as instructed in the sheet
882 b = a * math.sqrt(1-e**2)
883 E = np.arange(0,2*math.pi, 0.00001)
884
885
886
887 #set initial conditions
888 x0= a*(1+e)
889 y0=0
890 vx0=0
891 vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
892 t=tstart
893 xn=x0; yn=y0; vxn=vx0; vyn=vy0
894
895 #define arrays to store data for plotting/analysis
896 x=[]; y=[]; vx=[]; vy=[]
897
898
899 # # Energy and angular momentum arrays
900
901 energy_array= []
902 momentum_array = []
903
904 def energy(vxn, vyn, rn):
905     e = 1/2*(vxn**2 + vyn**2) - GM/rn
906     return e
907
908 def angular_momentum(xn, vxn, yn, vyn):
909     l = xn*vyn - yn*vxn
910     return l
911 time = []
912
913
914 #main time-stepping loop using 1st order Euler
915 while t<tend:
916  time.append(t)
917  rn=math.sqrt(xn*xn+yn*yn);
918  Fxn=-xn/rn**3; Fyn=-yn/rn**3
919  xnp1=xn+h*vxn
920  ynp1=yn+h*vyn
921  vxnp1=vxn+h*Fxn
922  vynp1=vyn+h*Fyn
923  t=t+h
924  xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
925  x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
926
927  #Append energy and momentum
928  energy_array.append(energy(vxn, vyn, rn))
929  momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
930
```

```
931
932
933 #plot results
934 plt.plot(x,y, label = "Forward Euler")
935 plt.title('Forward Euler')
936 plt.ylabel('y')
937 plt.xlabel('x')
938
939
940 #Predicted orbit
941 X = np.arange(0, 2*math.pi, h)
942 plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Predicted Orbit")
943 plt.title('Forward Euler')
944 # Star position
945 plt.plot(0, 0, 'o' , color ='red', label = "Star Position")
946 plt.legend(loc = "lower left")
947 plt.show()
948
949
950
951
952
953 #Energy conservation
954 energy_frac = []
955 for i in energy_array:
956     energy_frac.append(abs((i-energy_array[0])/energy_array[0]))
957
958 mom_frac = []
959 for i in momentum_array:
960     mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
961
962
963
964 # plot the log log scale of energy and angular momentum
965 plt.loglog(time, energy_frac)
966 plt.title("Forward Euler Energy conservation log-log (300 force
    evaluations per orbit)")
967 plt.show()
968 plt.loglog(time, mom_frac)
969 plt.title("Forward Euler momentum conservation log-log(300 force
    evaluations per orbit)")
970 plt.show()
971 #!/usr/bin/env python3
972 # -*- coding: utf-8 -*-
973 """
974 Created on Fri Nov 25 14:02:26 2022
975
976 @author: jai
977 """
978
979 import matplotlib.pyplot as plt
980 import math
981 import numpy as np
982
983
984 # Modified forward euler with part B conditions
985 # e = 0.5, 300 force evaluations (h/300) and N = 100
986
```

```
987
988 #define parameters
989 GM = 1 # from the sheet
990 a = 1 # from the sheet
991 P= 2*math.pi*(math.sqrt(a**3/GM))
992 e = 0.5 # the ellipicity from the sheet
993 tstart=0
994 tend=100*P # not sure what this is
995 h=P/300 #as instructed in the sheet, increasing this to p/100000 gets
        identical to predicted plot
996 b = a * math.sqrt(1-e**2)
997 E = np.arange(0,2*math.pi, 0.00001)
998
999
1000
1001 #set initial conditions
1002 x0= a*(1+e); y0=0; vx0=0; vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
1003 t=tstart
1004 xn=x0; yn=y0; vxn=vx0; vyn=vy0
1005
1006 #define arrays to store data for plotting/analysis
1007 x=[]; y=[]; vx=[]; vy=[]
1008
1009
1010 #ii) Modified Euler
1011
1012 # Energy and angular momentum arrays
1013
1014 energy_array= []
1015 momentum_array = []
1016
1017 def energy(vxn, vyn, rn):
1018     e = 1/2*(vxn**2 + vyn**2) - GM/rn
1019     return e
1020
1021 def angular_momentum(xn, vxn, yn, vyn):
1022     l = xn*vyn - yn*vxn
1023     return l
1024 time = []
1025
1026
1027 #Creating the loop and plugging in the rn+1 value into the Vn+1
1028
1029 while t<tend:
1030   time.append(t)
1031   rn=math.sqrt(xn**2+yn**2);
1032   # Calculate F(r')
1033   Fxn=-xn/rn**3
1034   Fyn=-yn/rn**3
1035
1036   # Calculate r'
1037   xnp1=xn+h*vxn
1038   ynp1=yn+h*vyn
1039
1040   rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)
1041
1042  # Calculate F(r')
1043   Fxnp1=-xnp1/rnp1**3
```

```
  Fynp1=-ynp1/rnp1**3


  # Calculate the new velocities
  vxnp1=vxn+h*Fxnp1
  vynp1=vyn+h*Fynp1


  t=t+h
  xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
  x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

  #Append energy and momentum
  energy_array.append(energy(vxn, vyn, rn))
  momentum_array.append(angular_momentum(xn, vxn, yn, vyn))


plt.plot(x,y, label = "Modfied Euler")
plt.ylabel('y')
plt.xlabel('x')

#Predicted orbit
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label ="Predicted Orbit")
plt.title('Modified Euler')
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "Star Postion")
plt.legend(loc = "upper left")
plt.show()




#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))



# plot the log log scale of energy and angular momentum
plt.loglog(time, energy_frac)
plt.title("Modified Euler Energy conservation log-log")
plt.show()
plt.loglog(time, mom_frac)
plt.title("Modified Euler momentum conservation log-log")
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 25 14:09:17 2022

```

```
@author: jai
"""
#Leapfrog method edited with Part B conditions:
# e = 0.5, 300 force evaluations (h/300) and N = 100
import matplotlib.pyplot as plt
import math
import numpy as np

#define parameters
P=2*math.pi
tstart=0
tend=100*P
h=P/150

e = 0.5
E = 0
b = math.sqrt((1-e*e))
a = 1
GM = a

#set initial conditions
x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]



#Energy and Angular Momentum arrays
energy_array = []; momentum_array = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vyn-yn*vxn
    return l
time = []



#main time-stepping loop
while t<tend:

    rn = math.sqrt(xn*xn+yn*yn);
    vn = math.sqrt(vyn*vyn+vxn*vxn)

    #Calculate r'

    xnd = xn + h*vxn/2
    ynd = yn + h*vyn/2
    rnd = math.sqrt(xnd*xnd + ynd*ynd)

    #Calculate F(r')
```

```
    Fxnd = -xnd/rnd**3
    Fynd = -ynd/rnd**3

    #Calculate new velocities
    vxnp1 = vxn + h*Fxnd
    vynp1 = vyn + h*Fynd

    #Calculate new coordinates
    xnp1 = xnd + h*vxnp1/2
    ynp1 = ynd + h*vynp1/2

    t=t+h
    xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

    #Append energy and momentum
    energy_array.append(energy(vxn, vyn, rn))
    momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
    time.append(t)



#plot results


plt.plot(x,y, label = "Leapfrog")
plt.ylabel('y')
plt.xlabel('x')
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Expected orbit")
plt.title('Leapfrog')
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "Star Postion")
plt.legend(loc = "upper left")
plt.show()



#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))



# plot the log log scale of energy and angular momentum
plt.loglog(time, energy_frac)
plt.title("Leapfrog Energy conservation log-log")
plt.show()
plt.loglog(time, mom_frac)
plt.title("Leapfrog momentum conservation log-log")
plt.show()
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 25 14:10:17 2022

@author: jai
"""

import matplotlib.pyplot as plt
import math
import numpy as np

#RK4 PART B e = 0.5, N = 100, 300 force evaluations

#define parameters
P=2*math.pi
tstart=0
tend=100*P
h=P/(75)

e = 0.5
E = 0
b = math.sqrt((1-e*e))
a = 1
GM = a

#set initial conditions
x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]



#Energy and Angular Momentum arrays
energy_array = []; momentum_array = []; time = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vyn-yn*vxn
    return l




#main time-stepping loop
#main time-stepping loop using 4th order RK4
while t<tend:
    time.append(t)
    rn=math.sqrt(xn*xn+yn*yn)
    wn=np.array([xn,yn, vxn, vyn] )
    wdotn=np.array([vxn,vyn,-xn/rn**3,-yn/rn**3])
```

```
    k1=h*wdotn
    wn1=wn+1/2*k1
    rn1=math.sqrt(wn1[0]**2+wn1[1]**2)
    k2= h*np.array ([wn1[2], wn1[3],-wn1[0]/rn1**3, -wn1[1]/rn1**3])
    wn2=wn+1/2*k2
    rn2 = math.sqrt(wn2[0]**2+wn2[1]**2)
    k3= h*np.array([wn2[2] , wn2[3],-wn2[0]/rn2**3, -wn2[1]/rn2**3])
    wn3=wn+k3
    rn3=math.sqrt(wn3[0]**2+wn3[1]**2)
    k4=h*np.array([wn3[2],wn3[3], -wn3[0]/rn3**3,-wn3[1]/rn3**3])
    wnp1=wn+1/6*(k1+2*k2+2*k3+k4)
    t=t+h

    xn =wnp1[0]; yn =wnp1[1]; vxn=wnp1[2]; vyn=wnp1[3]
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn)

    #Append energy and momentum
    energy_array.append(energy(vxn, vyn, rn))
    momentum_array.append(angular_momentum(xn, vxn, yn, vyn))



#plot results

plt.plot(x,y, linewidth = '3', label = "Runge-Kutta 4")
plt.ylabel('y')
plt.xlabel('x')
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Predicted orbit")
plt.title('Runge-Kutta 4')
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "Star Postion")
plt.legend(loc = "upper left")
plt.show()




#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))



# plot the log log scale of energy and angular momentum
plt.loglog(time, energy_frac)
plt.title("Part B: Runge Kutta 4 Energy conservation log-log")
plt.show()
plt.loglog(time, mom_frac)
plt.title("Part B: Runge-Kutta 4 Momentum conservation log-log")
plt.show()
```

```
1334
1335 #!/usr/bin/env python3
1336 # -*- coding: utf-8 -*-
1337 """
1338 Created on Tue Dec 13 15:55:36 2022
1339
1340 @author: jai
1341 """
1342 import matplotlib.pyplot as plt
1343 import math
1344 import numpy as np
1345 #Compiled energy plots part b e =0.5
1346
1347
1348 #forward euler
1349
1350
1351 #define parameters
1352 GM = 1 # from the sheet
1353 a = 1 # from the sheet
1354 P= 2*math.pi*(math.sqrt(a**3/GM))
1355 e = 0.5 # the ellipicity from the sheet
1356 tstart=0
1357 tend=100*P # not sure what this is
1358 h=P/300 #as instructed in the sheet
1359 b = a * math.sqrt(1-e**2)
1360 E = np.arange(0,2*math.pi, 0.00001)
1361
1362
1363
1364 #set initial conditions
1365 x0= a*(1+e)
1366 y0=0
1367 vx0=0
1368 vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
1369 r0 = math.sqrt(vx0**2 + vy0**2)
1370 t=tstart
1371 xn=x0; yn=y0; vxn=vx0; vyn=vy0
1372 initial_energy = 0.5*(vx0**2 + vy0**2)-1/r0
1373
1374
1375 #define arrays to store data for plotting/analysis
1376 x=[]; y=[]; vx=[]; vy=[]
1377
1378
1379 # Energy and angular momentum arrays
1380
1381 energy_array= []
1382 momentum_array = []
1383
1384 def energy(vxn, vyn, rn):
1385     e = 1/2*(vxn**2 + vyn**2) - GM/rn
1386     return e
1387
1388 def angular_momentum(xn, vxn, yn, vyn):
1389     l = xn*vxn - yn*vyn
1390     return l
1391 time = []
```

```
1392
1393
1394 #main time-stepping loop using 1st order Euler
1395 while t<tend:
1396  time.append(t)
1397  rn=math.sqrt(xn*xn+yn*yn);
1398  Fxn=-xn/rn**3; Fyn=-yn/rn**3
1399  xnp1=xn+h*vxn
1400  ynp1=yn+h*vyn
1401  vxnp1=vxn+h*Fxn
1402  vynp1=vyn+h*Fyn
1403  t=t+h
1404  xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
1405  x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
1406
1407  #Append energy and momentum
1408  rn=math.sqrt(xn*xn+yn*yn)
1409  energy_array.append(energy(vxn, vyn, rn))
1410  momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
1411
1412
1413
1414 #Energy conservation
1415 energy_frac = []
1416 for i in energy_array:
1417     energy_frac.append(abs((i-energy_array[0])/energy_array[0]))
1418
1419 mom_frac = []
1420 for i in momentum_array:
1421     mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
1422
1423 plt.loglog(time, energy_frac, label = "Forward Euler")
1424
1425 # modified forward euler
1426
1427 #define parameters
1428 GM = 1 # from the sheet
1429 a = 1 # from the sheet
1430 P= 2*math.pi*(math.sqrt(a**3/GM))
1431 e = 0.5 # the ellipicity from the sheet
1432 tstart=0
1433 tend=100*P # not sure what this is
1434 h=P/300 #as instructed in the sheet, increasing this to p/100000 gets
      identical to predicted plot
1435 b = a * math.sqrt(1-e**2)
1436 E = np.arange(0,2*math.pi, 0.00001)
1437
1438
1439
1440 #set initial conditions
1441 x0= a*(1+e); y0=0; vx0=0; vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
1442 t=tstart
1443 xn=x0; yn=y0; vxn=vx0; vyn=vy0
1444
1445 #define arrays to store data for plotting/analysis
1446 x=[]; y=[]; vx=[]; vy=[]
1447
1448
```

```
#ii) Modified Euler

# Energy and angular momentum arrays

energy_array= []
momentum_array = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn**2 + vyn**2) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vxn - yn*vyn
    return l
time = []


#Creating the loop and plugging in the rn+1 value into the Vn+1

while t<tend:
    time.append(t)

    rn=math.sqrt(xn**2+yn**2);
    # Calculate F(r')
    Fxn=-xn/rn**3
    Fyn=-yn/rn**3

    # Calculate r'
    xnp1=xn+h*vxn
    ynp1=yn+h*vyn

    rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)

   # Calculate F(r')
    Fxnp1=-xnp1/rnp1**3
    Fynp1=-ynp1/rnp1**3


    # Calculate the new velocities
    vxnp1=vxn+h*Fxnp1
    vynp1=vyn+h*Fynp1


    t=t+h
    xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

    #Append energy and momentum
    rn=math.sqrt(xn*xn+yn*yn)
    energy_array.append(energy(vxn, vyn, rn))
    momentum_array.append(angular_momentum(xn, vxn, yn, vyn))


#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

```

```
mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))



# plot the log log scale of energy and angular momentum
plt.loglog(time, energy_frac, label = "Modified Forward Euler")

# leapfrog

#define parameters
P=2*math.pi
tstart=0
tend=100*P
h=P/150

e = 0.5
E = 0
b = math.sqrt((1-e*e))
a = 1
GM = a

#set initial conditions
x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]



#Energy and Angular Momentum arrays
energy_array = []; momentum_array = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vxn-yn*vyn
    return l
time = []



#main time-stepping loop
while t<tend:
    time.append(t)


    rn = math.sqrt(xn*xn+yn*yn);
    vn = math.sqrt(vyn*vyn+vxn*vxn)

    #Calculate r'
```

```
1565
1566        xnd = xn + h*vxn/2
1567        ynd = yn + h*vyn/2
1568        rnd = math.sqrt(xnd*xnd + ynd*ynd)
1569
1570        #Calculate F(r')
1571        Fxnd = -xnd/rnd**3
1572        Fynd = -ynd/rnd**3
1573
1574        #Calculate new velocities
1575        vxnp1 = vxn + h*Fxnd
1576        vynp1 = vyn + h*Fynd
1577
1578        #Calculate new coordinates
1579        xnp1 = xnd + h*vxnp1/2
1580        ynp1 = ynd + h*vynp1/2
1581
1582        t=t+h
1583        xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
1584        x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
1585
1586
1587        #Append energy and momentum
1588        rn=math.sqrt(xn*xn+yn*yn)
1589        energy_array.append(energy(vxn, vyn, rn))
1590        momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
1591
1592
1593    #Energy conservation
1594    energy_frac = []
1595    for i in energy_array:
1596        energy_frac.append(abs((i-energy_array[0])/energy_array[0]))
1597
1598    mom_frac = []
1599    for i in momentum_array:
1600        mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
1601
1602
1603    # plot the log log scale of energy and angular momentum
1604    plt.loglog(time, energy_frac, label = "Leapfrog")
1605
1606    # runge kutta
1607    #define parameters
1608    P=2*math.pi
1609    tstart=0
1610    tend=100*P
1611    h=P/(75)
1612
1613    e = 0.5
1614    E = 0
1615    b = math.sqrt((1-e*e))
1616    a = 1
1617    GM = a
1618
1619    #set initial conditions
1620    x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
1621    t=tstart
1622    xn=x0; yn=y0; vxn=vx0; vyn=vy0
```

```
1623
1624
1625 #define arrays to store data for plotting/analysis
1626 x=[]; y=[]; vx=[]; vy=[]
1627
1628
1629
1630 #Energy and Angular Momentum arrays
1631 energy_array = []; momentum_array = []; time = []
1632
1633 def energy(vxn, vyn, rn):
1634     e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
1635     return e
1636
1637 def angular_momentum(xn, vxn, yn, vyn):
1638     l = xn*vxn-yn*vyn
1639     return l
1640
1641 #main time-stepping loop using 4th order RK4
1642 while t<tend:
1643     time.append(t)
1644     rn=math.sqrt(xn*xn+yn*yn)
1645     wn=np.array([xn,yn, vxn, vyn] )
1646     wdotn=np.array([vxn,vyn,-xn/rn**3,-yn/rn**3])
1647     k1=h*wdotn
1648     wn1=wn+1/2*k1
1649     rn1=math.sqrt(wn1[0]**2+wn1[1]**2)
1650     k2= h*np.array ([wn1[2], wn1[3],-wn1[0]/rn1**3, -wn1[1]/rn1**3])
1651     wn2=wn+1/2*k2
1652     rn2 = math.sqrt(wn2[0]**2+wn2[1]**2)
1653     k3= h*np.array([wn2[2] , wn2[3],-wn2[0]/rn2**3, -wn2[1]/rn2**3])
1654     wn3=wn+k3
1655     rn3=math.sqrt(wn3[0]**2+wn3[1]**2)
1656     k4=h*np.array([wn3[2],wn3[3], -wn3[0]/rn3**3,-wn3[1]/rn3**3])
1657     wnp1=wn+1/6*(k1+2*k2+2*k3+k4)
1658     t=t+h
1659
1660     xn =wnp1[0]; yn =wnp1[1]; vxn=wnp1[2]; vyn=wnp1[3]
1661     x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn)
1662
1663     #Append energy and momentum
1664     rn=math.sqrt(xn*xn+yn*yn)
1665     energy_array.append(energy(vxn, vyn, rn))
1666     momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
1667
1668 #Energy conservation
1669 energy_frac = []
1670 for i in energy_array:
1671     energy_frac.append(abs((i-energy_array[0])/energy_array[0]))
1672
1673 mom_frac = []
1674 for i in momentum_array:
1675     mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
1676 # plot the log log scale of energy and angular momentum
1677 plt.loglog(time, energy_frac, label = "RK4")
1678 plt.legend(loc="upper left")
1679 plt.xlabel("Time")
1680 plt.ylabel("Energy Fractional Error")
```

```
plt.title("Energy Conservation, e = 0.5, 300 force evalutations")
plt.show()
print(time)
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Dec 13 15:55:36 2022

@author: jai
"""
import matplotlib.pyplot as plt
import math
import numpy as np
#forward euler


#define parameters
GM = 1 # from the sheet
a = 1 # from the sheet
P= 2*math.pi*(math.sqrt(a**3/GM))
e = 0.5 # the ellipicity from the sheet
tstart=0
tend=100*P # not sure what this is
h=P/300 #as instructed in the sheet
b = a * math.sqrt(1-e**2)
E = np.arange(0,2*math.pi, 0.00001)



#set initial conditions
x0= a*(1+e)
y0=0
vx0=0
vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0

#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]


# Energy and angular momentum arrays

energy_array= []
momentum_array = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn**2 + vyn**2) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vyn - yn*vxn
    return l
time = []


#main time-stepping loop using 1st order Euler
while t<tend:
```

```
 time.append(t)
 rn=math.sqrt(xn*xn+yn*yn);
 Fxn=-xn/rn**3; Fyn=-yn/rn**3
 xnp1=xn+h*vxn
 ynp1=yn+h*vyn
 vxnp1=vxn+h*Fxn
 vynp1=vyn+h*Fyn
 t=t+h
 xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
 x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

 #Append energy and momentum
 rn=math.sqrt(xn*xn+yn*yn)
 energy_array.append(energy(vxn, vyn, rn))
 momentum_array.append(angular_momentum(xn, vxn, yn, vyn))


#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))

plt.loglog(time, mom_frac, label = "Forward Euler")

# modified forward euler

#define parameters
GM = 1 # from the sheet
a = 1 # from the sheet
P= 2*math.pi*(math.sqrt(a**3/GM))
e = 0.5 # the ellipicity from the sheet
tstart=0
tend=100*P # not sure what this is
h=P/300 #as instructed in the sheet, increasing this to p/100000 gets
    identical to predicted plot
b = a * math.sqrt(1-e**2)
E = np.arange(0,2*math.pi, 0.00001)



#set initial conditions
x0= a*(1+e); y0=0; vx0=0; vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0

#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]


#ii) Modified Euler

# Energy and angular momentum arrays

energy_array= []
```

```
momentum_array = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn**2 + vyn**2) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vyn - yn*vxn
    return l
time = []


#Creating the loop and plugging in the rn+1 value into the Vn+1

while t<tend:
    time.append(t)
    rn=math.sqrt(xn**2+yn**2);
    # Calculate F(r')
    Fxn=-xn/rn**3
    Fyn=-yn/rn**3

    # Calculate r'
    xnp1=xn+h*vxn
    ynp1=yn+h*vyn

    rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)

   # Calculate F(r')
    Fxnp1=-xnp1/rnp1**3
    Fynp1=-ynp1/rnp1**3


    # Calculate the new velocities
    vxnp1=vxn+h*Fxnp1
    vynp1=vyn+h*Fynp1


    t=t+h
    xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

    #Append energy and momentum
    rn=math.sqrt(xn*xn+yn*yn)
    energy_array.append(energy(vxn, vyn, rn))
    momentum_array.append(angular_momentum(xn, vxn, yn, vyn))


#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
```

```
1854
1855 # plot the log log scale of energy and angular momentum
1856 plt.loglog(time, mom_frac, label = "Modified Forward Euler")
1857
1858 # leapfrog
1859
1860 #define parameters
1861 P=2*math.pi
1862 tstart=0
1863 tend=100*P
1864 h=P/150
1865
1866 e = 0.5
1867 E = 0
1868 b = math.sqrt((1-e*e))
1869 a = 1
1870 GM = a
1871
1872 #set initial conditions
1873 x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
1874 t=tstart
1875 xn=x0; yn=y0; vxn=vx0; vyn=vy0
1876
1877
1878 #define arrays to store data for plotting/analysis
1879 x=[]; y=[]; vx=[]; vy=[]
1880
1881
1882
1883 #Energy and Angular Momentum arrays
1884 energy_array = []; momentum_array = []
1885
1886 def energy(vxn, vyn, rn):
1887     e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
1888     return e
1889
1890 def angular_momentum(xn, vxn, yn, vyn):
1891     l = xn*vyn-yn*vxn
1892     return l
1893 time = []
1894
1895
1896
1897 #main time-stepping loop
1898 while t<tend:
1899     time.append(t)
1900     rn = math.sqrt(xn*xn+yn*yn);
1901     vn = math.sqrt(vyn*vyn+vxn*vxn)
1902
1903     #Calculate r'
1904
1905     xnd = xn + h*vxn/2
1906     ynd = yn + h*vyn/2
1907     rnd = math.sqrt(xnd*xnd + ynd*ynd)
1908
1909     #Calculate F(r')
1910     Fxnd = -xnd/rnd**3
1911     Fynd = -ynd/rnd**3
```

```
1912
1913        #Calculate new velocities
1914        vxnp1 = vxn + h*Fxnd
1915        vynp1 = vyn + h*Fynd
1916
1917        #Calculate new coordinates
1918        xnp1 = xnd + h*vxnp1/2
1919        ynp1 = ynd + h*vynp1/2
1920
1921        t=t+h
1922        xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
1923        x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
1924
1925        #Append energy and momentum
1926        rn=math.sqrt(xn*xn+yn*yn)
1927        energy_array.append(energy(vxn, vyn, rn))
1928        momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
1929
1930    #Energy conservation
1931    energy_frac = []
1932    for i in energy_array:
1933        energy_frac.append(abs((i-energy_array[0])/energy_array[0]))
1934
1935    mom_frac = []
1936    for i in momentum_array:
1937        mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
1938
1939
1940
1941    # plot the log log scale of energy and angular momentum
1942    plt.loglog(time, mom_frac, label = "Leapfrog")
1943
1944    # runge kutta
1945    #define parameters
1946    P=2*math.pi
1947    tstart=0
1948    tend=100*P
1949    h=P/(75)
1950
1951    e = 0.5
1952    E = 0
1953    b = math.sqrt((1-e*e))
1954    a = 1
1955    GM = a
1956
1957    #set initial conditions
1958    x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
1959    t=tstart
1960    xn=x0; yn=y0; vxn=vx0; vyn=vy0
1961
1962
1963    #define arrays to store data for plotting/analysis
1964    x=[]; y=[]; vx=[]; vy=[]
1965
1966
1967
1968    #Energy and Angular Momentum arrays
1969    energy_array = []; momentum_array = []; time = []
```

```
1970
1971 def energy(vxn, vyn, rn):
1972     e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
1973     return e
1974
1975 def angular_momentum(xn, vxn, yn, vyn):
1976     l = xn*vyn-yn*vxn
1977     return l
1978
1979 #main time-stepping loop using 4th order RK4
1980 while t<tend:
1981     time.append(t)
1982     rn=math.sqrt(xn*xn+yn*yn)
1983     wn=np.array([xn,yn, vxn, vyn] )
1984     wdotn=np.array([vxn,vyn,-xn/rn**3,-yn/rn**3])
1985     k1=h*wdotn
1986     wn1=wn+1/2*k1
1987     rn1=math.sqrt(wn1[0]**2+wn1[1]**2)
1988     k2= h*np.array ([wn1[2], wn1[3],-wn1[0]/rn1**3, -wn1[1]/rn1**3])
1989     wn2=wn+1/2*k2
1990     rn2 = math.sqrt(wn2[0]**2+wn2[1]**2)
1991     k3= h*np.array([wn2[2] , wn2[3],-wn2[0]/rn2**3, -wn2[1]/rn2**3])
1992     wn3=wn+k3
1993     rn3=math.sqrt(wn3[0]**2+wn3[1]**2)
1994     k4=h*np.array([wn3[2],wn3[3], -wn3[0]/rn3**3,-wn3[1]/rn3**3])
1995     wnp1=wn+1/6*(k1+2*k2+2*k3+k4)
1996     t=t+h
1997
1998     xn =wnp1[0]; yn =wnp1[1]; vxn=wnp1[2]; vyn=wnp1[3]
1999     x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn)
2000
2001     #Append energy and momentum
2002     rn=math.sqrt(xn*xn+yn*yn)
2003     energy_array.append(energy(vxn, vyn, rn))
2004     momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
2005
2006
2007
2008 #Energy conservation
2009 energy_frac = []
2010 for i in energy_array:
2011     energy_frac.append(abs((i-energy_array[0])/energy_array[0]))
2012
2013 mom_frac = []
2014 for i in momentum_array:
2015     mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
2016
2017
2018
2019 # plot the log log scale of energy and angular momentum
2020 plt.loglog(time, mom_frac, label = "RK4")
2021 plt.legend(loc="upper left")
2022 plt.xlabel("Time")
2023 plt.ylabel("Momentum Fractional Error")
2024 plt.title("Momentum Conservation, e =0.5, 300 force evaluations")
2025 plt.show()
2026 #!/usr/bin/env python3
2027 # -*- coding: utf-8 -*-
```

```
"""
Created on Fri Nov 25 14:11:51 2022

@author: jai
"""

# Code to plot each of the integration schemes, angular momentum and energy conservation with different parameters:
# Now we  consider e = 0.9, N = 100 and 1000 force evalutations per orbit.
# FOWARD EULER

import matplotlib.pyplot as plt
import math
import numpy as np

#define parameters
GM = 1 # from the sheet
a = 1 # from the sheet
P= 2*math.pi*(math.sqrt(a**3/GM))
e = 0.9 # the ellipicity from the sheet
tstart=0
tend=100*P # not sure what this is
h=P/1000 #as instructed in the sheet
b = a * math.sqrt(1-e**2)
E = np.arange(0,2*math.pi, 0.00001)

#set initial conditions
x0= a*(1+e)
y0=0
vx0=0
vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0

#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]

#Energy and angular momentum arrays
energy_array= []
momentum_array = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn**2 + vyn**2) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vxn - yn*vyn
    return l
time = []


#main time-stepping loop using 1st order Euler
while t<tend:
 rn=math.sqrt(xn*xn+yn*yn);
 Fxn=-xn/rn**3; Fyn=-yn/rn**3
 xnp1=xn+h*vxn
 ynp1=yn+h*vyn
```

```
 vxnp1=vxn+h*Fxn
 vynp1=vyn+h*Fyn
 t=t+h
 xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
 x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

 #Append energy and momentum
 energy_array.append(energy(vxn, vyn, rn))
 momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
 time.append(t)

#plot results
plt.plot(x,y, label = "Forward Euler")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Forward Euler')

#Predicted orbit
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Predicted Orbit")
plt.title('Forward Euler')
# Star position
plt.plot(0, 0, 'o' , color ='red', label = "Star Position")
plt.legend(loc = "upper right")
plt.show()

#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))

# plot the log log scale of energy and angular momentum
plt.loglog(time, energy_frac)
plt.title("Forward Euler Energy conservation log-log")
plt.show()
plt.loglog(time, mom_frac)
plt.title("Forward Euler momentum conservation log-log")
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 25 14:31:30 2022

@author: jai
"""
import matplotlib.pyplot as plt
import math
import numpy as np

# Code to plot each of the integration schemes, angular momentum and
    energy conservation with different parameters:
# Now we  consider e = 0.9, N = 100 and 1000 force evalutations per
    orbit.
# Modified Euler
```

```
2140
2141 #define parameters
2142 GM = 1 # from the sheet
2143 a = 1 # from the sheet
2144 P= 2*math.pi*(math.sqrt(a**3/GM))
2145 e = 0.9 # the ellipicity from the sheet
2146 tstart=0
2147 tend=100*P
2148 h=P/1000 #as instructed in the sheet, increasing this to p/100000 gets
        identical to predicted plot
2149 b = a * math.sqrt(1-e**2)
2150 E = np.arange(0,2*math.pi, 0.00001)
2151
2152 #set initial conditions
2153 x0= a*(1+e); y0=0; vx0=0; vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
2154 t=tstart
2155 xn=x0; yn=y0; vxn=vx0; vyn=vy0
2156
2157 #define arrays to store data for plotting/analysis
2158 x=[]; y=[]; vx=[]; vy=[]
2159
2160 # Energy and angular momentum arrays
2161 energy_array= []
2162 momentum_array = []
2163
2164 def energy(vxn, vyn, rn):
2165     e = 1/2*(vxn**2 + vyn**2) - GM/rn
2166     return e
2167
2168 def angular_momentum(xn, vxn, yn, vyn):
2169     l = xn*vxn - yn*vyn
2170     return l
2171 time = []
2172
2173 #Creating the loop and plugging in the rn+1 value into the Vn+1
2174 while t<tend:
2175
2176   while t<tend:
2177     time.append(t)
2178     rn=math.sqrt(xn**2+yn**2);
2179     # Calculate F(r')
2180     Fxn=-xn/rn**3
2181     Fyn=-yn/rn**3
2182
2183     # Calculate r'
2184     xnp1=xn+h*vxn
2185     ynp1=yn+h*vyn
2186
2187     rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)
2188
2189    # Calculate F(r')
2190     Fxnp1=-xnp1/rnp1**3
2191     Fynp1=-ynp1/rnp1**3
2192
2193
2194     # Calculate the new velocities
2195     vxnp1=vxn+h*Fxnp1
2196     vynp1=vyn+h*Fynp1
```

```
2197
2198
2199        t=t+h
2200        xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
2201        x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
2202
2203        #Append energy and momentum
2204        energy_array.append(energy(vxn, vyn, rn))
2205        momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
2206
2207
2208    #Plot results
2209    plt.plot(x,y, label = "Modfied Euler")
2210
2211    #Predicted orbit
2212    X = np.arange(0, 2*math.pi, h)
2213    plt.plot(a*(np.cos(X)+e),b*np.sin(X), label ="Predicted Orbit")
2214    plt.title('Modified Euler')
2215    plt.xlabel('x')
2216    plt.ylabel('y')
2217    # Star position
2218    plt.plot(0, 0, 'o' , color ='Red', label = "Star Postion")
2219    plt.legend(loc = "upper left")
2220    plt.show()
2221
2222    #Energy conservation
2223    energy_frac = []
2224    for i in energy_array:
2225        energy_frac.append(abs((i-energy_array[0])/energy_array[0]))
2226
2227    mom_frac = []
2228    for i in momentum_array:
2229        mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))
2230
2231
2232
2233    # plot the log log scale of energy and angular momentum
2234    plt.loglog(time, energy_frac)
2235    plt.title("Modified Euler Energy conservation log-log")
2236    plt.show()
2237    plt.loglog(time, mom_frac)
2238    plt.title("Modified Euler momentum conservation log-log")
2239    plt.show()
2240    #!/usr/bin/env python3
2241    # -*- coding: utf-8 -*-
2242    """
2243    Created on Fri Nov 25 14:36:52 2022
2244
2245    @author: jai
2246    """
2247    # Code to plot each of the integration schemes, angular momentum and
        energy conservation with different parameters:
2248    # Now we  consider e = 0.9, N = 100 and 1000 force evaluations per
        orbit.
2249
2250    import matplotlib.pyplot as plt
2251    import math
2252    import numpy as np
```

```
2253
2254 #define parameters
2255 P=2*math.pi
2256 tstart=0
2257 tend=100*P
2258 h=P/500
2259
2260 e = 0.9
2261 E = 0
2262 b = math.sqrt((1-e*e))
2263 a = 1
2264 GM = a
2265
2266 #set initial conditions
2267 x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
2268 t=tstart
2269 xn=x0; yn=y0; vxn=vx0; vyn=vy0
2270
2271
2272 #define arrays to store data for plotting/analysis
2273 x=[]; y=[]; vx=[]; vy=[]
2274
2275 #Energy and Angular Momentum arrays
2276 energy_array = []; momentum_array = []
2277
2278 def energy(vxn, vyn, rn):
2279     e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
2280     return e
2281
2282 def angular_momentum(xn, vxn, yn, vyn):
2283     l = xn*vxn-yn*vyn
2284     return l
2285 time = []
2286 #main time-stepping loop
2287 while t<tend:
2288
2289     rn = math.sqrt(xn*xn+yn*yn);
2290     vn = math.sqrt(vyn*vyn+vxn*vxn)
2291
2292     #Calculate r'
2293
2294     xnd = xn + h*vxn/2
2295     ynd = yn + h*vyn/2
2296     rnd = math.sqrt(xnd*xnd + ynd*ynd)
2297
2298     #Calculate F(r')
2299     Fxnd = -xnd/rnd**3
2300     Fynd = -ynd/rnd**3
2301
2302     #Calculate new velocities
2303     vxnp1 = vxn + h*Fxnd
2304     vynp1 = vyn + h*Fynd
2305
2306     #Calculate new coordinates
2307     xnp1 = xnd + h*vxnp1/2
2308     ynp1 = ynd + h*vynp1/2
2309
2310     t=t+h
```

```
    xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

    #Append energy and momentum
    energy_array.append(energy(vxn, vyn, rn))
    momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
    time.append(t)
#plot results
plt.plot(x,y, label = "Leapfrog")
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Expected Orbit")
plt.xlabel('y')
plt.ylabel('y')

plt.title('Leapfrog')
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "Star Postion")
plt.legend(loc = "upper left")
plt.show()
#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))

# plot the log log scale of energy and angular momentum
plt.loglog(time, energy_frac)
plt.title("Leapfrog Energy conservation log-log")
plt.show()
plt.loglog(time, mom_frac)
plt.title("Leapfrog momentum conservation log-log")
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 25 14:39:06 2022

@author: jai
"""
# Code to plot each of the integration schemes, angular momentum and
    energy conservation with different parameters:
# Now we  consider e = 0.9, N = 100 and 1000 force evalutations per
    orbit.
# RK4 method
import matplotlib.pyplot as plt
import math
import numpy as np
#define parameters
P=2*math.pi
tstart=0
tend=100*P
h=P/250
e = 0.9
E = 0
b = math.sqrt((1-e*e))
```

```
a = 1
GM = a
#set initial conditions
x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0

#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]

#Energy and Angular Momentum arrays
energy_array = []; momentum_array = []; time = []

def energy(vxn, vyn, rn):
    e = 1/2*(vxn*vxn + vyn*vyn) - GM/rn
    return e

def angular_momentum(xn, vxn, yn, vyn):
    l = xn*vxn-yn*vyn
    return l
#main time-stepping loop using 4th order RK4
while t<tend:
    rn=math.sqrt(xn*xn+yn*yn)
    wn=np.array([xn,yn, vxn, vyn] )
    wdotn=np.array([vxn,vyn,-xn/rn**3,-yn/rn**3])
    k1=h*wdotn
    wn1=wn+1/2*k1
    rn1=math.sqrt(wn1[0]**2+wn1[1]**2)
    k2= h*np.array ([wn1[2], wn1[3],-wn1[0]/rn1**3, -wn1[1]/rn1**3])
    wn2=wn+1/2*k2
    rn2 = math.sqrt(wn2[0]**2+wn2[1]**2)
    k3= h*np.array([wn2[2] , wn2[3],-wn2[0]/rn2**3, -wn2[1]/rn2**3])
    wn3=wn+k3
    rn3=math.sqrt(wn3[0]**2+wn3[1]**2)
    k4=h*np.array([wn3[2],wn3[3], -wn3[0]/rn3**3,-wn3[1]/rn3**3])
    wnp1=wn+1/6*(k1+2*k2+2*k3+k4)
    t=t+h

    xn =wnp1[0]; yn =wnp1[1]; vxn=wnp1[2]; vyn=wnp1[3]
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn)

    #Append energy and momentum
    energy_array.append(energy(vxn, vyn, rn))
    momentum_array.append(angular_momentum(xn, vxn, yn, vyn))
    time.append(t)

#plot results
plt.plot(x,y, label = "Runge-Kutta 4")
X = np.arange(0, 2*math.pi, h)
plt.plot(a*(np.cos(X)+e),b*np.sin(X), label = "Predicted Orbit")

plt.title('Runge-Kutta 4')
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "Star Postion")
plt.legend(loc = "upper left")
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
#Energy conservation
energy_frac = []
for i in energy_array:
    energy_frac.append(abs((i-energy_array[0])/energy_array[0]))

mom_frac = []
for i in momentum_array:
    mom_frac.append(abs((i-momentum_array[0])/momentum_array[0]))

# plot the log log scale of energy and angular momentum
plt.loglog(time,energy_frac)
plt.title("Runge Kutta 4E nergy conservation log-log")
plt.show()
plt.loglog(time, mom_frac)
plt.title("Runge-Kutta 4 Momentum conservation log-log")
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Dec 20 12:49:07 2022

@author: jai
"""

# Time reversibility 300 FE

import matplotlib.pyplot as plt
import math
import numpy as np

#define parameters
GM = 1 # from the sheet
a = 1 # from the sheet
P= 2*math.pi*(math.sqrt(a**3/GM))
e = 0.5 # the ellipicity from the sheet
tstart=0
tend=10*P # not sure what this is
h=P/300 #as instructed in the sheet
b = a * math.sqrt(1-e**2)
E = np.arange(0,2*math.pi, 0.00001)

#set initial conditions
x0= a*(1+e)
y0=0
vx0=0
vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0

#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]


#main time-stepping loop using 1st order Euler
while t<tend:
 rn=math.sqrt(xn*xn+yn*yn);
 Fxn=-xn/rn**3; Fyn=-yn/rn**3
 xnp1=xn+h*vxn
```

```
 ynp1=yn+h*vyn
 vxnp1=vxn+h*Fxn
 vynp1=vyn+h*Fyn
 t=t+h
 xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
 x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

plt.plot(x,y , label = "Forward Euler", linewidth = '5')
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "star postion")


#Reversing forward euler:
vxn = -vxn
vyn = -vyn
t=0

while t<tend:
  rn=math.sqrt(xn*xn+yn*yn);
  Fxn=-xn/rn**3; Fyn=-yn/rn**3
  xnp1=xn+h*vxn
  ynp1=yn+h*vyn
  vxnp1=vxn+h*Fxn
  vynp1=vyn+h*Fyn
  t=t+h
  xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
  x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

plt.plot(x,y, label = "Reversed Forward Euler")
plt.title("Reversed Forward Euler")
plt.legend(loc = "upper left")
plt.show()

xdiff = abs(x[-1]-x0)
ydiff = abs(y[-1]- y0)
vxdiff = abs(-vx[-1] - vx0)
vydiff = abs(-vy[-1]-vy0)
print(xdiff, ydiff, vxdiff, vydiff)
#
   ------------------------------------------------------------------------


#define parameters
GM = 1 # from the sheet
a = 1 # from the sheet
P= 2*math.pi*(math.sqrt(a**3/GM))
e = 0.5 # the ellipicity from the sheet
tstart=0
tend=10*P # not sure what this is
h=P/300 #as instructed in the sheet, increasing this to p/100000 gets
   identical to predicted plot
b = a * math.sqrt(1-e**2)
E = np.arange(0,2*math.pi, 0.001)

#set initial conditions
x0= a*(1+e); y0=0; vx0=0; vy0= math.sqrt((GM*(1-e))/(a*(1+e)))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0
```

```
2538
2539 #define arrays to store data for plotting/analysis
2540 x=[]; y=[]; vx=[]; vy=[]
2541
2542
2543 #ii) Modified Euler
2544
2545
2546 #Creating the loop and plugging in the rn+1 value into the Vn+1
2547
2548 while t<tend:
2549
2550   rn=math.sqrt(xn**2+yn**2);
2551   # Calculate F(r')
2552   Fxn=-xn/rn**3
2553   Fyn=-yn/rn**3
2554
2555   # Calculate r'
2556   xnp1=xn+h*vxn
2557   ynp1=yn+h*vyn
2558
2559   rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)
2560
2561  # Calculate F(r')
2562   Fxnp1=-xnp1/rnp1**3
2563   Fynp1=-ynp1/rnp1**3
2564
2565
2566   # Calculate the new velocities
2567   vxnp1=vxn+h*Fxnp1
2568   vynp1=vyn+h*Fynp1
2569
2570
2571   t=t+h
2572   xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
2573   x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
2574
2575 plt.plot(x,y, label = "Modfied Euler",linewidth = '5')
2576
2577 # Star position
2578 plt.plot(0, 0, 'o' , color ='Red', label = "star postion")
2579
2580 #Reversing
2581 vxn = -vxn
2582 vyn = -vyn
2583 t=0
2584
2585 #Reversing
2586 while t<tend:
2587
2588   rn=math.sqrt(xn**2+yn**2);
2589   # Calculate F(r')
2590   Fxn=-xn/rn**3
2591   Fyn=-yn/rn**3
2592
2593   # Calculate r'
2594   xnp1=xn+h*vxn
2595   ynp1=yn+h*vyn
```

```
2596
2597   rnp1 = math.sqrt(xnp1*xnp1 + ynp1*ynp1)
2598
2599  # Calculate F(r')
2600   Fxnp1=-xnp1/rnp1**3
2601   Fynp1=-ynp1/rnp1**3
2602
2603
2604   # Calculate the new velocities
2605   vxnp1=vxn+h*Fxnp1
2606   vynp1=vyn+h*Fynp1
2607
2608
2609   t=t+h
2610   xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
2611   x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
2612
2613 plt.plot(x,y, label = " Reversed Modified Euler")
2614 plt.title("Reversed Modified Euler")
2615 plt.legend(loc = "upper left")
2616 plt.show()
2617
2618 xdiff_modified_euler = abs(x[-1]-x0)
2619 ydiff_modified_euler = abs(y[-1]- y0)
2620 vxdiff_modified_euler = abs(-vx[-1]-vx0)
2621 vydiff_modified_euler = abs(-vy[-1]-vy0)
2622 print(xdiff_modified_euler, ydiff_modified_euler, vxdiff_modified_euler
       , vydiff_modified_euler)
2623 #
      ----------------------------------------------------------------------------------------------------

2624 # LEAPFROG
2625 #define parameters
2626 P=2*math.pi
2627 tstart=0
2628 tend=100*P
2629 h=P/150
2630
2631 e = 0.5
2632 E = 0
2633 b = math.sqrt((1-e*e))
2634 a = 1
2635 GM = a
2636
2637 #set initial conditions
2638 x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
2639 t=tstart
2640 xn=x0; yn=y0; vxn=vx0; vyn=vy0
2641
2642
2643 #define arrays to store data for plotting/analysis
2644 x=[]; y=[]; vx=[]; vy=[]
2645
2646
2647 #main time-stepping loop
2648 while t<tend:
2649
2650     rn = math.sqrt(xn*xn+yn*yn);
```

```
    vn = math.sqrt(vyn*vyn+vxn*vxn)

    #Calculate r'

    xnd = xn + h*vxn/2
    ynd = yn + h*vyn/2
    rnd = math.sqrt(xnd*xnd + ynd*ynd)

    #Calculate F(r')
    Fxnd = -xnd/rnd**3
    Fynd = -ynd/rnd**3

    #Calculate new velocities
    vxnp1 = vxn + h*Fxnd
    vynp1 = vyn + h*Fynd

    #Calculate new coordinates
    xnp1 = xnd + h*vxnp1/2
    ynp1 = ynd + h*vynp1/2

    t=t+h
    xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
plt.plot(x,y, label ="Leapfrog", linewidth = '5')
# Star position
plt.plot(0, 0, 'o' , color ='Red', label = "star postion")


# Reversing

vxn = -vxn
vyn = -vyn
t=0

while t<tend:

    rn = math.sqrt(xn*xn+yn*yn);
    vn = math.sqrt(vyn*vyn+vxn*vxn)

    #Calculate r'

    xnd = xn + h*vxn/2
    ynd = yn + h*vyn/2
    rnd = math.sqrt(xnd*xnd + ynd*ynd)

    #Calculate F(r')
    Fxnd = -xnd/rnd**3
    Fynd = -ynd/rnd**3

    #Calculate new velocities
    vxnp1 = vxn + h*Fxnd
    vynp1 = vyn + h*Fynd

    #Calculate new coordinates
    xnp1 = xnd + h*vxnp1/2
    ynp1 = ynd + h*vynp1/2

    t=t+h
```

```
    xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);

plt.plot(x,y, label =" Reversed Leapfrog")
plt.title("Reversed Leapfrog")
plt.legend(loc = "upper left")
plt.show()



xdiff_leapfrog = abs(x[-1]-x0)
ydiff_leapfrog = abs(y[-1]- y0)
vxdiff_leapfrog = abs(-vx[-1]-vx0)
vydiff_leapfrog = abs(-vy[-1]-vy0)
print(xdiff_leapfrog, ydiff_leapfrog, vxdiff_leapfrog, vydiff_leapfrog)

#
    ------------------------------------------------------------------------------


#RK4
#define parameters
P=2*math.pi
tstart=0
tend=100*P
h=P/75

e = 0.5
E = 0
b = math.sqrt((1-e*e))
a = 1
GM = a

#set initial conditions
x0=1+e; y0=0; vx0=0; vy0=math.sqrt((1-e)/(1+e))
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[]

#main time-stepping loop
#main time-stepping loop using 4th order RK4
while t<tend:
    rn=math.sqrt(xn*xn+yn*yn)
    wn=np.array([xn,yn, vxn, vyn] )
    wdotn=np.array([vxn,vyn,-xn/rn**3,-yn/rn**3])
    k1=h*wdotn
    wn1=wn+1/2*k1
    rn1=math.sqrt(wn1[0]**2+wn1[1]**2)
    k2= h*np.array ([wn1[2], wn1[3],-wn1[0]/rn1**3, -wn1[1]/rn1**3])
    wn2=wn+1/2*k2
    rn2 = math.sqrt(wn2[0]**2+wn2[1]**2)
    k3= h*np.array([wn2[2] , wn2[3],-wn2[0]/rn2**3, -wn2[1]/rn2**3])
    wn3=wn+k3
    rn3=math.sqrt(wn3[0]**2+wn3[1]**2)
    k4=h*np.array([wn3[2],wn3[3], -wn3[0]/rn3**3,-wn3[1]/rn3**3])
```

```
    wnp1=wn+1/6*(k1+2*k2+2*k3+k4)
    t=t+h

    xn =wnp1[0]; yn =wnp1[1]; vxn=wnp1[2]; vyn=wnp1[3]
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn)

plt.plot(x,y, label = "RK4" ,linewidth = '5')
# Star position
plt.plot(0, 0, 'o', color ='Red', label = "star postion")

#Reversing

vxn = -vxn
vyn = -vyn
t = 0

while t<tend:
    rn=math.sqrt(xn*xn+yn*yn)
    wn=np.array([xn,yn, vxn, vyn] )
    wdotn=np.array([vxn,vyn,-xn/rn**3,-yn/rn**3])
    k1=h*wdotn
    wn1=wn+1/2*k1
    rn1=math.sqrt(wn1[0]**2+wn1[1]**2)
    k2= h*np.array ([wn1[2], wn1[3],-wn1[0]/rn1**3, -wn1[1]/rn1**3])
    wn2=wn+1/2*k2
    rn2 = math.sqrt(wn2[0]**2+wn2[1]**2)
    k3= h*np.array([wn2[2] , wn2[3],-wn2[0]/rn2**3, -wn2[1]/rn2**3])
    wn3=wn+k3
    rn3=math.sqrt(wn3[0]**2+wn3[1]**2)
    k4=h*np.array([wn3[2],wn3[3], -wn3[0]/rn3**3,-wn3[1]/rn3**3])
    wnp1=wn+1/6*(k1+2*k2+2*k3+k4)
    t=t+h

    xn =wnp1[0]; yn =wnp1[1]; vxn=wnp1[2]; vyn=wnp1[3]
    x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn)

plt.plot(x,y, label = "Reversed RK4")
plt.legend(loc = "upper left")
plt.title("Reversed RK4")
plt.show()


xdiff_RK4 = abs(x[-1] - x0)
ydiff_RK4 = abs(y[-1] - y0)
vxdiff_RK4= abs(-vx[-1]-vx0)
vydiff_RK4 = abs(-vy[-1]-vy0)
print(xdiff_RK4, ydiff_RK4, vxdiff_RK4, vydiff_RK4)
```

Appendix B

Appendix Chapter 2

B.1 Python Code: Random Numbers to Integrate

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Feb  6 16:33:33 2023

@author: jai
"""

# Random Numbers to Integrate

# Strater Task 1:
# Write python code to evaluate the mean and variance of a random
    sample drawn
# from some probability distrubutions.

import random
import statistics
import math
import numpy as np
import matplotlib.pyplot as plt
from numpy import pi


#random uniform
def random_uniform(a, b, trials):
    array1=[]

    # total = 0
    # squaretotal= 0
    for i in range(trials):
        number = random.uniform(a,b)
        # total =+  single
        # square_total = single*single
        array1.append(number)
    average = statistics.mean(array1)
    variance1 = statistics.variance(array1)

    return(average, variance1)

```

```
print(random_uniform(5,4,9))

#normal distrubution

def random_normal(trials, mu, sigma):
    array1=[]

    # total = 0
    # squaretotal= 0
    for i in range(trials):
        number = random.normalvariate(mu, sigma)
        # total =+  single
        # square_total = single*single
        array1.append(number)
    average = statistics.mean(array1)
    variance1 = statistics.variance(array1)

    return(average, variance1)

print(random_normal(5,4,9))

#random exponential
def random_exponential(trials, lmda):
    array1=[]

    # total = 0
    # squaretotal= 0
    for i in range(trials):
        number = random.expovariate(lmda)
        # total =+  single
        # square_total = single*single
        array1.append(number)
    average = statistics.mean(array1)
    variance1 = statistics.variance(array1)

    return(average, variance1)

print(random_exponential(5,1))

# Starter task 2 Obtain a formula for signa and hence an error estimate. How does this compare with
# quadrature schemes such as the trapezium rule or simposons rule? Try estimating the integral of something using monte carlo sampling

# Define function to integrate
def f(x):
    return 0.5*math.cos(x)**2

exact_area = 0.25*(2+0.5*math.sin(4))

#plotting the function
x = np.linspace(0, 2, 100)
y = 0.5 * np.cos(x)**2

fig, ax = plt.subplots()
ax.fill_between(x, y, 0, alpha=0.5)
ax.plot(x, y, label='f(x) = 0.5cos^2(x)')
ax.set_xlabel('x')
```

```
ax.set_ylabel('f(x) = 0.5cos^2(x)')
ax.set_title('Plot of the function f(x) = 0.5cos^2(x)')
ax.legend()
ax.grid()
plt.show()

#implementing the rectanglular method
#Rectangular midpoint method
def rectangle(a,b,n):
    h = (b-a)/n
    Area = 0
    for i in range(n):
        Area += f((a + h/2.0) + i*h)
    Area *= h
    rectangle_error = abs(Area-exact_area)
    return Area, rectangle_error
print("Rectangular Rule estimation:",rectangle(0, 2, 10))

# Implementing trapezoidal method # code sansar reference
def trapezium_integration(a,b,n):
    # calculating step size
    h = (b - a) / n
    # Finding sum
    trapezium_integration = f(a) + f(b)
    for i in range(1,n):
        k = a + i*h
        trapezium_integration = trapezium_integration + 2 * f(k)
    # Finding final integration value
    trapezium_integration = trapezium_integration * h/2
    #finding the trapzeium error
    trapezium_error = abs(exact_area-trapezium_integration)
    percentage_error = (trapezium_error)/exact_area * 100
    return (trapezium_integration,trapezium_error,percentage_error)
print("Trapezium Rule estimation", trapezium_integration(0,2,100))

result = 0

# Implementing Simpson's 1/3  #code sansar

def simpson(a,b,n):
    #calc the step size
    h=(b-a)/n
    simpson_integration=0
    #finding the sum
    for i in range(n):
        x1=a+i*h
        x2=x1 + h/2
        x3=x1+h
        simpson_integration+=(h/6)*(f(x1)+4*f(x2)+f(x3))
    simpsons_error = abs(simpson_integration-exact_area)
    percentage_error = simpsons_error/exact_area
    return simpson_integration,simpsons_error, percentage_error
print("Simpson Rule estimation", simpson(0,2,10))


#Implementing monte carlo integration
def monte_carlo(a,b,n):
```

```
152     area = (b-a)*f(random.uniform(a,b)) #i get more accurate results
            when it is random.random
153     for i in range (1,n):
154         area += (b-a)*f(random.uniform(a,b)) #i get more accurate
                results when it is random.random
155         monte_carlo_difference = abs(exact_area-area/n)
156     return (area/n, monte_carlo_difference)
157 print("Monte Carlo Rule estimation", monte_carlo(0,2,100))
158
159
160
161 #consider the error of the monte carlo integration - this gives the
        sigma error from the formula in sheet
162 def monte_carlo_integration_error(a,b,n):
163     array1=[]
164     for i in range(n):
165         number = random.uniform(a,b) #note here we are using a unifrom
                dist as we randomly selected points using the uniform
                distribution
166         # total =+  single
167         # square_total = single*single
168         array1.append(number)
169     #average = statistics.mean(array1)
170     variance1 = statistics.variance(array1)
171     mean_squared_error = (b-a)*math.sqrt(variance1)/math.sqrt(n)
172     return(mean_squared_error)
173 print("Monte Carlo Mean Squared error=",monte_carlo_integration_error
        (0, 2, 100))
174
175
176
177 def monte_carlo_pi(n): #with help from kaggle \cite{soachishti}
178     # Creating a square and circle stencil
179     square_x = [1,-1,-1,1,1]
180     square_y = [1,1,-1,-1,1]
181     circle_y,circle_x = [],[]
182
183     for i in range(361):
184         circle_y.append(np.cos(np.pi*i/180))
185         circle_x.append(np.sin(np.pi*i/180))
186
187     # vectors for values
188     in_x = []
189     in_y = []
190     out_x = []
191     out_y = []
192     Iteration = []
193     estimate_pi = []
194     inside = 0
195
196     # Uniform random selection between -1,1
197     for i in range(n):
198         x = random.uniform(-1, 1)
199         y = random.uniform(-1, 1)
200         r = np.sqrt(x**2+y**2)
201         Iteration.append(i)
202         if r <= 1:
203             inside +=1
```

```
            in_x.append(x)
            in_y.append(y)
        else:
            out_x.append(x)
            out_y.append(y)
        estimate_pi.append(4*inside/(i+1))

    pi_estimate = 4*inside / n
    pi_error = round(100*((pi_estimate-pi)/pi),4)

    # Draw a 2D plot of where our iterations landed compared to the square and circle
    plt.plot(square_x,square_y,color='Blue', label='Square')
    plt.plot(circle_y,circle_x,color='Red', label='Circle')
    plt.scatter(in_x,in_y,color='Pink',marker=".", label='Inside')
    plt.scatter(out_x,out_y,color='Orange',marker=".", label='Outside')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.axis('equal')
    plt.legend(loc ='lower left')
    plt.show()

    #plot of current estimate of pi vs. iteration number
    plt.loglog(Iteration,estimate_pi,color='Orange', label='Monte Carlo Estimation')
    plt.axhline(y=pi,color='pink',ls='--', label='Exact value')
    plt.xlabel('Iteration Number')
    plt.ylabel('Monte Carlo Estimation of pi')
    plt.title("Convergence towards Pi")
    plt.legend()
    plt.show()

    # Print out our final estimate and how it compares to the true value
    print('\n' + f'Pi is approximately {pi_estimate}\n')
    print(f'This is {pi_error}% off the true value.\n')


def exact_sphere_vol(N):#just the formula for the exact volume
    return math.pi**(N/2)/math.gamma(N/2+1)

#Unit sphere of N dimensions volume
def Nsphere_vol(N,n): #N is the number of dimensions and n is the number of samples
    inside=0
    for i in range(n):
        hypercube_point = np.random.uniform(low=-1, high=1, size=N) # generate N random numbers between -1 and 1
        if np.linalg.norm((hypercube_point)) <= 1:
            inside += 1
    cube_volume = (2*1)**N
    sphere_volume = cube_volume *(inside/n)
    return sphere_volume


## define the number of samples
n_values = 100000
# compute and print the estimated volumes for different values of N
```

```
N_values = [2,3,4,5,6,7,8,9,10,11,12,13,14,15]
estimated_vols = []
for N in N_values:
    estimated_vols.append(Nsphere_vol(N, n_values))
# plot the estimated volumes against N
plt.plot(N_values, estimated_vols, 'fo')
# add axis labels, title, and legend
plt.xlabel('Dimensionality (N)')
plt.ylabel('Estimated Volume')
plt.title('Monte Carlo Estimates of N-Dimensional Sphere Volume')
# display the plot
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 24 16:53:18 2023

@author: jai
"""

import math
import matplotlib.pyplot as plt
#This script will show the convergence for each of the methods towards
    the exact area
#Trazpezium method to include time series plot
# Define function to integrate
def f(x):
    return 0.5*math.cos(x)**2

exact_area = 0.25*(2+0.5*math.sin(4))
def rectangle(a,b,n):
    h = (b-a)/n
    Area = 0
    for i in range(n):
        Area += f((a + h/2.0) + i*h)
    Area *= h
    return Area

rectangle_area=[]
n_values=[]
n=10000
for n in range(1,n):
    b=rectangle(0,2,n)
    n_values.append(n)
    rectangle_area.append(b)
# plot time series and exact value
plt.loglog(n_values, rectangle_area, label='Rectangle Rule Estimation',
    color ="red")
plt.xlabel('Number of interations')
plt.ylabel('Integration value')
plt.legend()

# Implementing trapezoidal method
def trapezium_integration(a,b,n):
    # calculating step size
    h = (b - a) / n
    # Finding sum
    trapezium_integration = f(a) + f(b)
```

```
    for i in range(1,n):
        k = a + i*h
        trapezium_integration = trapezium_integration + 2 * f(k)
    # Finding final integration value
    trapezium_integration = trapezium_integration * h/2
    return (trapezium_integration)

trapezium_area=[]
n_values=[]
n=10000
for n in range(1,n):
    b=trapezium_integration(0,2,n)
    n_values.append(n)
    trapezium_area.append(b)
# plot pseudo time series and exact value
plt.loglog(n_values, trapezium_area, label='Trapezium Rule Estimation',
    color = "green")
plt.xlabel('Number of Iterations')
plt.ylabel('Integration value')
plt.legend()


def simpson(a,b,n):
    #calc the step size
    h=(b-a)/n
    simpson_integration=0
    #finding the sum
    for i in range(n):
        x1=a+i*h
        x2=x1 + h/2
        x3=x1+h
        simpson_integration+=(h/6)*(f(x1)+4*f(x2)+f(x3))

    return simpson_integration

simpson_area=[]
n_values=[]
n=10000
for n in range(1,n):
    b=simpson(0,2,n)
    n_values.append(n)
    simpson_area.append(b)
# plot  time series and exact value
plt.loglog(n_values, simpson_area, label="Simpson's 1/3 Rule Estimation
   ", color= 'blue')
plt.xlabel('Number of interations')
plt.ylabel('Integration value')
plt.legend()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 24 16:01:04 2023

@author: jai
"""
import matplotlib.pyplot as plt
import random
import math
```

```
import numpy as np

# This script will utilise the functions I previously defined for the approximations of areas
# and plot the errors for them on a log log graph as the number of interations increases.
# Define the function to integrate
def f(x):
    return 0.5*math.cos(x)**2


exact_area = 0.25*(2+0.5*math.sin(4))

def rectangle(a,b,n):
    h = (b-a)/n
    Area = 0
    for i in range(n):
        Area += f((a + h/2.0) + i*h)
    Area *= h
    return Area

#Create the time series for the errors
rectangle_area=[]
rectangle_error=[]
n_values=[]
n=1000
for n in range(1,n):
   e=rectangle(0, 2, n)
   n_values.append(n)
   rectangle_area.append(e)
   rectangle_error.append(abs(e-exact_area))
plt.loglog(n_values,rectangle_error, label="Rectangle Rule",color='red')
plt.legend()

def trapezium_integration(a,b,n):
    # calculating step size
    h = (b - a) / n
    # Finding sum
    trapezium_integration = f(a) + f(b)
    for i in range(1,n):
        k = a + i*h
        trapezium_integration = trapezium_integration + 2 * f(k)
    # Finding final integration value
    trapezium_integration = trapezium_integration * h/2
    return (trapezium_integration)
print("Trapezium Rule estimation", trapezium_integration(0,2,100))

#Creating a time series for the errors
trapezium_area=[]
n_values=[]
trap_error=[]
n=1000
for n in range(1,n):
    b=trapezium_integration(0,2,n)
    n_values.append(n)
    trapezium_area.append(b)
    trap_error.append(abs(b-exact_area))
```

```
plt.loglog(n_values,trap_error, label='Trapezium Rule',color='green')
plt.title("Trapezium Rule Estimation Errors")
plt.legend()


def simpson(a,b,n):
    #calc the step size
    h=(b-a)/n
    simpson_integration=0
    #finding the sum
    for i in range(n):
        x1=a+i*h
        x2=x1 + h/2
        x3=x1+h
        simpson_integration+=(h/6)*(f(x1)+4*f(x2)+f(x3))

    return simpson_integration


#Creating a time series for the errors
simpson_area=[]
simpson_error=[]
n=1000
n_values=[]
for n in range(1,n):
   c=simpson(0,2,n)
   n_values.append(n)
   simpson_area.append(c)
   simpson_error.append(abs(c-exact_area))
plt.loglog(n_values,simpson_error, label="Simpson's 1/3 Rule",color='
   blue')
plt.title("Simpson's 1/3 Rule Errors")
plt.legend()


def monte_carlo(a,b,n):
    area = (b-a)*f(random.uniform(a,b)) #i get more accurate results
       when it is random.random
    for i in range (1,n):
        area += (b-a)*f(random.uniform(a,b)) #i get more accurate
           results when it is random.random
    return area/n

#Creating a time series for the errors
monte_carlo_area=[]
monte_carlo_error=[]
n_values=[]
n=1000

for n in range(1,n):
   d=monte_carlo(0, 2, n)
   n_values.append(n)
   monte_carlo_area.append(d)
   monte_carlo_error.append(abs(d-exact_area))


# Fit a polynomial curve to the data
x = np.log(n_values)
```

```
y = np.log(monte_carlo_error)
fit = np.polyfit(x, y, deg=1)
poly = np.poly1d(fit)

# Plot the Monte Carlo estimation and the line of best fit
plt.loglog(n_values, monte_carlo_error, label="Monte Carlo Estimation",
    color='Orange')
plt.plot(np.exp(x), np.exp(poly(x)), label="Monte Carlo Line of Best
   Fit", linestyle="--", color='purple')
plt.title("Estimation Errors for Each Scheme, n = 10000")
plt.xlabel("Number of Iterations")
plt.ylabel("Integration Error")
plt.legend()
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Mar  4 18:09:53 2023

@author: jai
"""
#comparison script
import random
import math
import numpy as np
import matplotlib.pyplot as plt

#Unit sphere of N dimensions volume
exact_volume = math.pi**(10/2)/math.gamma(10/2+1) #for 9 dimension
   sphere

#Unit sphere of N dimensions volume
def Nsphere_vol(N,n): #N is the number of dimensions and n is the
   number of samples
    N = 10
    inside=0
    for i in range(n):
        hypercube_point = np.random.uniform(-1,1, size=N) # generate N
           random numbers between -1 and 1
        if np.linalg.norm((hypercube_point)) <= 1:
            inside += 1
    cube_volume = (2*1)**N
    sphere_volume = cube_volume *(inside/n)
    return sphere_volume

n= 100
estimated_volume=[]
n_values=[]
for n in range(1,n):
    v=Nsphere_vol(10,n)
    n_values.append(n)
    estimated_volume.append(v)
# plot  time series and exact value
plt.loglog(n_values, estimated_volume, label="Uniform Estimation",
   color= 'Red')
plt.xlabel('Number of interations')
plt.ylabel('Estimated Volume')
plt.legend()
```

```
531
532
533 def Nsphere_vol2(N, n):
534     inside = 0
535     N=10
536     for i in range(n):
537         hypercube_point = random.normalvariate(1,0.5)# generate random
                numbers between -1 and 1
538         if np.linalg.norm((hypercube_point)) <= 1:
539             inside += 1
540     cube_volume = (2*1)**N
541     sphere_volume = cube_volume *(inside/n)
542     return sphere_volume
543
544 # define the number of samples
545 n= 100
546 estimated_volume=[]
547 n_values=[]
548 for n in range(1,n):
549     v=Nsphere_vol2(10,n)
550     n_values.append(n)
551     estimated_volume.append(v)
552 # plot  time series and exact value
553 plt.loglog(n_values, estimated_volume, label="Normal Estimation", color
    = 'blue')
554 plt.xlabel('Number of interations')
555 plt.ylabel('Estimated Volume')
556 plt.axhline(y=exact_volume, color='purple', linestyle='--', label='
    Exact value')
557 plt.title("Monte Carlo Estimate for a 10 dimension sphere")
558 plt.legend()
559 plt.show()
560 #!/usr/bin/env python3
561 # -*- coding: utf-8 -*-
562 """
563 Created on Sat Mar  4 17:52:17 2023
564
565 @author: jai
566 """
567 import numpy as np
568 import matplotlib.pyplot as plt
569 #Unit sphere of N dimensions volume
570 def Nsphere_vol(N,n): #N is the number of dimensions and n is the
    number of samples
571     inside=0
572     for i in range(n):
573         hypercube_point = np.random.uniform(low=-1, high=1, size=N) #
                generate N random numbers between -1 and 1
574         if np.linalg.norm((hypercube_point)) <= 1:
575             inside += 1
576     cube_volume = (2*1)**N
577     sphere_volume = cube_volume *(inside/n)
578     return sphere_volume
579
580 ## define the number of samples
581 n_values = 10000
582 # compute and print the estimated volumes for different values of N
583 N_values = [2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

```
584 estimated_vols = []
585 for N in N_values:
586     estimated_vols.append(Nsphere_vol(N, n_values))
587 
588 # plot the estimated volumes against N
589 plt.plot(N_values, estimated_vols, '-o')
590 # add axis labels, title, and legend
591 plt.xlabel('Dimensionality (N)')
592 plt.ylabel('Estimated Volume')
593 plt.legend()
594 #!/usr/bin/env python3
595 # -*- coding: utf-8 -*-
596 """
597 Created on Wed Mar  1 16:27:33 2023
598 
599 @author: jai
600 """
601 
602 import math
603 import numpy as np
604 import matplotlib.pyplot as plt
605 
606 exact_value = math.pi
607 
608 def monte_carlo_pi(n):
609     # Draw a square and a circle to frame out simulation
610     circle_y,circle_x = [],[]
611 
612     for i in range(361):
613         circle_y.append(np.cos(np.pi*i/180))
614         circle_x.append(np.sin(np.pi*i/180))
615 
616     # vectors for values
617     in_x,in_y,out_x,out_y,Iteration,current_pi = [],[],[],[],[],[]
618     inside = 0
619 
620     # Generate a bunch of values of x and y between -1 and 1
621     for i in range(n):
622         x = 2*(np.random.uniform()-0.5)
623         y = 2*(np.random.uniform()-0.5)
624         r = np.sqrt(x**2+y**2)
625         Iteration.append(i)
626         if r <= 1:
627             inside +=1
628             in_x.append(x)
629             in_y.append(y)
630         else:
631             out_x.append(x)
632             out_y.append(y)
633         current_pi.append(4*inside/(i+1))
634 
635     pi_estimate = 4*inside / n
636     return pi_estimate
637 
638 def monte_carlo_oi_error(n):
639     #Creating a time series for the errors
640     monte_carlo_area=[]
641     monte_carlo_error=[]
```

```
    n_values=[]

    for n in range(1,n):
        d=monte_carlo_pi(n)
        n_values.append(n)
        monte_carlo_area.append(d)
        monte_carlo_error.append(abs(d-exact_value))


    # Fit a polynomial curve to the data
    x = np.log(n_values)
    y = np.log(monte_carlo_error)
    fit = np.polyfit(x, y, deg=1)
    poly = np.poly1d(fit)
    # Plot the Monte Carlo estimation and the line of best fit
    plt.loglog(n_values, monte_carlo_error, label="Monte Carlo
        Estimation", color='Orange')
    plt.plot(np.exp(x), np.exp(poly(x)), label="Monte Carlo Line of
        Best Fit", linestyle="--", color='purple')
    plt.title("Error for Monte Carlo Pi Estimate")
    plt.xlabel("Number of Iterations")
    plt.ylabel("Absolute Error")
    plt.legend()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Mar  1 17:45:07 2023

@author: jai
"""
import math
import matplotlib.pyplot as plt
import numpy as np
from numpy import pi

# Simpson's 1/3 rule to estimate unit circle and MC estimation
# x^2 +y^2 = 1 and rearrange for y to be subject
# its botched because I have to enter simpson(0,1,n) but i want between
    -1 and 1?
# resutls still look viable however
def f(x):
    return np.sqrt(1 - x**2)

exact_value = math.pi

def simpson(a,b,n):
    #calc the step size
    h=(b-a)/n
    simpson_integration=0
    #finding the sum
    for i in range(n):
        x1=a+i*h
        x2=x1 + h/2
        x3=x1+h
        simpson_integration+=(h/6)*(f(x1)+4*f(x2)+f(x3))

    return simpson_integration*4

```

```
697
698 simpson_area=[]
699 simpson_error=[]
700 n_values=[]
701 n=1000
702 for n in range(1,n):
703     b=simpson(0,1,n)
704     n_values.append(n)
705     simpson_area.append(b)
706     simpson_error.append(abs(b-exact_value))
707 # plot pseudo time series and exact value
708 plt.loglog(n_values, simpson_area, label="Simpson's 1/3 Rule Estimation
    ", color= 'blue')
709 plt.xlabel('Number of interations')
710 plt.ylabel('Integration value')
711 plt.legend()
712
713 def monte_carlo_pi(n):
714     # Draw a square and a circle to frame out simulation
715     circle_y,circle_x = [],[]
716
717     for i in range(361):
718         circle_y.append(np.cos(np.pi*i/180))
719         circle_x.append(np.sin(np.pi*i/180))
720
721     # vectors for values
722     in_x,in_y,out_x,out_y,Iteration,current_pi = [],[],[],[],[],[]
723     inside = 0
724
725     # Generate a bunch of values of x and y between -1 and 1
726     for i in range(n):
727         x = 2*(np.random.uniform()-0.5)
728         y = 2*(np.random.uniform()-0.5)
729         r = np.sqrt(x**2+y**2)
730         Iteration.append(i)
731         if r <= 1:
732             inside +=1
733             in_x.append(x)
734             in_y.append(y)
735         else:
736             out_x.append(x)
737             out_y.append(y)
738         current_pi.append(4*inside/(i+1))
739
740     pi_estimate = 4*inside / n
741     return pi_estimate
742
743 #Creating a time series for the errors
744 monte_carlo_area=[]
745 n_values=[]
746 n=1000
747
748 for n in range(1,n):
749     d=monte_carlo_pi(n)
750     n_values.append(n)
751     monte_carlo_area.append(d)
752
753
```

```
# Plot the Monte Carlo estimation and the line of best fit
plt.loglog(n_values, monte_carlo_area, label="Monte Carlo Estimation", color='Orange')
plt.title("Convergence towards Pi, n = 1000")
plt.xlabel("Number of Iterations")
plt.ylabel("Integration Value")
plt.axhline(y=pi,color='pink',ls='--', label='Exact value')
plt.legend()
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Feb 25 11:06:20 2023

@author: jai
"""

#Python script to work out the volume of higher order hyper spheres: extension task 1
  # this is from stack exchange or chat gpt
import numpy as np
from scipy.special import gamma

def sphere_volume(n):
    vol = (np.pi**(n/2) / gamma(n/2 + 1))
    return vol


def monte_carlo_sphere_volume(N, n):
    inside = 0
    for i in range(n):
        x = np.random.uniform(-1, 1, size=N)
        r = np.linalg.norm(x)
        if r <= 1:
            inside += 1
    vol = (2**n) * (inside / n)
    return vol
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Mar  4 15:48:36 2023

@author: jai
"""
import numpy as np
from scipy.stats import qmc
import matplotlib.pyplot as plt
import math

def exact_sphere_vol(N):
    return math.pi**(N/2)/math.gamma(N/2+1)

def Nsphere_vol(N,n):
    inside=0
    for i in range(n):
        hypercube_point = np.random.uniform(low=-1, high=1, size=N)
        if np.linalg.norm((hypercube_point)) <= 1:
            inside += 1
```

```
    cube_volume = (2*1)**N
    sphere_volume = cube_volume *(inside/n)
    return sphere_volume

Nsphere_volume=[]
n_values=[]
n=1000
N=10
for n_value in range(1,n):
    b=Nsphere_vol(N,n_value)
    n_values.append(n_value)
    Nsphere_volume.append(b)

exact_vol = exact_sphere_vol(N)
plt.loglog(n_values, Nsphere_volume, label="Monte Carlo Estimation",
    color= 'orange')
plt.axhline(y=exact_vol, color='red', label="Exact Volume", ls="--")
plt.xlabel('Number of iterations')
plt.ylabel('Volume')
plt.legend()

# Define the dimensionality of the problem
n = 2
# Define the number of points to generate
N = 1000
# Generate a Sobol sequence of points in the n-dimensional unit cube
sobol = qmc.Sobol(d=n, scramble=True)
# Initialize variables for plotting the convergence
iterations = []
estimates = []
# Estimate the sphere volume for increasing numbers of points
for i in range(1, N+1):
    points_cube = sobol.random(n=i)
    norms = np.linalg.norm(points_cube, axis=1)
    points_sphere = points_cube / norms[:, None]
    count_inside = np.sum(norms <= 1)
    volume_sphere = count_inside / i * 2**n
    # Append the iteration number and estimated volume to the plot
        variables
    iterations.append(i)
    estimates.append(volume_sphere)
# Plot the estimated sphere volume versus the number of iterations
plt.loglog(iterations, estimates, label = "Quassi Monte Carlo Estimate"
    )
plt.xlabel('Number of Iterations')
plt.ylabel('Estimated Volume of Sphere')
plt.title('Convergence estimating Nsphere volume')
plt.legend()
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Feb 11 15:03:38 2023

@author: jai
"""
import numpy as np
import matplotlib.pyplot as plt
```

```
865
866 #rectangle rule
867
868 x = np.linspace(0, 2, 10)
869 y = 0.5 * np.cos(x)**2
870
871 fig, ax = plt.subplots()
872 for i in range(1, len(x)):
873     xs = [x[i-1], x[i], x[i], x[i-1]]/2
874     ys = [0, 0, y[i-1], y[i-1]]/2
875     ax.fill(xs, ys, color='gray', alpha=0.5)
876 ax.plot(x, y, label='f(x) = 0.5cos^2(x)')
877 ax.set_xlabel('x')
878 ax.set_ylabel('f(x) = 0.5cos^2(x)')
879 ax.set_title('Plot of the function f(x) = 0.5cos^2(x)')
880 ax.legend()
881 ax.grid()
882 plt.show()
883
884
885 x = np.linspace(0, 2, 10)
886 y = 0.5 * np.cos(x)**2
887
888 fig, ax = plt.subplots()
889 ax.fill_between(x, y, 0, alpha=0.5)
890 ax.plot(x, y, label='f(x) = 0.5cos^2(x)')
891 ax.set_xlabel('x')
892 ax.set_ylabel('f(x) = 0.5cos^2(x)')
893 ax.set_title('Plot of the function f(x) = 0.5cos^2(x)')
894 ax.legend()
895 ax.grid()
896 plt.show()
897
898
899 x = np.linspace(0, 2, 10)
900 y = 0.5 * np.cos(x)**2
901
902 fig, ax = plt.subplots()
903 for i in range(1, len(x)):
904     xs = [x[i-1], x[i], x[i], x[i-1]]
905     ys = [0, 0, y[i], y[i-1]]
906     ax.fill(xs, ys, color='gray', alpha=0.5)
907 ax.plot(x, y, label='f(x) = 0.5cos^2(x)')
908 ax.set_xlabel('x')
909 ax.set_ylabel('f(x) = 0.5cos^2(x)')
910 ax.set_title('Plot of the function f(x) = 0.5cos^2(x)')
911 ax.legend()
912 ax.grid()
913 plt.show()
914
915 x = np.linspace(0, 2, 10)
916 y = 0.5 * np.cos(x)**2
917
918 fig, ax = plt.subplots()
919 for i in range(1, len(x)):
920     xs = [x[i-1], x[i], x[i], x[i-1]]
921     ys = [0, 0, y[i], y[i-1]]
922     ax.fill(xs, ys, color='gray', alpha=0.5)
```

```
ax.plot(x, y, label='f(x) = 0.5cos^2(x)')
ax.set_xlabel('x')
ax.set_ylabel('f(x) = 0.5cos^2(x)')
ax.set_title('Plot of the function f(x) = 0.5cos^2(x)')
ax.legend()
ax.grid()
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 18 10:10:09 2023

@author: jai
"""
#
    ================================================================================

# Python Code to Average the Error and plot it
#
    ================================================================================

import matplotlib.pyplot as plt
import random
import math
import numpy as np

# Define your function here
def f(x):
    return 0.5*math.cos(x)**2
exact_area = 0.25*(2+0.5*math.sin(4))

def rectangle(a,b,n):
    h = (b-a)/n
    Area = 0
    for i in range(n):
        Area += f((a + h/2.0) + i*h)
    Area *= h
    return Area

#Create the time series for the errors
rectangle_area=[]
rectangle_error=[]
n_values=[]
n=10000
for n in range(1,n):
   e=rectangle(0, 2, n)
   n_values.append(n)
   rectangle_area.append(e)
   rectangle_error.append(abs(e-exact_area))
plt.loglog(n_values,rectangle_error, label="Rectangle Rule",color='red'
    )
plt.legend()

def trapezium_integration(a,b,n):
    # calculating step size
    h = (b - a) / n
    # Finding sum
    trapezium_integration = f(a) + f(b)
```

```
    for i in range(1,n):
        k = a + i*h
        trapezium_integration = trapezium_integration + 2 * f(k)
    # Finding final integration value
    trapezium_integration = trapezium_integration * h/2
    return (trapezium_integration)
print("Trapezium Rule estimation", trapezium_integration(0,2,100))

#Creating a time series for the errors
trapezium_area=[]
n_values=[]
trap_error=[]
n=10000
for n in range(1,n):
    b=trapezium_integration(0,2,n)
    n_values.append(n)
    trapezium_area.append(b)
    trap_error.append(abs(b-exact_area))
plt.loglog(n_values,trap_error, label='Trapezium Rule',color='green')
plt.title("Trapezium Rule Estimation Errors")
plt.legend()


def simpson(a,b,n):
    #calc the step size
    h=(b-a)/n
    simpson_integration=0
    #finding the sum
    for i in range(n):
        x1=a+i*h
        x2=x1 + h/2
        x3=x1+h
        simpson_integration+=(h/6)*(f(x1)+4*f(x2)+f(x3))

    return simpson_integration


#Creating a time series for the errors
simpson_area=[]
simpson_error=[]
n=10000
n_values=[]
for n in range(1,n):
   c=simpson(0,2,n)
   n_values.append(n)
   simpson_area.append(c)
   simpson_error.append(abs(c-exact_area))
plt.loglog(n_values,simpson_error, label="Simpson's 1/3 Rule",color='
   blue')
plt.title("Simpson's 1/3 Rule Errors")
plt.legend()



def monte_carlo(a, b, n):
    area = (b - a) * f(random.uniform(a, b))
    for i in range(1, n):
        area += (b - a) * f(random.uniform(a, b))
```

```
    return area / n

def average_monte_carlo_error(a, b, n, num_trials, exact_area):
    errors = []
    for _ in range(num_trials):
        d = monte_carlo(a, b, n)
        errors.append(abs(d - exact_area)) #append each error for each point
    return sum(errors) / num_trials



monte_carlo_error = []
n_values = []
n = 10000
num_trials = 10

for n in range(1, n):
    avg_error = average_monte_carlo_error(0, 2, n, num_trials, exact_area)
    n_values.append(n)
    monte_carlo_error.append(avg_error)

# Fit a polynomial curve to the data
x = np.log(n_values)
y = np.log(monte_carlo_error)
fit = np.polyfit(x, y, deg=1)
poly = np.poly1d(fit)

# Plot the Monte Carlo estimation and the line of best fit
plt.loglog(n_values, monte_carlo_error, label="Monte Carlo Estimation", color='Orange')
plt.plot(np.exp(x), np.exp(poly(x)), label="Monte Carlo Line of Best Fit", linestyle="--", color='purple')
plt.title("Estimation Errors for Each Scheme, n = 10000")
plt.xlabel("Number of Iterations")
plt.ylabel("Integration Error")
plt.legend()
plt.show()
```

Appendix C

Appendix Chapter 3

C.1 Python Code: Period of a Pendulum

```
    #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Mar  9 20:20:31 2023

@author: jai
"""

# Graphs comparing small angle approixmations of trig functions
import matplotlib.pyplot as plt
import numpy as np

# Define the functions
def cos(x):
    return np.cos(x)

def tan(x):
    return np.tan(x)

def sin(x):
    return np.sin(x)

def small_angle_sin(x):
    return x

def small_angle_cos(x):
    return 1 - x**2/2

def small_angle_tan(x):
    return x


# Set up the x-axis range
x = np.linspace(-0.3*np.pi, 0.3*np.pi, 1000)

# Plot the functions
plt.plot(x, sin(x), label='sin(x)', color = 'orange')
plt.plot(x, cos(x), label='cos(x)', color ='blue')
plt.plot(x, tan(x), label='tan(x)',color = 'green')
```

```
plt.plot(x, small_angle_sin(x), label='small angle sin(x)', linestyle =
    'dotted', color = 'red')
plt.plot(x, small_angle_cos(x), label='small angle cos(x)',linestyle =
   'dotted', color ='black')
plt.plot(x, small_angle_tan(x), label ='small angle tan(x)',linestyle =
    'dotted', color = 'purple')
plt.title('Small Angle Approximations')
plt.legend()
plt.xlabel(' Theta')
plt.ylabel('y')
plt.show()
# Define the angle range in radians
theta = np.linspace(0, np.pi/4, 1000)

# Calculate the exact values of sin, cos, and tan
sin_exact = np.sin(theta)
cos_exact = np.cos(theta)
tan_exact = np.tan(theta)

# Calculate the small angle approximations of sin, cos, and tan
sin_approx = theta
cos_approx = 1 - theta**2/2
tan_approx = theta

# Calculate the percent error between the exact and approximated values
sin_error = np.abs(sin_exact - sin_approx)/sin_exact*100
cos_error = np.abs(cos_exact - cos_approx)/cos_exact*100
tan_error = np.abs(tan_exact - tan_approx)/tan_exact*100

# Create a plot of the percent error vs angle
plt.plot(theta, sin_error, label='sin', color = 'orange')
plt.plot(theta, cos_error, label='cos', color = 'blue')
plt.plot(theta, tan_error, label='tan', color = 'green')
plt.legend()
plt.title('Percentage Error of Small Angle Approximations')
plt.xlabel('Theta')
plt.ylabel('Percentage Error (%)')
plt.show()

x = np.linspace(0, 2*np.pi, 1000)
y = np.cos(x)
plt.plot(x, y)
plt.xlabel('Radians')
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 11 12:18:04 2023

@author: jai
"""
import numpy as np
import matplotlib.pyplot as plt

# Define constants, this is to non-dimensionalise the
L = 1
g = 1

```

```
# Define release angles (radians)
theta1 = 0.05
theta2 = 0.1
theta3 = 0.025

# Calculate periods using small angle approximation
T1 = 2*np.pi*np.sqrt(L/g)
T2 = 2*np.pi*np.sqrt(L/g)
T3 = 2*np.pi*np.sqrt(L/g)

# Define time arrays for one period
t1 = np.linspace(0, T1, 1000)
t2 = np.linspace(0, T2, 1000)
t3 = np.linspace(0, T3, 1000)

# Calculate release angles as a function of time
theta1_values = theta1 * np.cos(np.sqrt(g/L) * t1)
theta2_values = theta2 * np.cos(np.sqrt(g/L) * t2)
theta3_values = theta3 * np.cos(np.sqrt(g/L) * t3)

# Plot release angles vs time
plt.plot(t1, theta1_values, label='Release angle = 0.05 rad')
plt.plot(t2, theta2_values, label='Release angle = 0.1 rad')
plt.plot(t3, theta3_values, label='Release angle = 0.025 rad')
plt.xlabel('Time')
plt.ylabel('Release angle (radians)')
plt.title('Simple pendulum with small angle approximation')
plt.grid()
plt.legend()

# Add horizontal line for zero release angle
plt.axhline(y=0, color='r', linestyle='dashed')
# Show the plot
plt.show()
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 13 20:11:02 2023

@author: jai
"""
#
    ================================================================================

# Python code to work out the elliptical integral using simpsons rule
    or trap rule
#
    ================================================================================

import numpy as np
import matplotlib.pyplot as plt
import scipy
import math

exact_period=scipy.special.gamma(0.25)**2/math.sqrt(math.pi)

def f(x):
    return 1 / np.sqrt(np.cos(x))
```

```
148
149
150 def trapezium_integration(a,b,n):
151     # calculating step size
152     b = b - (np.pi/2)/n
153     h = (b - a) / n
154     #redefine b to remove the singularity by taking away 1 trap width (
        the end one)
155     # Finding sum
156     trapezium_integration = f(a) + f(b)
157     for i in range(1,n):
158         k = a + i*h
159         trapezium_integration = trapezium_integration + 2 * f(k)
160     # Finding final integration value
161     trapezium_integration = trapezium_integration * h/2
162     return (trapezium_integration * 2*np.sqrt(2))
163
164
165 n=500
166 trapezium_area=[]
167 trap_error=[]
168 n_values=[]
169 a = 0
170 b = np.pi/2
171 print(trapezium_integration(a, b, n))
172 for n in range(1,n):
173     est=trapezium_integration(0,b,n)
174     n_values.append(n)
175     trapezium_area.append(est)
176     trap_error.append(abs(est-exact_period))
177 # plot pseudo time series
178 plt.loglog(n_values, trap_error, label='Trapezium Rule Estimate, n=5000
    ', color='orange', linewidth = 1)
179 plt.xlabel('Number of Iterations')
180 plt.ylabel('Estimation Error')
181 plt.title("Estimate Error using the Trapezium Rule")
182 plt.legend()
183
184 #
    ================================================================================

185 # Now implement simpsons 1\3
186 #
    ================================================================================

187 def simpson(a,b,n): # change the b when inputing
188     #calc the step size
189     #redfine b to remove the end point
190     b = b - ((np.pi/2)/n)
191     h=(b-a)/n
192     #print(b)
193     simpson_integration=0
194     #finding the sum
195     for i in range(1,n):
196         x1=a+i*h
197         x2=x1 + h/2
198         x3=x1+h
199         simpson_integration+=(h/6)*(f(x1)+4*f(x2)+f(x3))
```

```
200
201     return simpson_integration * 2 * np.sqrt(2)
202
203 n=500
204 simpsons_area=[]
205 simpsons_error=[]
206 n_values=[]
207 a = 0
208 b = np.pi/2
209 for n in range(1,n):
210     est=simpson(0,b,n)
211     n_values.append(n)
212     simpsons_area.append(est)
213     simpsons_error.append(abs(est-exact_period))
214 # plot pseudo time series
215 plt.loglog(n_values, simpsons_error, label='Simpsons Rule Estimate, n
        =5000', color = "DarkBlue", linewidth = 1)
216 plt.xlabel('Number of Iterations')
217 plt.ylabel('Estimation Error')
218 plt.title("Estimation Errors Using Numerical Methods")
219
220
221 p = 500
222 x = np.linspace(1, p, p)
223 y = 1 / np.sqrt(x)
224
225 plt.plot(x, y, '--', color = 'black',label = '1/sqrt(n)')
226 plt.legend()
227 plt.show()
228 #!/usr/bin/env python3
229 # -*- coding: utf-8 -*-
230 """
231 Created on Wed Mar 22 21:10:31 2023
232
233 @author: jai
234 """
235 import numpy as np
236 import matplotlib.pyplot as plt
237 import scipy
238 import math
239
240 exact_period=scipy.special.gamma(0.25)**2/math.sqrt(math.pi) -4*math.
        sqrt(np.pi)
241
242 def f(x):
243     if math.isclose(np.pi/2,x,rel_tol=0,abs_tol=0.0001):
244         return 0
245     else:
246         return (2*math.sqrt(2))/math.sqrt(math.cos(x)) - (2*math.sqrt
            (2))/math.sqrt(np.pi/2 - x)
247
248 def trapezium_integration(a,b,n):
249     # calculating step size
250     h = (b - a) / n
251     # Finding sum
252     trapezium_integration = f(a) + f(b)
253     for i in range(1,n):
254         k = a + i*h
```

```
        trapezium_integration = trapezium_integration + 2 * f(k)
    # Finding final integration value
    trapezium_integration = trapezium_integration * h/2
    return (trapezium_integration)

#Trapezium area estimate
n=5000
trapezium_area=[]
trap_error=[]
n_values=[]
a = 0
b = np.pi/2
for n in range(1,n):
    est=trapezium_integration(0,b,n)
    n_values.append(n)
    trapezium_area.append(est)
    trap_error.append(abs(est-exact_period))
# plot pseudo time series
plt.loglog(n_values, trap_error, label='Trapezium Rule Estimate, n=5000
    ', color='orange', linewidth = 1)
plt.xlabel('Number of Iterations')
plt.ylabel('Estimation Error')
plt.title("Estimate Error using the Trapezium Rule")

# Simpson Rule

def simpson(a,b,n): # change the b when inputing
    h=(b-a)/n
    simpson_integration=0
    #finding the sum
    for i in range(n):
        x1=a+i*h
        x2=x1 + h/2
        x3=x1+h
        simpson_integration+=(h/6)*(f(x1)+4*f(x2)+f(x3))
    return simpson_integration

#Simpson area estimate
n=5000
simpsons_area=[]
simpsons_error=[]
n_values=[]
a = 0
b = np.pi/2
for n in range(1,n):
    est=simpson(0,b,n)
    n_values.append(n)
    simpsons_area.append(est)
    simpsons_error.append(abs(est-exact_period))
# plot pseudo time series
plt.loglog(n_values, simpsons_error, label='Simpsons Rule Estimate, n
    =5000', color = "DarkBlue", linewidth = 1)
plt.xlabel('Number of Iterations')
plt.ylabel('Estimation Error')
plt.legend()
plt.title("Estimation Errors Using Numerical Methods")
```