PH30110 Computational Astrophysics

Hand-in exercise 2024-2025 Semester 2 Python Coursework

Computational Modeling of Orbital Dynamics

University of Bath

Candidate Number: 24367

March 5, 2025

PH30110: Computational Astrophysics

Candidate Number: 24367

Department of Physics,
University of Bath, Bath
BA2 7AY, UK

Date Submitted: March 5, 2025

Contents

Introduction

The accurate simulation of celestial orbits requires solving a set of ordinary differential equations (ODEs) that describe the motion of an orbiting object under the gravitational influence of heavier objects, such as the Sun. In principle, these equations can be solved analytically for simple two-body systems using Keplerian mechanics [1]. However, for systems involving highly eccentric orbits, complex perturbations, or N-body interactions where N $\geq$ 3, no general closed-form solution exists [2]. Instead, numerical integration methods have become indispensable for the prediction of long-term orbital evolution.

One of the simplest approaches to solving ODEs is the Euler method, which estimates the state of a system at the next time step by assuming that the derivative remains constant over the interval. Although conceptually straightforward and computationally inexpensive, the Euler method is only first-order accurate, meaning that its local truncation error is proportional to the square of the step size ($O(h^2)$). This results in significant accumulation of errors, requiring extremely small time steps to maintain precision [3]. Consequently, the Euler method is impractical for orbital simulations, particularly near perihelion (closest approach), where rapid variations in velocity and positions occur over short time frames.

To overcome the challenges and inaccuracies, higher-order methods such as the fourth-order Runge-Kutta (RK4) method are widely used in celestial mechanics [4]. The RK4 method improves on Euler's approach by evaluating four intermediate slopes within each time step, dt, computing a weighted average to approximate the final solution. This increases accuracy to fourth order ($O(h^5)$), allowing for larger time steps while maintaining precision. However, even the RK4 struggles with highly eccentric orbits, where a fixed step size may either be too small (increasing computational cost) or too large (reducing accuracy near perihelion).

A more sophisticated approach is to employ adaptive Runge-Kutta methods, which dynamically adjust step size based on an estimate of the local truncation error. In the case of an adaptive fourth-order Runge-Kutta (RK4) method, the numerical solution is computed at different step sizes, and the difference between these estimates serves as an error measure. If the estimated error exceeds a predefined tolerance, the step size is reduced to improve accuracy. Conversely, if the error is below the threshold, the step size is increased, optimising computational efficiency. The adaptive strategy is particularly advantageous for simulating Halley's Comet, where velocity changes rapidly near perihelion and slows considerably at aphelion (furthest point in orbit).

This report presents the development, implementation, and testing of Python-based programs to simulate orbital dynamics. By comparing the fixed-step RK4 method with adaptive RK4, the study highlights the limitations of fixed-step approaches and the advantages of adaptive techniques for highly eccentric systems. The method is then taken further, handling a three-body system, and finally used to model the system from the centre of mass (CoM) frame. The results reinforce the importance of robust numerical methods in celestial mechanics, where analytical solutions remain impractical beyond simple cases.

Fixed-step RK4 Model

2.1 RK4 Theory

Consider the Sun, of mass M, to be stationary at the origin, with an orbiting comet, of mass m, at position vector $\vec{r}$. The coordinate system can be orientated such that the comet is in the $z = 0$ plane, allowing for z to be ignored and letting $r = \sqrt{x^2 + y^2}$. The force between the Sun and the comet is therefore in the direction of $\frac{\vec{r}}{r}$, such that it is towards the Sun,

$$F = m\vec{a} = m\frac{d^2\vec{r}}{dt^2} = -\left(\frac{GMm}{r^2}\right)\frac{\vec{r}}{r}. \tag{1}$$

This then simplifies to,

$$\frac{d^2\vec{r}}{dt^2} = -\frac{GM}{r^3}\vec{r}.$$

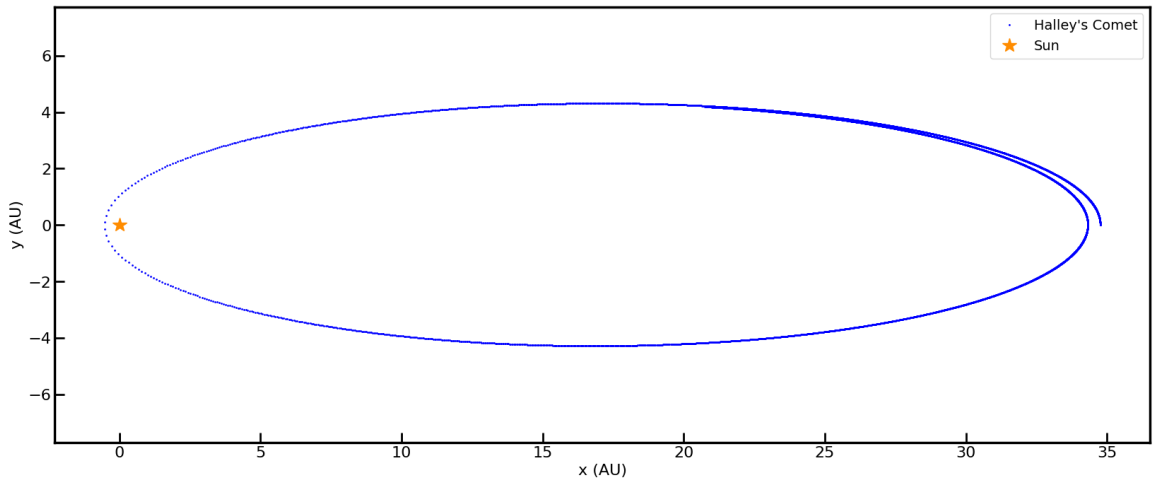

Figure 1: Simulated orbit of Halley's Comet around the Sun using the fixed-step Runge-Kutta integration method. The blue points represent the comet's trajectory, while the orange star marks the Sun's position at the origin. The highly elliptical orbit illustrates the periodic nature of Halley's Comet, and it is clear to see that as the comet's speed increases, the points become more greatly spaced. This plot was computed for a 100 year period and a fixed step of 7.3 days.

Since the comet remains confined to the $z = 0$ plane, the motion can be described in two dimensions with position $\vec{r} = (x, y)$ and velocity $\vec{v} = (v_x, v_y)$, leading to a system of first-order equations:

$$\frac{d}{dx} = v_x, \quad \frac{d}{dy} = v_y \tag{2}$$

$$\frac{dv_x}{dt} = -\frac{GM}{r^3}\vec{x}, \quad \frac{dv_y}{dt} = -\frac{GM}{r^3}\vec{y}. \tag{3}$$

Since an analytical solution to this system is not feasible due to the non-linearity of the equations, a numerical method is required. The Runge-Kutta 4th Order (RK4) was implemented to integrate the system over time. The RK4 computes the next state, y_{n+1}, using a weighted average of four derivative approximations,

$$k_1 = h \cdot f(y, t),$$
$$k_2 = h \cdot f(y + k_1/2, t + h/2),$$
$$k_3 = h \cdot f(y + k_2/2, t + h/2),$$
$$k_4 = h \cdot f(y + k_3, t + h).$$

The final update for y_{n+1} is then given by

$$y_{n+1} = y_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}. \tag{4}$$

Halley's Comet follows a highly eccentric orbit, with an aphelion distance of $r_0 = 5.2 \times 10^9$ km and an initial velocity of $v_0 = 880$ m/s, assumed to be perpendicular to the position vector. To simplify the initial setup, the comet was orientated such that the conditions are,

$$\vec{r} = [r_{aphelion}, 0], \quad \vec{v} = [0, v_{aphelion}]$$

2.2 Implementation and Results

The simulation begins by defining the gravitational acceleration function, `acceleration(f)`, which calculates the acceleration experienced by the comet at any given position. The function takes a state vector $f = [x, y, v_x, v_y]$ computes the radial distance r, and applies Newton's gravitational law to determine the acceleration components a_x and a_y, using Eq.3. The statement `'if r > 1e-12'` prevents division by zero if it were to arise near the origin. The `rk4_step(f,dt)` function implements the RK4 integration scheme, which updates the comet's state by computing intermediate

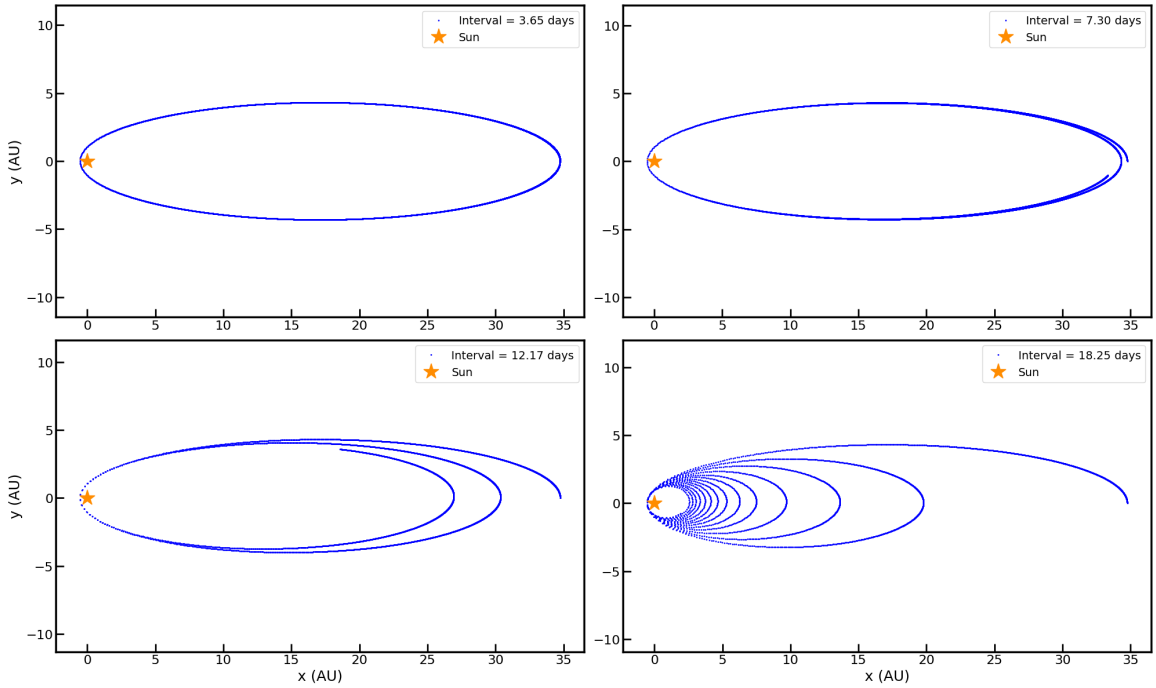

Figure 2: Effect of time step size on the stability of Halley's Comet orbital simulation. Each subplot represents the computed trajectory using different time step intervals $3.65, 7.30, 12.17$ and 18.25 days. The top-left panel (smallest time step) closely follows an elliptical orbit, while increasing the time step introduces noticeable numerical errors, leading to orbital drift and divergence. This demonstrates the importance of choosing an appropriate time step in numerical simulations to maintain accuracy.

slopes and using their weighted average to advance the system in time through Eq.4. The comet's state at aphelion is initialised in the `run_simulation()` function. The iteration through a `while` loop updates the position and velocity using the RK4 method at each step, with a fixed step of 0.02 of a year (≈ 7.3 days) and total runtime of 100 years. The trajectory is stored in lists for visualisation and shown in Fig.1. The Sun is fixed at the origin, permissible due to the mass difference between the Sun and the comet, meaning there is negligible gravitational attraction on the Sun.

To investigate the impact of time-step selection on the accuracy of Halley's Comet's orbit, multiple simulations were conducted using different fixed time-step sizes. The time-step intervals tested were $3.65, 7.30, 12.17$ and 18.25 days, with each simulation running for 140 years. The result from this is shown in Fig.2, ensuring the Sun is once again fixed at the origin.

As expected, smaller time steps, such as 3.65 days, produce more accurate and smooth trajectories, closely resembling the known highly elliptical orbit of Halley's Comet. In contrast, larger time steps introduce increasing numerical errors, causing the orbit to deviate from the expected path due to accumulated integration inaccuracies. This demonstrates the trade-off between computational efficiency and accuracy; while larger time steps reduce the number of calculations and speed up simulations, they compromise precision. Conversely, smaller time steps capture the orbit with higher reliability but require more computational resources. The code for this is provided in the appendix.

Improving the RK4 Model

3.1 Adaptive Theory

In celestial mechanics, objects accelerate near perihelion and slow down near aphelion. A fixed-step integrator, as described above and demonstrated in Fig.1, fails to capture rapid changes in velocity near perihelion unless the step size is very small and wastes computation at aphelion by using unnecessarily small time steps. An adaptive method addresses this by automatically reducing the

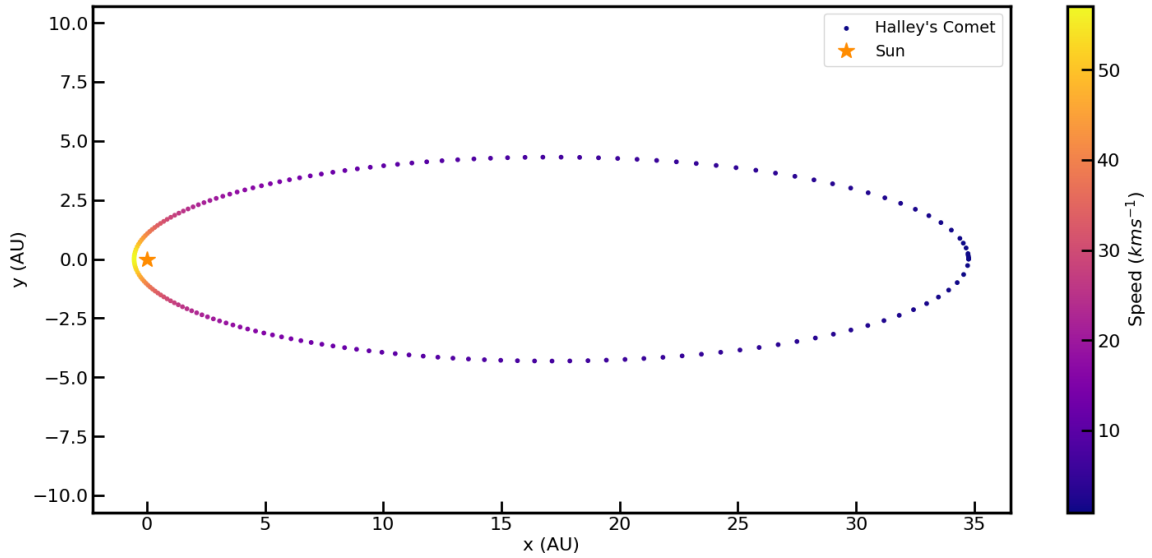

Figure 3: Simulated orbit of Halley's Comet around the Sun using an adaptive RK4 integration method. The colour gradient represents the speed of the comet, with higher velocities near perihelion and lower velocities near aphelion. The Sun is fixed at the origin. The simulation was run for 76 years, with an initial time step of 3.65 days and an adaptive tolerance of 1000 m/s. The adaptive step size is dynamically adjusted to maintain accuracy, reducing computational cost while capturing rapid velocity changes near the perihelion. Total integration steps were 174 with a computation time of 0.02 s.

step-size, dt, when changes are rapid and increases the step-size when motion is slow.

The adaptive RK4 method estimates the truncation error by computing two solutions, f_{full} using a full time step and f_{half} using two half-steps. The difference between these two values is called the error,

$$\text{Error} = \frac{||f_{\text{full}} - f_{\text{half}}||}{30},$$

where the factor of 30 is a correction constant derived from the error estimation properties of the RK4 scheme [3]. This factor scales the difference to provide an appropriate measure of the local truncation error. If this estimated error exceeds the preset tolerance, ϵ, the time-step is reduced by a factor of typically a half and then recalculated. If the error is below the threshold, the step size is increased according to the relation

$$dt_{\text{new}} = dt \times \left(\frac{\epsilon}{\text{Error}}\right)^{\beta},$$

where β is the control parameter, typically 0.2. This is chosen as a standard control parameter for stability and efficiency.

3.2 Implementation and Results

The code is structured into two classes, a `Comet` class defining the object representing the comet and a `RunSimulation` class handling the numerical integration and adaptive RK4 method.

The comet class initialises the comet's position, velocity, and mass while storing its trajectory, energy values and adaptive time step adjustments. The `compute_energy` function tracks kinetic and potential energy over time, which is crucial for analysing the system's energy conservation.

The `Runsimulation` class executes the numerical integration using an adaptive RK4 method. The function `rk4_step` computes the comet's position at the next time step using four intermediate estimates, improving accuracy over simpler integration techniques. Instead of using a fixed time step, the function `adaptive_step` dynamically adjusts the step size to balance accuracy and efficiency. It does this by comparing a full RK4 step with two half RK4 steps, estimating the numerical error. If

the estimated error is large, the step size is reduced to maintain precision, which is essential near perihelion, where velocity changes rapidly. Conversely, when the comet moves slowly at aphelion, the step size increases to reduce computational cost while preserving accuracy. The addition of the small constant, $1e-10$, is used to avoid the division by zero, preventing a runtime error.

```
if err > self.rel_tol:
    self.dt *= 0.5  # Reduce step size if error is too large continue
else:
    scale = (self.rel_tol / (err + 1e-10)) ** 0.2
    dt_new = self.dt * np.clip(scale, 0.5, 2.0)
    self.comet.f_vec, _, _ = self.step_method(f_orig, dt_new)  # Unpacking
    self.comet.history.append(self.comet.f_vec.copy())
    self.comet.compute_energy()
    return dt_new
```

The `Run` method executes the simulation loop, progressively updating the comet's position, velocity, energy, and time step values while tracking computational steps and simulation time. The `plot_trajectory_speed` function visualises the orbital motion of the comet, as in Fig.3, with colour-coded velocity values, confirming that the adaptive time-step effectively captures both high and low-speed regions. The number of steps required for a full orbit is significantly reduced (271 steps) compared to a fixed-step RK4 approach (3800 steps), demonstrating the efficiency and accuracy of the adaptive integration method for the same 76-year cycle.

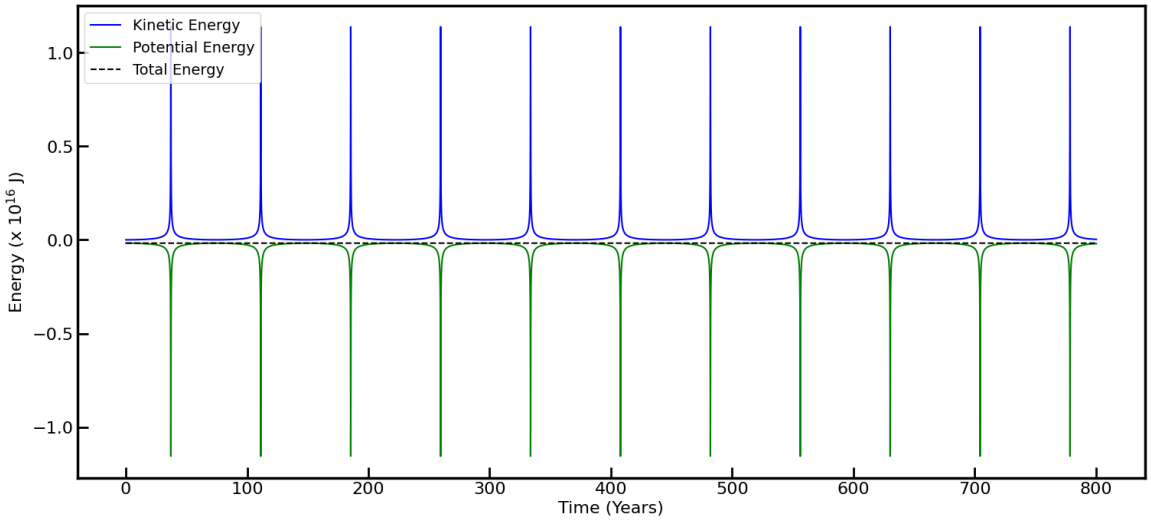

Figure 4: Energy conservation in Halley's Comet's orbit over 500 years, simulated using an initial time step of 3.65 days with an adaptive Runge-Kutta method. The kinetic energy (blue) peaks sharply at perihelion, where the comet moves fastest, while the potential energy (green) is minimised due to the strong gravitational interaction with the Sun. The total energy (dashed black) remains nearly constant, confirming numerical stability and conservation of energy within the adaptive time-stepping scheme.

The `compute_energy` function can be used to produce the plot as shown in Fig.4. The shape of the plot is influenced by the highly elliptical orbit of the comet, with sharp kinetic energy peaks at perihelion, while the potential energy drops to minima at this point. The total energy, E, serves as a metric for validating the numerical accuracy of the simulation. Ideally, E should remain constant, with minor deviations indicating numerical drift, occurring due to finite floating-point precision or step-size variation in the integration. Table 1 shows the energy rate loss from the system and compares to alternative methods explored later.

The evolution of the adaptive time step is shown in Fig.5. The time step reaches its smallest values at perihelion, ensuring precision in this fast-moving region. Fluctuations in each perihelion-aphelion cycle arise from error estimates inherent to the adaptive RK4 model.

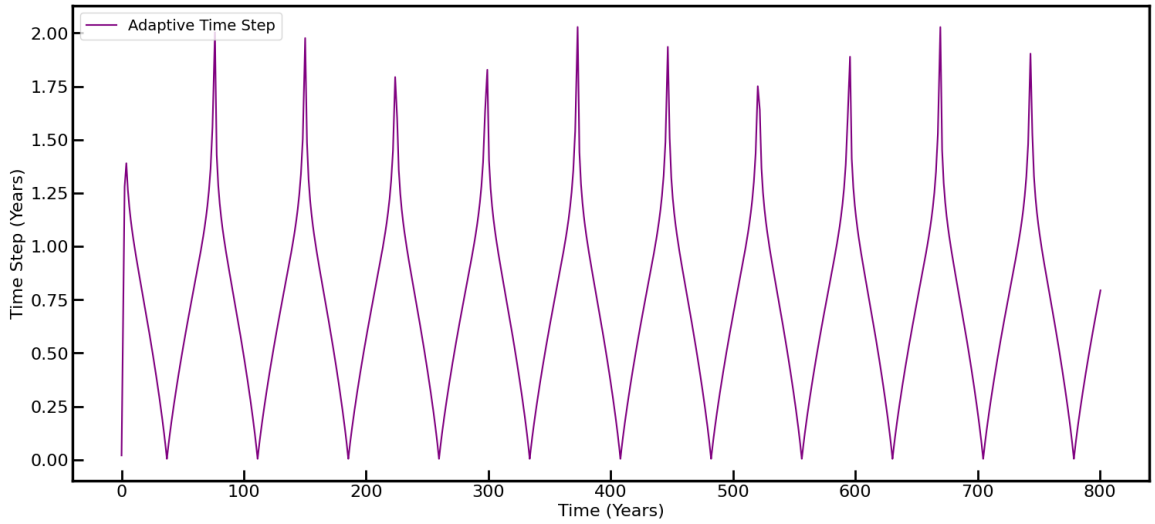

Figure 5: Adaptive time step evolution over the course of Halley's Comet's orbit, simulated using an initial time step of ≈ 3.65 days and a total simulation time of 500 years. The time step (dt) varies periodically, decreasing significantly at perihelion, where the comet's velocity is highest, and increasing at aphelion, where motion is slower. This adaptive approach, implemented with a relative tolerance of 100, ensures numerical accuracy while optimising computational efficiency.

3.3 RKF45 Method

Over longer timescales, small numerical drifts in energy and time, observed above, may accumulate when using non-symplectic integration methods. The Runge-Kutta Fehlberg, RKF45, method is one improvement that can be made. The RKF45 method used two embedded Runge-Kutta formulas, a fourth-order (RK4) estimate and a fifth-order estimate. The difference between these gives a more direct and precise error estimate.

The RKF45 method has been implemented as an extension of the `Runsimulation` class, creating a `RunsimulationRKF45` subclass. This subclass retains all functionality from the base class, but instead, the RK4 method is overridden. The program assigns `self.rkf45_step` as the new Runge-Kutta function. The constructor ensures that the simulation still initialises with the `Comet` class. The RKF45 method calculates six intermediate slopes using the Fehlberg coefficients [5]. Each slope represents a function evaluation at different points along the trajectory, improving accuracy. The fourth-order solution, f_4, is calculated using the weighted sum of slopes, and the fifth-order, f_5, is calculated separately using different coefficients and an extra slope, k_6. The code can be used in the main python file, by un-commenting the RKF45 method.

```
k1 = dt * self.comet.acceleration(f)
k2 = dt * self.comet.acceleration(f + k1 * 1/4)
k3 = dt * self.comet.acceleration(f + k1 * 3/32 + k2 * 9/32)
k4 = dt * self.comet.acceleration(f + k1 * 1932/2197 - k2 * 7200/2197 +
    k3 * 7296/2197)
k5 = dt * self.comet.acceleration(f + k1 * 439/216 - k2 * 8 + k3 * 3680/513 -
    k4 * 845/4104)
k6 = dt * self.comet.acceleration(f - k1 * 8/27 + k2 * 2 - k3 * 3544/2565 +
    k4 * 1859/4104 - k5 * 11/40)

f4 = f + k1 * 25/216 + k3 * 1408/2565 + k4 * 2197/4104 - k5 * 1/5
f5 = f + k1 * 16/135 + k3 * 6656/12825 + k4 * 28561/56430 - k5 * 9/50 + k6 * 2/55
```

The six intermediate slopes are calculated, by evaluating the acceleration at different estimated states of the system. The results of the comparison in computational efficiency and accuracy is presented in Table 1. The code for the RKF45 method is provided in the appendix under the relevant section.

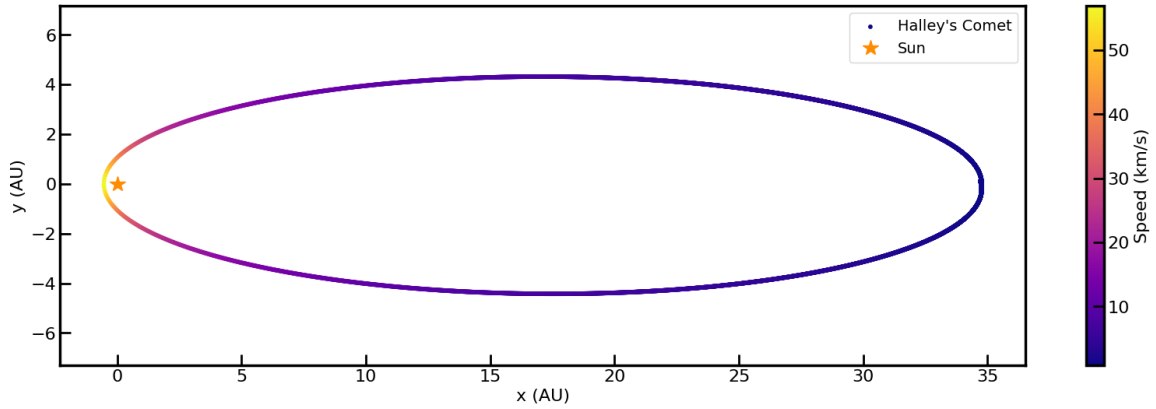

Figure 6: An elliptical orbit of Halley's Comet around the Sun, simulated using the velocity Verlet method. The colour scale represents the comet's speed (in km/s), with cooler colours (purple/blue) indicating lower speeds near aphelion and warmer colours (yellow/orange) indicating higher speeds near perihelion. The star (Sun) is shown at the focus of the ellipse. The simulation was performed with a fixed step of 3.65 days and a total time of 76 years.

3.4 Velocity - Verlet Method

The final method implemented for comparison is the Verlet Integration Method, specifically the velocity-Verlet variant. The Verlet method, as implemented in the `RunVerlet` class, was originally introduced by L. Verlet in his seminal 1967 paper [6] and is also well described in standard texts such as Allen and Tildesley [7]. The `RunVerlet` class inherits from the `RunSimulation` class. The tolerance set to 'None', as the Verlet method does not use a tolerance to adapt steps but instead uses a fixed step size over the duration of the simulation. The `verlet_step` function begins by extracting the position (x, y) and velocity (v_x, v_y) components from the current state vector. The method then computes the current acceleration using the `acceleration(f)` function.

The method updates the positions using the standard velocity-Verlet formula.

```
# Update positions
x_new = x + vx * dt + 0.5 * a_current[0] * dt**2
y_new = y + vy * dt + 0.5 * a_current[1] * dt**2
```

A temporary state `f_temp` is created at the new position to compute updated acceleration. The velocity update can then be computed as,

```
# Temporary state to compute new acceleration
f_temp = np.array([x_new, y_new, vx, vy], dtype=np.float64)
a_next = self.comet.acceleration(f_temp)[2:]  # Acceleration at the new position

# Update velocities
vx_new = vx + 0.5 * (a_current[0] + a_next[0]) * dt
vy_new = vy + 0.5 * (a_current[1] + a_next[1]) * dt
```

The velocity is updated using the average acceleration from the old and new positions. The correction ensures the velocity updates remain second-order accurate. Finally, the state vector is returned in the same structure as used by the RK method, with placeholders None for error estimates. Since the Verlet method does not inherently provide an error estimate for adaptive time stepping, the `adaptive_step` method is overridden in the `RunVerlet` class to simply perform the integration with a fixed time step. This simplicity leads to lower computational cost per step, although it may require a large number of steps for long simulations. The results for computational efficiency is shown in Table 1, with the full code shown in the appendix.

The comparative data shown in Table 1 demonstrates that while RK4 and RKF45 typically require fewer steps, due to their adaptive time stepping, the Verlet method uses many more steps in short simulations but can preserve energy effectively over longer periods. Computation times generally increase with the number of steps; Adaptive RK4 tends to be efficient for moderate runs, whereas RKF45 can incur higher overhead. Meanwhile, the Verlet method, despite its large step count in

Table 1: Comparison of numerical integration methods Adaptive RK4 (ARK4), RKF45, and Verlet for orbital simulations over different time spans. The table presents computation time, the number of integration steps taken, and the energy dissipation rate for each method. The step size is set to 100, and the tolerance is set to 100 for the RK methods.

Method	**Simulation Length (Years)**	**Computation Time (s)**	**Steps Taken**	**Energy Dissipation Rate (J/year)**
ARK4	100	0.03	296	1.4717
RKF45	100	0.06	187	3.8624
Verlet	100	0.10	10,000	4.8961
ARK4	1,000	0.31	3,443	1.0894
RKF45	1,000	0.68	2,139	3.4072
Verlet	1,000	0.78	100,000	0.0531
ARK4	10,000	3.02	34,947	6.343
RKF45	10,000	6.57	21,557	3.3892
Verlet	10,000	8.20	1,000,000	0.0005

shorter runs, demonstrates strong energy conservation over very long simulations. Consequently, the choice of method depends on the balance between computational cost (time and steps) and long-term accuracy (energy dissipation rate). The remainder of the questions focused on the use of the adaptive RK4 (ARK4) due to the balance it offers.

Three-Body Problem Around a Fixed Mass

The three-body problem is a fundamental challenge in celestial mechanics, where three masses interact gravitationally according to Newton's laws. Unlike the two-body problem, the three-body problem lacks a general closed-form solution [2]. In this section, the orbits of two planets around a central star, mass M, are simulated using the adaptive RK4 method. The two planets have mass ratios of $m_1/M = 10^{-3}$ and $m_2/M = 4 \times 10^{-2}$, where M is the Mass of the Sun. Initial circular orbits begin at $a_1 = 2.52$ AU and $a_2 = 5.24$ AU, respectively.

The `Planet` class is a subclass of `Comet` class, meaning it inherits properties previously defined. The constructor initialises each of the planets with a semi-major axis, a, mass, m, and an initial velocity derived from the circular orbit condition,

$$v = \sqrt{\frac{GM}{a}}.$$

```
class Planet(Comet):
    def __init__(self, name, a, mass):
        # Inherit from Comet and add semi-major axis
        super().__init__(name, [a, 0.0], [0.0, np.sqrt(G * M_sun / a)], mass)
```

This ensures that, in the absence of perturbations, each planet follows a stable circular orbit.

The `SolarSystemClass` class is responsible for running the numerical integration, extended to multiple bodies from the `Runsimulation` class. At each iteration, the system updates each planet independently, using the minimum time step amount for all planets to synchronise integration. This ensures that fast-moving planets do not introduce numerical instability by forcing all planets to take smaller time steps than necessary.

```
dt_list = []  # Collect time steps for each planet

for planet in self.planets:
    dt_list.append(self.adaptive_step(planet))  # Compute dt for each planet

self.dt = min(dt_list)  # Use the smallest dt among all planets to synchronise the simulation
current_t += self.dt
```

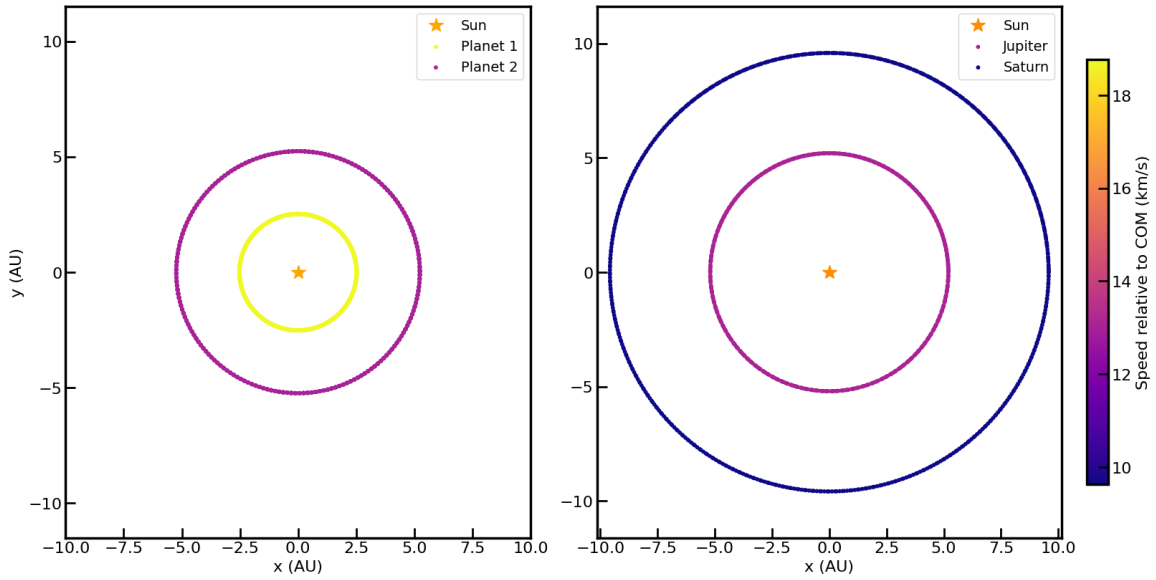

Figure 7: Side-by-side comparison of two planetary system simulations using the adaptive RK4 integration method. The left panel displays the orbits of two hypothetical planets, Planet 1 (closer in) and Planet 2 (further out), with circular orbits at 2.52 AU and 5.24 AU around a central Sun (orange). The right panel presents the orbits of Jupiter (red) and Saturn (purple), which are also modelled with circular initial conditions at their respective semi-major axes of 5.2 AU and 9.58 AU. The simulations accurately reproduce stable planetary orbits, demonstrating the effectiveness of the numerical method in modelling gravitational dynamics.

The results are shown in Fig.7, demonstrating that the planets follow a circular path around the central star. In addition, Planet 2 follows a Jupiter-like path around the star at the centre of the plot, fixed at the origin as previously done.

System Centre of Mass Transformation

To extend the numerical model and improve its applicability to multi-body problems, the coordinate system can be transformed into the centre-of-mass (CoM) frame. This transformation ensures that the motion is analysed relative to the system's overall CoM, providing a more stable reference frame for studying interactions between celestial bodies.

In a system of N bodies with masses m_i, the position of the CoM, $\vec{r}_{cm}(t)$, is given by,

$$\vec{r}_{cm}(t) = \frac{\sum_i m_i \vec{r}_i(t)}{M_{\text{total}}}$$

where M_{total} is the total mass of the system. Similarly, the velocity of the CoM, $\vec{v}_{cm}(t)$, is given by,

$$\vec{v}_{cm}(t) = \frac{\sum_i m_i \vec{v}_i(t)}{M_{\text{total}}}. \tag{5}$$

To shift into the CoM frame, the positions and velocities of all objects in the system are transformed as follows,

$$\vec{r}_i'(t) = \vec{r}_i(t) - \vec{r}_{cm}(t), \qquad \vec{v}_i'(t) = \vec{v}_i(t) - \vec{v}_{cm}(t). \tag{6}$$

This ensures that the CoM remains at the origin of the coordinate system throughout the simulation, reducing numerical drift and maintaining consistency when simulating multi-body interactions [8].

In the program, each celestial body is represented by the `CelestialBody` class, which stores its position, velocity, mass, energy, and a history of these values. The main `MultiBodySimulation` class runs the dynamics using the adaptive RK4 integrator. At initialisation, all bodies are shifted into the CoM frame so that the central reference is the CoM rather than an arbitrarily fixed star.

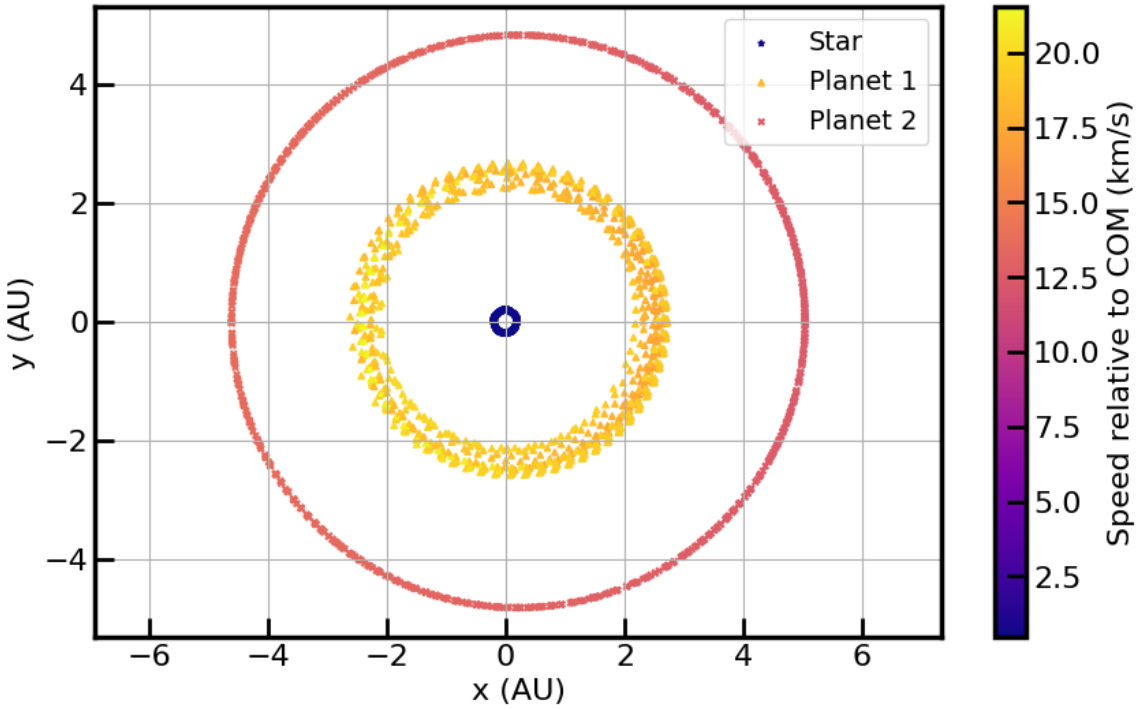

Figure 8: Orbits of the three bodies, a central star (blue circle near origin), Planet 1 (yellow triangles) and Planet 2 (red squares) in the centre of mass frame. The simulation was performed with the adaptive RK4 method, with an initial step of 3.65 days, a final time of 50 years and an error tolerance of 1000 m/s. As expected, the star in the central orbits the CoM, albeit very slowly.

This approach ensures that even a massive star may exhibit a slight orbital motion around the CoM if it is influenced by other bodies.

```
def compute_centre_of_mass(self):
    total_mass = sum(body.mass for body in self.bodies)
    pos_cm = np.zeros(2)
    vel_cm = np.zeros(2)

    for body in self.bodies:
        pos_cm += body.mass * body.state_vec[:2]
        vel_cm += body.mass * body.state_vec[2:]

    return pos_cm / total_mass, vel_cm / total_mass
```

In the code, `body.state_vec[:2]` extracts the x and y coordinates of position and `body.state_vec[2:]` extracts the velocity. The sum of these, weighted by each body's mass, is normalised by the total mass to obtain the CoM position.

Once the CoM is calculated, the `shift_to_centre_of_mass` method adjusts each body's state so that the CoM becomes the origin. This is done by subtracting the CoM position from the bodies position and the CoM velocity from each body's velocity.

```
def shift_to_centre_of_mass(self):
    r_cm, v_cm = self.compute_centre_of_mass()
    for body in self.bodies:
        body.state_vec[:2] -= r_cm
        body.state_vec[2:] -= v_cm
        # Reset the history to reflect the shift
        body.state_history = [body.state_vec.copy()]
```

The plot in Fig.8 shows that bodies closer to the star orbit more quickly and that even the star itself may exhibit a small orbit around the CoM if influenced by sufficiently massive companions,

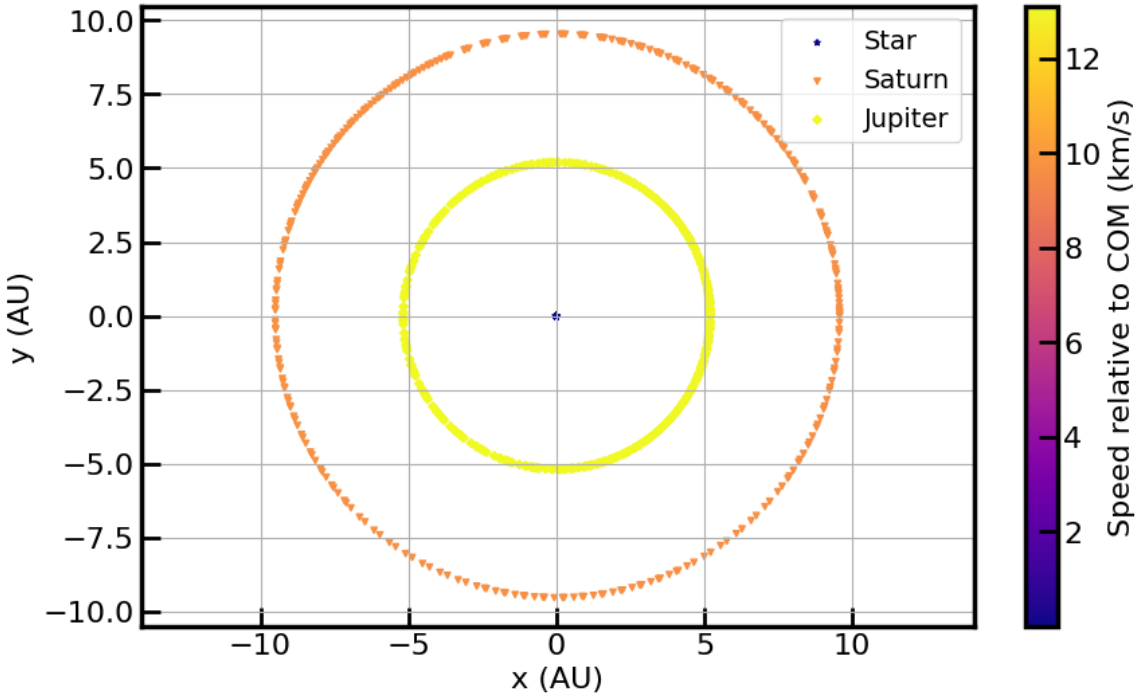

Figure 9: Orbits of the three bodies, a central star (blue circle near origin), Saturn (Orange triangles) and Jupiter (Yellow squares) in the centre of mass frame. The simulation was performed with the adaptive RK4 method, with an initial step of 3.65 days, a final time of 50 years and an error tolerance of 1000 m/s. As expected, the star in the central orbits the CoM, albeit very slowly.

reflecting physically consistent orbital motion. Conversely, the orbits of Jupiter and Saturn, as plotted in Fig.9 show the Sun has little to no wobble compared to the hypothetical planet setup. The simulation can be run for a longer period of time, yielding the result as shown in Fig.10, showing the long-term orbital dynamics of all bodies involved.

The CoM transformation is particularly useful when dealing with multiple gravitational interactions, such as the motion of exoplanets around binary stars [9]. By shifting to the CoM frame, the numerical model can be extended to study complex orbital dynamics with improved stability and reduced long-term drift, as shown in Fig.11. The setup shows two binary stars, orbiting around a common centre of mass. In addition to the stars, there is a small planet, of negligible mass, orbiting around star 1. The code for the setup of this system is displayed in the relevant section of the Appendix.

The successful implementation of this transformation in numerical simulations provides a foundation for investigating dynamical stability in multi-body celestial systems.

Conclusion

This report has addressed a number of applications of the Runge-Kutta (RK) methods for modelling dynamic orbits. The report has explored variations from the Runge-Kutta fixed time-step method, firstly implementing an adaptive step, the Runge-Kutta Fehlberg method, and then, finally an alternative through the Velocity-Verlet method. While each had their positives, the adaptive RK4 method proved its place when simulating orbital dynamics, reducing the number of steps and computing power required. Given more time, the report naturally leads to the modelling of other systems, such as black hole systems, which provide the opportunity to implement general relativity into the program design.

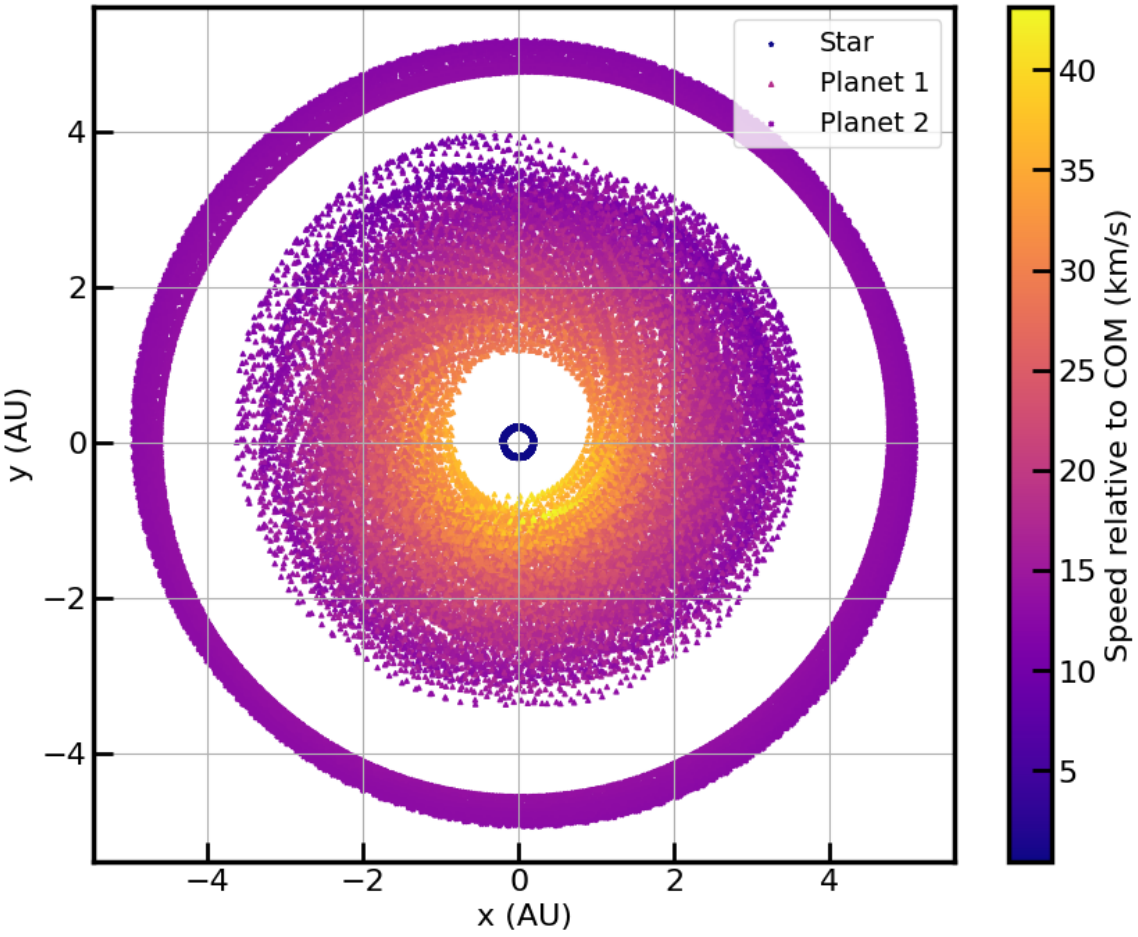

Figure 10: Orbital plot for planets 1 and 2 around the central star of one solar mass. The simulation was run over a period of 1000 years to visualise the variation in path experience by the innermost planet (planet 1).

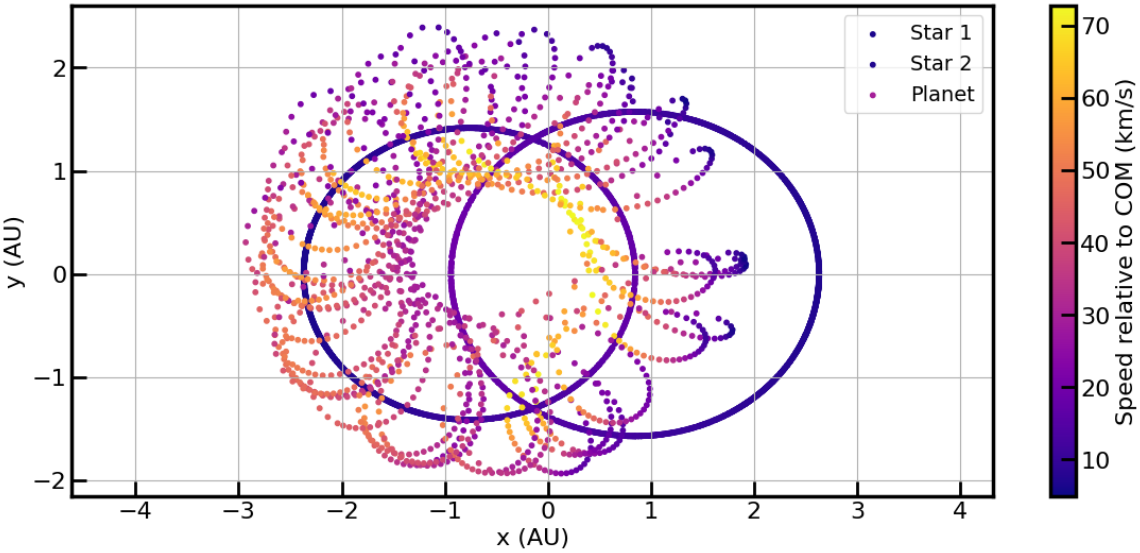

Figure 11: Binary star system with two stars orbiting their common centre of mass and a planet gravitationally bound near one of the stars. Each orbit is shown in the centre-of-mass frame, and colour-coding indicates the orbital speed relative to the CoM. Warmer colours (orange/yellow) correspond to higher velocities, while cooler colours (purple) indicate slower motion. The figure demonstrates how both stars circle their shared barycentre while the planet's path remains closely tied to the more massive star. Star 1 has a mass of 1 solar mass, while star 2 has 0.8 solar mass.

References

[1] J. M. A. Danby. *Fundamentals of Celestial Mechanics*. Willmann-Bell, Inc., 1988.

[2] M. Valtonen and H. Karttunen. *The Three-Body Problem*. Cambridge University Press, 2006.

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

[4] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer, 2006.

[5] E. Fehlberg. Low order classical runge-kutta formulas with step size control. Technical Report R-315, NASA, 1969. NASA Technical Report.

[6] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical Review*, 159(1):98–103, 1967.

[7] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 1987.

[8] C. D. Murray and S. F. Dermott. *Solar System Dynamics*. Cambridge University Press, Cambridge, UK, 1999.

[9] G. Laughlin, P. Bodenheimer, and F. C. Adams. The end of the main sequence. *The Astrophysical Journal*, 652:1187–1195, 2006.

Appendix

7.1 Time Step Stability Plot

The code below was used to produce Fig.2 along with the code presented in Q1a.

```
# Time steps and total simulation time
time_steps = [Year / 100, Year / 50, Year / 30, Year / 20]
end_t = 140 * Year  # Simulate for 140 years

fig, axs = plt.subplots(2, 2, figsize=(20, 12))
# Loop over time_steps
for i, dt in enumerate(time_steps):
    x, y = [], []
    current_t, steps_taken = 0.0, 0.0
    # Initial state: [x, y, vx, vy]
    f = np.array([r_aphelion, 0, 0, v_aphelion], dtype=float)
    while current_t < end_t:
        f = rk4_step(f, dt)  # Update state vector

        # Store results
        x.append(f[0])
        y.append(f[1])

        current_t += dt
        steps_taken += 1
    ax = axs[i // 2, i % 2]
    ax.plot(np.array(x) / AU, np.array(y) / AU, label=f"Interval = {(dt / 86400):.2f} days",
            color='blue', markersize=2, linestyle='None', marker='.')
    ax.plot(0, 0, '*', color='darkorange', markersize=20, label="Sun")
    # Formatting where axis labels are placed
    if i // 2 == 1:
        ax.set_xlabel('x (AU)', fontsize=18)
    else:
        ax.set_xlabel('')
    if i % 2 == 0:
```

```
        ax.set_ylabel('y (AU)', fontsize=18)
    else:
        ax.set_ylabel('')

    ax.axis('equal')
    ax.legend(loc='upper right')

plt.tight_layout()
plt.show()
```

7.2 RKF45 Method

The code below is an adaption of the adaptive RK4 method presented for question 1b. This inherits properties from the classes used in that, and thus should be used in conjunction.

```
# Comet class remains the same - only changing the method used
class RunsimulationRKF45(Runsimulation): # Inherit parameters from above
    def __init__(self, comet, dt_init, end_time, error_tolerance):
        super().__init__(comet, dt_init, end_time, error_tolerance, step_method =
            self.rkf45_step)

    #Define the new method - RKF45
    def rkf45_step(self, f, dt):
        # Computes extra terms and uses Fehlberg coefficients
        k1 = dt * self.comet.acceleration(f)
        k2 = dt * self.comet.acceleration(f + k1 * 1/4)
        k3 = dt * self.comet.acceleration(f + k1 * 3/32 + k2 * 9/32)
        k4 = dt * self.comet.acceleration(f + k1 * 1932/2197 -
            k2 * 7200/2197 + k3 * 7296/2197)
        k5 = dt * self.comet.acceleration(f + k1 * 439/216 -
            k2 * 8 + k3 * 3680/513 - k4 * 845/4104)
        k6 = dt * self.comet.acceleration(f - k1 * 8/27 +
            k2 * 2 - k3 * 3544/2565 + k4 * 1859/4104 - k5 * 11/40)
        # Compute 4th and 5th-order estimates
        f4 = f + k1 * 25/216 + k3 * 1408/2565 + k4 * 2197/4104 - k5 * 1/5
        f5 = f + k1 * 16/135 + k3 * 6656/12825 + k4 * 28561/56430 - k5 * 9/50 + k6 * 2/55
        # Compute error estimate
        error = np.linalg.norm(f5 - f4)

        return f4, f5, error

# Run RKF45 without redefining anything
comet = Comet(name = "Halley",
              initial_pos = [r_aphelion, 0],
              initial_vel = [0, v_aphelion],
              mass=2.2e14)

sim_rkf45 = RunsimulationRKF45(comet,
                               dt_init = Year / 100,
                               end_time = Year * 80,
                               error_tolerance = 1000) # m/s

sim_rkf45.run_simulation()        # Uses RKF45 above
sim_rkf45.plot_trajectory_speed() # Plot speed of comet
sim_rkf45.plot_energy()           # Plot energy of comet
```

7.3 Velocity-Verlet Method

```
# Comet class remains the same - only changing the method used
class RunVerlet(Runsimulation): # Inherit parameters from adaptive RK4 method
    def __init__(self, comet, dt_init, end_time):
```

```
        # Verlet method does not need a tolerance so can be set to None
        super().__init__(comet, dt_init, end_time, error_tolerance = None,
            step_method=self.verlet_step)

    # Define verlet method
    def verlet_step(self, f, dt):
        x, y, vx, vy = f
        # Compute acceleration at current position
        a_current = self.comet.acceleration(f)[2:]  # [ax, ay]
        # Update positions and temporaily store them
        x_new = x + vx * dt + 0.5 * a_current[0] * dt**2
        y_new = y + vy * dt + 0.5 * a_current[1] * dt**2
        f_temp = np.array([x_new, y_new, vx, vy], dtype=np.float64)
        # Acceleration at the new position
        a_next = self.comet.acceleration(f_temp)[2:]
        # Update velocities
        vx_new = vx + 0.5 * (a_current[0] + a_next[0]) * dt
        vy_new = vy + 0.5 * (a_current[1] + a_next[1]) * dt

        return np.array([x_new, y_new, vx_new, vy_new], dtype=np.float64), None, None

    # Need to adapt the adaptive_step() to not use error estimate
    def adaptive_step(self, body):
        f_new, _, _ = self.step_method(self.comet.f_vec, self.dt)
        self.comet.f_vec = f_new
        self.comet.history.append(f_new.copy())
        self.comet.compute_energy()
        # Fixed step size, so dt remains unchanged
        return self.dt

# Create Comet and Simulation
comet = Comet("Halley",
              [r_aphelion, 0],
              [0, v_aphelion],
              mass = 2.2e14)

sim = RunVerlet(comet,
                dt_init=Year / 100,
                end_time=Year * 76)

sim.run_simulation()          # Uses Verlet integration
sim.plot_trajectory_speed()   # Plot orbit coloured by speed
sim.plot_energy()             # Plot energy evolution
```

7.4 Binary Star Setup

The code can be added onto that of Q2b, and used to demonstrate a binary star setup, as shown in Fig.11

```
M_star1 = M_sun
M_star2 = 0.9 * M_sun
a_binary = 5 * AU  # Separation between stars

# Compute distances from centre of mass
r_star1 = (M_star2 / (M_star1 + M_star2)) * a_binary
r_star2 = (M_star1 / (M_star1 + M_star2)) * a_binary

# Compute orbital velocities for a stable binary system
v_star1 = np.sqrt(G * M_star2 / a_binary) * (r_star2 / a_binary)
v_star2 = np.sqrt(G * M_star1 / a_binary) * (r_star1 / a_binary)
```

```
# Place stars symmetrically around the centre of mass
star1 = CelestialBody(name = "Star 1",
                      mass = M_sun,
                      initial_pos = [-r_star1, 0],
                      initial_vel = [0, v_star1])  # Counterclockwise

star2 = CelestialBody(name = "Star 2",
                      mass = 0.9 * M_sun,
                      initial_pos = [r_star2, 0],
                      initial_vel = [0, -v_star2])  # Clockwise

# Define a planet orbiting one of the stars
planet = CelestialBody(name = "Planet",
                       mass = 1e-20 * M_sun,
                       initial_pos = [-r_star1 + 1 * AU, 0],
                       initial_vel = [0, np.sqrt(G * M_star1 / (1 * AU))])

# Create the binary star system
bodies = [star1, star2, planet]

# Run the simulation
sim = MultiBodySimulation(bodies, dt_init=Year/100, end_time=20*Year, error_tolerance=1000)
sim.run_simulation()

# Plot the system
sim.plot_trajectory_speed()
```

End of Report