James Kavanagh-Cranston
40254673
02.11.2021

Using a Numerical Model of the Solar System in the Investigation of Kirkwood Gaps.

Abstract

Utilizing the Runge-Kutta method to simulate motions of solar-system bodies, it was calculated that a body orbiting at radius 0.58692 AU has an orbital period of 164.28 days. N-body calculations were used to investigate the effect of the planets' motions on the Sun and the effect Jupiter has on bodies in the main asteroid belt. It was seen that bodies in Kirkwood gaps had unstable orbits with irregular kinetic energy – time graphs and as a result, would deviate from these orbits, leaving behind gaps in the belt.

Introduction

The objective of this program is to investigate Kirkwood gaps and the kinetic energy of asteroids with circular orbits located in these gaps. In order to simulate the motions of the planets and asteroids, a numerical model must be used.

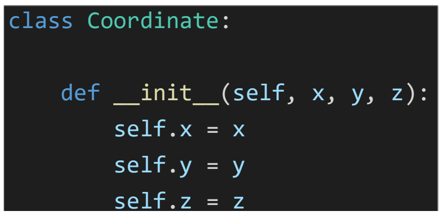
The Runge-Kutta approximation is a good method to use for this application as it is an efficient numerical model, allowing for a smaller time-step than Euler and Euler-Cromer methods. This results in faster computation times, meaning longer simulations can be run quickly which is required in the investigation of Kirkwood gaps.

Kirkwood gaps are regions in the main asteroid belt in which very few asteroids can be found. Asteroids in these orbits are in orbital resonance with Jupiter, for example, for an asteroid in a circular orbit of radius 2.5AU, the ratio between the asteroid's orbit and Jupiter's orbit is 3:1[1]. This means that once every three orbits of the asteroid, the relative positions of Jupiter and the asteroid are the same. As the asteroid has repeated encounters with Jupiter at the same point in its orbit, it eventually gets pulled out of the circular 2.5AU orbit, leaving behind a gap in the main asteroid belt.

To allow for comprehensive investigation, the program should be able to calculate orbits of bodies with consideration of the gravitational effects of each of the bodies in the system. These N-body calculations allow the investigation of Kirkwood gaps caused by Jupiter; the most massive planet in our solar system. The program should also include the ability to allow the Sun to move as to investigate the effects of the orbiting bodies on the Sun.

Method

The program written for this project is object oriented, allowing for simplistic variable organisation and expandability when compared to procedural programming techniques. An interesting use of classes implemented in this program is the `Coordinate` class seen in **Code 1**. This class groups the x, y and z components of a position or velocity vector in one object which can be acquired in a simple manner.



```
class Coordinate:

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

[Code.1] Class used to organise position & velocity data.

Utilizing the Runge-Kutta method in this program requires a very specific order of operations to allow N-body calculations. **Code 2** shows the `rungeKutta` method inside the `Simulation`

```
def rungeKutta(self, t, dt, stationarySun):
    a = dt * calcDiffEqs(t, self.joinedVec, self.massList, stationarySun)
    b = dt * calcDiffEqs(t + 0.5*dt, self.joinedVec + a/2, self.massList, stationarySun)
    c = dt * calcDiffEqs(t + 0.5*dt, self.joinedVec + b/2, self.massList, stationarySun)
    d = dt * calcDiffEqs(t + dt, self.joinedVec + b, self.massList, stationarySun)

        yNew = self.joinedVec + ((a + 2*b + 2*c + d) / 6.0)
        return yNew
```

[Code.2] Runge-Kutta method in the Simulation class which can be used to calculate the positions and velocities of the bodies at a timestep dt forward in time.

class. This method is the main code running the Runge-Kutta mathematics and calls the `calcDiffEqs` function seen in **Code 3**.

```
def calcDiffEqs(t, y, masses, stationarySun=True):
    G = 6.67e-11 #m^3 kg^-1 s^-2
    N_bodies = int(len(y) / 6)
    solvedVector = np.zeros(y.size)

    if stationarySun == True:
        start = 1
    elif stationarySun == False:
        start = 0

    for i in range(start, N_bodies):
        ioffset = i * 6
        for j in range(N_bodies):
            joffset = j * 6
            solvedVector[ioffset] = y[ioffset + 3]
            solvedVector[ioffset + 1] = y[ioffset + 4]
            solvedVector[ioffset + 2] = y[ioffset + 5]
            if i!= j:
                xrad = y[ioffset] - y[joffset]
                yrad = y[ioffset + 1] - y[joffset + 1]
                zrad = y[ioffset + 2] - y[joffset + 2]
                r = sqrt(xrad**2 + yrad**2 + zrad**2)
                ax = (-G * masses[j] / r**3) * xrad
                ay = (-G * masses[j] / r**3) * yrad
                az = (-G * masses[j] / r**3) * zrad
                solvedVector[ioffset + 3] += ax
                solvedVector[ioffset + 4] += ay
                solvedVector[ioffset + 5] += az

    return solvedVector
```

*[Code.3] Function which calculates the acceleration and therefore the velocity and position of a body, i, as a result of gravitational forces exerted by bodies, j returning a numpy array of the format: **[body1x, body1y, body1z, body1vx, body1vy, body1vz, body2x, body2y, body2z, body2vx, body2vy, body2vz, …]**.*

The following describes the mathematics behind the Runge-Kutta method used in these sections of code:

Let an initial value problem be specified as follows:

$$\frac{dy}{dx} = f(t, y), \qquad y(t_0) = y_0$$

y is an unknown function of time, t, which we would like to approximate.

Function f and initial conditions t_0 and y_0 are given.

Select a time-step, $dt > 0$

$$y_{n+1} = y_n + \frac{\delta t}{6}(a_i + 2b_i + 2c_i + d_i)$$

$$t_{n+1} = t_n + \delta t$$

For $n = 0, 1, 2, 3, 4, \dots$ where,

$$a = f_i(t_n, y_n)$$

$$b = f\left(t_n + \frac{\delta t}{2}, y_n + \delta t\frac{a}{2}\right)$$

$$c = f\left(t_n + \frac{\delta t}{2}, y_n + \delta t\frac{b}{2}\right)$$

$$d = f(t_n + \delta t, y_n + \delta t c)$$

For the case of calculating orbits,

$$f(t, y) = \frac{d^2 x_i}{dt^2} = \sum_{j=1, j\neq i}^{n} \frac{G m_j (x_j - x_i)}{r_{ij}^3}$$

Where $f(t, y)$ is calculating the acceleration of a body i with respect to bodies j.

It is noted that the above summation does not occur if $i = j$ as this would be suggesting that the body, i, causes itself to accelerate as a result of its own gravitational forces. Furthermore, it is necessary that the calculations are carried out in a specific order: each body must be advanced in its orbit by the appropriate time-step in order to calculate the force exerted on the body in question at a particular time-step.

Code 4 is the section of code responsible for calculating the motions of the bodies in the system. The `rungeKutta` method and `calcDiffEqus` function are controlled by the `runOrbits` method in the `Simulation` class. This method takes a time-step, `dt`, time period, `T`, a reporting frequency, `reportFreq`, and a Boolean variable stating if the Sun is allowed to move, `stationarySun`.

```python
def runOrbits(self, dt, stationarySun, T, reportFreq):
    self.path = [self.joinedVec]
    clock_time = 0
    nsteps = int(T / dt)

    for step in range(nsteps):
        yNew = self.rungeKutta(0, dt, stationarySun)

        if step % reportFreq == 0:
            self.path.append(yNew)
            self.t.append(clock_time)

        self.joinedVec = yNew
        clock_time += dt

    self.path = np.array(self.path)
```

[Code.4] Method of the Simulation class which uses the rungeKutta method to create a numpy array, 'path' of the form: ***[body1x, body1y, body1z, body1vx, body1vy, body1vz, body2x, body2y, body2z, body2vx, body2vy, body2vz, …].***

The `stationarySun` variable is passed to the `calcDiffEqus` function through the `rungeKutta` method and modifies the starting index of the for loop iterating through i, seen in **Code 3**. This ignores the first body in the list passed to it through the variable `y`, therefore ignoring the Sun when calculating new position values.

The variable names here (`stationarySun`) are appropriate for this application, however, this program can handle any system of bodies, allowing for the investigation of orbits in any system, e.g., Jupiter and its moons or a binary star system.

The `reportFreq` variable is used to keep the amount of data saved to a minimum. This aides the plotting process as, without this, the matplotlib plots take a very long time to process and are slow when zooming in and moving the plots.

Using **Code 2-4**, it is possible to simulate the motions of an N-body system and a 2-body system; by only passing two bodies into the program, the `calcDiffEqus` function acts in the same manner as a program written exclusively for two bodies.

Code 4 can be modified to calculate the orbital period of a planet and can be seen in **Code 5**. In this method, the orbital period is found using the angle which the radius vector makes with the x-axis giving an angle ranging from $0 \rightarrow 2\pi$. This poses a challenge as, for all bodies orbiting the origin, at some point in its orbit, its angle will go from 2π to 0. This means that it is not possible to use a single conditioned while loop and two conditions must be used:

`currentAngle > prevAngle` (condition 1)

`currentAngle < initAngle` (condition 2)

The first condition advances the bodies position while the angle increases and the second condition advances the bodies position while the current angle is less than the initial angle. Regardless of a bodies initial position, the point at which the conditions must be switched is always when the body crosses the +ve x-axis.

On the final step forward, where condition 2 is no longer satisfied, it is almost always the case that there will be some 'over-shoot' where the body will be moved more than is required to complete the orbit. This over-shoot in time can be approximated using the following logic:

$$time\ overshoot = \delta t \times \frac{overShoot}{overShoot + underShoot}$$

where,

`overShoot = currentAngle - initAngle`

and,

`underShoot = (2 * pi) - prevAngle`

This method to remove the over-shooting inaccuracy assumes that over a small time-step, δt, the angular velocity is constant. This is a reasonable assumption to make as the orbital period, $T \gg \delta t$.

Results / Discussion

This program performed well and was able to simulate the paths of bodies quickly given the number of calculations required.

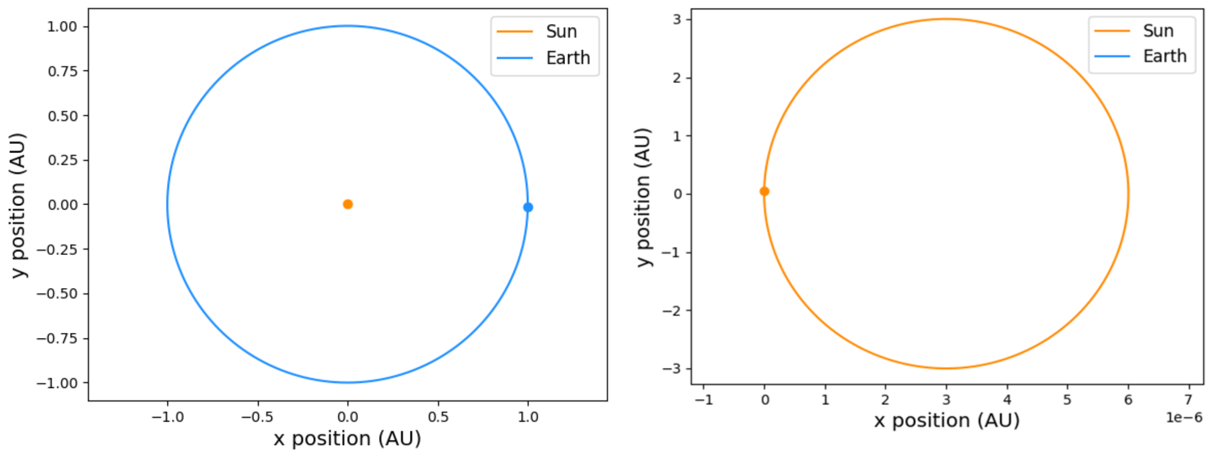
```python
def calcPeriod(self, dt, stationarySun=True, T=None):
    self.path = [self.joinedVec]
    clock_time = 0
    initAngle = self.bodies[1].angle()
    currentAngle = initAngle
    prevAngle = initAngle - 1
    step = 0
    plotx = []
    ploty = []
    passed = False

    while (currentAngle > prevAngle and not passed) or
                        (currentAngle < initAngle):
        prevAngle = currentAngle
        yNew = self.rungeKutta(0, dt, stationarySun)
        self.path.append(yNew)
        self.joinedVec = yNew
        currentAngle = angle(yNew[6], yNew[7])
        plotx.append(yNew[6])
        ploty.append(yNew[7])
        step += 1
        clock_time += dt

        if not (currentAngle > prevAngle) and not passed:
            passed = True

    self.path = np.array(self.path)

    overShoot = currentAngle - initAngle
    underShoot = (2 * pi) - prevAngle
    timeOverShoot = dt * (overShoot/(overShoot + underShoot))

    clock_time -= timeOverShoot

    if clock_time/(60*60*24) <= 365*1.5:
        return f"{clock_time/(60*60*24)} days"

    else:
        return f"{clock_time/(60*60*24*365)} years"
```

[Code.5] Method of the Simulation class to calculate the orbital period of the 1st body in the provided list (excluding the Sun).

Figure 1a shows the Earth's path around the Sun. This is an example of the program handling a two-body case and it can be seen from the plot that the Earth has made one complete orbit of the Sun during a period of 365 days.

For this simulation, the Sun was allowed to move. An insight into the effects of this can be seen in **Figure 1b**, which is a zoomed in plot from the same simulation as **Figure 1a**. Here, the Sun can be seen orbiting the centre of mass of the system, known as the barycentre, and completes a circular orbit. This phenomenon also occurs with an increased number of bodies, however, the path is more chaotic and does not trace out a perfect circle

The `calcPeriod` function can be tested on this two-body system and it is found that the Earth orbits the Sun with an orbital period of 365.35 days. This, and other accurate measurements



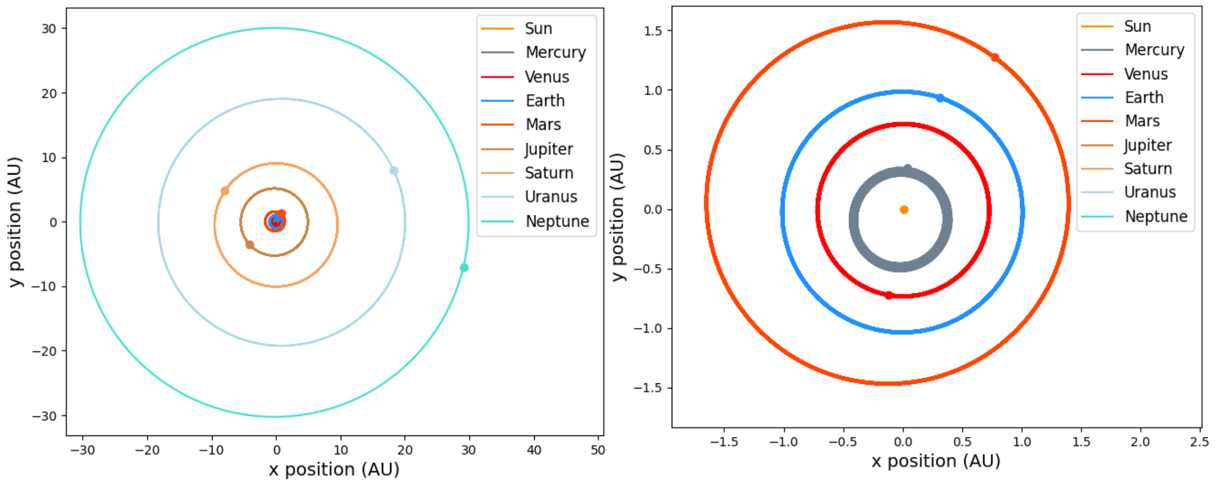
[Figure.1] (a) Plot of the Earths motion around the Sun. (b) Plot of the motion of the Sun in a two-body system consisting of the Earth and the Sun.

of different planets' orbital periods allows for confident measurement of unknown orbital periods of bodies.

The orbital period of a planet with a circular orbit about the Sun with radius,

$$r = \left(0.4 + \frac{4673}{25000}\right) AU = 0.58692\ AU$$

was calculated and found to be 164.28 days. Through these calculations, it was noted that the orbital period had a dependency on the chosen time-step, δt. This dependency, however, was very small and for the case of the planet mentioned above, when comparing a time-step of $\delta t =$ 24 hours and $\delta t = 1$ hour, the difference in the calculated orbital period was 5.2 min.

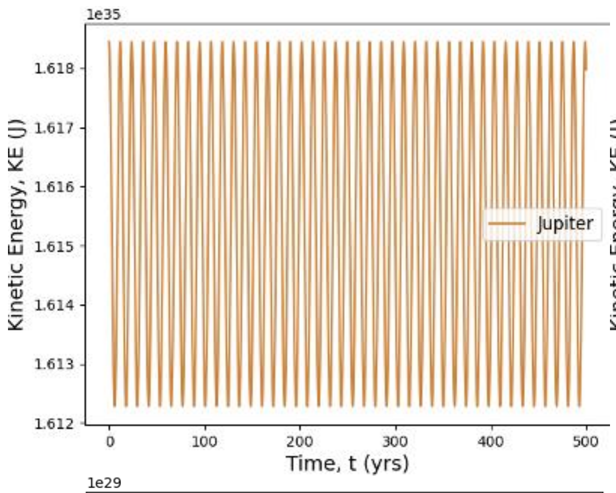


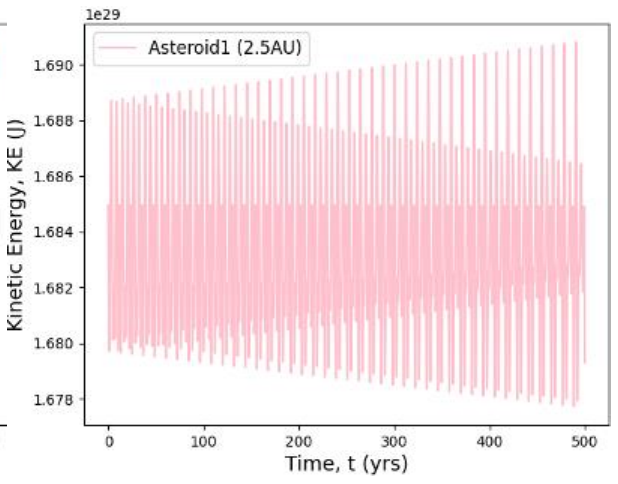
[Figure.2] (a) Plot of the planet's motion around the Sun. (b) Plot of the rocky planet's motion around the Sun.

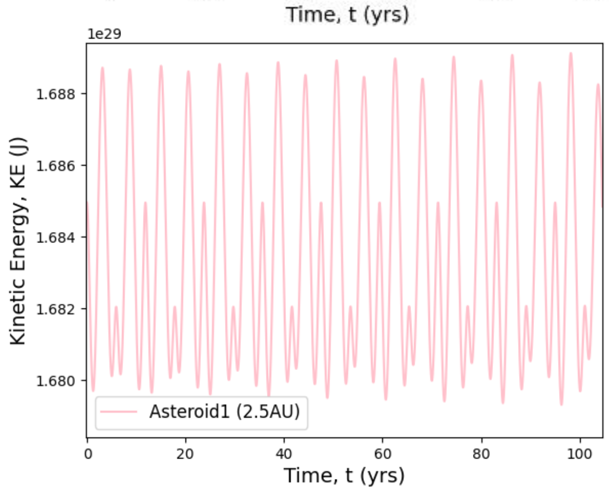
Figure 2a shows the entire solar system, simulated for a period of 165 years; the orbital period of Neptune as calculated by the program. A limitation of the program can be noted from **Figure 2b**, particularly in the orbit of Mercury. With a time-step of $\delta t = 24$ hours, Mercury's orbit becomes inaccurate which can be seen by the 'smearing' of its orbit in **Figure 2b**.

This problem can be remedied at the cost of computation time by decreasing the time-step to a value in the range of $\delta t = 1$ hour however, using a computer with an Intel Core i5-8400 CPU at 2.8GHz, this calculation would take ~ 36.5 min. These computation times can be decreased by simulating a reduced number of bodies and this was the strategy used when investigating the orbits of asteroids in Kirkwood gaps.

Figure 3a shows the kinetic energy – time graph for Jupiter. A regular, sinusoidal plot can be seen which corresponds to the planet's varying velocity as it gets nearer and further away from







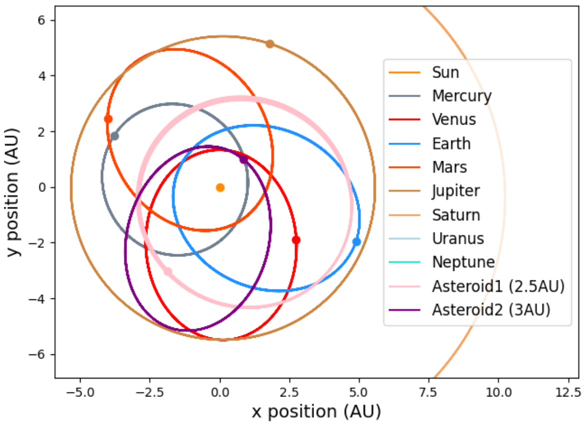
[Figure.3] Simulation of 500 *yrs with a* $\delta t = 24$ *hrs. (a) Plot of the Kinetic Energy against time for the planet Jupiter about the Sun. (b) Plot of the Kinetic Energy against time for an asteroid of mass* 0.95×10^{21} *kg and orbital radius 2.5 AU. (c) Zoomed in section of plot (b), showing the Kinetic Energy for the asteroid over* 100 *years*

the Sun throughout its orbit. **Figure 3b/c** shows the KE– time graphs for an asteroid of mass 0.95×10^{21} kg at an orbital radius of 2.5 AU. It is evident that this plot is significantly more chaotic than Jupiter's, suggesting the body may be in an unstable orbit.

From observations, it can be seen that at a radius of 2.5 AU from the sun, there are very few asteroids in the main asteroid belt. This particular Kirkwood gap corresponds to an orbital resonance with Jupiter of 3:1. By plotting a KE-time graph, the difference between a stable orbit and an unstable orbit can be seen. **Figure 3a** has a regular, periodic form whilst **Figure 3b/c** is irregular and has a linearly increasing 'amplitude'. This change in 'amplitude' corresponds with an increasing eccentricity which, on this time scale, is difficult to determine from a regular orbital plot such as those seen in **Figure 2**. This instability will eventually cause the body to completely leave its circular orbit becoming highly elliptical, leaving behind a baron ring in the main asteroid belt.

It must be noted that it is important to check the validity of this result as it has been observed that this program can perform poorly for smaller orbits such as Mercury's mentioned previously. In order to check the validity of this result, another simulation is run, this time with a time-step, $\delta t = 24$ hr. If the results found in **Figure 3** are physical results as opposed to limitations of the numerical model, the graph should look almost identical despite the change in time-step.

It was found that for this asteroid of orbital radius 2.5 AU, the results were accurate and no

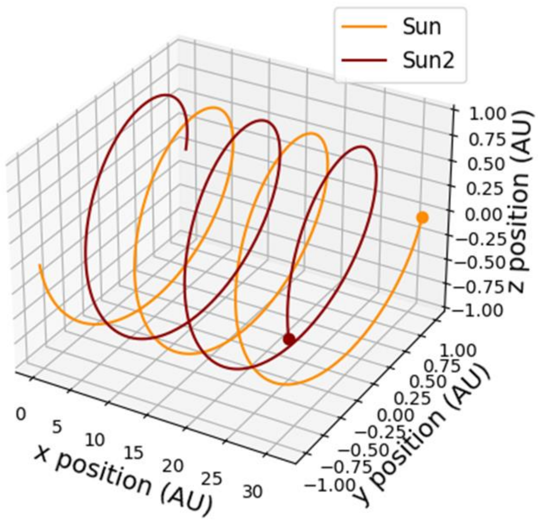


[Figure.4] Plot showing the effects of large time-step ($\delta t = 2$ *weeks) over a long time period* ($T = 100{,}000$ *yrs) on the rocky planets. It is clear from this result that care must be taken to ensure that real physics is being explored as opposed to errors caused my numerical approximations.*

discernible difference was seen when comparing graphs made using $\delta t = 1$ hr and $\delta t = 24$ hr.

The above check must be made as, from **Figure 4**, the limitations of this program are evident. From visual inspection, it is easy to spot large errors such as the one seen here, however, for more subtle errors, this is non-trivial and therefore, the above method should be employed.

An interesting use of this program is to investigate binary star systems and the result of this can be seen in **Figure 5**. It is more obvious that the two stars are orbiting the barycentre in this system than it is in the Earth – Sun system seen in **Figure 1**. This is due to the comparable masses of the two bodies which moves the barycentre of the system to, in this case, the half-way point between the two bodies.



[Figure.5] 3D plot of a system consisting of two bodies of equal mass 1.98892×10^{30} *kg. These bodies can be seen orbiting barycentre and travelling with an average velocity in the x-direction.*

An example of further research using this program is the effects of orbiting satellites on bodies which are orbiting the sun. It was found that it is possible to simulate the Moon orbiting the Earth as it orbits the Sun if accurate initial positions and velocities are used and further investigation into these scenarios would yield interesting results.[2]

Conclusion

This program has provided insight into the behaviour of systems of bodies subject to gravitational interactions and demonstrated interesting physical phenomena.

Using this program, it was possible to calculate the orbital period of the Earth to be 365.35 days and the orbital period of a body which orbits at a radius of $r = \left(0.4 + \frac{4673}{25000}\right) AU$ was found to be 164.28 days.

Kirkwood gaps were investigated and the consequence of a bodies orbit being in resonance with Jupiter's was demonstrated using a Kinetic Energy – time graph. This provided an insight into what defines a stable orbit and it was demonstrated that for a body who's KE – time graph is regular and periodic, that body is said to be in a stable orbit. A body with an irregular KE – time graph is said to have an unstable orbit and as result will tend to become elliptical and drift from its original path.

Through simulations of 2-body and N-body systems, it was found that the gravitational force of the planets causes the Sun to move and orbit about the barycentre of the system. This effect was further explored in the case of a binary star system where two bodies of equal mass orbit the barycentre; in this case, an equidistant point in space between the two bodies.

Further research could yield interesting results while using this program to investigate the effect an orbiting satellite has on a body orbiting the Sun. This would be particularly interesting if satellites have a notable effect on a bodies orbital period which could perhaps allow the prediction of the existence of currently unknown satellites.

References

[1] nineplanets.org. *Kirkwood Gap Facts & Information*. [online] Available at: < https://nineplanets.org/kirkwood-gap/>

[2] ssd.jpl.nasa.gov. *Horizons System*. [online] Available at: <https://ssd.jpl.nasa.gov/horizons/app.html#/>

Appendix

```
"""
# Object Oriented Solar System Model using the Runge-Kutta Method
# 3rd year Project
# James Kavanagh-Cranston
# 40254673
# 07/10/2021
"""
from math import *
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time
import sys
import os
import pickle
import platform
import datetime

'''
#   defining a class which is used throughout in order to store
#   position and velocity data in a cartesian coordinate system
'''

class Coordinate:

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

'''
#   this class is for each body included in the simulation assigning
#   variables to individual bodies for use in calculations
'''

class Body:
    bodies = []

    def __init__(self, position, mass, velocity, name, colour):
        self.position = position
        self.mass = mass
        self.velocity = velocity
        self.name = name
        self.colour = colour
        self.xdat = []
        self.ydat = []
        self.zdat = []
        self.vxdat = []
        self.vydat = []
        self.vzdat = []
        self.radius = []
        self.KE = []

    #   returns a list in the form [x, y, z, vx, vy, vz] for the body's
    #   initial position and velocity
    def returnVec(self):
        return [self.position.x, self.position.y, self.position.z, self.velocity.x, self.velocity.y,
                self.velocity.z]

    #   returns the mass of the body
    def returnMass(self):
        return self.mass

    #   returns the name of the body
    def returnName(self):
        return self.name

    #   returns the resultant velocity of the body
    def returnVel(self):
        return sqrt(self.velocity.x ** 2 + self.velocity.y ** 2 + self.velocity.z ** 2)

    #   returns the angle the body makes with the +ve x-axis, resulting
    #   in a range of angles between 0 - 2pi
    #   note, needs to include exceptions for scenarios where a bodies
    #   position is (0, y)
    def angle(self):
        #   x = +, y = +
        if self.position.x > 0 and self.position.y >= 0:
            angle = atan(abs(self.position.y/self.position.x))
        #   x = -, y = +
        elif self.position.x <= 0 and self.position.y >= 0:
            try:
                angle = pi - atan(abs(self.position.y/self.position.x))
            except:
                angle = pi / 2
        #   x = -, y = -
        elif self.position.x <= 0 and self.position.y < 0:
            try:
                angle = pi + atan(abs(self.position.y/self.position.x))
            except:
                angle = pi * 3/2
        #   x = +, y = -
        elif self.position.x > 0 and self.position.y < 0:
            angle = 2*pi - atan(abs(self.position.y/self.position.x))

        return angle

'''
#   Simulation class is used in order to store variables related to the
#   simulation as a whole allows for expandability
'''

class Simulation:
```

```
def __init__(self, bodies):
    self.bodies = bodies
    self.N_bodies = len(bodies)
    self.joinedVec = np.zeros(0)
    for body in bodies:
        for element in body.returnVec():
            self.joinedVec = np.append(self.joinedVec, element)
    self.massList = np.array([body.returnMass() for body in self.bodies])
    self.nameList = [body.returnName() for body in self.bodies]
    self.t = [0]

#   Runge-Kutta method
def rungeKutta(self, t, dt, stationarySun):
    a = dt * calcDiffEqs(t, self.joinedVec, self.massList, stationarySun)
    b = dt * calcDiffEqs(t + 0.5*dt, self.joinedVec + a/2, self.massList, stationarySun)
    c = dt * calcDiffEqs(t + 0.5*dt, self.joinedVec + b/2, self.massList, stationarySun)
    d = dt * calcDiffEqs(t + dt, self.joinedVec + b, self.massList, stationarySun)

    yNew = self.joinedVec + ((a + 2*b + 2*c + d) / 6.0)
    return yNew

#   method calculating the orbits of the bodies
def runOrbits(self, dt, stationarySun, T, reportFreq):
    self.path = [self.joinedVec]
    clock_time = 0
    nsteps = int(T / dt)

    start_time = time.time()

    for step in range(nsteps):
        sys.stdout.flush()
        sys.stdout.write(f"Integrating: step = {step} / {nsteps} | Simulation time =
                         {round(clock_time, 3)} Percentage = {round(100 * step / nsteps, 1)}%\r")
        yNew = self.rungeKutta(0, dt, stationarySun)

        if step % reportFreq == 0:
            self.path.append(yNew)
            self.t.append(clock_time)

        self.joinedVec = yNew
        clock_time += dt

    runtime = time.time() - start_time

    print(f"\nSimulation completed in {runtime} seconds")

    #   output to metrics.txt to provide insight into hardware performance
    metricsFileName = 'metrics.txt'
    with open(metricsFileName, 'a') as metrics:
        metrics.write(f"\n\n{datetime.datetime.now()}\n")
        metrics.write(f"Platform = {platform.platform()}\n")
```

```
            metrics.write(f"Simulation length = {T/60/60/24/365} yrs\n")
            metrics.write(f"Time step, dt = {dt/60/60} hr\n")
            metrics.write(f"Number of steps = {nsteps}\n")
            metrics.write(f"Completed simulation in {round(runtime, 2)} s\n")
            metrics.write(f"Computation time per time-step = {round(1000 *
                          (runtime / nsteps), 3)} ms\n")

        self.path = np.array(self.path)

        #   assigning position lists to each body
        for index, i in enumerate(range(0, len(self.bodies) * 6, 6)):
            self.bodies[index].xdat = self.path[:,i]
            self.bodies[index].ydat = self.path[:,i+1]
            self.bodies[index].zdat = self.path[:,i+2]

        #   assigning velocity lists to each body
        for index, i in enumerate(range(0, len(self.bodies) * 6, 6)):
            self.bodies[index].vxdat = self.path[:,i+3]
            self.bodies[index].vydat = self.path[:,i+4]
            self.bodies[index].vzdat = self.path[:,i+5]

    #   method calculating the orbital period of the first body
    #   (excluding the sun) in the provided list
    def calcPeriod(self, dt, stationarySun=True, T=None):
        self.path = [self.joinedVec]
        clock_time = 0
        initAngle = self.bodies[1].angle()
        currentAngle = initAngle
        prevAngle = initAngle - 1
        step = 0
        plotx = []
        ploty = []
        start_time = time.time()
        passed = False

        #   while the current angle increases with respect to the previous angle
        while (currentAngle > prevAngle and not passed) or currentAngle < initAngle:
            prevAngle = currentAngle
            sys.stdout.flush()
            sys.stdout.write(f"Integrating: step = {step} | Simulation time =
                             {round(clock_time, 3)}s, {round(clock_time/60/60/24/365, 3)} years\r")
            yNew = self.rungeKutta(0, dt, stationarySun)
            self.path.append(yNew)
            self.joinedVec = yNew
            currentAngle = angle(yNew[6], yNew[7])
            plotx.append(yNew[6])
            ploty.append(yNew[7])
            step += 1
            clock_time += dt

            if not (currentAngle > prevAngle) and not passed:
                passed = True
```

```python
        self.path = np.array(self.path)

        #   assigning position lists to each body
        for index, i in enumerate(range(0, len(self.bodies) * 6, 6)):
            self.bodies[index].xdat = self.path[:,i]
            self.bodies[index].ydat = self.path[:,i+1]
            self.bodies[index].zdat = self.path[:,i+2]

        #   assigning velocity lists to each body
        for index, i in enumerate(range(0, len(self.bodies) * 6, 6)):
            self.bodies[index].vxdat = self.path[:,i+3]
            self.bodies[index].vydat = self.path[:,i+4]
            self.bodies[index].vzdat = self.path[:,i+5]

        #   correcting for the ammount that the time step overshoots a full orbit
        #   assumes that on the small scale of dt, the planet orbits with a
        #   constant angular velocity
        overShoot = currentAngle - initAngle
        underShoot = (2 * pi) - prevAngle
        timeOverShoot = dt * (overShoot/(overShoot + underShoot))

        clock_time -= timeOverShoot

        runtime = time.time() - start_time

        #   debugging plot to check body has made one full orbit
        plot = False
        if plot:
            plt.plot(self.bodies[1].position.x, self.bodies[1].position.y, 'o',
color=self.bodies[1].colour)
            plt.plot(plotx, ploty, color=self.bodies[1].colour)
            plt.axis('equal')
            plt.show()

        print(f"\nSimulation completed in {runtime} seconds")

        #   output to metrics.txt to provide insight into hardware performance
        metricsFileName = 'metrics.txt'
        with open(metricsFileName, 'a') as metrics:
            self.metrics(metrics, dt, step, runtime)

        #   output result in days for orbital periods under 1.5 years
        if clock_time/(60*60*24) <= 365*1.5:
            return f"{clock_time/(60*60*24)} days"

        #   output result in years for orbital periods over 1.5 years
        else:
            return f"{clock_time/(60*60*24*365)} years"

    def metrics(self, metrics, dt, step, runtime):
        metrics.write(f"\n\n{datetime.datetime.now()}\n")
        metrics.write(f"Platform = {platform.platform()}\n")
        metrics.write('Simulation length = None yrs\n')
        metrics.write(f"Time step, dt = {dt/60/60} hr\n")
        metrics.write(f"Number of steps = {step}\n")
        metrics.write(f"Completed simulation in {round(runtime, 2)} s\n")
        metrics.write(f"Computation time per time-step = {round(1000 * (runtime / step), 3)} ms\n")



#   function returning the angle a point at position (x, y) makes
#   with the x-axis, resultingin a range of angles between 0 - 2pi
#   note, needs to include exceptions for scenarios where a bodies
#   position is (0, y)
def angle(x, y):
    #   x = +, y = +
    if x > 0 and y >= 0:
        angle = atan(abs(y/x))
    #   x = -, y = +
    elif x <= 0 and y >= 0:
        try:
            angle = pi - atan(abs(y/x))
        except:
            angle = pi / 2
    #   x = -, y = -
    elif x <= 0 and y < 0:
        try:
            angle = pi + atan(abs(y/x))
        except:
            angle = pi * 3/2
    #   x = +, y = -
    elif x > 0 and y < 0:
        angle = 2*pi - atan(abs(y/x))

    else:
        print("angle() failed")

    return angle

#   calculates the acceleration exerted on each body as a result of
#   gravitational forces exerted on the body by all other bodies
def calcDiffEqs(t, y, masses, stationarySun=True):
    G = 6.67e-11 #m^3 kg^-1 s^-2
    N_bodies = int(len(y) / 6)
    solvedVector = np.zeros(y.size)

    if stationarySun:
        start = 1
    elif not stationarySun:
        start = 0

    for i in range(start, N_bodies):
        ioffset = i * 6
        for j in range(N_bodies):
            joffset = j * 6
            solvedVector[ioffset] = y[ioffset + 3]
            solvedVector[ioffset + 1] = y[ioffset + 4]
            solvedVector[ioffset + 2] = y[ioffset + 5]
            if i!= j:
                xrad = y[ioffset] - y[joffset]
                yrad = y[ioffset + 1] - y[joffset + 1]
                zrad = y[ioffset + 2] - y[joffset + 2]
                r = sqrt(xrad**2 + yrad**2 + zrad**2)
                ax = (-G * masses[j] / r**3) * xrad
                ay = (-G * masses[j] / r**3) * yrad
                az = (-G * masses[j] / r**3) * zrad
                solvedVector[ioffset + 3] += ax
                solvedVector[ioffset + 4] += ay
                solvedVector[ioffset + 5] += az

    return solvedVector

#   calculating the kinetic energy at each time-step for a list of bodies
def calcKE(bodies):
    for body in bodies:
        for i in range(len(body.xdat)):
            resultantVelSqr = body.vxdat[i] ** 2 + body.vydat[i] ** 2 + body.vzdat[i] ** 2
            body.KE.append(0.5 * body.mass * resultantVelSqr)

#   calculating the orbital radius at each time step for a list of bodies
def calcRadius(bodies):
    for body in bodies:
        for i in range(len(body.xdat)):
            body.radius.append(sqrt(body.xdat[i] ** 2 + body.ydat[i] ** 2 + body.zdat[i] ** 2))



##########  Variables   ##########


#   time period in years
T = 12
#   dt in hours
dt = 1
#   sun stationary or dynamic
sunStationary = True
#   report frequency in days
reportFreq = 1
#   give the Sun momentum to prevent system drift
momentumFix = True



##########  Body initial positions  ##########
```

```
'''
#   initial positions are held in dictionaries.
#   the below data are representative of the corresponding bodies as they were on 24th Nov 2019
#   data was obtained from NASA JPL at:
#   https://ssd.jpl.nasa.gov/horizons/app.html#/
'''

AU = 1.495978707e11
G = 6.67e-11

R1 = 2.06 * AU
V1 = sqrt((G * 1.98892e30) / R1)

R2 = 2.5 * AU
V2 = sqrt((G * 1.98892e30) / R2)

R3 = 2.82 * AU
V3 = sqrt((G * 1.98892e30) / R3)

R4 = 2.6 * AU
V4 = sqrt((G * 1.98892e30) / R4)

#   4025 4673
assignmentPlanetRad = ( 0.4 + (4673) / 25000 ) * AU
assignmentPlanetVel = sqrt((G * 1.98892e30) / assignmentPlanetRad)

#   information about initial body positions and velocities are
#   input using dictionaries for easy reading and manipulation

sun = {"position":Coordinate(0,0,0), "mass":1.98892e30, "velocity":Coordinate(0,0,0),
"colour":'darkorange', "name":'Sun'} #darkorange  pink
mercury = {"position":Coordinate(-2.754973475923117E+10, 3.971482635075326E+10,
5.772553348387497E+09), "mass":3.302e23, "velocity":Coordinate(-4.985503186396708E+04, -
2.587115609586964E+04, 2.459423100025674E+03), "colour":'slategrey', "name":'Mercury'}#3.302e23
venus = {"position":Coordinate(6.071824980347975E+10, -9.031478095293820E+10, -4.743119158717781E+09),
"mass":48.685e23, "velocity":Coordinate(2.882992914024795E+04, 1.941822077492687E+04, -
1.397248807063850E+03), "colour":'red', "name":'Venus'}
earth = {"position":Coordinate(7.081801535330121E+10, 1.304740594736121E+11, -4.347298831932247E+06),
"mass":5.972e24, "velocity":Coordinate(-2.659141632959534E+04, 1.428195558990953E+04,
1.506587338520049E-01), "colour":'dodgerblue', "name":'Earth'}
mars = {"position":Coordinate(-2.338256705323077E+11, -6.744910716399051E+10, 4.323713396453075E+09),
"mass":6.417e23, "velocity":Coordinate(7.618913418166695E+03, -2.120844917567340E+04, -
6.313535649528479E+02), "colour":'orangered', "name":'Mars'}
jupiter = {"position":Coordinate(3.640886585245620E+10, -7.832464736633219E+11,
2.438628231032491E+09), "mass":1.898e27, "velocity":Coordinate(1.290779089733536E+04,
1.225548372134438E+03, -2.938400770294290E+01), "colour":'peru', "name":'Jupiter'}
saturn = {"position":Coordinate(5.402930559845881E+11, -1.401101262550307E+12, 2.852732020323873E+09),
"mass":5.683e26, "velocity":Coordinate(8.493464320782032E+03, 3.448137125239667E+03, -
3.974048933366207E+01), "colour":'sandybrown', "name":'Saturn'}
uranus = {"position":Coordinate(2.440195048138449E+12, 1.684964594605102E+12, -2.534637639068711E+10),
"mass":86.813e24, "velocity":Coordinate(-3.907126929953547E+03, 5.287630406417824E+03,
7.019651631039658E+00), "colour":'lightblue', "name":'Uranus'}
```

```
neptune = {"position":Coordinate(4.370905565958221E+12, -9.700637945288652E+11, -
8.076862923992699E+10), "mass":102.409e24, "velocity":Coordinate(1.155455528773291E+03,
5.342194379391015E+03, -1.363189398672890E+01), "colour":'turquoise', "name":'Neptune'}
pluto = {"position":Coordinate(1.924334541511769E+12, -4.695917904493715E+12, -5.411176029132771E+10),
"mass":1.307e22, "velocity":Coordinate(5.165981607247869E+03, 9.197384040335644E+01, -
1.566614784409940E+03), "colour":'rosybrown', "name":'Pluto'}
moon = {"position":Coordinate(7.048971389599609E+10, 1.303131748973128E+11, 2.721761216115206E+07),
"mass":7.349e22, "velocity":Coordinate(-2.612964553516142E+04, 1.331468891451992E+04, -
2.786657700313278E+01), "colour":'slategrey', "name":'Moon'}#slategrey pink
asteroid1 = {"position":Coordinate(R1, 0, 0), "mass":0.95e21, "velocity":Coordinate(0, V1, 0),
"colour":'pink', "name":f'Asteroid1 ({R1/AU}AU)'}
asteroid2 = {"position":Coordinate(R2, 0, 0), "mass":0.95e21, "velocity":Coordinate(0, V2, 0),
"colour":'purple', "name":f'Asteroid2 ({R2/AU}AU)'}
asteroid3 = {"position":Coordinate(R3, 0, 0), "mass":0.95e21, "velocity":Coordinate(0, V3, 0),
"colour":'lawngreen', "name":f'Asteroid3 ({R3/AU}AU)'}
asteroid4 = {"position":Coordinate(R4, 0, 0), "mass":0.95e21, "velocity":Coordinate(0, V4, 0),
"colour":'red', "name":f'Asteroid4 ({R4/AU}AU)'}

# sun = {"position":Coordinate(0,-AU,0), "mass":1.98892e30, "velocity":Coordinate(sqrt((G *
1.98892e30) / AU), 0, -0.5*sqrt((G * 1.98892e30) / AU)), "colour":'darkorange', "name":'Sun'}
#darkorange  pink
# sun2 = {"position":Coordinate(0,AU,0), "mass":1.98892e30, "velocity":Coordinate(sqrt((G *
1.98892e30) / AU), 0, 0.5*sqrt((G * 1.98892e30) / AU)), "colour":'darkred', "name":'Sun2'}
# earth = {"position":Coordinate(AU, 0, 0), "mass":5.9742e24, "velocity":Coordinate(0, sqrt((G *
1.98892e30) / AU), 0), "colour":'dodgerblue', "name":'Earth'}
# mercury = {"position":Coordinate(0.387 * AU, 0, 0), "mass":3.302e23, "velocity":Coordinate(0,
sqrt((G * 1.98892e30) / (0.387 * AU)), 0), "colour":'slategrey', "name":'Mercury'}
# venus = {"position":Coordinate(0.723 * AU, 0, 0), "mass":48.685e23, "velocity":Coordinate(0, sqrt((G
* 1.98892e30) / (0.723 * AU)), 0), "colour":'red', "name":'Venus'}
# jupiter = {"position":Coordinate(5.2*AU, 0, 0), "mass":1.898e27, "velocity":Coordinate(0, sqrt((G *
1.98892e30) / (5.2 * AU)), 0), "colour":'peru', "name":'Jupiter'}
# neptune = {"position":Coordinate(30.1 * AU, 0, 0), "mass":102.409e24, "velocity":Coordinate(0,
sqrt((G * 1.98892e30) / (30.1 * AU)), 0), "colour":'turquoise', "name":'Neptune'}
# assignmentPlanet = {"position":Coordinate(assignmentPlanetRad, 0, 0), "mass":5.9742e24,
"velocity":Coordinate(0, assignmentPlanetVel, 0), "colour":'crimson', "name":f'Assignment Asteroid
({assignmentPlanetRad / AU}AU)'}

'''
#   the list, bodyNames, is used to select which bodies
#   should be included in the simulation
'''

# bodyNames = [sun, mercury, venus, earth, mars, jupiter, saturn, uranus, neptune, asteroid1,
asteroid2]

# bodyNames = [sun, mercury, venus, earth, mars, jupiter, saturn, uranus, neptune, pluto, moon]
# bodyNames = [sun, mercury, venus, earth, moon, mars, jupiter, saturn, uranus, neptune, pluto]
# bodyNames = [sun, mercury, venus, earth, moon, mars, jupiter, saturn, uranus, neptune]
# bodyNames = [sun, mercury, venus, earth, mars, jupiter, saturn, uranus, neptune]
# bodyNames = [sun, mercury, venus, earth, moon, mars, jupiter, saturn, uranus]
# bodyNames = [sun, mercury, venus, earth, moon, mars, jupiter, saturn]
bodyNames = [sun, mercury, venus, earth, moon, mars, jupiter]
```

```
# bodyNames = [sun, mercury, venus, earth, moon, mars]
# bodyNames = [sun, mercury, venus, earth, moon]
# bodyNames = [sun, mercury, venus, earth]
# bodyNames = [sun, mercury, venus]
# bodyNames = [sun, mercury]
# bodyNames = [sun, venus]
# bodyNames = [sun, earth]
# bodyNames = [sun, assignmentPlanet]
# bodyNames = [sun, earth, moon]
# bodyNames = [sun, neptune]
# bodyNames = [sun, mercury, venus, asteroid]
# bodyNames = [sun, asteroid1, asteroid2, asteroid3, jupiter]
# bodyNames = [sun, sun2]
# bodyNames = [sun, asteroid1, asteroid2, jupiter]
# bodyNames = [sun, asteroid1, jupiter]
# bodyNames = [sun, jupiter, asteroid1, asteroid2, asteroid3, asteroid4]
# bodyNames = [sun, jupiter]

'''
#   if the sun is allowed to move during the simulation, this if statement
#   will apply momentum to the sun in order to prevent a drift of the whole system
'''

if not sunStationary and momentumFix:

    momx = momy = momz = 0
    for i in range(len(bodyNames) - 1):
        momx += bodyNames[i+1]['velocity'].x * bodyNames[i+1]['mass']
        momy += bodyNames[i+1]['velocity'].y * bodyNames[i+1]['mass']
        momz += bodyNames[i+1]['velocity'].z * bodyNames[i+1]['mass']

    velx = -momx / bodyNames[0]['mass']
    vely = -momy / bodyNames[0]['mass']
    velz = -momz / bodyNames[0]['mass']

    sunVel = Coordinate(velx, vely, velz)

    bodyNames[0]['velocity'] = sunVel

#   a list of objects of the Body class

bodyObjects = [Body( body['position'], body['mass'], body['velocity'], body['name'], body['colour'] )
               for body in bodyNames]

simulation = Simulation(bodyObjects)



##########  Output  ##########

'''
#   this section will run trhe simulation and save the calculated data to
#   a file using the pandas library
#   this prevents the unnecessary recalculation of simulations which have
#   been run previously, saving time throughout the development of the software
'''

#   outer planet
planetTo = bodyObjects[-1].name

#   saving the data with a filename which can be used to retrieve the data
#   at a later date
sunString = '__dynamicSun' if not sunStationary else '__staticSun'
filename = f'data/T={T}__dt={dt}hr__to_planet={planetTo}{sunString}.data'

#   checking if the current simulation settings have been calculated previously
if not os.path.isfile(filename):

    #   running simulation for specified dt, T and reporting frequency
    print(f"\nRunning simulation with:\ndt = {dt}\nT = {T}\nStationary Sun =
{sunStationary}\nReporting Freq = {reportFreq}\n")
    simulation.runOrbits(dt*60*60, sunStationary, T*365*24*60*60, reportFreq*24/dt)
    path = simulation.path

    # for index, i in enumerate(range(0, le

    calcKE(bodyObjects)
    calcRadius(bodyObjects)

    KE = [body.KE for body in bodyObjects]
    t = simulation.t
    radius = [body.radius for body in bodyObjects]
    print("\nCalculations complete.")

    #   selecting the information to save in the .data file
    dump = [path, KE, bodyObjects, t, radius]

    #   saving the .data file
    print(f"Saving data as {filename}...")
    with open(filename, 'wb') as f:
        pickle.dump(dump, f)
    print("Data saved.")

#   retrieving the pre-calculated data
elif os.path.isfile(filename):
    print(f"\nLoading data from {filename}...")
    with open(filename, 'rb') as f:
        load = pickle.load(f)
        path = load[0]
        KE = load[1]
        bodyObjects = load[2]
        t = load[3]
        radius = load[4]
    print("Data loaded.")
```

```python
##########  Plotting  ##########



#   selecting the final 1000 position points before plotting
#   this reduces the number of points which are plotted, reducing the
#   time taken for the matplotlib plot to load
print(f"Plotting...")
points = 150000
for body in bodyObjects:
    body.xdatPruned = body.xdat[len(body.xdat)-points+1:len(body.xdat)-1]
    body.ydatPruned = body.ydat[len(body.ydat)-points+1:len(body.ydat)-1]
    body.zdatPruned = body.zdat[len(body.zdat)-points+1:len(body.zdat)-1]

tYrs = np.array(t) / (60*60*24*365)

#   2d plotting the orbits
fig = plt.figure(figsize=(6.5,5))
for body in bodyObjects:
    xpos = np.array(body.xdatPruned) / AU
    ypos = np.array(body.ydatPruned) / AU

    plt.plot(xpos, ypos, color=body.colour, label=body.name)
    plt.plot(xpos[-1], ypos[-1], 'o', color=body.colour)

plt.legend(fontsize=12)
plt.axis('equal')
plt.xlabel('x position (AU)', fontsize=14)
plt.ylabel('y position (AU)', fontsize=14)
plt.show()


# #   3d plotting the orbits
# fig = plt.figure(figsize=(9,5))
# ax = plt.axes(projection='3d')

# #   uncomment below to plot with time in the z-axis
# # tYrs = tYrs[0:-1]
# for body in bodyObjects:
#     xpos = np.array(body.xdatPruned) / AU
#     ypos = np.array(body.ydatPruned) / AU
#     zpos = np.array(body.zdatPruned) / AU

#     plt.plot(xpos, ypos, zpos, color=body.colour, label=body.name)
#     plt.plot(xpos[-1], ypos[-1], zpos[-1], 'o', color=body.colour)

# plt.legend(fontsize=12)
# ax.set_xlabel('x position (AU)', fontsize=14)
# ax.set_ylabel('y position (AU)', fontsize=14)
# ax.set_zlabel('z position (AU)', fontsize=14)
# plt.show()


#   plotting the KE against time
for body in bodyObjects:
    fig = plt.figure(figsize=(6.5,5))
    plt.plot(tYrs, body.KE, color=body.colour, label=body.name)

    plt.legend(fontsize=12)
    plt.xlabel('Time, t (yrs)', fontsize=14)
    plt.ylabel('Kinetic Energy, KE (J)', fontsize=14)
    plt.show()

'''
# #   plotting total KE against time
# totalKE = []
# for i in range(len(bodyObjects[0].KE)):
#     totalKE.append(0)
#     for body in bodyObjects:
#         totalKE[i] += body.KE[i]

# fig = plt.figure(figsize=(6.5,5))
# plt.plot(tYrs, totalKE)
# plt.xlabel('Time, t (yrs)', fontsize=14)
# plt.ylabel('Total KE of System, KE (J)', fontsize=14)
# plt.show()
'''

#   plotting distance from origin against time
# radAU = np.array(radius) / AU

# for i, body in enumerate(bodyObjects):
#     fig = plt.figure(figsize=(6.5,5))
#     plt.plot(tYrs, radAU[i], color=body.colour, label=body.name)

#     plt.legend(fontsize=12)
#     plt.xlabel('Time, t (yrs)', fontsize=14)
#     plt.ylabel('Orbital Radius, r (AU)', fontsize=14)
#     plt.show()

# #   plotting the normaliesed KE and distance from origin against time
# for i, body in enumerate(bodyObjects):
#     #   normalising the KE numpy array
#     KE = np.array(body.KE)
#     y1 = 100 * (KE / np.amax(KE))

#     #   normalising the orbRad numpy array
#     orbRad = np.array(radius[i])
#     y2 = 100 * (orbRad / np.amax(orbRad))

#     fig = plt.figure(figsize=(6.5,5))
#     plt.plot(tYrs, y1, color=body.colour, label=f"{body.name} KE")
#     plt.plot(tYrs, y2, color='green', label=f"{body.name} rad")

#     plt.legend(fontsize=12)
#     plt.xlabel('Time, t (yrs)', fontsize=14)
#     plt.ylabel('Normalised Kinetic Energy, KE (%) & Orbital Radius, r (%)', fontsize=14)
#     plt.show()



##########  Orbital Period  ##########



# print(simulation.calcPeriod(dt*60*60, sunStationary))
```