Simulating Trojan Asteroids in Python

Candidate Number: 6946S

University of Cambridge

May 3, 2021

Abstract

This paper is dedicated to creating a simulation of Trojan asteroids in Python. This simulation is then used to investigate the stability of the region around the Lagrange point L4. Using the planar circular restricted three-body problem revealed that the stability of an asteroid was primarily determined whether the specific angular momentum of the asteroid was similar to the specific angular momentum of a mass stationary at the Lagrange point. For asteroids with this angular momentum, looking in the radial displacement and radial velocity space stability was observed in egg-shaped regions extending in the outwards radial direction further than inwards. The stability for different masses of the orbiting planet was also measured. Increasing the mass of the orbiting planet caused the maximum distance an asteroid wandered to increase. At a Sun to planet mass ratio of 0.001, a maximum in the wander distance was observed. Beyond this, the maximum wander distance decreased with no stable asteroids observed near mass ratios of 0.025 ± 0.001. Above this, some stable asteroids were observed up to a mass ratio of 0.042. These results are consistent with other work in the field. Word count: 2992.

1 Introduction

Trojan asteroids are collections of asteroids located at approximately either 60° ahead or behind a body in orbit at the L4 and L5 Lagrange points of a two-body system. These asteroids orbit in a 1:1 resonance with the smaller of the two bodies. The stability of these Lagrange points was theorised before any stable bodies were discovered, since then, thousands of Trojan asteroids have been discovered for Jupiter and more discovered elsewhere in the solar system. This type of asteroid promises details of the history of the early solar system and is therefore of particular interest.

While an object stationary at a Lagrange point will remain there, objects close to L4 and L5 are known to wander. The action of the Coriolis force causes the asteroids to oscillate about the Lagrange point while remaining stable. Whether the asteroid remains in a stable orbit near the Lagrange point is therefore of interest and is dependent on the position and velocity of the asteroid.

It is this dependence; through building a numerical simulation of Trojan asteroids, that I aim to explore. To do this, the correlation between an asteroid's initial position in phase space and its wander distance from the Lagrange point will be investigated. How the ratio of the celestial body masses changes the behaviour of the asteroids will also be examined. This investigation will be limited to the planar circular restricted three-body model. This model will be used primarily because of the limited computational resources available. The majority of Trojan asteroids in the solar system are located at the Lagrange points of Jupiter so the majority of experiments in this investigation will use the mass and orbital radius of Jupiter. Trojan asteroid simulations have been run many times before, therefore comparisons can be made between the relatively simple model investigated here and the more complex analysis in other works. During this paper, the larger mass will be referred to as the 'Sun' and the smaller 'planet'.

An analysis of the computational physics used and the simplifications made will be discussed in the next section, followed by a summary of the implementation and performance of the simulation. A discussion of the results obtained is in section 4, with some conclusions in section 5.

2 Analysis

2.1 Physical Model

In general, gravitational simulations are complicated and time-consuming, therefore in this paper a simple model of a two-body system is used so that the paths of asteroids can be solved in reasonable time while preserving the key physics.

A key simplification made was to build a two-dimensional simulation; only considering the co-orbital motion of the asteroids. This was done in principle to reduce the time complexity without losing significant physical insight.

Beyond this the following simplifications were made:

1. The asteroids are point test masses.
2. The planet moves in a circular orbit.
3. Other gravitational bodies are ignored.

The first simplification means asteroids do not have gravitational fields and will not collide with each other. Therefore there can be no exchange of energy between asteroids and they can be simulated independently, while the second and third simplifications mean the rotating frame implementation discussed in section 2.3

Quantity	Name	Symbol
Mass	Solar Mass	$M_\odot$
Distance	astronomical unit	au
Time	year	y

Table 1: Scaling of the simulation.

can be used. Together these simplifications correspond to the planar circular restricted three-body problem (CR3BP).

In the solar system, differences are observed between L4 and L5. These differences are generally attributed to effects that have been removed by these simplifications[1]. Therefore the Lagrange points L4 and L5 are identical in this simulation and accordingly only L4 will be considered during this investigation.

2.2 Scaling

The simulation is scaled using an astronomical system of units (Table 1). In this system the gravitational constant, G, is given by $4\pi^2 \mathrm{au}^3 \mathrm{y}^{-2} M_\odot{}^{-1}$. These units have the advantage that they are both physically meaningful while ensuring that the majority of variables in the program are of the same order of magnitude which may reduce the error propagation due to floating-point arithmetic.

2.3 Rotating Frame

As mentioned, the CR3BP means the simulation can be implemented in the rotating frame of the planet. This has several advantages over implementing in an inertial frame. Most importantly, in the rotating frame, the planet and Sun are stationary, therefore the motion of the planet does not need to be simulated which would add sources of error. In the rotating frame, the equations of motion can be written as a single second-order differential equation which can then be solved by standard computational techniques. The equation of motion in a rotating frame are given by:

$$\boldsymbol{a}_r = \boldsymbol{a}_i - 2\boldsymbol{\Omega} \times \boldsymbol{v}_r - \boldsymbol{\Omega} \times (\boldsymbol{\Omega} \times \boldsymbol{r}) - \frac{d\boldsymbol{\Omega}}{dt} \times \boldsymbol{r}, \quad (1)$$

where $\boldsymbol{a}_r$ and $\boldsymbol{v}_r$ are acceleration and velocities in the rotating frame. $\boldsymbol{r}$ is the position vector relative to the centre of rotation, which here will be the barycenter of the Sun and planet. $\boldsymbol{a}_i$ is the acceleration in the inertial frame, in this case, given by the Newtonian gravitational force due to the Sun and planet. $\boldsymbol{\Omega}$ (the axis of rotation) is constant due to the circular motion approximation. In the co-orbital plane of the planet, equation (1) reduces to:

$$\boldsymbol{a}_r = \boldsymbol{g_s} + \boldsymbol{g_p} - |\boldsymbol{\Omega}|^2 \boldsymbol{r} - 2\boldsymbol{\Omega} \times \boldsymbol{v}_r, \quad (2)$$

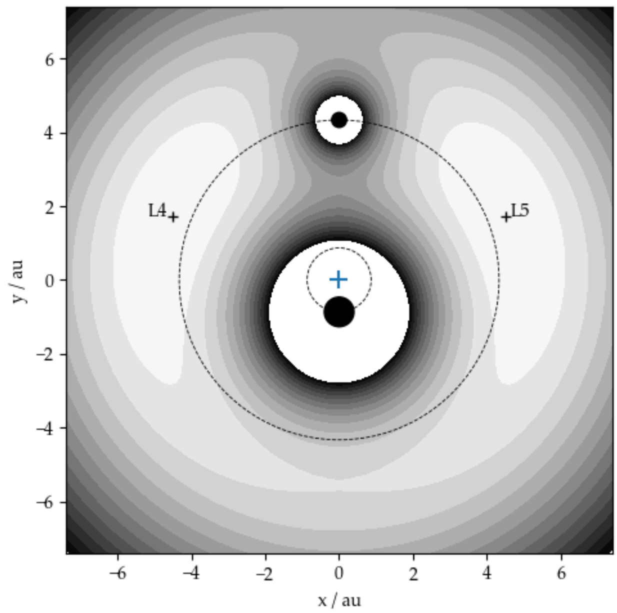

Figure 1: The effective potential in the frame rotating with the Sun and an orbiting planet. This shows the L4 and L5 Lagrange points being located at maxima in the potential. The ratio of the Sun and planet masses is 5:1 to exaggerate the effect that a smaller planet such as Jupiter would have.

where $\boldsymbol{g}_{s/p}$ are the gravitational forces from the Sun and planet respectively. Here the first three terms correspond to the gradient of the effective potential. This is plotted in Fig.1. Equation (2) can then be separated into the acceleration in two Cartesian directions which when implemented in the code are then split into four coupled first-order ODEs.

$\boldsymbol{\Omega}$ is perpendicular to the co-orbital plane meaning the perpendicular velocity does not contribute to equation (2). Hence, building this simulation in 2D will not drastically limit the capability of this simulation. The magnitude of $\boldsymbol{\Omega}$ is given by:

$$|\boldsymbol{\Omega}| = \sqrt{\frac{G(m_s + m_p)}{R^3}}, \quad (3)$$

where m_s and m_p are the masses of the Sun and planet respectively. m_p will be varied in section 4.4 but otherwise $m_s = 1M_\odot$, $m_p = 0.001M_\odot$ and $R = 5.2\mathrm{au}$ corresponding to Jupiter's parameters.

3 Implementation

The code is written in Python 3 and is structured into one Simulation class that contains all of the information and functions to run individual simulations. This class is then initialised by other programs that calculate and plot the results. This means there is consistency between the different experiments in this project. The

Method	Min Steps per Orbit	Time / s	H error / %
RK45	100	25	1.1e-7
RK23	100	17	4.0e-3
LSODA	100	10	2.2e-4
DOP853	20	12	1.5e-9

Table 2: The relative speeds and accuracy of different algorithms in the SciPy library. The same asteroid was used for each test. The asteroid was simulated for 800 periods with the error in the Hamiltonian taken from the maximum variation over this time. The run times showed some variability, of the order of 1s.

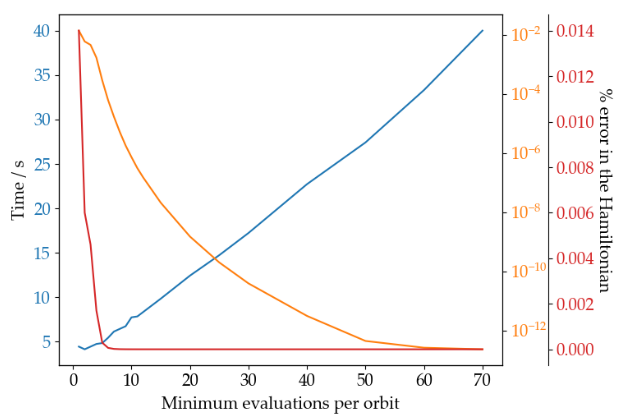

Figure 2: Run time and error in the Hamiltonian for an asteroid with a wander distance of 7.6au is shown as a function of the minimum number of evaluations per orbit. The error is shown on both log and linear scales. The time appears to increase linearly with the minimum evaluations per orbit while the error decays rapidly, but with diminishing returns. The error in the Hamiltonian was evaluated from the maximum variation in the Hamiltonian accumulated over 800 orbits.

asteroid generation is abstracted from the main simulation so that the same calculating/plotting program may run for different asteroid patterns. This greatly improves the versatility of the code.

For simplicity, the simulation is implemented with the barycentre at the coordinate origin with other metrics such as radius and angular velocity being measured about this point. The source code can be found in appendix A.

3.1 ODE Algorithm

At the heart of this simulation will be a numerical ODE solver. Here I have chosen to implement an open-source algorithm. This is done to improve speed and accuracy over attempting to build my own. For this project the DOP853[1] algorithm has been chosen from the SciPy integration library[3]. As an 8$^{\text{th}}$ order integrator this algorithm requires fewer integration steps (hence less execution time) to achieve the same accuracy as a lower order integrators as shown in Table 2. Where the accuracy of each algorithm is quantified using the maximum variation in the Hamiltonian accumulated over 800 orbits for an asteroid with a wander distance of 7.6au; this is discussed further in section 4.2.1.

The implementation of DOP853 also permits changing the value of the minimum steps to reach the desired end time, with more being used if required. The specific choice of this value is somewhat arbitrary as within the scope and simplifications of this project, highly precise measurements of the solar system would not be possible and therefore it is foolish to strive for the highest possible accuracy at the expense of compute time. However, insufficient precision may be unable to successfully determine if an asteroid is stable as the asteroid may become (un)stable due to errors in each integration step accumulating over time. Different simulation lengths may be run so the minimum steps to reach the end time is standardised into the minimum evaluations per orbit. Fig.2 shows a dramatic drop in the error around 7 evaluations per orbit and hence it would be unwise to use any value less than this. The compute time appears to increase linearly with minimum evaluations per orbit while the error shows diminishing returns for larger values. For these reasons, 12 will be used as the minimum evaluations per orbit, which represents a good compromise between speed and accuracy.

[1]See E. Hairer (2000)[2] for details.

Threaded?	Time / s
Yes	30
No	57

Table 3: Comparison between a multi-threaded and a single threaded program. This was calculated for 25 asteroids initially located at L4 simulated over 500 periods. Times are recorded to the nearest second.

3.2 Optimisations

A limitation of the investigation is the number of asteroids that can be simulated. Using a faster program will mean more asteroids can be simulated in reasonable time. Two major optimisations in addition to the choice of ODE algorithm and the physical simplifications discussed are made. These optimisations are outlined below.

3.2.1 Multi-threading

Due to the CR3BP implemented, the asteroids are independent of each other and therefore can be solved separately, thus, multi-threading can easily be taken advantage of. The Python multiprocessing library[2] was used for this task.

Table 3 shows the dramatic increase in performance that multiprocessing provides, in this case, the simu-

[2]https://docs.python.org/3/library/multiprocessing.html

lation executed in approximately half the time when using multi-threading compared to single-threading.

3.2.2 Removal of Non-Trojan Asteroids

Another optimisation is that asteroids were no longer considered 'Trojan' if the asteroid wandered more than 8au from L4. As discussed by M. Janson(2013)[4] larger amplitude orbits are possible but may only be stable due to the simple model investigated here and will add significant complexity to the analysis, so will not be considered.

To implement this optimisation, the simulation would run for intervals of time before checking the wander distances and removing asteroids that wandered too far from L4. This meant that asteroids were not simulated for longer than necessary. To take further advantage of this, the first simulation interval was short (≈ 8 orbits). This would eliminate highly unstable asteroids before the main simulation would be performed.

4 Results and Discussion

4.1 The Shape of Trojan Orbits

The first step is to look at the path of a simple stable orbit to confirm the shape and stability of asteroids in the simulation.

Fig.3 shows the typical tadpole orbit of Trojan asteroids. Importantly, this plot is in the rotating frame. In an inertial frame, the orbit looks similar to a simple Keplerian orbit due to the relative size of the wander compared to the orbit radius.

4.2 Accuracy and Stability

To ensure the simulation was working as expected a number of tests were performed.

4.2.1 Conservation of the Hamiltonian

In a rotating frame the energy, $E = E_k + V$, of an asteroid is not conserved while the Hamiltonian:

$$H = \frac{1}{2}|\boldsymbol{v}|^2 - \frac{1}{2}|\boldsymbol{\Omega} \times \boldsymbol{v}|^2 + V(\boldsymbol{r}), \tag{4}$$

is[5]. Where $V(\boldsymbol{r})$ is the gravitational potential energy due to the Sun and planet.

Fig.4 shows that over time the small errors accumulate and cause the Hamiltonian to vary. From Fig.2 the strong dependence between the precision of the ODE solver and the error in the Hamiltonian of an asteroid suggests that the main source of this error is the numerical procedure used and not the physical model suggesting that H is indeed conserved (within the precision of the integrator) implying the simulation is working as expected. The maximum wander distance of an asteroid in Fig.4 is 7.3au, therefore the error shown is representative of the maximum error of any

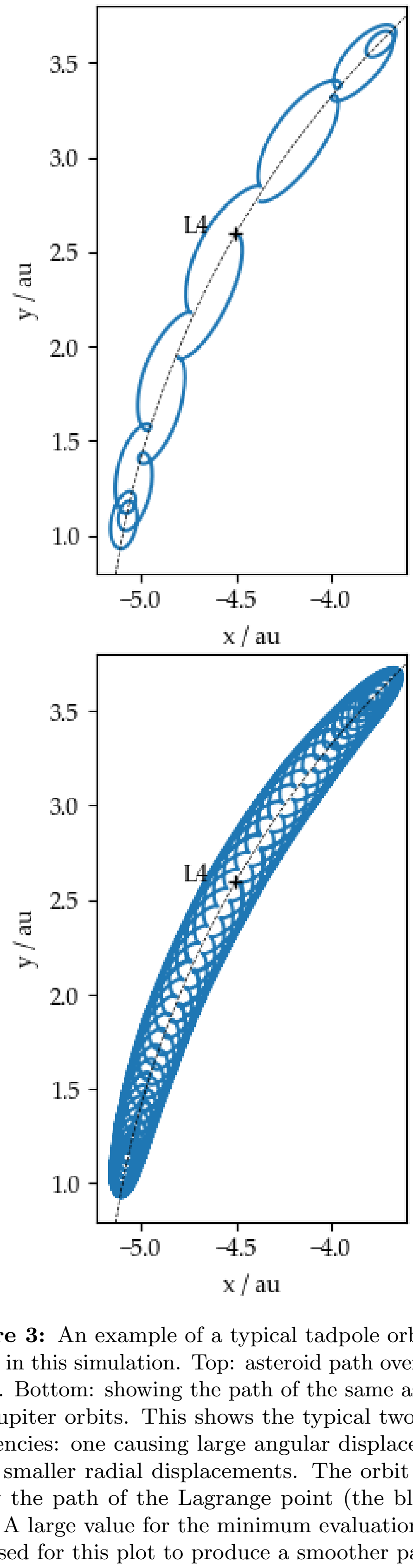

Figure 3: An example of a typical tadpole orbit of an asteroid in this simulation. Top: asteroid path over 12 Jupiter orbits. Bottom: showing the path of the same asteroid over 100 Jupiter orbits. This shows the typical two oscillation frequencies: one causing large angular displacements, the other smaller radial displacements. The orbit appears to follow the path of the Lagrange point (the black dashed line). A large value for the minimum evaluations per orbit was used for this plot to produce a smoother path.

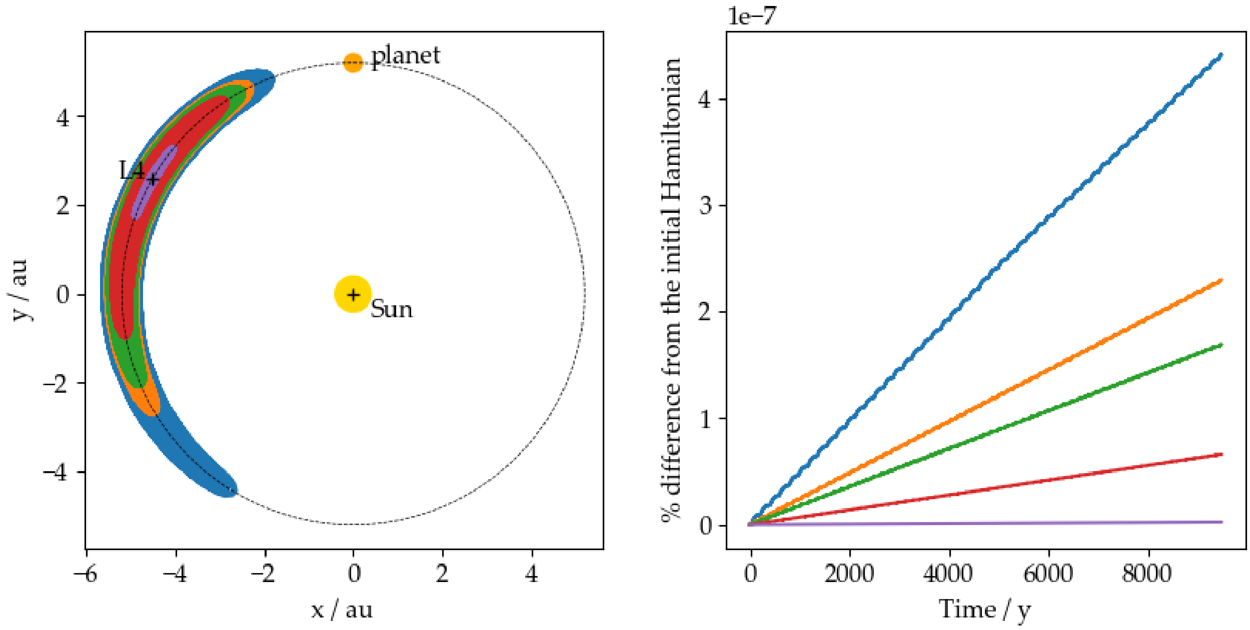

Figure 4: Variation of the Hamiltonian over integration time. All asteroids began at L4 with random velocities up to 0.12auy^{-1}. Left: Plot showing the paths of the asteroids in the rotating frame. The maximum wander distance of an asteroid is 7.3au. Right: The percentage change in the Hamiltonian for the asteroids shown in the left plot. Here, one colour corresponds to the same asteroids in both plots. The Hamiltonian appears to change linearly with simulation time and for asteroids that wander farther from the Lagrange point, the error increases at a faster rate. A minimum of 12 evaluations per orbit was used.

stable asteroid in the simulation. Even so, this error is significantly smaller than the precision that will be possible later in this project because moving forward, the data will be limited by the number of asteroids that can be simulated rather than the precision with which those asteroids are simulated.

4.2.2 Period of Oscillation

To compare this simulation to previous work in the field; the period of oscillation of an asteroid is compared to the work of G. Laughlin (2002)[6] which gives approximate values for the oscillation frequencies of stable Trojan asteroids. Here, asteroids were simulated and then a sinusoid was fitted to the angular displacement from L4 to calculate the oscillation frequency.

Fig.5 shows an example of an asteroid's angular position versus time and the corresponding sinusoidal fit made to calculate the frequency of the longer period oscillations. Similar fits were performed for 208 stable asteroids with random initial velocities placed at L4. The angular frequency was found to be 0.042 ± 0.003y^{-1} with a range of 0.01y^{-1}. This is consistent with the value given by G. Laughlin (2002)[6] of approximately 0.0435y^{-1}. This is further evidence for the validity of the simulation.

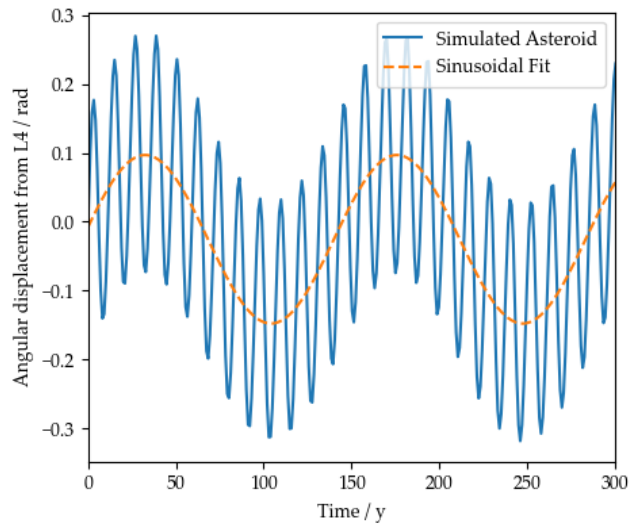

Figure 5: Angular position of an asteroid against time. This shows the sinusoidal fit made to calculate the angular frequency fits the longer period oscillations. The period of the asteroid shown here was calculated as 0.0438 y^{-1}

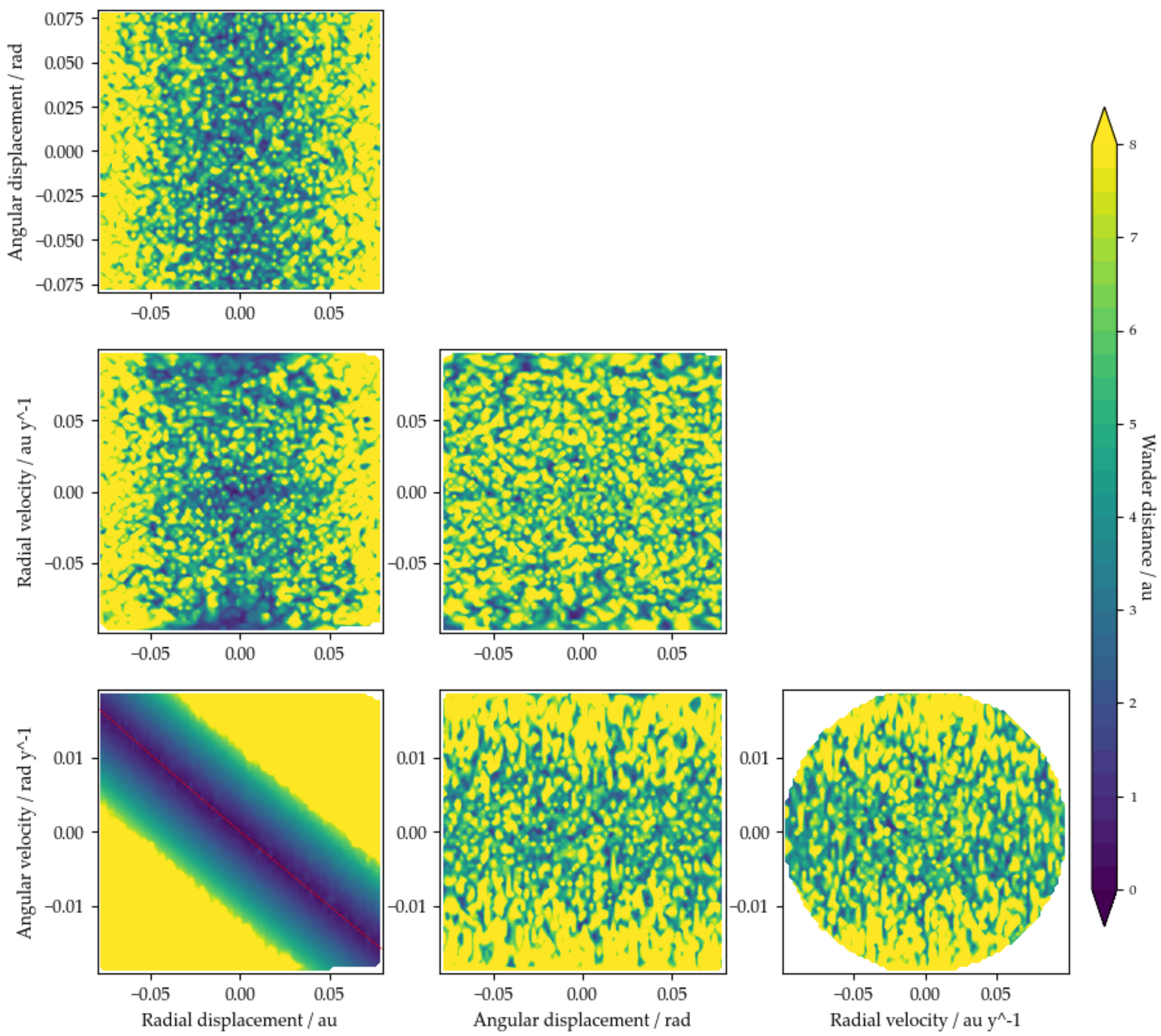

Figure 6: Correlation between initial position, velocity and wander distance for 12,000 asteroids with random initial positions and velocities simulated for 800 periods of the planet. The results show a strong dependence between $\dot{\theta}$ and r which is well understood as being related to the specific angular momentum of the asteroid compared to the Lagrange point. This plot does also indicate that this is the primary driving force for stability. The plot of θ against $\dot{r}$ shows no obvious correlation.

Other tests of the accuracy of the simulation were performed, including setting the planet mass to zero and ensuring that the paths of the asteroids were simple elliptical orbits.

4.3 Wander Distance from L4

Here, how far asteroids 'wander' from L4 is investigated. This is done by calculating the maximum distance an asteroid has travelled from the Lagrange point during the simulation. A critical parameter here is how many orbital periods the simulation should run for. If the simulation is not run for sufficient simulation time the asteroids may not reach their maximum wander distance. However long simulations quickly become impractical. Section 4.2.2 shows that the period of oscillation is of the order of 12 orbits of the Planet. This represents the minimum time that a simulation should be run. However, asteroids that are initially stable but become unstable must be accounted for. Therefore simulations are run for 800 orbits of the Planet ($\approx 9500y$ for a Jupiter sized planet). This, through trial and error, ensured that the final number of stable asteroids was approximately constant. However, many asteroids were stable for long periods of time (≈ 500 orbits). Longer simulations would be beneficial, however, shorter simulations meant that more asteroids total could be simulated overall.

To investigate the stability of L4, asteroids were placed randomly in an annulus sector covering 0.16au radially and 0.16rad annularly centred on L4. This was done to maximise the density of initial positions investi-

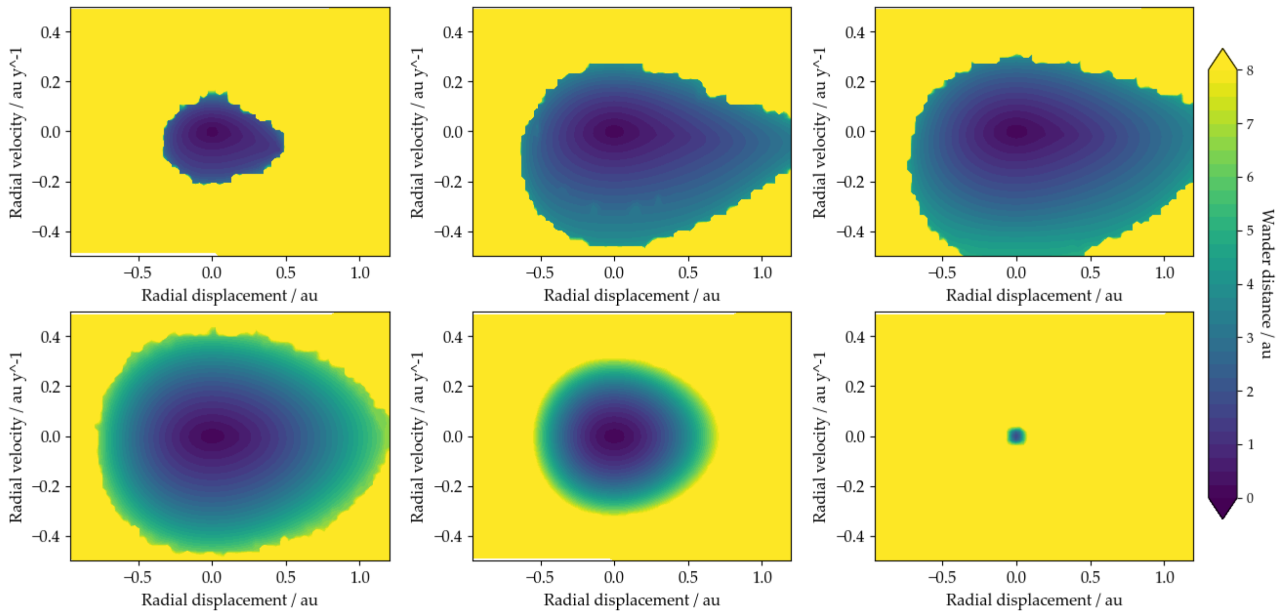

Figure 7: From top left to bottom right masses of Planet are 0.02, 0.01, 0.006 , 0.001, 0.0001 and 0 $M_\odot$. For each mass 2,400 asteroids were simulated. This shows that the size of the stable region appears to have a maximum at around $0.001 M_\odot$. The larger mass plots also appear to have a steeper boundary with fewer asteroids having larger wander distances. All plots show the stable region extending further outward radially than inward.

gated along the path of the Lagrange point. Velocities were uniformly random in the rotating frame, with a maximum speed of 0.1auy^{-1}. Limiting the velocity has a significant drawback, where an unstable radius may become stable only for larger initial velocities than were permitted. However this is unavoidable as only a finite number of asteroids can be simulated.

Fig.6 shows the correlation between wander distance and an asteroid's initial position for 12,000 asteroids. This shows the primary force driving stability is the relationship between radial displacement (r) and angular velocity ($\dot{\theta}$). This relationship developed quicker (for fewer simulated asteroids) than the other plots shown. The red line plotted represents:

$$r^2(\dot{\theta} + \Omega) = r_{L4}^2 \Omega, \tag{5}$$

where Ω is the angular velocity of the rotating frame and r_{L4} is the radius of the Lagrange point from the barycentre. This represents the asteroid having the same initial specific angular velocity as a mass stationary at L4.

Fig.6 also shows the lack of dependence on θ and $\dot{r}$ with the corresponding plot of these parameters showing no obvious correlation, implying that these parameters are less important for determining the stability of an asteroid.

4.4 The Effect of Planetary Mass

To investigate how the planet mass changes the stability of Trojan asteroids; the results discussed in section 4.3 are used. Accordingly, asteroids were simulated with an initial angular velocity as given by equation (5). Asteroids were placed with the same angular displacement as the Lagrange point leaving only r and $\dot{r}$ as free parameters. This was done to maximise the number of asteroids that were stable. To more directly compare different planet masses, asteroids were placed on a grid in r, $\dot{r}$ space so that the simulation for each mass is performed on the same data removing any variation, between masses, due to the random placement of the asteroids.

The wander distance in the r and $\dot{r}$ plane for different masses is shown in Fig.7. This shows the roughly egg-shaped stable regions around the zero radial displacement, zero radial velocity position. These plots show larger outward radii being more stable than inward radii, although it is worth noting that asteroids displaced radially inwards often remained within the orbit of the planet for a long time and so could be classified as stable but would no longer be classified as Trojan. The plots also appear to show more symmetry with decreasing mass. The maximum number of stable asteroids was observed for masses around $0.001 M_\odot$.

Fig.8 shows the number of stable asteroids varies with both the wander distance and mass of the Planet. This shows a decay in maximum wander distance with the mass of the planet which is consistent with the steep change in wander distance around the edges of the stable regions observed in Fig.7. Most interestingly, this plot shows that for specific masses there are fewer than expected stable asteroids, most notably at $m_p =$

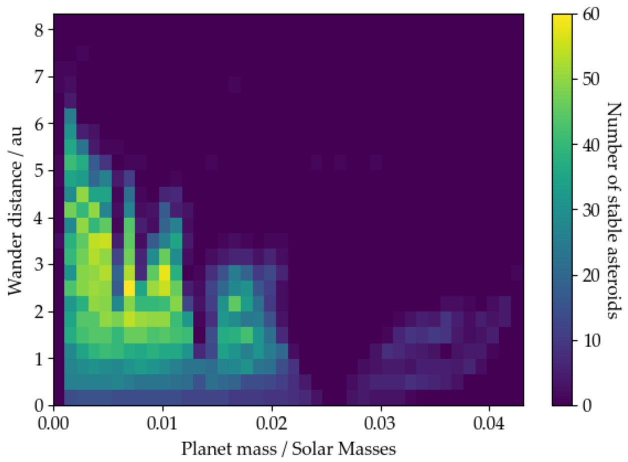

Figure 8: How the wander distance of stable asteroids changes with the mass of the planet. 1000 asteroids were simulated for each of 40 masses. This shows that the wander distance decreases with increasing mass, however, there are some masses where Trojan asteroids appear anomalously unstable. Particularly at $m = 0.025M_\odot$ where no stable asteroids were observed. The lack of asteroids on the left edge is representative of the very rapid decay of stability in this regime.

$0.025 \pm 0.001M_\odot$ where no stable asteroids were found. Beyond this value, some stable asteroids were observed with wander distance increasing with increasing mass. These features are remarkably similar to the results of R. Schwarz (2009)[7] and B. Erdi (2007)[8].

No stable asteroids were found beyond $m_p = 0.042M_\odot$. This is consistent with the work of S. Ciulli (2008)[9] which predicts a maximum stable mass of $m_p = 0.040M_\odot$. The difference observed is likely as a result of the simulation time of 800 orbital periods not being sufficient. Jupiter's mass ($m_J = 0.001M_\odot$) is approximately at the peak of the graph so would be expected to have a large stabilising influence compared with other two body systems with smaller ratios of m_p/m_s, hence may explain why Jupiter has over 2000 discovered Trojans.

5 Conclusions

In this investigation, a simulation of the CR3BP was created to investigate stability of Trojan asteroids by measuring the distance that the asteroids wandered from the Lagrange point L4. To this end, the primary driving force in the stability was found to be the specific angular momentum of the asteroid being close to the specific angular momentum of a mass stationary at the Lagrange point itself. In the radial and radial velocity, space stability was found to form egg-shaped regions which were more stable in the outwards radial direction than inwards. These plots would have benefited from more detailed analysis as other interesting behaviours were observed, however, due to limited resources and time this was not possible.

The stability against mass was also investigated and found to show a decrease in wander distance for increasing masses. No stable asteroids found at ratios of m_s to m_p of 0.025 ± 0.001 and no stable asteroids found above 0.042. Unfortunately, the resolution of these measurements is poor due to the limited number of asteroids that could be simulated.

Overall, I was able to create a capable simulation that successfully modelled the dynamics of Trojan asteroids and produced results consistent with other work in the field. From here, the Hilda asteroids represent an interesting extension which could easily be instigated using this simulation.

References

[1] F. Marzari, H. Scholl, C. Murray, and C. Lagerkvist, *Origin and Evolution of Trojan Asteroids*, pp. 725–738. 2002.

[2] E. Hairer, S. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I Nonstiff problems.* Berlin: Springer, second ed., 2000.

[3] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[4] M. Janson, "A Systematic Search for Trojan Planets in the Kepler Data," *The Astrophysical Journal*, vol. 774, p. 156, Sept. 2013.

[5] X.-Y. Wang, S.-P. Gong, and J.-F. Li, "A method for classifying orbits near asteroids," *Acta Mechanica Sinica*, vol. 30, p. 316–325, May 2014.

[6] G. Laughlin and J. E. Chambers, "Extrasolar trojans: The viability and detectability of planets in the 1:1 resonance," *The Astronomical Journal*, vol. 124, pp. 592–600, jul 2002.

[7] R. Schwarz and S. Eggl, "Dynamics of possible Trojan planets in binary systems-the 3 dimensional case," *Publications of the Astronomy Department of the Eotvos Lorand University*, vol. 20, pp. 135–144, June 2011.

[8] B. Érdi, I. Nagy, Z. Sándor, A. Süli, and G. Fröhlich, "Secondary resonances of co-orbital motions," *Monthly Notices of the Royal Astronomical Society*, vol. 381, p. 33–40, Sep 2007.

[9] S. Ciulli and C. Sebu, "Stable Lagrangian Points of Large Planets as Possible Regions where WIMPs could be Sought.," *Journal of Physics A: Mathematical and Theoretical*, vol. 41, p. 335201, July 2008. 11 pages.

Appendix A: Source Code

This section contains the source code. Included here is the main simulation code which is contained in Simulation.py. Also included here is an example of how the simulation code was implemented to generate asteroid data. The example given here is Mass_Calculator.py. This was used to calculate the data shown in Fig.7 and Fig.8.

Simulation.py

This is the file containing the core functionality of the simulation. This is used to calculate the paths of the asteroids as well as other quantities such as wander distance and the Hamiltonian.

```
#-------------------------------------------------------------------------------
# Purpose:     To calculate the paths of asteroids in the CR3BP

# BGN:         6946S
# Created:     21/02/2021
#-------------------------------------------------------------------------------

# Scientific Python
import numpy as np
from numpy import pi
from scipy.integrate import solve_ivp

# Optimisation Libraries
import cProfile
import multiprocessing as mp


class AsteroidPath:
    """A struct to contain information on the path of each asteroid."""
    def __init__(self, t, y):
        self.t = t
        self.rx = y[0]
        self.ry = y[1]
        self.vx = y[2]
        self.vy = y[3]
        self.v = y[2:].T
        self.r = y[:2].T

    def appened(self, path):
        """
        Appends the paths of one AsteroidPath instance to the end of another

        Parameters:

            path (AsteroidPath): path to add to end of the path contained in
                this instance of AsteroidPath.
        Returns:

            (AsteroidPath): The new asteroid path instance with the paths of
                the asteroid in the paths parameter added to the end.
        """
        return AsteroidPath(np.append(self.t, path.t),
                            np.array([np.append(self.rx, path.rx),
                                      np.append(self.ry, path.ry),
                                      np.append(self.vx, path.vx),
                                      np.append(self.vy, path.vy)]))

```

```

class CelestialBody:
    """A struct containing the information on a celestial body."""
    def __init__(self, position, mass):
        self.position = np.array(position)
        self.mass = mass


class Simulation:
    """
    A class to simulate the motion of asteroids in a sun and planet system in
    the planar circular restricted three-body problem (CR3BP).

    This class also includes functions to calculate other parameters of the
    simulation such as the Hamiltonian and the maximum wander distance of
    asteroids.

    Coordinates used here are cartesian with the origin at the barycentre of
    the planet and sun. The length unit is au and masses are in solar masses.
    Time is measured in earth years and therefore, G = 4*pi^2.

    The planet is placed along the positive y which is in the upwards direction
    in the drawing plane. Positive x is to the right. Angles are measured from
    the planet (hence from the y axis with positive angles being clockwise).
    Hence to convert from a coordinate to an angle one should use arctan(x, y).

    Methods:

        __init__: Constructor for the simulation. This sets up the parameters
            of the simulation and generates the asteroids to be simulated.

        run: Runs the simulation. This generates the paths of the asteroids.

        hamiltonian: Calculates the Hamiltonian of the asteroids in the
            rotating frame. The Hamiltonian is calculated as is a conserved
            quantity of the simulation.

        wander_distance: Calculates the maximum distance that individual
            asteroids have wandered from a chosen Lagrange point.

        remove_asteroid: Removes an asteroid from the system. Often useful if
            an asteroid has left the area of interest and therefore does not
            need to be simulated further.

        rotation_transformation: Preforms a transformation to move into a
            different rotating reference frame than the one of the planet. This
            transformation is NOT saved within the instance of the Simulation.

        _equations_of_motion: The equations of motion of the system used to
            calculate asteroid paths.

        _gravity_equation: Calculates the acceleration due to gravity at a
            point in space due to a gravitational body. Note the result needs
            to be multiplied by the gravitational constant (G) to get the true
            force on the body.

        _potential_energy: Calculates the potential energy at a point in space
            due to a gravitational body. Note the result needs to be multiplied
```

```
            by the gravitational constant (G) to get the true force on the
            body.

        _ODE: Uses SciPy integrator to calculate the paths of a given
            asteroid(s).

        _initial_position_to_index: Converts the initial position of an
            asteroid to the index of that asteroid in the other arrays in the
            simulation.

        _check_if_sim_performed: Check if a simulation has been performed or
            not.
    """
    # Physical Constants
    G = 4 * pi**2
    SUN_MASS = 1

    def __init__(self, asteroid_generator, N, *args, planet_mass=0.001, R=5.2):
        """
        Creates and initialises the parameters of the simulation for use
        later. This also generates the asteroids for the simulation.

        Parameters:

            asteroidGenerator (function): The function used to generate the
                initial position and velocities of the asteroids. The calling
                signature is asteroidGenerator(N, Simulation, *args). Where N
                is the number of asteroids, Simulation an instance of the
                Simulation class - This is used to extract specific
                parameters of each simulation. *args are any additional
                arguments required, for example the maximum velocity permitted.
                asteroidGenerator must return np.array of size (4,N).
                The rows must represent [rx, ry, vx, vy] for the initial
                positions and velocities of each asteroid in cartesian
                coordinates. If N asteroids are not generated, then the
                simulation will run with the number of asteroids generated but
                an error message will be displayed.

            N (int): Number of asteroids to attempt to generate.

            *args: Additional parameters (if needed) for the asteroid
                generation function.

            planet_mass (float): Mass of the planet. The default value is 0.001
                for Jupiter.

            R(float): Radius of the orbiting planet from the sun. The default
                value is 5.2 for Jupiter.
        """
        if N <= 0:
            raise Exception('The simulation must have at least 1 asteroid.')
        if int(N) != N:
            raise Exception('N must be an integer.')
        if planet_mass < 0:
            raise Exception('The planet mass must be positive.')
        if R < 0.005:
            raise Exception('R must be larger than the radius of the sun.')

```

```
        self.R = R
        self.planet_mass = planet_mass
        self.omega = (Simulation.G*(Simulation.SUN_MASS+planet_mass)/R**3)**0.5
        self.T = 2*pi / self.omega  # Period of oscillation

        # Move barycentre to origin.
        # r_sun is negative corresponding to the sun being below the barycentre
        self.r_sun = -R * planet_mass / (planet_mass + Simulation.SUN_MASS)
        self.r_planet = self.r_sun + R

        # Positions of the Lagrange points.
        self.L4 = [-R*np.sin(pi/3), R*np.cos(pi/3)+self.r_sun]
        self.L5 = [R*np.sin(pi/3), R*np.cos(pi/3)+self.r_sun]

        # For convenience when generating asteroids
        self.r_L4 = (self.L4[0]**2 + self.L4[1]**2)**0.5
        self.theta_L4 = np.arctan2(self.L4[0], self.L4[1])
        self.r_L5 = (self.L5[0]**2 + self.L5[1]**2)**0.5
        self.theta_L5 = np.arctan2(self.L5[0], self.L5[1])

        # Create the celestial bodies with a gravitational field.
        sun = CelestialBody([0, self.r_sun], Simulation.SUN_MASS)
        planet = CelestialBody([0, self.r_planet], planet_mass)
        self._bodies = np.array([sun, planet])

        # Generate asteroids.
        self.initial_positions = np.array(asteroid_generator(N, self, *args))

        if len(self.initial_positions) != N:
            print('Failed to produce', N, 'asteroids. '
                  'Simulation will run with', len(self.initial_positions),
                  'asteroids instead.')

        # If asteroidGenerator does not return N asteroids.
        self.N = len(self.initial_positions)

        # Initialise for later
        self.paths = None
        self._t_start = None
        self._t_end = None
        self._max_step = None

    def run(self, periods, extend=False,
            min_steps_per_orbit=12, threaded=True):
        """
        Runs the simulation.

        This function uses the scipy.integrate.solve_ivp ODE solver to solve
        for the path of each asteroid for the number of periods specified.

        Parameters:

            periods (float): The number of orbital periods of the planet to run
                the simulation for.

            extend (bool): Whether to run the simulation as an extension to
                the simulation if the simulation has already been run. This
                will start the simulation at the time where it previously
```

```
                finished. The default value is False.

            min_steps_per_orbit (int): The minimum number of evaluations per
                orbit used when calculating the paths of each asteroid. The
                default value is 12 which is provides a good balance for speed
                and accuracy.

            threaded (bool): Whether to run the simulation using
                multithreading. This significantly improves speed for large
                data sets although maybe slower for small data sets. The
                default value is True.

        Reutrns:

            (A list of class AsteroidPath dim(Simulation.N)): A list of the
                paths of each asteroid. The path information is contained
                within the class AsteroidPath for ease of extraction and
                plotting. This is also saved within the class so can be
                accessed later if required.
        """
        self._max_step = self.T / min_steps_per_orbit

        if extend:
            # Get end positions for each asteroid.
            initial_positions = [[asteroid.rx[-1],
                                  asteroid.ry[-1],
                                  asteroid.vx[-1],
                                  asteroid.vy[-1]] for asteroid in self.paths]
            self._t_start = self._t_end

        else:
            initial_positions = self.initial_positions
            self._t_start = 0

        self._t_end = periods * self.T + self._t_start

        # Run the simulation for each asteroid.
        if threaded:
            with mp.Pool(processes=mp.cpu_count()) as pool:
                paths = np.array(pool.map(self._ODE, initial_positions))

                # Handle memory issues.
                pool.close()
                pool.join()
        else:
            paths = np.array([self._ODE(asteroid)
                              for asteroid in initial_positions])

        # Reformat result to make plotting more intuitive.
        paths = [AsteroidPath(asteroid.t, asteroid.y) for asteroid in paths]

        # If extended then add paths to the end of the old ones.
        if extend:
            self.paths = [self.paths[n].appened(asteroid_path)
                          for n, asteroid_path in enumerate(paths)]
        else:
            self.paths = paths

```

```python
        return self.paths

    def remove_asteroid(self, initial_position):
        """
        Function to remove an asteroid from the simulation.

        Parameters:
            initial_position (np.array dim(4)): The initial position
                (in format [x, y, vx, vy]) of the asteroid to be removed.
                If two asteroids have the same initial position, then only the
                first will be removed.
        """
        self._check_if_sim_performed()
        '''
        Cannot rely on the positions within the array remaining constant
        throughout operation therefore cannot provide the index so need to use
        the initial position to find the asteroid. If two asteroids had the
        same initial position then they would have evolved the same way so it
        should not matter which is removed.
        '''
        n = self._initial_position_to_index(initial_position)

        # Delete the asteroid
        self.initial_positions = np.delete(self.initial_positions, n, 0)
        self.paths = np.delete(self.paths, n, 0)

    def hamiltonian(self, initial_position):
        """
        Calculates the Hamiltonian of the asteroids in the simulation.
        The Hamiltonian is calculated instead of the energy because in the
        rotating frame energy is not conserved whereas the Hamiltonian is.

            H = E - 1/2 (w x r)^2
        where:
            E = 1/2 v^2 + U(r)

        Parameters:

            initial_position (np.array dim(4)): The initial position
                (in format [x, y, vx, vy]) of the asteroid to calculate the
                Hamiltonian for.

        Returns:

            (np.array dim(2, n)): First column corresponds to times in years.
                The second column corresponds to the Hamiltonian of the
                asteroid at the time in the first column.
        """
        self._check_if_sim_performed()
        '''
        The ODE solver can take variable step sizes.
        This would mean that different asteroids would take a different
        number of steps to reach t_max. It is therefore difficult to sum
        the individual asteroid's energies over time. Or even return all the
        energies as lists as these lists would be of different lengths
        therefore this is implemented to only look at one asteroid at a time.
        '''
        # Find index of asteroid in question
```

```
        n = self._initial_position_to_index(initial_position)
        asteroid = self.paths[n]

        H = np.zeros(len(asteroid.t))

        for dt in range(len(asteroid.t)):

            v, r = asteroid.v[dt], asteroid.r[dt]

            # Correction for rotating frame (tangential velocity).
            v_T = np.cross([0, 0, self.omega], r)

            PE = - Simulation.G * sum(map(self._potential_energy,
                                          (r, r), self._bodies))

            H[dt] = 1/2 * v.dot(v) + PE - 1/2 * v_T.dot(v_T)

        return np.array([asteroid.t, H])

    def wander_distance(self, L5=False):
        """
        Calculates the maximum distance that each asteroid has travelled
        from a given Lagrange point during the simulation.

        Parameters:

            L5 (bool): A parameter to determine which Lagrange point to measure
                distances from. False means to calculate from L4 (the Lagrange
                point on the leading side of the planet, this is on the left
                for this simulation). True means calculating distances from L5
                (on the trailing side of the planet). The default value is
                False.

        Returns:

            (np.array dim(5,N)): A list of the initial positions and the
                maximum wander distance for each of the asteroids in the
                simulation. The format is: [rx, ry, vx, vy, wd], with wd being
                the wander distance. The other parameters are the initial
                displacement and velocity of the asteroid.
        """
        self._check_if_sim_performed()

        if L5:
            L = self.L5
        else:
            L = self.L4
        '''
        Loop over each asteroid and find maximum distance from L
        then add this distance and the initial position of that asteroid
        to the array.
        '''
        return np.array([[self.initial_positions[n][0],
                          self.initial_positions[n][1],
                          self.initial_positions[n][2],
                          self.initial_positions[n][3],
                          max(np.sqrt(np.sum((ast.r-L)**2, axis=-1)))]
                         for n, ast in enumerate(self.paths)])
```

```python

    def rotation_transformation(self, omega):
        """
        Transforms the result of the simulation into a different rotating
        frame.

        The result of this is NOT saved to the instance of the simulation.
        Therefore the Simulation.paths parameter will always be in the planet's
        rotating frame. This is done to avoid confusion.

        Parameters:

            omega (float): Angular velocity of the output frame. This angular
                velocity should be given relative to an inertial -
                non-rotating frame. Therefore using an angular velocity of the
                planet in the simulation (self.omega) is therefore the same as
                no transformation and the function will simply return the same
                as self.paths.

        Returns:

            (list of class (AsteroidPath) dim(Simulation.N)): The paths of the
                asteroids in the new rotating frame formatted as a list of
                AsteroidPath classes.
        """
        self._check_if_sim_performed()

        # Subtract the current rotation of the simulation.
        omega -= self.omega

        # For simplicity
        array = np.array

        '''
        Performs the transformation for each time and each asteroid the
        resultant transformed coordinates are then used to create new
        AsteroidPath classes with the transformed data. This is simply a
        standard rotating frame transformation for velocity and position.
        '''
        return [AsteroidPath(ast.t,  # No change in time coordinate.
                             array([[ast.rx[n] * np.cos(-omega * t) -
                                     ast.ry[n] * np.sin(-omega * t),
                                     ast.rx[n] * np.sin(-omega * t) +
                                     ast.ry[n] * np.cos(-omega * t),
                                     ast.vx[n] +
                                     np.cross([0, 0, omega], ast.r[n])[0],
                                     ast.vy[n] +
                                     np.cross([0, 0, omega], ast.r[n])[1]]
                                    for n, t in enumerate(ast.t)]).T)
                for ast in self.paths]

    def _equations_of_motion(self, t, y):
        """
        Calculates the right-hand side of the equations of motion of the
        in a frame rotating at the velocity of the planet.

        This is solely intended to be used as the fun parameter for
        scipy.integration.solve_ivp.
```

```

        Parameters:

            t (float): Time of evaluation, required by solve_ivp.

            y (np.array(4)): Position to evaluate equations at. This is a
                vector with components [rx,ry,vx,vy] corresponding to both
                displacement and velocity of an asteroid.

        Returns:

            (list(4)): The derivatives of y in the rotating frame.

        """
        r = y[:2]
        vx, vy = y[2:]
        '''
        Calculate the acceleration in the rotating frame. This is the sum
        of the gravitational force and centripetal force. The Coriolis force
        is excluded here for increased performance.
        G is also multiplied here as is faster compared to multiplying
        within the gravity function.
        '''
        accel = (Simulation.G * sum(map(self._gravity_equation,
                                        (r, r), self._bodies)) +
                 self.omega**2*r)

        return [vx,                            # drx / dt
                vy,                            # dry / dt
                accel[0] + 2*self.omega*vy,    # dvx / dt, coriolis force added
                accel[1] - 2*self.omega*vx]    # dvy / dt

    def _gravity_equation(self, r, body):
        """
        Calculates the gravitational acceleration at position r (np.array(2))
        created by body (CelestialBody class). The gravitational constant
        (G) is omitted for performance reasons.
        """
        r = body.position - r  # Displacement from body
        return body.mass * r / r.dot(r)**1.5

    def _potential_energy(self, r, body):
        """
        Calculates the gravitational potential energy at position r
        (np.array(2)) created by body (CelestialBody class). The gravitational
        constant (G) is omitted for performance reasons.
        """
        r = body.position - r  # Displacement from body
        return body.mass / r.dot(r)**0.5

    def _ODE(self, asteroid):
        """
        Runs the scipy.integrate.solve_ivp using the DOP853 algorithm to
        calculate the path of asteroid. This is separated from Simulation.run
        to allow multithreading.

        Parameters:

```

```
              asteroid (np.array dim(4)): The initial position of the asteroid to
                  be simulated. The components of asteroid are [x, y, vx, vy].
        """
        return solve_ivp(self._equations_of_motion,
                         [self._t_start, self._t_end],
                         asteroid,
                         method='DOP853',
                         max_step=self._max_step,
                         dense_output=True)

    def _initial_position_to_index(self, ip):
        """
        Finds the index of an asteroid in the initial positions and hence the
        paths arrays. This is done using the the initial position of the
        asteroid given by the ip (np.array(4)) parameter. The components of
        ip are [x, y, vx, vy].
        """
        return np.nonzero(np.all(self.initial_positions == ip, axis=-1))[0][0]

    def _check_if_sim_performed(self):
        """
        Check if a Simulation.run has been performed and raise exception if
        not.
        """
        if self.paths is None:
            raise Exception('No simulation performed. Use Simulation.run() '
                            'first.')
```

Mass Calculator.py

This is an example of how Simulation.py was used to calculate asteroid wander distances. This code was used to calculate how the wander distances varied with different planet masses.

```
#--------------------------------------------------------------------------------
# Purpose:     Calculate wander distances for different masses of planets
#              results are saved to a file.

# BGN:         6946S
# Created:     01/04/2021
#--------------------------------------------------------------------------------
import numpy as np
import csv, os

from Simulation import Simulation
from AsteroidGenerators import AstGen

FILE_NAME = '--.csv'

SPATIAL_LIM = 1.2   # Spatial size of anulus.
VEL_LIM = 0.5       # Maximum speed of asteroids.

N_AST = 1000        # Number of asteroids to simulate for each mass.
WD_LIM = 8          # Wander distance where an asteroid is not Trojan.

T_MAX = 800         # Total length of simulation / periods.
START_INTERVAL = 10 # Length of initial run to remove highly unstable asteroids.
T_INTERVAL = 79     # Time intervals to check wander distance after initial run.

N_MASS = 45         # Number of masses to simulate.
```

```
MASSES = np.linspace(0.042, 0, N_MASS) # Masses to check.


def calculate():
    """
    Calculate the wander distances of asteroid for various masses and save
    the results to a file.
    """

    for mass in MASSES:

        print(mass)

        # Container of the wander distance of the asteroids
        wander_distances = np.empty((0,5))

        # Create simulation
        sim = Simulation(AstGen.uniform_r_correct_angular_velocity,
                         N_AST,
                         SPATIAL_LIM,
                         VEL_LIM,
                         planet_mass=mass)

        # Run one short sim to remove all the initially unstable asteroids
        sim.run(START_INTERVAL)

        for times in range(START_INTERVAL, T_MAX+1, T_INTERVAL):

            intermediate_wander_distances = sim.wander_distance()
            for asteroid in intermediate_wander_distances:

                # Remove asteroids that are no longer Trojan
                if asteroid[4] > WD_LIM:
                    wander_distances = np.append(wander_distances,
                                                 [asteroid],
                                                 axis=0)
                    sim.remove_asteroid(asteroid[:4])

            # If no asteroids left then leave the loop
            if len(sim.paths) == 0 :
                break

            # Check number of remaining stable asteroids.
            print('After', times, 'periods', len(sim.paths), 'asteroids remain')

            if times < T_MAX:
                sim.run(T_INTERVAL, True)

        # Add remaining asteroids
        intermediate_wander_distances = sim.wander_distance()

        if len(intermediate_wander_distances) != 0:
            wander_distances = np.append(wander_distances,
                                         intermediate_wander_distances,
                                         axis=0)


        # Write data to file
```

```
        with open(FILE_NAME, 'a', newline='') as writefile:
            writer = csv.writer(writefile, delimiter=',')

            # Write header only once per file.
            if os.path.getsize(FILE_NAME) == 0:
                writer.writerow([N_MASS, N_AST])

            # Write header of each mass for the specific simulation parameters.
            writer.writerow(['--', sim.R, sim.omega,
                             sim.L4[0], sim.L4[1], mass])

            writer.writerows(wander_distances)


if __name__ == '__main__':
    calculate()
```

Here the relevant asteroid generation function used in the code above is included for completeness. This is an extract from AsteroidGenerators.py which contains other asteroid generation functions. The full listing of which has not been included for brevity.

```
def random_r_correct_angular_velocity(N, sim, r_lim, vel_lim):
    """
    Creates a random distribution of asteroids across a radial range of
    -0.8*r_lim to r_lim centred on L4, with radial velocity range -vel_lim
    to vel_lim. Asteroids are placed at the same angle as L4 and their
    angular velocity is such that the angular momentum of the asteroid
    matches that of a mass stationary at the Lagrange point L4.
    """
    asteroids = np.zeros((N, 4))

    for ast in range(N):
        '''
        The 0.8 included here is to limit the inward r as the plots
        are asymmetric in the r axis.
        '''
        r = uniform(-r_lim*(0.8), r_lim) + sim.r_L4
        theta = sim.theta_L4
        v_r = uniform(-vel_lim, vel_lim)
        v_theta = sim.omega * (sim.r_L4**2 - r**2) / r

        asteroids[ast] = [r*np.sin(theta),
                          r*np.cos(theta),
                          v_r*np.sin(theta) - v_theta*np.cos(theta),
                          v_r*np.cos(theta) + v_theta*np.sin(theta)]

    return asteroids
```