Part II Computational Physics: Trojan Asteroids

Blind Grade Number: 6899V

April 27, 2020

Abstract

This report summarises the development and results of a simulation of Trojan asteroid systems in Python 3, focusing on the behaviour of asteroids perturbed from the stable Lagrangian points. The maximum displacement of an asteroid perturbed about the stable Lagrangian points was found to be quadratic in the size of the perturbation in both position and velocity space, and perturbations in the z direction gave rise to approximately harmonic oscillations. The system was also found to be unstable for planet-Sun mass ratios greater than 0.043, consistent with previous analytic results.

(Word count: 2987)

1 Introduction

A Lagrangian point is a point in space where the gravitational forces of two large bodies balance the centrifugal force felt by a third, much smaller body. This is an equilibrium point where, for example, a spacecraft could be parked in order to make observations. A planetary system has 5 Lagrangian points where a third much smaller mass can be placed. $\mathbf{L}_1$, $\mathbf{L}_2$ and $\mathbf{L}_3$ are collinear with the axis of the massive bodies, and are unstable. $\mathbf{L}_4$ and $\mathbf{L}_5$ form equilateral triangles with the positions of the massive bodies in the orbital plane, and are stable.

The Greek and Trojan asteroids are the groups of asteroids in elongated, curved regions around $\mathbf{L}_4$ and $\mathbf{L}_5$ respectively in the Sun-Jupiter planetary system, known collectively as the Trojans. They are named from Greek mythology after figures of the Trojan War, Achilles was the first to be discovered by Max Born in 1906 [1]. Other planets, including Mars [2] and Neptune [3], have since been observed also to have Trojans.

The project was to write a program to produce numerical solutions for the system, then use it to investigate the behaviour of asteroids perturbed about the stable Lagrangian points, and the effect of varying the mass of the planet. Since the behaviour around $\mathbf{L}_4$ and $\mathbf{L}_5$ should be symmetrical, I used $\mathbf{L}_4$ for all the investigation in the rotating frame.[4]

As Jupiter is not unique in having Trojan asteroids, the smaller mass will be referred to as "the planet" with mass M_P at a distance R from the Sun. Unless otherwise stated however, Jupiter's properties will be used for the simulations.

2 Theoretical Background

2.1 Solar system units

For this problem I worked in solar system units, where the mass of the Sun $M_\odot$ is taken as unit mass, the astronomical unit is the unit distance and one year unit time. In these units the gravitational constant is $G = 4\pi^2$. I then took the mass of Jupiter to be $M_\mathrm{J} = 0.001 M_\odot$, the Sun-Jupiter distance to be $R_0 = 5.2\,\mathrm{au}$, and the masses of the asteroids to be negligible.

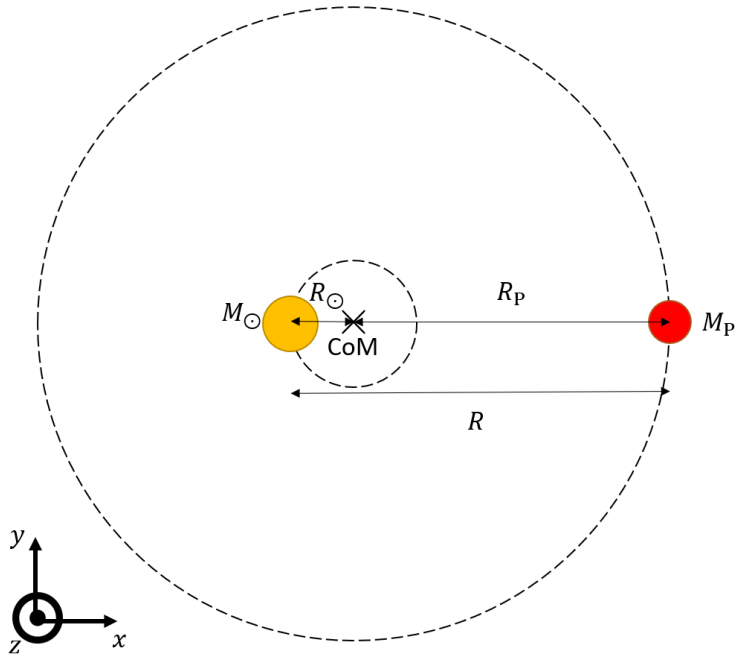

Figure 1: Cartoon showing the distances of the Sun and planet from the origin at the centre of mass, where $R_\mathrm{P} = R\frac{M_\mathrm{P}}{M_\odot + M_\mathrm{P}}$ and $R_\odot = R - R_\mathrm{P}$. Also shown are the Cartesian coordinate directions in both frames at time=0.

2.2 Coordinate system

A coordinate system needed to be chosen in order to represent the system for the simulations. The origin was placed at the centre of mass (CoM), and at the start of the simulation the x direction is aligned with the sun-planet axis, then the y and z directions being normal to this in the plane and out of the plane respectively, as shown in figure 1. I then defined the stationary frame where this coordinate system remained fixed, and the rotating frame where the axes remained fixed relative to the Sun and planet. The axes would be coincident at the start of the simulation.

2.3 Finding the Lagrangian points

I approximated the orbits of the Sun and the planet to be circular about the CoM. Mathematically, this orbit can be considered the same as an object of reduced mass $\mu = \frac{M_\odot M_\mathrm{P}}{M_\odot + M_\mathrm{P}}$ orbiting a fixed mass with the same gravitational force between them.[5] Balancing gravitational and centrifugal forces gives an angular frequency of the orbit ω:

$$\mu\omega^2 R = \frac{GM_\odot M_\mathrm{P}}{R^2} \quad \Longrightarrow \quad \omega = \sqrt{\frac{G(M_\odot + M_\mathrm{P})}{R^3}} \tag{1}$$

The effective potential in the rotating frame has saddle points at $\mathbf{L}_1$, $\mathbf{L}_2$ and $\mathbf{L}_3$, and local maxima at $\mathbf{L}_4$ and $\mathbf{L}_5$, which would suggest all five points are unstable. This is where the Coriolis force comes in, curving the paths of object that deviate from $\mathbf{L}_4$ and $\mathbf{L}_5$ to help prevent their escape. This allows asteroids to collect around these locations.

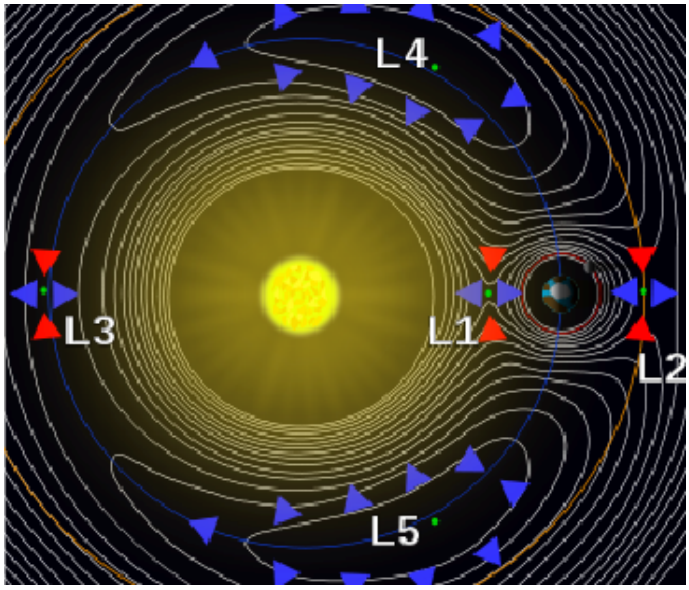

Figure 2: Diagram showing effective potential in the rotating frame, and the location of the Lagrangian points. Arrows indicate direction of decreasing potential. Figure adapted from Greenspan. [6]

In the stationary frame, the acceleration $\boldsymbol{a}$ of the asteroid is the due to the gravitational forces from the Sun and planet:

$$\boldsymbol{a}_{\text{stationary}} = -G\left[\frac{M_{\odot}(\boldsymbol{r}-\boldsymbol{r}_{\odot})}{|\boldsymbol{r}-\boldsymbol{r}_{\odot}|^3} + \frac{M_{\text{P}}(\boldsymbol{r}-\boldsymbol{r}_{\text{P}})}{|\boldsymbol{r}-\boldsymbol{r}_{\text{P}}|^3}\right] \tag{2}$$

In the rotating frame the Coriolis and centrifugal virtual forces need to be included too:

$$\boldsymbol{a}_{\text{rotating}} = \boldsymbol{a}_{\text{stationary}} - 2\boldsymbol{\omega}\times\boldsymbol{v} - \boldsymbol{\omega}\times(\boldsymbol{\omega}\times\boldsymbol{r}) = \boldsymbol{a}_{\text{stationary}} + \omega\begin{pmatrix} 2v_{\text{y}} + \omega r_{\text{x}} \\ -2v_{\text{x}} + \omega r_{\text{y}} \\ 0 \end{pmatrix} \tag{3}$$

Here the position and velocity vectors now refer to those in the rotating frame, and the second equality arises as $\boldsymbol{\omega} = \omega\hat{\boldsymbol{z}}$ and then explicitly calculating the cross products. The Lagrangian points can be then found by setting $\boldsymbol{a}_{\text{rotating}} = 0$ and $\boldsymbol{v} = 0$, resulting in $\mathbf{L}_4$ and $\mathbf{L}_5$ at:

$$\mathbf{L}_{4/5} = \begin{pmatrix} \frac{R}{2} - R_{\odot} \\ \pm\frac{\sqrt{3}}{2}R \\ 0 \end{pmatrix} \tag{4}$$

in the rotating frame, where $R_{\odot}$ is the distance from the CoM to the Sun.

For such a three body system with asteroid mass m, Gascheau found that $\mathbf{L}_4$ and $\mathbf{L}_5$ are only stable given the condition [1]:

$$\frac{(M_{\odot}+M_{\text{P}}+m)^2}{M_{\odot}M_{\text{P}} + M_{\odot}m + M_{\text{P}}m} > 27 \tag{5}$$

Taking m to be negligible, $M_{\text{P}} < M_{\odot}$ and using solar system units, equation 5 reduces to:

$$M_P < \frac{25 - 3\sqrt{69}}{2} \approx 0.04 \tag{6}$$

This condition is actually satisfied for all planet-Sun and satellite-planet pairs in the solar system except Charon-Pluto, though the majority don't possess Trojan objects. [4]

2.4 Orbit shape

The stable orbits near the Lagrangian points are known as tadpole orbits due to their shape. These arise as a low frequency elongated elliptical motion about the Lagrangian point, then a higher frequency ellipse about that. Larger amplitude tadpole orbits begin to curve around the planet's orbital circle, eventually resulting in a horseshoe orbit which encompasses $\mathbf{L}_3$, $\mathbf{L}_4$ and $\mathbf{L}_5$. Yet further from the Lagrangian points are passing orbits, which circulate either inside or outside the planetary circle. [4]

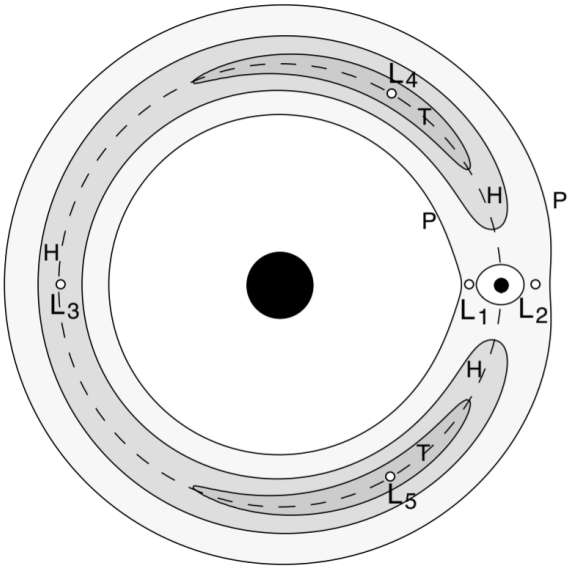

Figure 3: The location of the Lagrangian points. The large and small filled circles represent the sun and planet respectively. The areas labelled T, H and P are regions of tadpole, horseshoe and passing orbits respectively. [4]

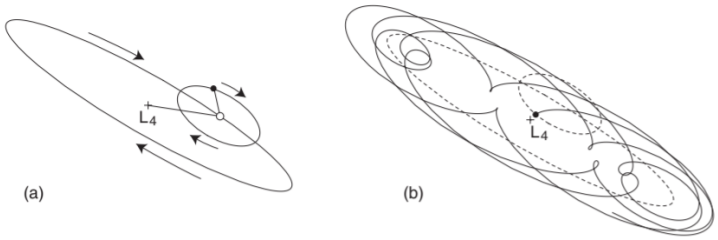

Figure 4: (a) The tadpole motion has two separate contributions, low frequency elongated elliptical motion about $\mathbf{L}_4$, then a high frequency ellipse about that. (b) The resulting trajectory. [4]

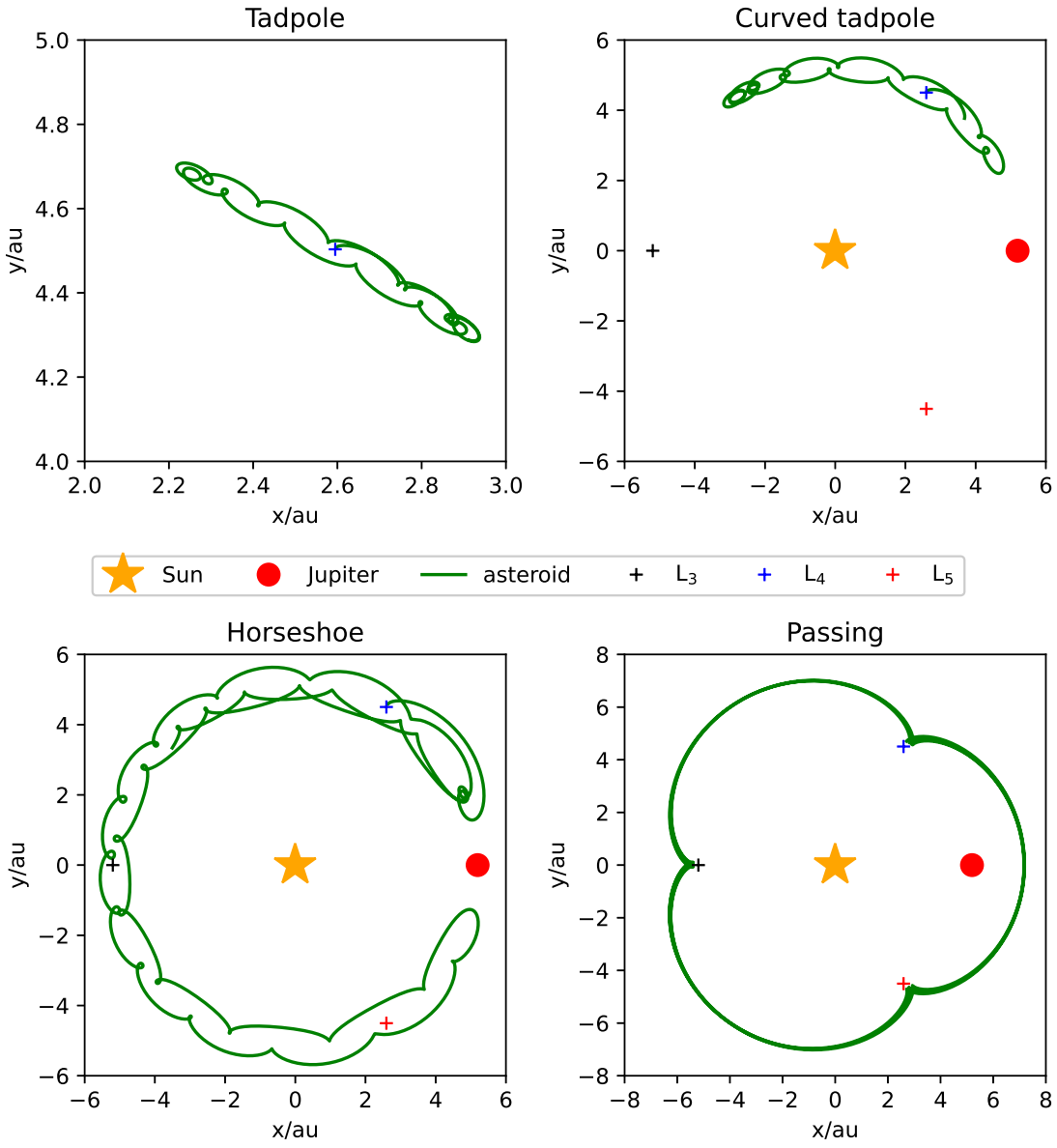

Figure 5: Simulations resulting in tadpole, curved tadpole, horseshoe and passing orbits. All four started with zero velocity, starting at $1.001\mathbf{L}_4$, $1.01\mathbf{L}_4$, $1.02\mathbf{L}_4$ and $1.045\mathbf{L}_4$ respectively.

2.5 z oscillations

Perturbations in z can cause oscillations. Considering the forces in the z-direction in figure 6 for small angles, oscillations at $\mathbf{L}_4$ should have angular frequency given by:

$$\omega_z = \sqrt{\frac{G(M_\odot + M_\mathrm{P})|\mathbf{L}_4|}{R^4}} = \omega\sqrt{\frac{|\mathbf{L}_4|}{R}} \tag{7}$$

This results in a time period $T_z = \frac{2\pi}{\omega_z}$ of 11.8549 years, slightly longer than the orbit time of 11.8519 years since $|\mathbf{L}_4|$ is slightly smaller than R.

Displacement in the z direction will introduce movement in the x-y plane as the centrifugal force isn't balanced by the slightly weaker gravity higher up. This will cause the asteroid to begin to move outwards, which brings in the Coriolis force, complicating the system beyond simple harmonic behaviour.

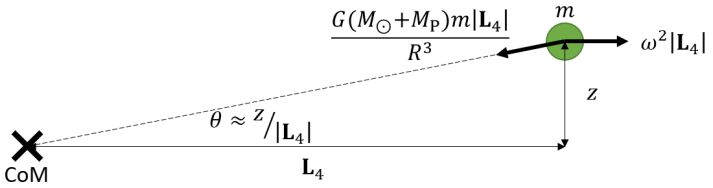

Figure 6: Cartoon showing the forces resulting in z oscillations, where r is the . Balancing the forces, using Newton II and taking $|\mathbf{L}_4| \approx R$ results an angular frequency of approximately $\sqrt{\frac{G(M_\odot+M_P)}{R^3}}$, the same as the planet's orbit.

3 Analysis and Implementation

The simulation was implemented in Python 3, making use of the NumPy and SciPy libraries for the main simulation code, and Matplotlib for plotting results. The code formatter Black was used throughout for consistent style.

Using the solar system units was beneficial as it kept numerical values around $\mathcal{O}(1)$, which helped avoid issues with underflow or (more likely) overflow that would more likely arise using SI for astronomical quantities.

I wanted to produce simulations in both the stationary frame and the frame rotating with the planet and Sun, so that I could compare their performance and then use the better one for the experiments. In order to be able to do this, there would either need to be a Boolean passed to every function to tell it which frame was being used, or two separate sets of code. After having a go at both, it became clear that the first method would require almost every function would need to have essentially two definitions inside. Therefore, it was most sensible to tackle the two frames separately.

To avoid duplicating the values of the physical parameters used by the two simulations, a separate file `constants.py` containing these was created and imported by both the simulation and plotting files. Initially, this file contained G, $M_\odot$ M_J, R_0, and all the other quantities that could be derived from them. This included distances of the Sun and Jupiter from the CoM, and the positions of the Lagrangian points. However, this meant the code was essentially stuck simulating the Jupiter system, so a different approach was necessary. I decided instead to create a class for each simulation, which would be instantiated with the planet mass and planet-Sun distance which default to the Jupiter values from `constants.py`. The derived quantities were calculated in the simulations' constructors, and getters added for the plotting files to access them. Correspondingly, `constants.py` was reduced to just the first four quantities.

In addition to the `trajectory` method to return the asteroid path for given initial conditions, both classes had a `wander` method to return the greatest distance an asteroid wanders from the equilibrium point. This method was used to quantify the stability of an asteroid's orbit. As wander will depend on the duration, the wanders were all for simulations of 100 planetary orbits, sampled at 100 points per year.

In the stationary frame, the motion of the Sun and Jupiter needed to be included. Taking their orbits to be circular about their CoM meant the analytic solutions could be used, rather than adding unnecessary complexity and error by solving their behaviour numerically.

To generate the plots many simulations needed to be run. To reduce the running time simulations were run in parallel, making use of the multitasking ability of modern processors. The `Pool` class from the `multiprocessing` module was used, using its `map` and `starmap` methods for 1D and 2D inputs respectively. Both map methods worked with single-argument functions, but the `trajectory` and `wander` methods required multiple arguments. This could be gotten around by combining the arguments into an array of tuples, or using a wrapper to take care inputting the arguments that weren't being varied. I chose to use wrappers

for better clarity and ease for making changes. While developing I allowed the `Pool` object to decide how many threads to use in to allow my laptop to perform other tasks, then when it came to timing in order to assess performance it was set to use the full number of threads available. My laptop has an Intel i7 processor with 4 dual-threaded cores, which `multiprocessing` sees as 8 cores. However this won't correspond to an 8-fold time reduction, as the threads compete for other resources, such as memory access.

The vectors in the simulation were represented by NumPy arrays. This meant equations 2 and 3 could be translated directly using NumPy's vectorised functions. This had two main benefits: it is easily readable, and vectorisation allowed for a speed increase as operations can be carried out in parallel. For 2D arrays representing all components of a vector with increasing time, the convention [component, time] was used throughout, which was easier for plotting and matched the output of the numerical solver. However, in order to add a single vector to every vector represented in the array, the array had to be transposed to allow NumPy broadcasting to work.

To generate the solutions, I used the `solve_ivp` function from the sub-package `scipy.integrate`. This required the initial $\boldsymbol{r}$ and $\boldsymbol{v}$ to be bundled together into a 6-vector $\boldsymbol{y}$, and the second order differential equations to be rewritten as an array of first order differential equations for $\dot{\boldsymbol{y}}$:

$$\dot{\boldsymbol{y}} = \begin{pmatrix} \dot{r_x} \\ \dot{r_y} \\ \dot{r_z} \\ \dot{v_x} \\ \dot{v_y} \\ \dot{v_z} \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ a_x \\ a_y \\ a_z \end{pmatrix} \tag{8}$$

`solve_ivp` has several integration methods provided, and can use any child of `OdeSolver`. However, I decided to restrict selection to the built-in options as these should all be well tested and reliable. The methods must satisfy three requirements: asteroids placed in the L_4 and L_5 orbits should stay there in both the stationary and rotating frames, and energy should be conserved in the stationary frame.

The RK45 method was unsuitable as Lagrangian asteroids spiraled in towards the Sun in the stationary frame. For the rest, I compared the wander for starting at the Lagrangian points in both frames for 100 orbits. The results are shown in table 1. BDF had the most stable performance in the rotating frame, but by far the worst in the stationary frame as it ended up performing passing orbits. Therefore, I chose to use Radau throughout. This test also showed that the Lagrangian points seemed to be less stable in the stationary frame, with extra numerical error accumulating through the motion of the Sun and the planet. Therefore, the wander experiments will all be performed in the rotating frame.

Method	stationary frame wander / au	rotating frame wander / au
DOP853	0.185	1.52×10^{-4}
Radau	0.100	1.23×10^{-10}
BDF	10.656	2.22×10^{-13}
LSODA	3.344	1.49×10^{-5}

Table 1: Wanders using different integration methods for 100 planetary orbits. Radau has the best performance in the stationary frame and excellent performance in the rotating frame, so was used for the rest of the simulations.

To test that my simulations were working properly I introduced small random perturbations in position-velocity space about $\mathbf{L}_4$ and $\mathbf{L}_5$, and checked that they were indeed stable. I extended this into a video in the stationary frame to check that the asteroids visually formed the shapes expected, which is included in the outputs file.

The stationary frame code was also tested by looking at how the total energy varied with time for random sets of initial conditions. The specific energy of random perturbations about $\mathbf{L}_4$ tended to oscillate with relative amplitudes around 2%. Variation is inevitable due to the limited precision to which numbers are stored and the inaccuracy of the solving method itself, so small deviations around the initial value are acceptable. Only if the specific energy was clearly increasing or decreasing with time would a fundamental issue be apparent. Unfortunately the Coriolis force complicates the conservation of energy, so a similar test couldn't be easily implemented in the rotating frame.

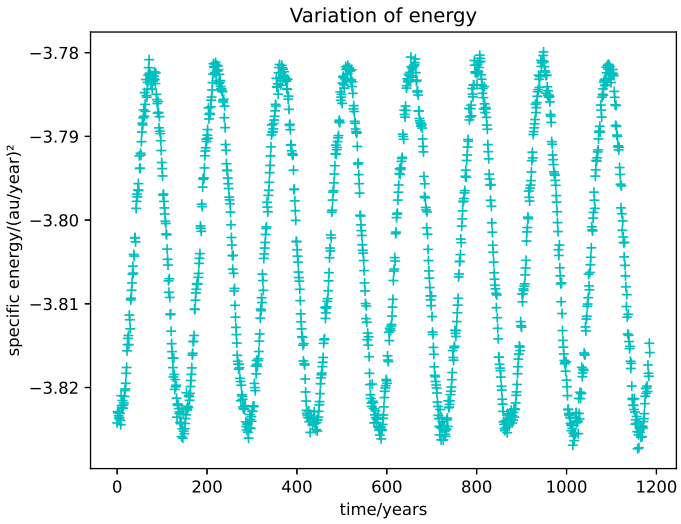

Figure 7: Typical oscillation of specific energy with time, here for a perturbation from $\mathbf{L}_4$ of $\boldsymbol{r} = (0.014, 0.039, -0.025)$au and $\boldsymbol{v} = (0.042, -0.029, 0.023)$au/year, with a relative amplitude of 1.2%.

4 Results

4.1 Position perturbations

The effect of perturbing the starting position about $\mathbf{L}_4$ on wander investigated first. It would be very difficult to gain insight by considering all directions at once, so the x-y plane was considered first, then the z direction.

4.1.1 Position perturbations in the orbital plane

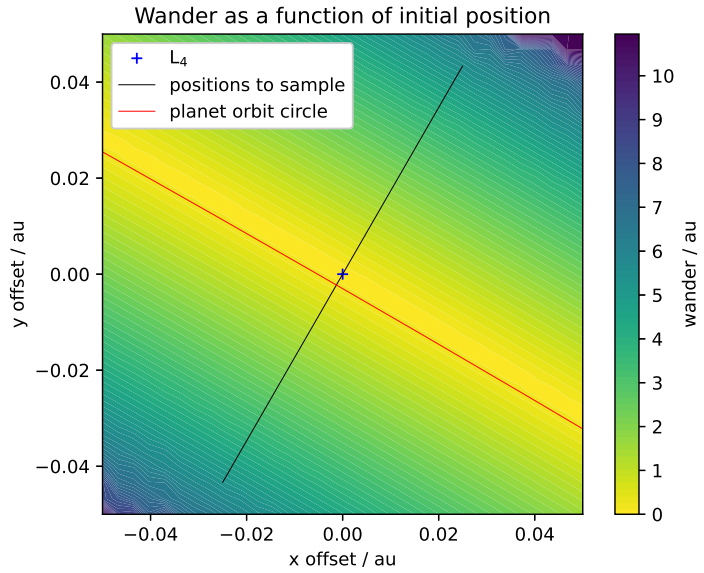

Figure 8: Contour plot of the wander for position perturbations about $\mathbf{L}_4$. Also shown are the planet's orbital circle, and the radial positions normal to it that will be investigated further.

To gain insight into the behaviour in the x-y plane a contour plot of the wander due to perturbations about $\mathbf{L}_4$ was produced, shown in figure 8. This showed that perturbations along the $\mathbf{L}_4$ direction became less stable much more quickly than those along the radial direction. The variation along the radial direction was then plotted and found to follow quadratic trends for both going inwards and outwards, $681.4x^2 + 74.4x + 0.026$ and $576.3x^2 - 75.8x + 0.020$ respectively, shown in figure 9. The similar coefficients suggest that the shapes of the orbits are similar for asteroids displaced both inwards and outwards (note that the differing signs of the second coefficient is due to x being negative for inward displacements), and the small y-intercept is as expected as an asteroid starting at $\mathbf{L}_4$ shouldn't wander. The quadratic form for small displacements can be reasoned intuitively: the tadpole orbits are made up of two elliptical contributions, both of which might be expected to grow linearly with sufficiently small perturbations.

4.1.2 z position perturbations and oscillations

The wander due to perturbations in the z direction was then investigated. For perturbations less than 1.0 au the wander was quadratic, fitting a curve of $4.75z^2 - 1.67 \times 10^{-2}$ as shown in figure 10. The symmetry of the curve is reassuring, as the system is symmetric in the x-y plane.

I then plotted the variation of z with time and found that it did oscillate almost harmonically with varying amplitude, as shown in figure 11. The period decreased with increasing amplitude, though started off greater than the planet orbit period. The simple harmonic analysis in equation 7 suggested that the oscillation period should be shorter than the orbit period for small oscillations, this has turned out to not be the case. They are close however, as was expected. This is due to the asteroid remaining at roughly the same radius despite moving in the orbital plane.

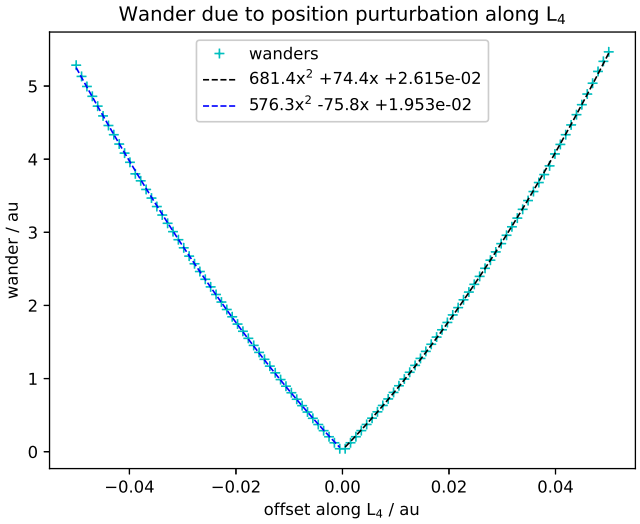

Figure 9: Wander as a function of radial displacement outwards from $\mathbf{L}_4$. A quadratic has been fitted to both inward and outward perturbations.

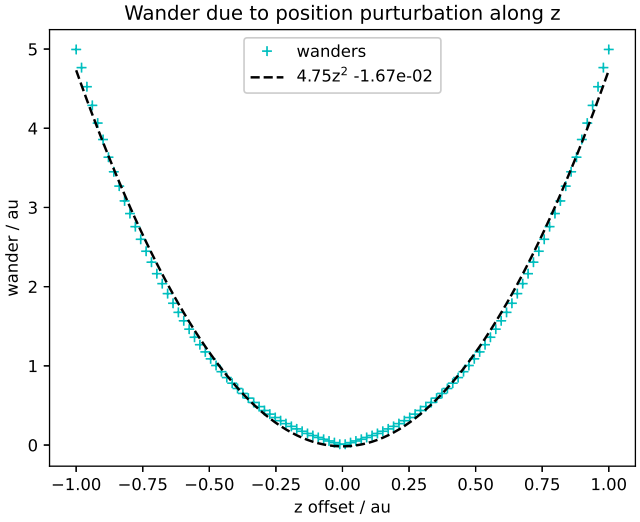

Figure 10: Wanders resulting from z perturbations. Note that the extent of the wander is largely due to the resulting motion in the orbital plane, rather than in the z direction.

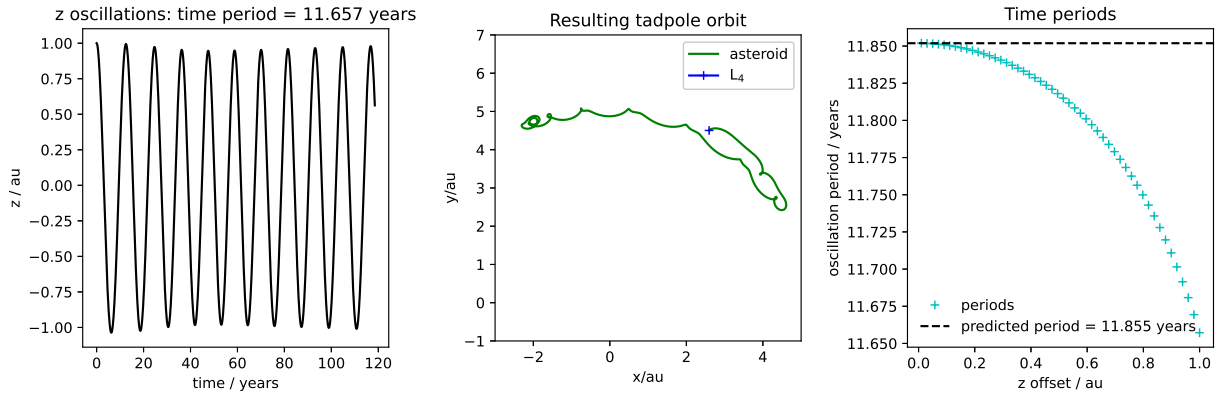

Figure 11: Left: oscillations in the z direction for a perturbation of 1.0 au. Centre: the same asteroid trajectory viewed in the x-y plane. Right: The time period of oscillations for varying perturbations compared to the planet orbit period. The smallest oscillations closely match the period expected from a simple harmonic analysis.

4.2 Velocity perturbations

Like the position perturbations, I considered first velocity perturbations in the x-y plane then the z direction separately, for asteroids starting from $\mathbf{L}_4$.

4.2.1 Velocity perturbations in the orbital plane

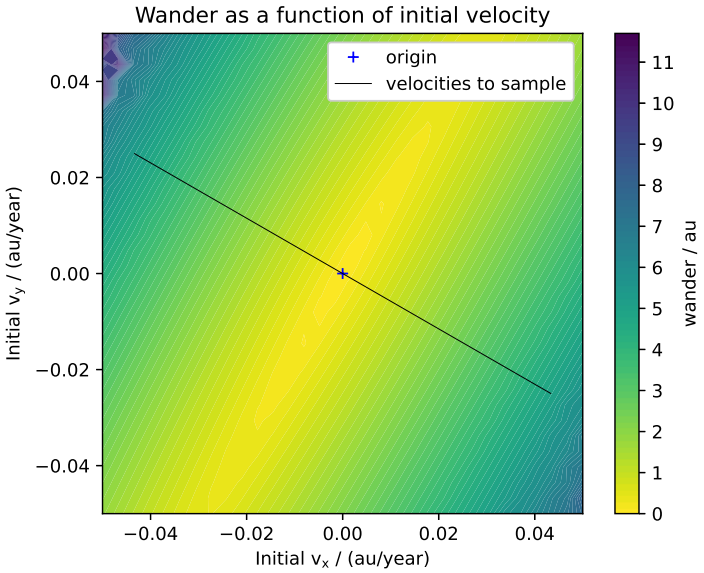

Figure 12: Contour plot of the wander for velocity perturbations starting at $\mathbf{L}_4$. Also shown are the velocities in the direction tangent to the planet's orbit which will be investigated further.

Again I produced a contour plot shown in figure 12; unlike the position perturbations the stable region was along the radial direction. Therefore, I investigated along the tangential direction perpendicular to this.

Unlike the position perturbations, the wander didn't appear quite symmetric going tangentially toward or away from the planet, fitting equations of $627.9v^2 + 72.5x + 0.023$ and $461.1v^2 - 74.8v + 0.013$ respectively,

shown in figure 13. This is perhaps due to the way curved tadpole orbits grow more towards $\mathbf{L}_3$ than towards the planet, so it seems logical that asteroids perturbed towards $\mathbf{L}_3$ would wander further.

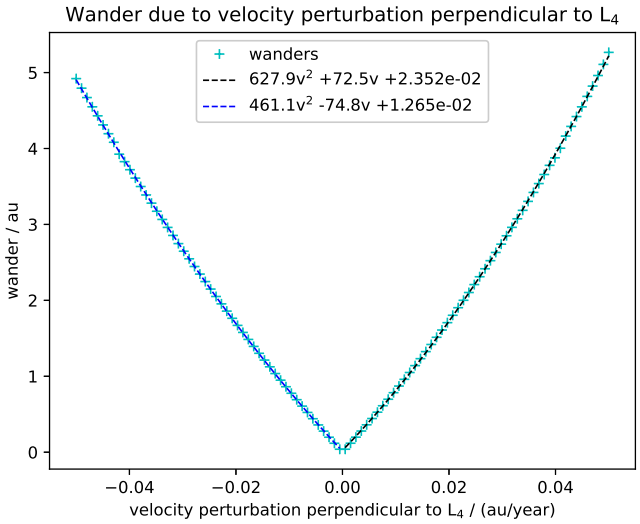

Figure 13: Wander as a function of velocity perturbation v along the tangential direction (perpendicular to $\mathbf{L}_4$). Note: positive v is in the tangential direction away from the planet.

4.2.2 $\mathbf{v}_z$ perturbations

Velocity perturbations along z followed a similar trend to the position perturbations, again with a symmetric shape following $17.54v_z^2 - 7.56 \times 10^{-3}$ shown in figure 14, again as expected due to the symmetry in the orbital plane. Oscillations were not investigated, as the findings would be similar to those for position perturbations.

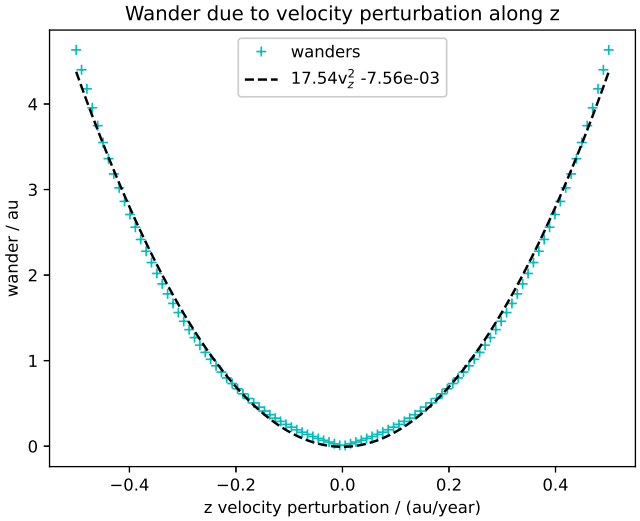

Figure 14: Wander as a function of velocity perturbation in the z direction.

4.3 Varying mass ratio

To investigate the effect of varying the mass ratio, simulations were instantiated with varying M_{P}, then looking at the wander for an asteroid starting 1×10^{-4} au radially outwards of $\mathbf{L}_4$. The wander initially decreased with increasing mass ratio, and was empirically found to fit $\frac{2.32\times10^{-4}}{\sqrt{m}} + 6.10 \times 10^{-4}$. The system then became unstable beyond $M_{\mathrm{P}} = 0.043 M_{\odot}$. This is within the unstable region $M_{\mathrm{P}} > 0.04 M_{\odot}$ predicted by Gascheau. [1]

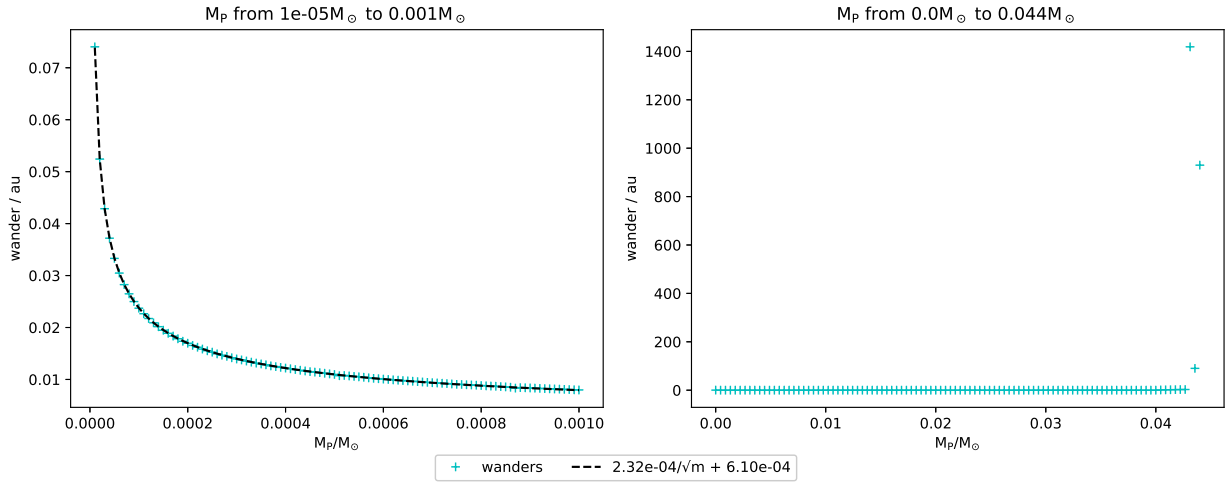

Figure 15: Wander for varying mass ratios, with a starting perturbation of 1×10^{-4} au radially outwards. The wander decreases as $\frac{2.32\times10^{-4}}{\sqrt{m}} + 6.10 \times 10^{-4}$ until the system becomes unstable beyond $M_{\mathrm{P}} = 0.043 M_{\odot}$.

5 Performance

To assess the performance, I used the package `time` to record how long it took to perform the simulations for the position perturbation contour plot. This required $32 \times 32 = 1024$ simulations and took 318.4 s using a laptop with an Intel i7 processor, on average 0.3109 s per simulation.

6 Conclusions

The project has been successful in simulating the behaviour of Trojan asteroids, using the Radau integration method from SciPy. The Lagrangian points have been found to be stable, and perturbations about them resulted in the tadpole, horseshoe and passing orbits expected.

The stable region in the orbital plane has been found to be elongated parallel to the the line of the planet's orbit direction in position space, and perpendicular to it in velocity space. In the directions perpendicular to the stable axes the wander was found to fit a quadratic relationship with the size of the perturbations, with a small y-intercept due to the stability at the Lagrangian points.

Wander due to perturbations perpendicular to the orbital plane were also quadratic, and resulted in oscillations in that direction while the asteroid wandered in the plane direction. The period tends to that of the planetary orbit for small oscillations, as expected by analysing the gravitational force in the z direction for small oscillations. However, the behaviour is complicated by the the wandering in the orbital plane caused by the perturbation.

The mass ratio m of the planet to the Sun also affected the wander, for perturbations of 1×10^{-4} au radially outwards it was found to follow $\frac{2.32\times10^{-4}}{\sqrt{m}} + 6.10 \times 10^{-4}$. The system then became unstable for $m > 0.043$.

For further investigation, the stability regions about the Lagrangian points could be mapped out further, and the behaviour of the system around the transition from stable to unstable mass ratio investigated more closely.

References

[1] Souchay J.J. and Dvorak R. *Dynamics of Small Solar System Bodies and Exoplanets*. Lecture Notes in Physics. Springer Berlin Heidelberg, 2010.

[2] Buscher D. Part II Computational Physics Projects. NST Part II Computational Physics handout, Department of Physics, University of Cambridge, Lent Term 2020.

[3] Scott S. Sheppard and Chadwick A. Trujillo. A Thick Cloud of Neptune Trojans and Their Colors. *Science*, 313(5786):511–514, July 2006.

[4] Murray Lagerkvist Marzari, Scholl. Origin and evolution of trojan asteroids. In *Asteroids III*, pages 725–736, 2002.

[5] Green D. Dynamics: Handout II. NST Part IB lecture handout, Department of Physics, University of Cambridge, Lent Term 2019.

[6] Greenspan T. Semantic Scholar. In *Stability of the Lagrange Points, L4 and L5*, pages 1–2, January 2014.

A constants.py

```
"""
Constants required for Trojan asteroid simulations in solar system units
Contains:
Gravitational constant G
Mass of Jupiter M_JUPITER
Mass of Sun M_SUN
Sun-Jupiter distance R_0
"""
import numpy as np

G = 4 * np.pi ** 2
M_JUPITER = 0.001
M_SUN = 1
R_0 = 5.2
```

B rotatingframe.py

```
"""
Code for performing Trojan asteroid simulations in the rotating frame

Vectors are represented by NumPy arrays.

scipy.integrate.solve_ivp is used to integrate the equations of motion
"""
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from constants import G, M_JUPITER, M_SUN, R_0


class RotatingAsteroid:
    """Simulation of a Trojan asteroid in the rotating frame

    The class should be initialised with the desired planetary mass
    and Sun-planet distance R in solar system units.

    The properties default to those of Jupiter
    """

    def __init__(self, M_P=M_JUPITER, R=5.2):
        """
        Simulation of a Trojan asteroid in the rotating frame.
        M_P = planetary mass
        R = Sun-planet distance
        """
        self._M_P = M_P
        self._R = R

        self._R_SUN = R * self._M_P / (self._M_P + M_SUN)
        self._R_P = R * M_SUN / (self._M_P + M_SUN)
        self._r_sun = np.array([-self._R_SUN, 0, 0])
        self._r_p = np.array([self._R_P, 0, 0])

        self._W = np.sqrt(G * (self._M_P + M_SUN) / R ** 3)
        self._T = 2 * np.pi / self._W

        self._L4 = np.array([self._R / 2 - self._R_SUN, self._R * np.sqrt(3) / 2, 0])
        self._L5 = np.array([self._R / 2 - self._R_SUN, -self._R * np.sqrt(3) / 2, 0])

    ### Getter methods
    @property
    def L4(self):
        """L4"""
        return self._L4

    @property
    def L5(self):
        """L5"""
        return self._L5
```

```
    @property
    def M_P(self):
        """Mass of Planet"""
        return self._M_P

    @property
    def R(self):
        """Distance between Sun and planet"""
        return self._R

    @property
    def R_SUN(self):
        """Distance from origin to Sun"""
        return self._R_SUN

    @property
    def R_P(self):
        """Distance from origin to Jupiter"""
        return self._R_P

    @property
    def T(self):
        """Time period of planetary orbit"""
        return self._T

    @property
    def W(self):
        """Angular frequency of planetary orbit"""
        return self._W

    def _acceleration(self, t, r, v):
        """Acceleration of an asteroid at position r with velocity v at time t"""
        real_accel = -G * (
            M_SUN * (r - self._r_sun) / np.linalg.norm(r - self._r_sun) ** 3
            + self._M_P * (r - self._r_p) / np.linalg.norm(r - self._r_p) ** 3
        )

        # explicit form for virtual acceleration with rotation in the z direction
        virtual_accel = self._W * np.array(
            [2 * v[1] + self._W * r[0], -2 * v[0] + self._W * r[1], 0]
        )
        return real_accel + virtual_accel

    def _derivs(self, t, y):
        """derivatives for solver"""
        return np.hstack((y[3:6], self._acceleration(t, y[0:3], y[3:6])))

    def trajectory(self, t_eval, r_0, v_0, events=None, method="Radau"):
        """Trajectory of asteroid calculated using solve_ivp"""
        y0 = np.append(r_0, v_0)

        return solve_ivp(
            self._derivs,
```

```
            (0, t_eval[-1]),
            y0,
            t_eval=t_eval,
            method=method,
            events=events,
        )

    def wander(self, t_eval, r_0, v_0, stability_point, method="Radau"):
        """Find the maximum distance from the stability point for given initial conditions in the rotat
        sol = self.trajectory(t_eval, r_0, v_0, method=method)
        rs = sol.y[0:3]  # extract positions from solution

        deltas = (rs.T - stability_point).T  # Displacements from L4
        # Transpose used to stick to array convention
        norms = np.linalg.norm(deltas, axis=0)
        return norms.max()
```

C stationaryframe.py

```
"""
Code for performing Trojan asteroid simulations in the stationary frame.

Vectors are represented by NumPy arrays.

scipy.integrate.solve_ivp is used to integrate the equations of motion.
"""
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from constants import G, M_JUPITER, M_SUN, R_0


class StationaryAsteroid:
    """
    Simulation of a Trojan asteroid in the stationary frame.

    The class should be initialised with the desired planetary mass M_P
    and Sun-planet distance R in solar system units.

    The properties default to those of Jupiter.
    """

    def __init__(self, M_P=M_JUPITER, R=R_0):
        """
        Simulation of a Trojan asteroid in the stationary frame.
        M_P = planetary mass
        R = Sun-planet distance
        """
        self._M_P = M_P
        self._R = R

        self._R_SUN = R * self._M_P / (self._M_P + M_SUN)
        self._R_P = R * M_SUN / (self._M_P + M_SUN)
```

```
        self._r_sun = np.array([-self._R_SUN, 0, 0])
        self._r_p = np.array([self._R_P, 0, 0])

        self._W = np.sqrt(G * (self._M_P + M_SUN) / R ** 3)

        self._L4 = np.array([self._R / 2 - self._R_SUN, self._R * np.sqrt(3) / 2, 0])
        self._L5 = np.array([self._R / 2 - self._R_SUN, -self._R * np.sqrt(3) / 2, 0])

    ### Getter methods
    @property
    def M_P(self):
        """Mass of planet"""
        return self._M_P

    @property
    def R(self):
        """Distance between Sun and planet"""
        return self._R

    @property
    def R_SUN(self):
        """Distance from origin to Sun"""
        return self._R_SUN

    @property
    def R_P(self):
        """Distance from origin to planet"""
        return self._R_P

    @property
    def T(self):
        """Time period of planetary orbit"""
        return 2 * np.pi / self._W

    @property
    def W(self):
        """Angular frequency of planetary orbit"""
        return self._W

    def l4(self, t):
        """Position of L_4 at time t"""
        # np.stack for correct behaviour if t is an array
        # 0.0 * t used to create element of correct length
        return np.stack(
            (
                self._L4[0] * np.cos(self._W * t) - self._L4[1] * np.sin(self._W * t),
                self._L4[0] * np.sin(self._W * t) + self._L4[1] * np.cos(self._W * t),
                0.0 * t,
            )
        )

    def l5(self, t):
        """Position of L_4 at time t"""
        # np.stack for correct behaviour if t is an array
```

```
        # 0.0 * t used to create element of correct length
        return np.stack(
            [
                self._L5[0] * np.cos(self._W * t) - self._L5[1] * np.sin(self._W * t),
                self._L5[0] * np.sin(self._W * t) + self._L5[1] * np.cos(self._W * t),
                0.0 * t,
            ]
        )

    def r_sun(self, t):
        """Position of the sun at time t"""
        # np.stack for correct behaviour if t is an array
        # 0.0 * t used to create element of correct length
        return np.stack(
            [
                -self._R_SUN * np.cos(self._W * t),
                -self.R_SUN * np.sin(self._W * t),
                0.0 * t,
            ]
        )

    def r_p(self, t):
        """Position of planet at time t"""
        # np.stack for correct behaviour if t is an array
        # 0.0 * t used to create element of correct length
        return np.stack(
            [self._R_P * np.cos(self._W * t), self._R_P * np.sin(self._W * t), 0.0 * t]
        )

    def specific_energy(self, t, r, v):
        """Specific energy of an asteroid at time t, position r and velocity v"""
        kinetic = 0.5 * np.linalg.norm(v, axis=0) ** 2
        potential = -G * (
            M_SUN / np.linalg.norm(r - self.r_sun(t), axis=0)
            + self._M_P / np.linalg.norm(r - self.r_p(t), axis=0)
        )
        return kinetic + potential

    def omega_cross(self, r):
        """W x r (for convenience)"""
        return np.array([-self.W * r[1], self.W * r[0], 0])

    def _acceleration(self, t, r):
        """Acceleration of an asteroid at position r and time t"""
        return -G * (
            M_SUN * (r - self.r_sun(t)) / np.linalg.norm(r - self.r_sun(t)) ** 3
            + self._M_P * (r - self.r_p(t)) / np.linalg.norm(r - self.r_p(t)) ** 3
        )

    def _derivs(self, t, y):
        """Derivatives for solver"""
        return np.hstack((y[3:6], self._acceleration(t, y[0:3])))

    def trajectory(self, t_eval, r_0, v_0, events=None, method="Radau"):
```

```
    """Trajectory of asteroid calculated using solve_ivp"""
    y0 = np.append(r_0, v_0)
    return solve_ivp(
        self._derivs,
        (0, t_eval[-1]),
        y0,
        t_eval=t_eval,
        method=method,
        events=events,
    )

def wander(self, t_eval, r_0, v_0, method="Radau"):
    """Find the maximum distance from L4 point for given initial conditions
    in the rotating frame"""
    sol = self.trajectory(t_eval, r_0, v_0, method=method)

    deltas = sol.y[0:3] - self.l4(t_eval)  # displacements from L4
    norms = np.linalg.norm(deltas, axis=0)
    return norms.max()
```