INVESTIGATING THE STABILITY OF THE L4 AND L5 LAGRANGINAN POINTS

April 27, 2020

ABSTRACT

Lagrangian Points are points within two body orbits where a body with a much smaller mass can be placed and will maintain its relative position to two larger bodies in orbit. Points L4 and L5 form an equilateral triangle with the two large bodies, and are a stable equilibrium. The allowed perturbations from the Lagrangian points were investigated for the Sun and Jupiter system. The maximum allowed radial perturbations was $\pm 0.065 \pm 0.001 AU$ and the allowed angular perturbations were $+2\pi/3 \leq \theta \leq -0.2 \pm 0.01\pi$, where $+2\pi/3$ is the maximum possible positive perturbation.

The effect of the relative masses of the two bodies on maximum stable perturbations was also investigated. When the relative mass was decreased, the allowed angular perturbations continued to increase, whereas the peak of the radial perturbations occurred at a relative mass of 0.0075. When the relative mass was increased, the allowed angular perturbations very quickly decreased, and while the radial perturbations also decreased while having several smaller peaks and troughs. However at a relative masses of 0.045 and higher, all perturbations became unstable.

1 Introduction

Lagrangian Points are points within two body orbits where a body with a much smaller mass can be placed and will maintain its relative position to two larger bodies in orbit. Five such points exist as shown in Figure 1. We will be examining points L4 and its mirror L5. These points are special as they are stable [1] and such are points where asteroids can be found. These are called ”Trojan” and ”Greek” Asteroids.

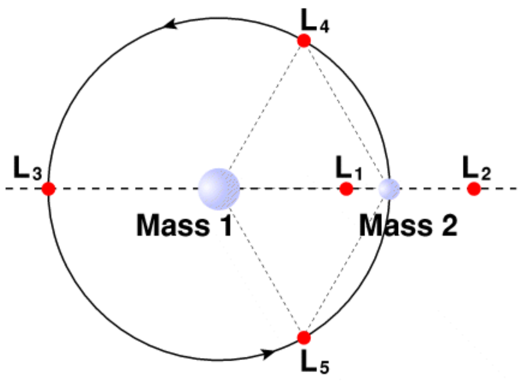


Figure 1: Location of the 5 Lagrangian Points. Image from Wikipedia Commons

The stability of these points are investigated, including the maximum allowed perturbations and how they are affected by the relative masses of the two larger bodies.

2 Theoretical Background

2.1 Orbits

Planetary orbits can be modelled using Newton's law of gravity,

$$m\ddot{\mathbf{r}} = -\frac{GMm}{|\mathbf{r}|^2}\hat{\mathbf{r}} \tag{1}$$

For objects in circular orbits, the circular motion equation can be used

$$F_{centri} = mr\omega^2 \tag{2}$$

Two body orbits, will result in both bodies $(M_1, \mathbf{r_1})$ and $(M_2, \mathbf{r_2})$ in a circular orbit around their centre of mass, orbiting with the same angular velocity. The distance from the centre of mass can be calculated through

$$r = r_1 + r_2 \tag{3}$$
$$M_1 r_1 = M_2 r_2 \tag{4}$$

where r_1 and r_2 are the distances from the centre of mass and r is the distance between the two bodies. The angular velocity can be calculated using equation (2) to be

$$\omega = \sqrt{\frac{G(M_1 + M_2)}{r^3}} \tag{5}$$

2.2 Lagrangian Points

To examine the Lagrangian points, it is easier to switch into a frame that is rotating around the centre of mass [1]. In this frame, the two orbiting bodies $(M_1, \mathbf{r_1})$ and $(M_2, \mathbf{r_2})$ will remain stationary and any other bodies $(m, \mathbf{r})$ will experiences the forces of gravity as well as the fictitious Centrifugal $(m\omega^2\mathbf{r})$ and Coriolis forces $(-2m\omega \times \dot{\mathbf{r}})$.

$$\mathbf{F} = \frac{GM_1 m}{|\mathbf{r_1} - \mathbf{r}|^3}(\mathbf{r_1} - \mathbf{r}) + \frac{GM_2 m}{|\mathbf{r_2} - \mathbf{r}|^3}(\mathbf{r_2} - \mathbf{r}) + m\omega^2\mathbf{r} - 2m\omega \times \dot{\mathbf{r}} \quad [2] \tag{6}$$

For a body to remain stationary, the force on it will have to be 0. By substituting Equation (6) into Equation (7) and assuming the body is initially at rest, we can write

$$0 = \frac{GM_1 m}{|\mathbf{r_1} - \mathbf{r}|^3}(\mathbf{r_1} - \mathbf{r}) + \frac{GM_2 m}{|\mathbf{r_2} - \mathbf{r}|^3}(\mathbf{r_2} - \mathbf{r}) + \frac{G(M_1 + M_2)m}{|\mathbf{r_2} - \mathbf{r_1}|^3}\mathbf{r} \tag{7}$$

Which has a solution of $|\mathbf{r_1} - \mathbf{r}| = |\mathbf{r_2} - \mathbf{r}| = |\mathbf{r_2} - \mathbf{r_1}|$ [2], or more simply if the 3 bodies form an equilateral triangle. This is the case for L4 and L5.

3 Modelling the Orbits

3.1 Selecting an ODE solver

To model orbits, we can solve the relevant differential equations with the correct initial conditions. Python's libraries only have first order differential equation solvers. Hence equation (1) needs to be split into two coupled first order differential equations.

$$\dot{\mathbf{r}} = \mathbf{v} \tag{8}$$
$$\dot{\mathbf{v}} = -\frac{GM}{r^2}\hat{\mathbf{r}} \tag{9}$$

Converting these into their Cartesian form,

$$\dot{x} = v_x \tag{10}$$

$$\dot{y} = v_y \tag{11}$$

$$\dot{v_x} = -\frac{GMx}{(x^2+y^2)^{\frac{3}{2}}} \tag{12}$$

$$\dot{v_y} = -\frac{GMy}{(x^2+y^2)^{\frac{3}{2}}} \tag{13}$$

These can be solved using Python's ODE solvers. The two possibilities for ODE solvers were SciPy's $scipy.integrate.odeint$ and $scipy.integrate.solve\_ivp$. Both methods require the inputs of the initial x and y positions and velocities, amount of time to calculate values for and the relevant ODEs, then they will output each bodies x and y coordinate and velocity at all discreetly defined times. To test which method would be better for this problem, a simple circular orbit of radius 5.2 around the origin was tested. The initial positions are (5.2, 0) and the initial velocities can be calculated using the equation (2). Hence the initial velocity will be (0, $\sqrt{GM/5.2}$).

With these initial conditions, the ODEs were solved for over 100 orbits using both $scipy.integrate.odeint$ and $scipy.integrate.solve\_ivp$. A plot of the x and y coordinates was made.

It was found that $scipy.integrate.odeint$ preserved the circular orbit better over time, with very little deviations, whereas $scipy.integrate.solve\_ivp$ spiraled inwards. $Solve\_ivp's$ minumum steps could be adjusted to maintain the circular orbit, however this led to it taking a long time to solve the ODEs.

Hence $scipy.integrate.odeint$ was used continuing forward.

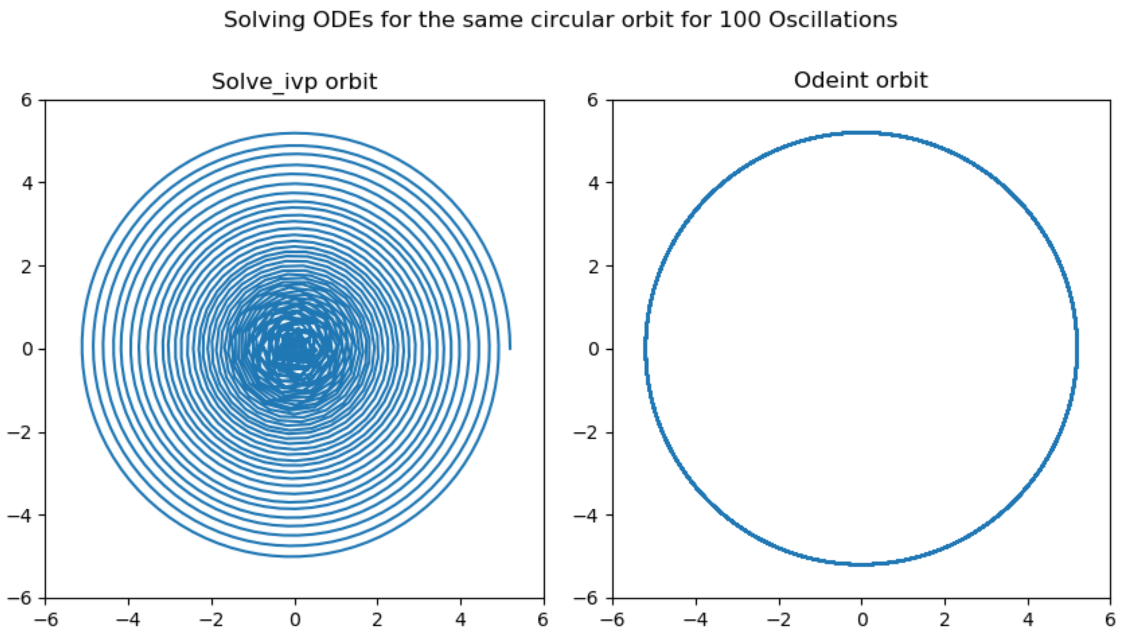


Figure 2: Circular orbits with same initial conditions and ODEs, solved using different Python ODE solvers, for 100 oscillations. The $Solve\_ivp$ orbit spirals inwards while $Odeint$ preserves the circular orbit.

3.2 Jupiter and Sun's two body orbit

The sun and Jupiter orbit each other in a 2-body system. Using solar system units, the masses of Jupiter and Sun were, $m_s = 1$ and $m_j = 0.001$, the distance between them was set as $r = 5.2AU$ and the Gravitational constant was set as $G = 4\pi^2$. The differential equations of the system were

$$r^2 = (x_s - x_j)^2 + (y_s - y_j)^2 \tag{14}$$

$$\dot{x_j} = v_x \tag{15}$$

$$\dot{y_j} = v_y \tag{16}$$

$$\dot{x_s} = u_x \tag{17}$$

$$\dot{y_s} = u_y \tag{18}$$

$$\dot{v_x} = \frac{Gm_s(x_s - x_j)}{r^{\frac{3}{2}}} \tag{19}$$

$$\dot{v_y} = \frac{Gm_s(y_s - y_j)}{r^{\frac{3}{2}}} \tag{20}$$

$$\dot{u_x} = \frac{Gm_j(x_j - x_s)}{r^{\frac{3}{2}}} \tag{21}$$

$$\dot{u_y} = \frac{Gm_j(y_j - y_s)}{r^{\frac{3}{2}}} \tag{22}$$

Where x_j, y_j, x_s, y_s are the x and y coordinates of Jupiter and the sun respectively. Continuing forward, the notation (x, y, v_x, v_y) will be used to describe the positions and velocities of a body.

The initial conditions for the sun and Jupiter orbit can be calculated as they are in a circular orbit around their centre of mass. The distance from the centre of mass can be calculated through using equations (3) and (4)). This gives

$$r_s = r\frac{m_j}{m_j + m_s} \tag{23}$$

$$r_j = r\frac{m_s}{m_j + m_s} \tag{24}$$

Their velocities can be calculated using equation (5). This gives

$$v_j = \sqrt{\frac{Gm_s^2}{r(m_j + m_s)}} \tag{25}$$

$$v_s = \sqrt{\frac{Gm_j^2}{r(m_j + m_s)}} \tag{26}$$

Hence the initial conditions for Jupiter were $(r_j, 0, 0, v_j)$ and for the sun $(-r_s, 0, 0, -v_s)$. To store the output of the ODE solver, "objects" were used, which stored (x, y, v_x, v_y) for each body allowing them to be easily called when needed.

A plot of the x and y coordinates of each body was made to observe the orbits for 100 orbits. This produced a stable orbit between the two bodies as shown in Figure (3). The masses were also varied to test the orbit was indeed stable. The x and y coordinates were also made into an animation to clearly observe if the orbit behaved at expected.

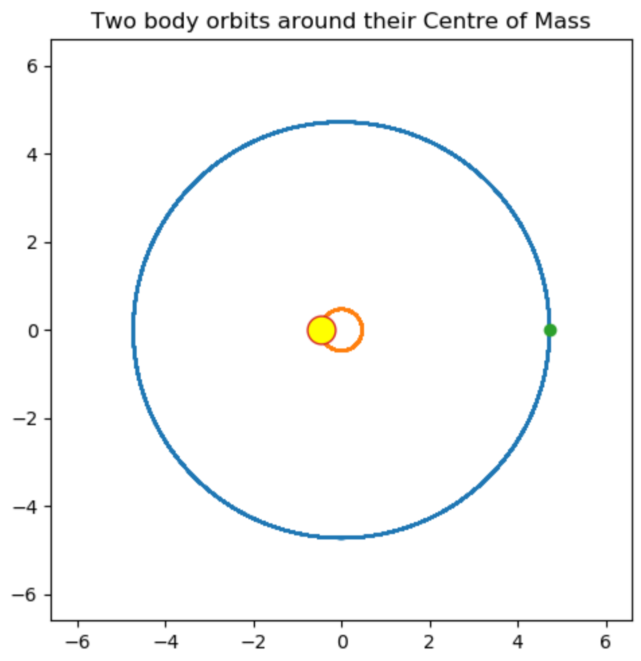


Figure 3: Two body orbit after 100 oscillations, which produces circular orbits around their centre of mass positioned at (0,0). The masses have been varied so the orbit of the Sun is seen clearly

3.3 Introducing the asteroids

The masses of the asteroid are much smaller than both Jupiter and the Sun, hence its gravitational effect may be neglected, meaning the differential equations used in Section (**3.2**) are still perfectly valid. In addition to them, the ODEs of the asteroid would need to be added. These are affected by both the gravity of Jupiter and the Sun. The differential equations for the Trojan asteroids (x_t, y_t, w_x, w_y) are:

$$r_{st}^2 = (x_s - x_t)^2 + (y_s - y_t)^2 \tag{27}$$

$$r_{jt}^2 = (x_j - x_t)^2 + (y_j - y_t)^2 \tag{28}$$

$$\dot{x_t} = w_x \tag{29}$$

$$\dot{y_t} = w_y \tag{30}$$

$$\dot{w_x} = \frac{Gm_s(x_s - x_t)}{r_{st}^{\frac{3}{2}}} + \frac{Gm_j(x_j - x_t)}{r_{jt}^{\frac{3}{2}}} \tag{31}$$

$$\dot{w_y} = \frac{Gm_s(y_s - y_t)}{r_{st}^{\frac{3}{2}}} + \frac{Gm_j(y_j - y_t)}{r_{jt}^{\frac{3}{2}}} \tag{32}$$

And the equivalent for the Greek asteroids were also added. To calculate the initial positions of the asteroids, we would need to calculate the distance and angle from the centre of mass of the system, as the centre of mass was the origin. This can be done with some geometry, using the values shown in Figure 4, r_a and ϕ can be calculated using the cosine rule and the already known values of r, r_s, r_j and θ as $\pm\pi/3$. Then the initial positions are:

$$r_a^2 = r^2 + r_s^2 - 2rr_s cos(\theta) \tag{33}$$
$$r_{aj}^2 = 2r^2(1 - cos(\theta)) \tag{34}$$
$$cos(\phi) = \frac{r_a^2 + r_j^2 - r_{aj}^2}{2r_a r_j} \tag{35}$$
$$x_a = r_a cos(\phi) \tag{36}$$
$$y_a = r_a sin(\phi) \tag{37}$$

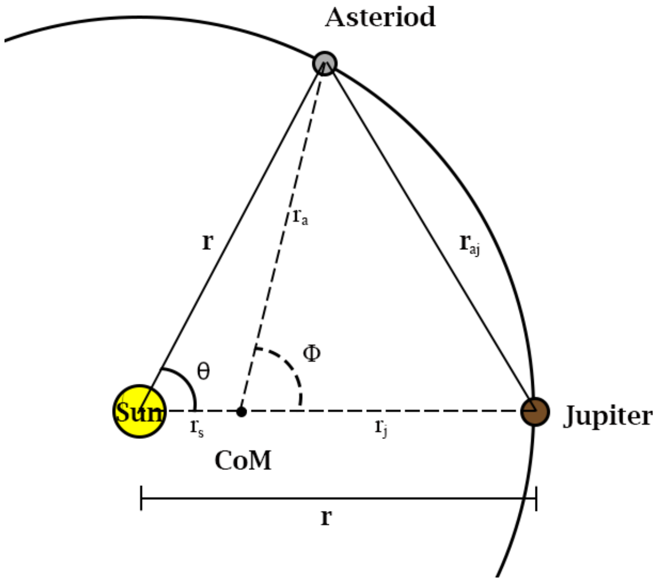


Figure 4: Diagram of the angles and distance between the Sun, Jupiter and an Asteroid

It is worth noting that equation (35) gives $cos(\phi)$, so it is important the correct sign for ϕ is used. The calculated values were $r_a = 5.1974AU$ and $\phi = 0.33360\pi$

As the Lagrangian point is stationary in a rotating frame, it must be orbiting the CoM with the same angular velocity as the Sun and Jupiter. Using this and the previously calculated r_a, ϕ, r_j and v_J (from equations 25 and 26) the initial velocities can be calculated

$$v_a = \frac{v_j r_a}{r_j} \tag{38}$$
$$v_x = -v_a sin(\phi) \tag{39}$$
$$v_y = v_a cos(\phi) \tag{40}$$
$$\tag{41}$$

The initial velocities were calculated for both Greek $(+\pi/3)$ and Trojan $(-\pi/3)$ asteroids. To test their orbits, the x and y coordinates for the asteroids were plotted to see if the orbit was stable and that the two asteroids had the same orbit. Then the distance between the asteroids and Jupiter and the sun was also plotted to see if it remained constant, this is shown in Figure (6). Additionally the radial distance and angular displacement of the asteroids from Jupiter, as measured from the CoM, was also measured to make sure it remained constant, as shown in Figure (7).

As both points have been shown to have stable orbits, from this point forward, only one point will be investigated, as one is just a mirror of the other and what is true for one point will also be true for the other.

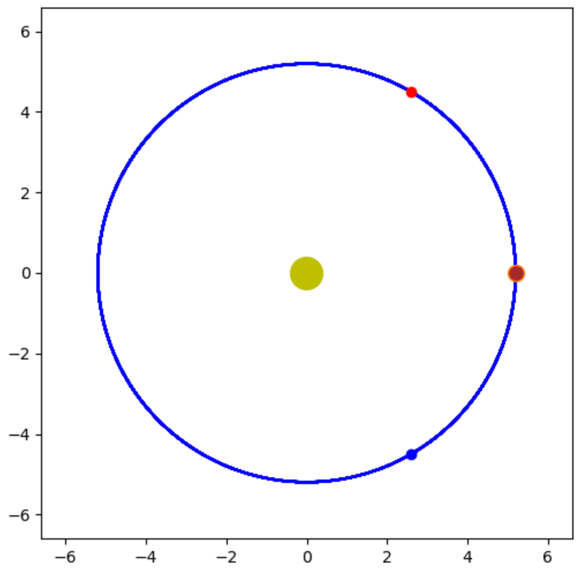


Figure 5: Plot of the orbits of Jupiter and the two asteroids after 100 oscillations. The orbits are stable and all very similar as expected.

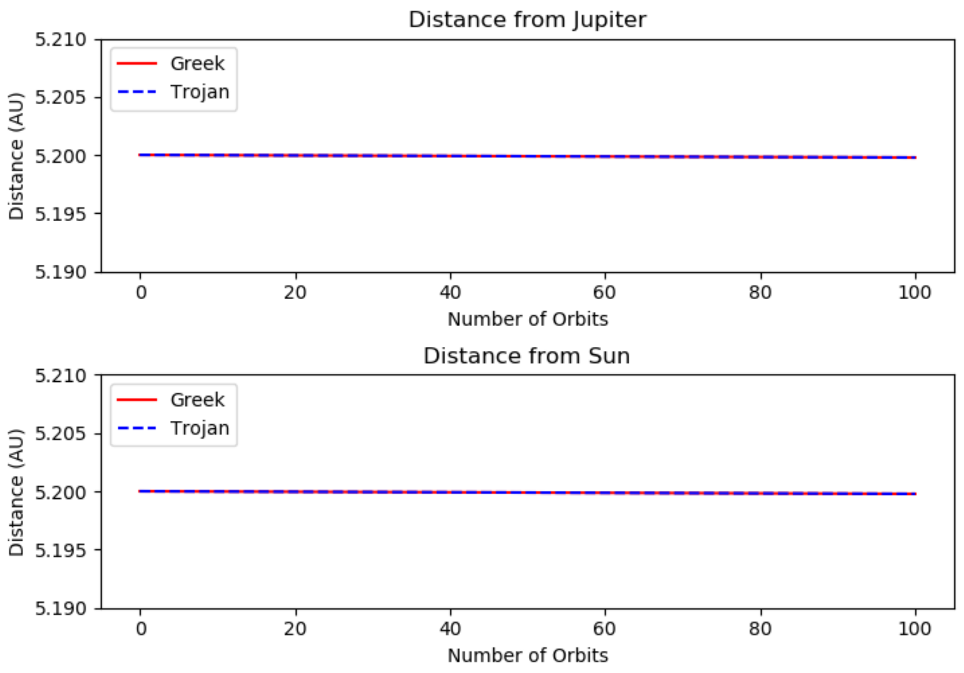


Figure 6: Plot of the distances between the asteroids and the Sun and Jupiter. It is shown that all values remain constant as expected.

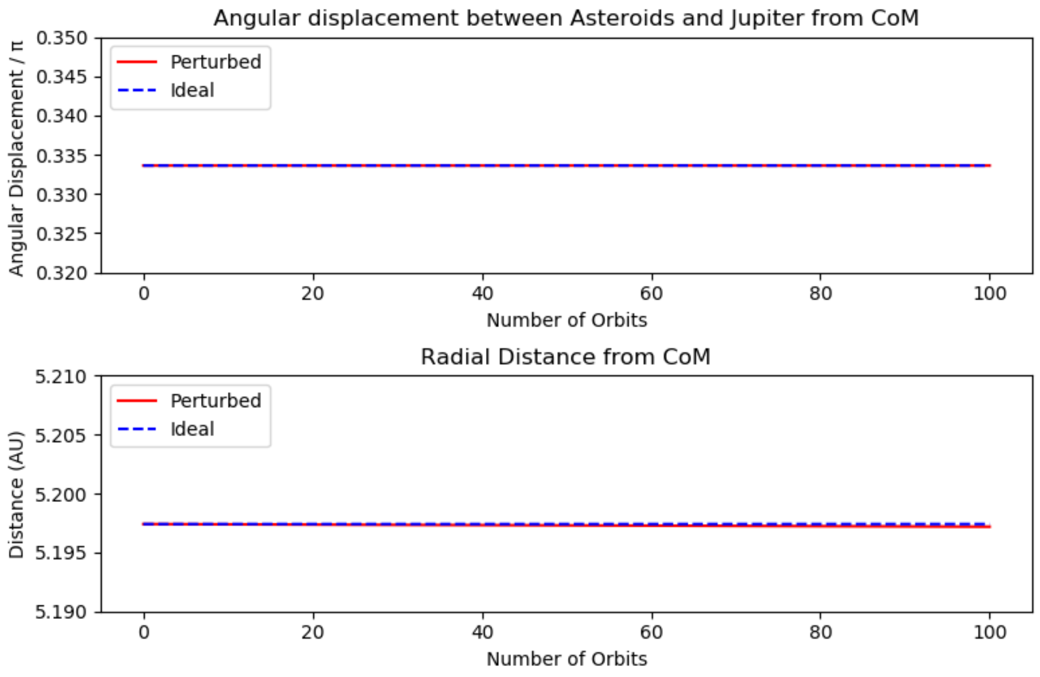


Figure 7: Plot of the radial distance and absolute angular displacement of the Greek and Trojan Orbits from Jupiter, as measured from the CoM. It is shown that all values remain constant as expected.

3.4 Measurements

To further study the Trojan Asteroids, several measurement were made. The first measurement made was the distances from the origin and to the other bodies at each discrete time. This could easily be done using Pythagoras's theorem. The distances were again stored as "objects" for each body.

Additionally angles of the asteroids and Jupiter from the x axis was also measured. This can be done using $\theta = arctan(y/x)$. As the arctan function will always output the acute angle, the angle must be correctly adjusted depending on which quadrant the body is in. Due to this, a function was used to calculate the angle at each discrete time. The recorded angles were in the range $0 \leq \theta \leq 2\pi$.

From this, the angular displacement between bodies can be calculated. Again, care was taken with this, as the angles were between $0 \leq \theta \leq 2\pi$, and if one body completed an orbit then its angle would return to 0 and simply taking the difference between the angles would not give the correct displacement. To correct for this, if the difference was found to be over π then $2\pi - \Delta\theta$ was used as the angular displacement.

3.5 Switching to a rotating Frame

As described in section 2.2, it may be easier to examine the Lagrangian points in a frame rotating around the CoM at angular velocity ω. This transformation was be made using

$$x' = xcos(\omega t) + ysin(\omega t) \tag{42}$$

$$y' = -xsin(\omega t) + ycos(\omega t) \tag{43}$$

Where x' and y' are the coordinates in the new frame. This can be differentiated to find the velocities

$$\dot{x'} = \dot{x}cos(\omega t) - \omega xsin(\omega t) + \dot{y}sin(\omega t) + y\omega cos(\omega t) \tag{44}$$

$$\dot{y'} = -\dot{x}sin(\omega t) - \omega xcos(\omega t) + \dot{y}cos(\omega t) - y\omega sin(\omega t) \tag{45}$$

This transformation was done for all the bodies using their values calculated by solving the ODEs. This frame was tested by plotting the x' and y' coordinates of all bodies, which remained stationary.

3.6 Ideal Lagrangian Point

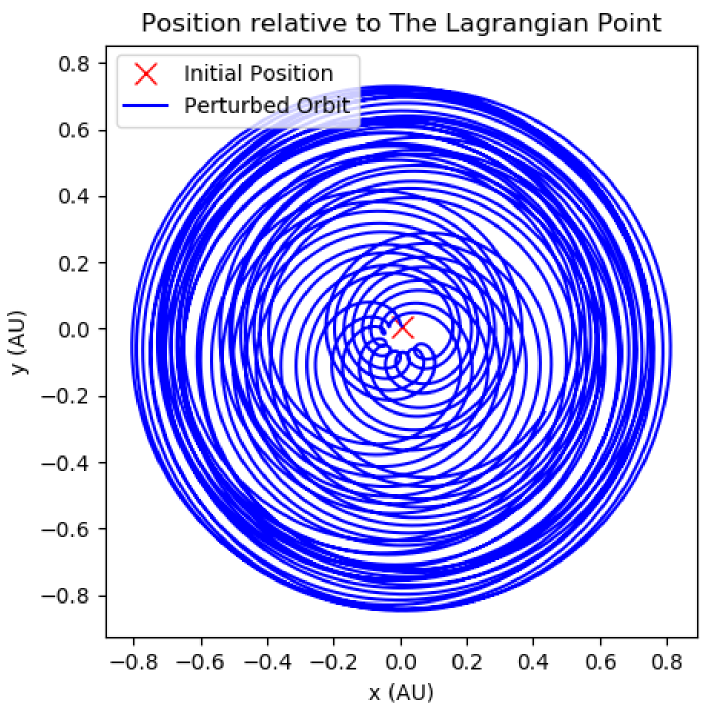
As the asteroids would be perturbed, to measure the difference from the normal orbit, an "ideal" orbit was made. This was done by using the radius and angle of the Lagrangian point calculated equations (33) and (35) and then holding this angle constant relative to Jupiter's orbit. This orbit can be easily tested by finding the distance between this point and the unperturbed asteroid, which was found be to near 0.

4 Investigating the Perturbations

To explore the stability of the Lagrangian Points, the initial conditions of the asteroid were perturbed. The perturbations were split into the radial directions and azimuth directions from the origin. This was done by simply adding perturbations to the radius and angle calculated in equations (33) and (35), which meant that the initial conditions were also adjusted accordingly.

4.1 Radial Perturbations

Initially, a radial perturbation of $+0.01AU$ was made. The radius of the asteroid, the angle between the asteroid and Jupiter and the relative position of the asteroid compared to the "ideal" Lagrangian point in both the normal and rotating reference frames were measured. These values are shown in Figures 8 and 9.



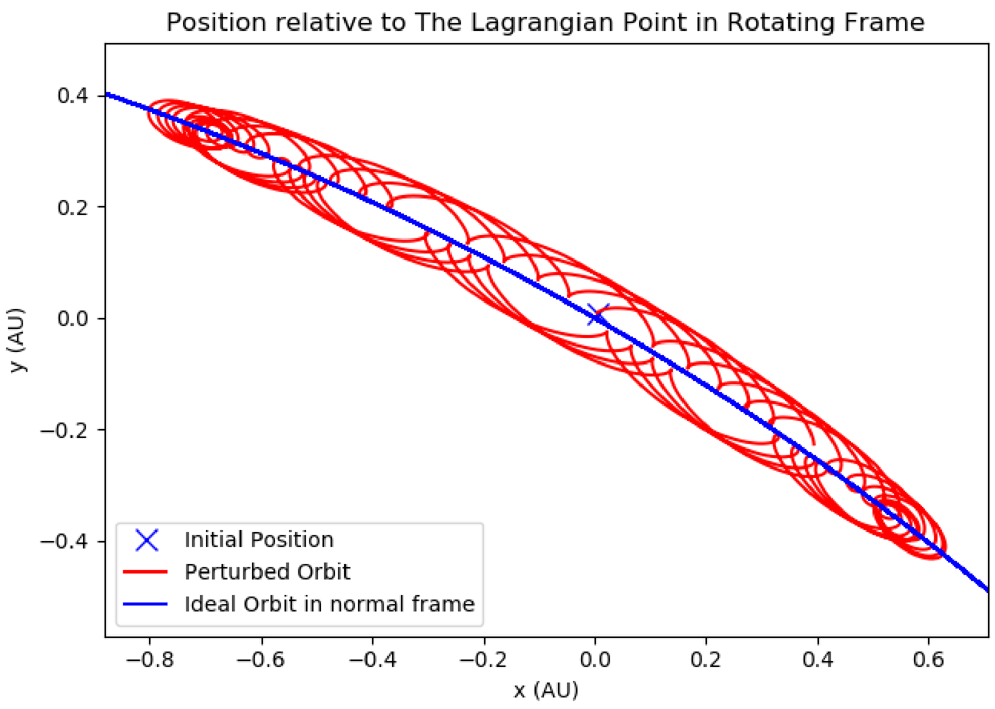


Figure 8: Plot of displacement from the ”ideal” Lagrangian point is the original reference frame and a rotating reference frame after 50 orbits. The perturbation was 0.01AU in the radial direction. Both plots show complicated patters with lots of loops, which oscillate around the ”ideal” point.

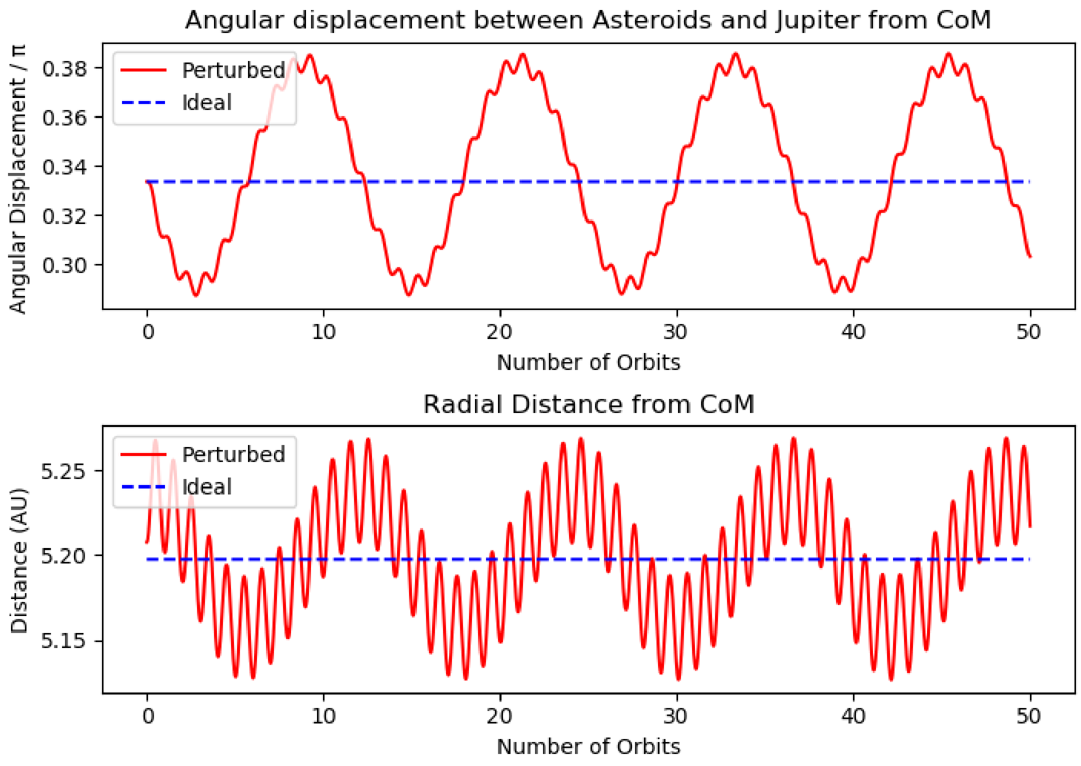


Figure 9: Plot of the radial distance and angular displacement of the perturbed asteroid from Jupiter, as measured from the CoM. The perturbation was 0.01AU in the radial direction, however the plots show that both the radial distance and the angular displacements are affected by this. They both oscillate around the ”ideal” value.

Although the initial displacement was only in the radial direction, Figure 9 shows that both the radial distance and angular displacements are affected. They both oscillate around the ”ideal” value as one might expect and two distinct frequencies can be seen. Additionally, both oscillations initially move away from the stable point, unlike what you might expect from a SHM. The maximum and minimum values of the angular displacement were $0.3855 \pm 0.0001\pi$ and $0.2872 \pm 0.0001\pi$, which gives a midpoint of $0.3364 \pm 0.0001\pi$, slightly higher than the stable value of $0.3336 \pm 0.0001\pi$. The maximum and minimum values of the radius were $5.2687 \pm 0.0002AU$ and $5.1265 \pm 0.0002AU$ with a midpoint of $5.1976 \pm 0.0002AU$.

Figure 8 shows that in the original frame of reference, the relative position around the Lagrangian point oscillates in a circular form with lots of loops, the pattern is very chaotic and complicated. In the rotating frame however, the pattern is simpler although it still contains lots of loops. It also clearly shows that the displacements are not symmetric as the angular displacement is larger in one side that the other.

Further radial perturbations were made. With higher perturbations, the oscillations of the angular displacement and the radial distances lost their symmetry, as shown in Figures 10.

When the perturbation was increased to $+0.1AU$, the asteroid became unstable. Figure 11 shows the orbit of the asteroid.

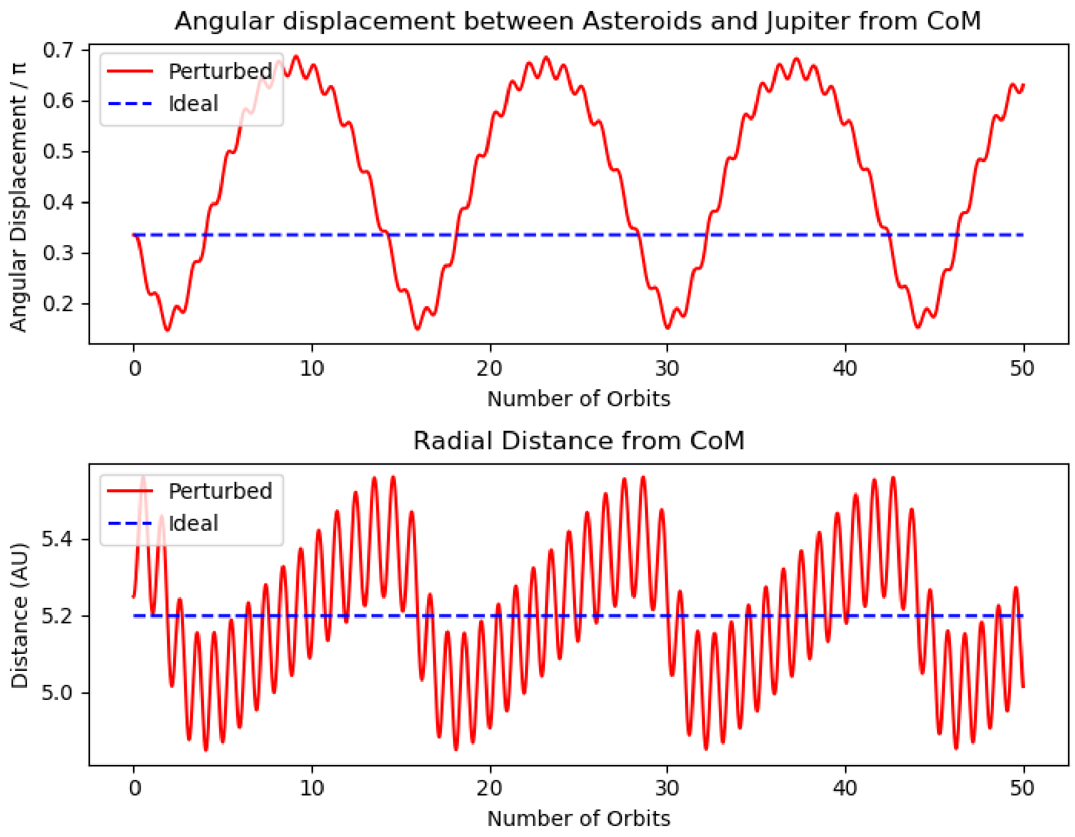


Figure 10: Plot of the radial distance and angular displacement of the perturbed asteroid from Jupiter, as measured from the CoM. The perturbation was 0.50 AU in the radial direction. They both oscillate around the ”ideal” value, however the radial distance seems to be symmetric around this value where as the angular displacement is not.

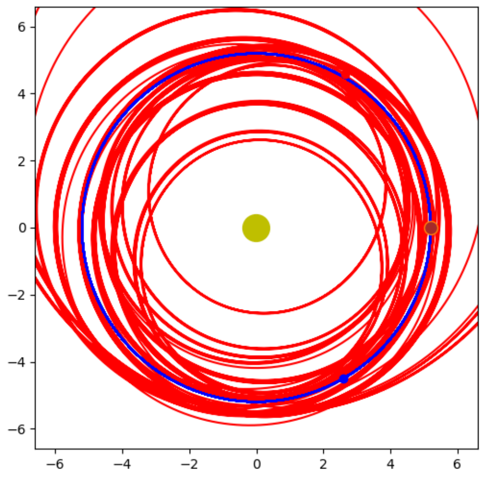


Figure 11: Orbit of perturbed asteroid with +0.1 AU radial perturbation after 100 oscillations. The orbit is no longer stable.

To investigate when radial perturbations become unstable, the program was looped for a variety of perturbations for 50 orbits. A plot of the maximum radius values was made, as unstable orbits will have a noticeably larger maximum value. This is shown in Figure 12. The maximum radius linearly rises as you move away from 0 perturbations, until there is a point where the orbits are no longer stable, these occur at $\pm 0.065 \pm 0.001 AU$.

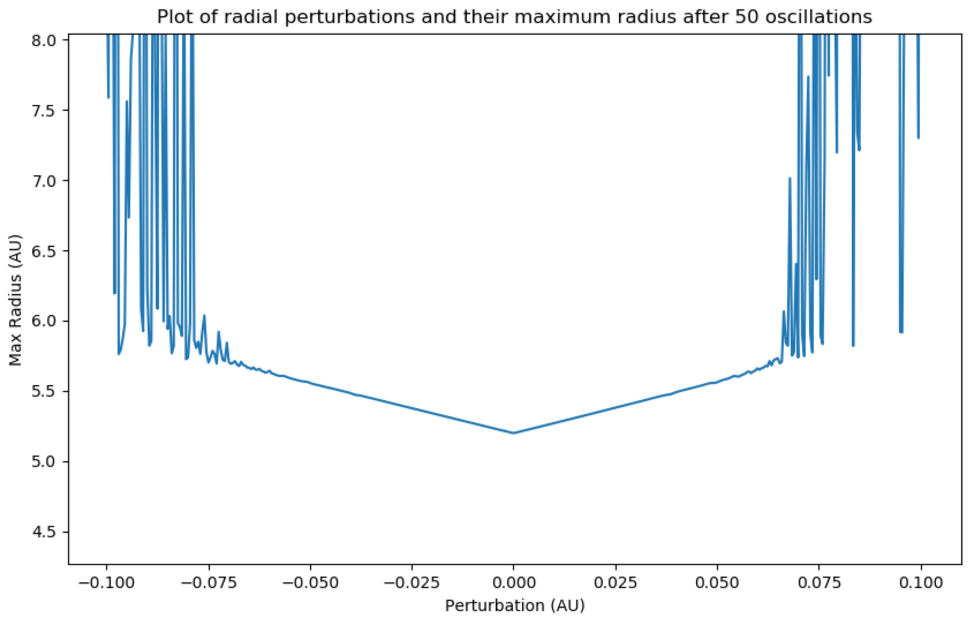
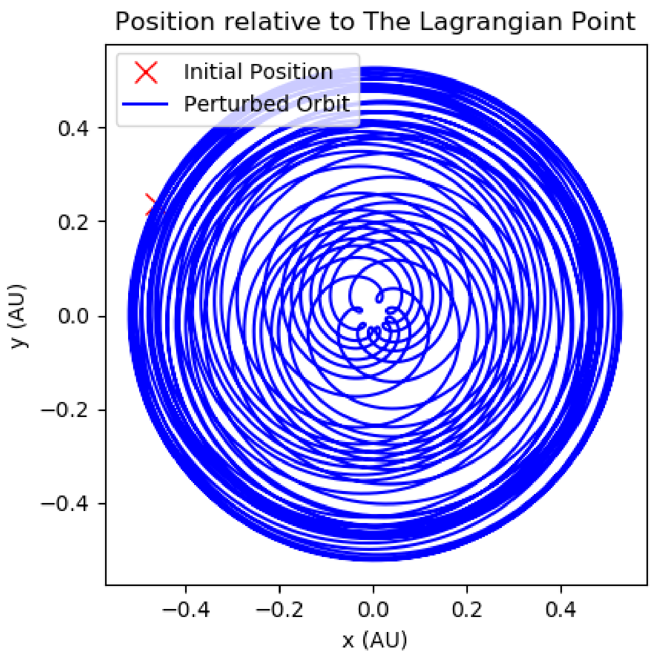


Figure 12: Plot of maximum radius after 50 orbits for a variety of radial perturbations. There is a linear rise in the maximum value until the orbits are no longer stable and the max values spike.

4.2 Angular Perturbations

Initially, a perturbation of $0.1 Rads$ was investigated



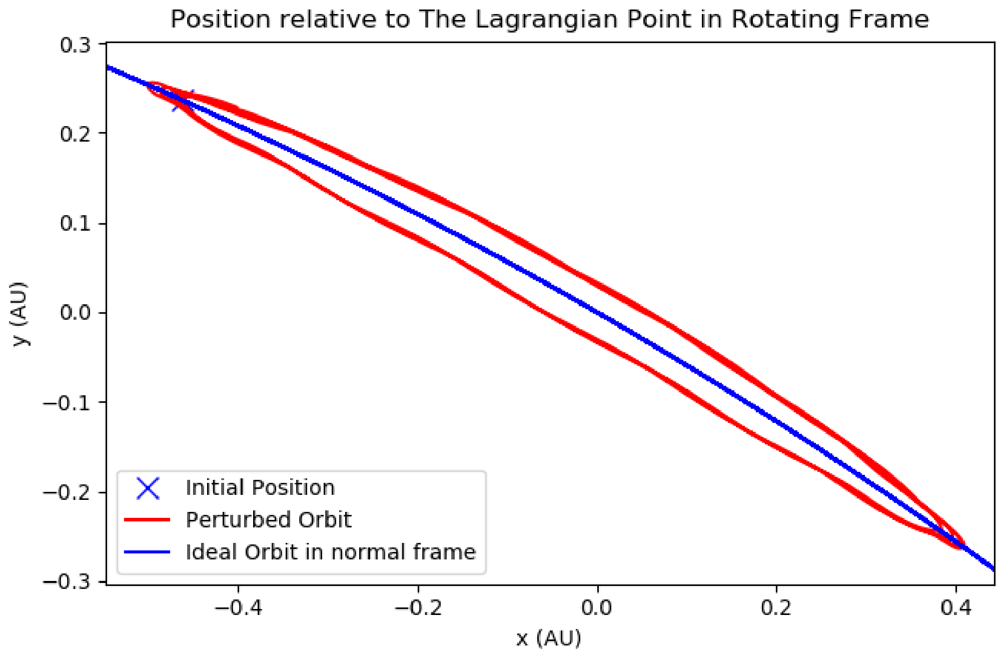


Figure 13: Plot of displacement from the ”ideal” Lagrangian point is the original reference frame and a rotating reference frame after 50 orbits. The perturbation was +0.1 Rads. Both plots show complicated patters with lots of loops, which oscillate around the ”ideal” point.

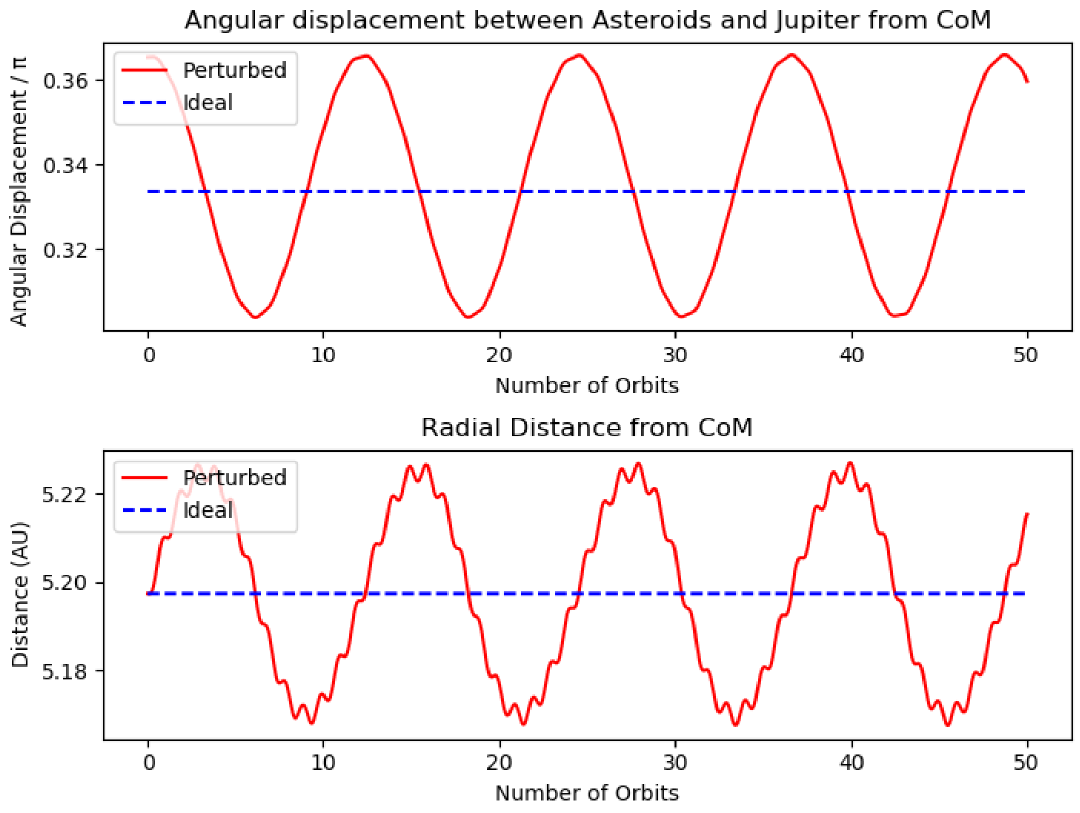


Figure 14: Plot of the radial distance and angular displacement of the perturbed asteroid from Jupiter, as measured from the CoM. The perturbation was 0.1 Rads, the plots show that both the radial distance and the angular displacements are affected by this. They both oscillate around the ”ideal” value.

Figure 14 shows that again, both the radial distance and angular displacements are affected. They both oscillate around the ”ideal” value. In the radial oscillations two distinct frequencies were seen as before, but the angular displacement only has one frequency visible. Similarly as before the radial distance and angular displacement initially increase, although the angular displacement only increases slightly.

The maximum and minimum values of the angular displacement were $0.3660 \pm 0.0001\pi$ and $0.3038 \pm 0.0001\pi$, which gives a midpoint of $0.3349 \pm 0.0001\pi$, slightly higher than the stable value of $0.3336 \pm 0.0001\pi$. The maximum and minimum values of the radius were $5.2270 \pm 0.0002AU$ and $5.1676 \pm 0.0002AU$ with a midpoint of $5.1973 \pm 0.0002AU$.

Figure 13 shows that the in the original frame of reference, the displacement is very chaotic and complicated, same as with a radial perturbation. In the rotating frame, the pattern is a lot simpler, making a loop around the ”Ideal Orbit in normal frame”.

As before, further perturbations were tested, and it was found again that the plots lost their symmetry as the perturbations increased, as shown in Figure 15. When the perturbation was increased to $+2\pi/3Rads$, it was found the orbit was still stable. This meant that all perturbations that are purely angular and increase the angle are stable as $\pi Rads$ is the maximum angle away from Jupiter.

Negative angular perturbations were also examined. At small negative values, the oscillations were the same as for small positive values. However at larger negative values, as the asteroid got closer to Jupiter, the perturbations became unstable.

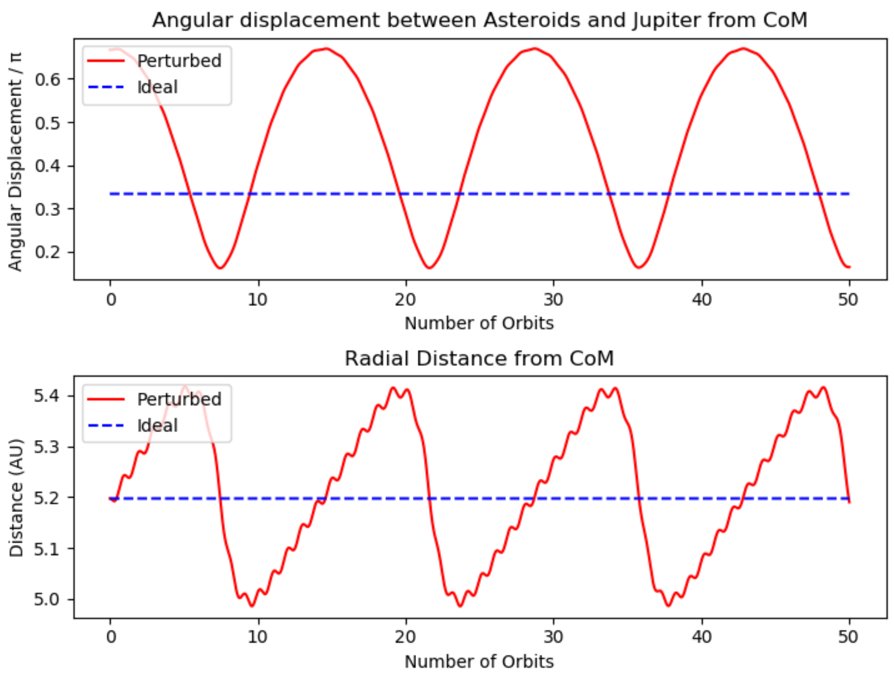


Figure 15: Plot of the radial distance and angular displacement of the perturbed asteroid from Jupiter, as measured from the CoM. The perturbation was $+\pi/3Rads$. They both oscillate around the "ideal" value.

To test the limits of stability, the same method as before was used. The perturbation was then increased to $+\pi/3Rads$. Figure 16 shows that all perturbations that are purely angular and increase the angle are stable as $+2\pi/3Rads$ is the maximum angle away from Jupiter. However negative perturbations become unstable at $-0.22 \pm 0.01\pi Rads$.

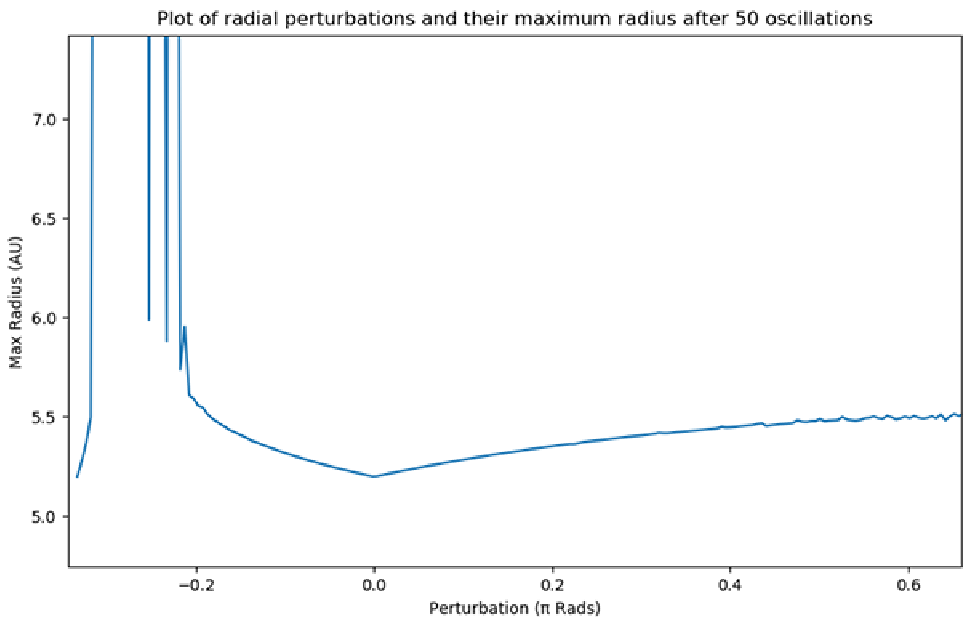


Figure 16: Plot of maximum radius after 100 orbits for a variety of angular perturbations.

4.3 Combining both Perturbations

Finally, both perturbations were applied to examine the points of stability. The program was looped for a variety of radial and angular perturbations, giving unique x and y initial conditions, and then the maximum distances were plotted for each initial x and y positions in a contour plot.

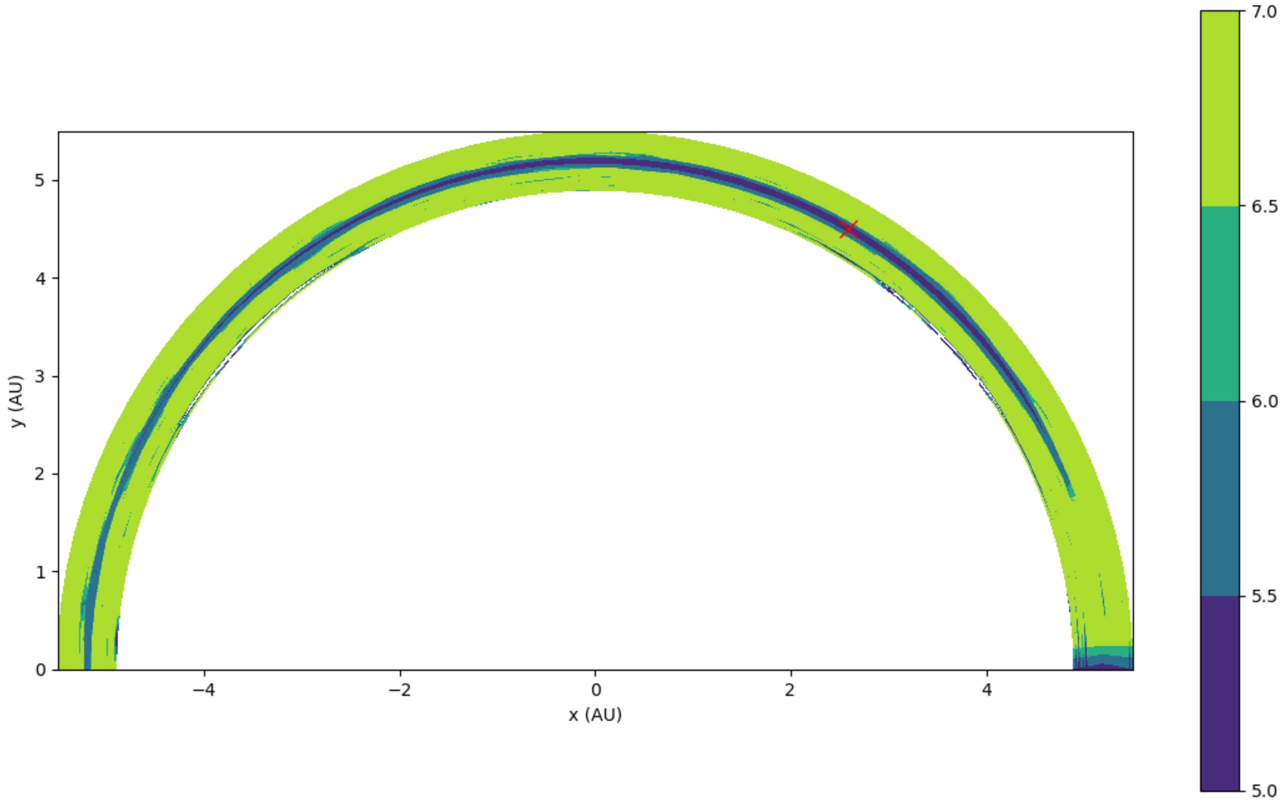


Figure 17: Contour plot of maximum radius of orbit for asteroid with (x, y) initial conditions. The maximum distance has been capped at 7 to provide a clearer plot of points of stability, hence the blue regions are stable and green regions are not.

In Figure 17, the maximum radius has been capped at 7 to give a clearer representation of points of stability. From figures 12 and 16, it can be assumed that orbits with a maximum radius higher than 6.5 are unstable. The plot clearly shows that large angular perturbations will still produce a stable orbit. Additionally, as you increase the angular perturbations, the range of radial perturbations that will produce a stable orbit decreases, as shown in Figure 21 by the blue area decreasing.

4.4 Stability Discussion

Switching to the rotating frame to examine the perturbations and using equation (6), for points away from the Lagrangian point there will be a resultant force. Initially this force will be away from the Lagrangian point, explaining why initially the perturbation increases, however once the asteroid is moving the Coriolis force will act perpendicular to the direction of motion. The combination of all the forces lead to the complicated orbits seen. The Coriolis force acting perpendicular to the direction of motion also explains how having only radial or angular perturbations leads to oscillations in both radial and angular directions, which further explains how both perturbations lead to very similar plots in angular displacement and radius.

The errors of $\pm 0.0002 AU$ for radial measurements and $\pm 0.0001\pi$ for angular measurements were chosen as that was how much the the values increased by after 100 orbits for a non perturbed orbit. The errors in the maximum perturbation values were chosen from the discrete steps in perturbation used when looping the program to calculate maximum radius.

4.5 The effect of the relative masses

The effect of the relative mass values on maximum perturbation was explored, through changing Jupiter's mass. The maximum radial perturbation and the maximum and minimum angular perturbations were measured for a variety of relative masses. The related plots are shown below.

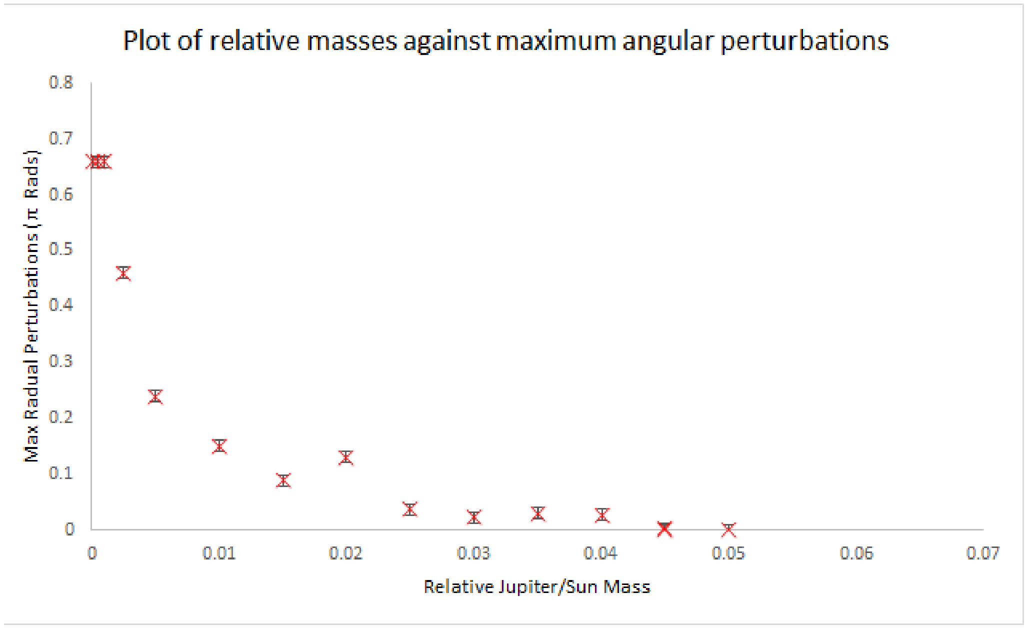


Figure 18: Plot of relative masses against maximum angular perturbation. At low relative masses, the maximum value of $+2\pi/3$ is allowed, however this quickly drops off and at a relative mass of 0.045, no angular perturbations are stable.

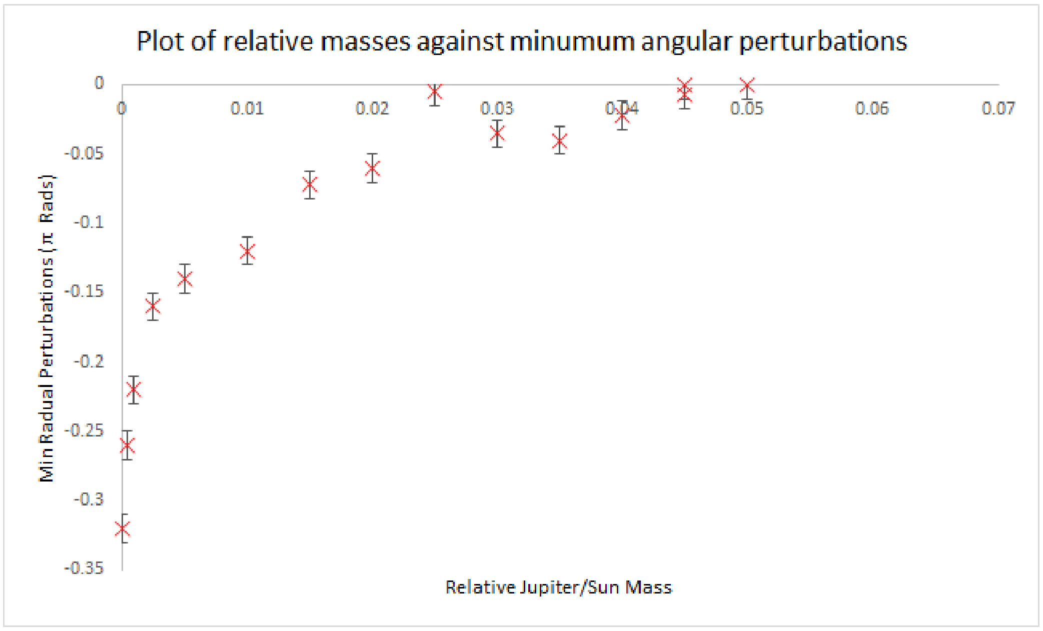


Figure 19: Plot of relative masses against minimum angular perturbation (Negative angular perturbation). At very low relative masses, a value very near to the maximum value of of $-\pi/3$ is allowed, however this quickly drops off and at a relative mass of 0.045, no angular perturbations are stable.

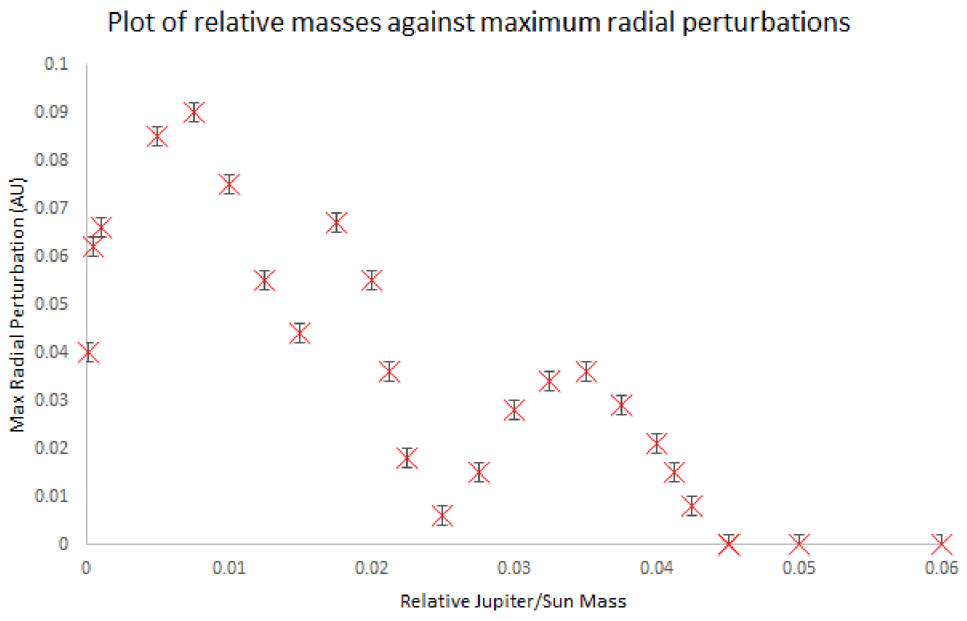


Figure 20: Plot of relative masses against maximum radial perturbation. Several peaks and troughs can be seen, and at relative mass of 0.045, no radial perturbations are stable.

Figure 20 shows that the maximum radial perturbation has several peaks and troughs at a variety of relative masses. There seems to be an overall peak at 0.0075, which allows a perturbation up to $0.90AU \pm 0.02$. At very low relative masses Figure 18 shows that the maximum angular perturbations of $+2\pi/3$ is allowed and the value very near to the maximum value of of $-\pi/3$ is also allowed, however the allowed perturbations decrease very quickly as the mass increases. For both radial and angular perturbations, there does not seem to be any allowed perturbations after the relative mass increases past 0.045. This is also seen in the contour plot for a relative mass of 0.005, Figure 21, where there are no points of stability outside the ideal Lagrangian point.

The increase in the allowed angular perturbations as the relative masses may be explained as the relative mass decreases, the system tends to a single body orbit, and the asteroid simply has a circular orbit around the sun.

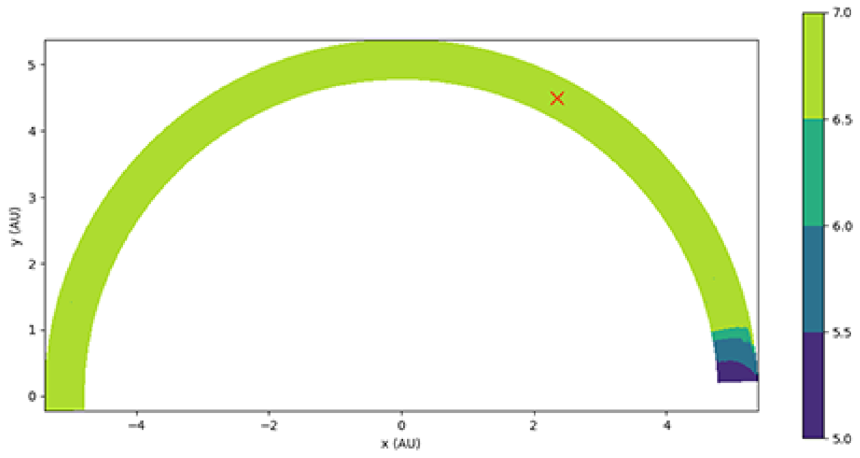


Figure 21: Contour plot of maximum radius of orbits for asteroid with (x, y) initial conditions for relative mass of 0.5. The maximum distance has been capped at 7 to provide a clearer plot of points of stability, hence the blue regions are stable and green regions are not. It can be seen there are no stable perturbations.

5 Performance

The bulk of the program's processing was solving the initial ODE's, as once this was done, all the coordinates and velocities of all bodies were known and all other information can be calculated with simple calculations. Therefore, the time taken to run the program was very dependant on the amount of orbits to solve for, hence most measurements were made for 50 orbits. Due to this, the program ran almost instantly (5s). However, this was not the case when calculating when the maximum perturbations, as the program had to be looped many times.

Therefore to minimise the time taken, the second asteroid was removed to reduce the number of ODEs and all unnecessary calculations were removed, as only the max radius was required. Additionally, 400 perturbations data points were chosen to have enough to plot while trying to minimise the number of loops. Even with these measures, the time taken was still fairly long (90s). However this was amplified further when the both radial and angular perturbations were applied simultaneously to create a contour plot, as the number of loops increased drastically. Due to this, only 50 perturbations values were taken for each perturbations leading to 2500 loops,the program took 15 minutes to run.

Additionally, the program was optimised by storing all data in arrays to speed up calculations, and information was stored in objects to make it simpler to access when required.

6 Conclusion

For the two body orbit of Jupiter and the Sun, the maximum allowed radial perturbations was $\pm 0.065 \pm 0.001 AU$ and the allowed angular perturbations were $+2\pi/3 \leq \theta \leq -0.2 \pm 0.01\pi$, where $+2\pi/3$ is the maximum possible positive perturbation. When combining both perturbations, the allowed radial values decreased as you increase the angular perturbations. The radial and angular perturbations were also closely linked, as applying only one perturbation led to oscillations in both the asteroid's angular displacement and radius.

When the relative masses of the two bodies Jupiter and the Sun decreased, the stable angular perturbations increased and continued increasing, whereas the peak of the radial perturbations occurred at a relative mass of 0.0075. When the relative mass was increased, the allowed angular perturbations very quickly decreased, and while the radial perturbations also decreased, it also had several smaller peaks and troughs. However at relative masses of 0.045 and higher, all perturbations became unstable.

References

[1] ”The Lagrange Points” by Neil J. Cornish for WMAP Education and Outreach 1998

[2] ”Lagrangian Points” by Dennis Westra, 2017

Appendices

A Wordcount

2991 Words

B Relevant Files

TrojanAstroids.py

Contains main code and most of the plots used.

TrojanAstroidSingleLoop.py

Used to calculate maximum perturbation for just radial or just angular perturbations.

TrojanAstroidDoubleLoop.py

Used to calculate maximum perturbation for both radial and angular perturbations and plot relevant contour plot.

C Code

```
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
from matplotlib import animation

"""
All in units of solar system units
"""
#Set up constants
G = 4*np.pi**2
massSun = 1
massJup = 0.001
massTotal= massSun + massJup
radius = 5.2

#Number of oscillations to plot
numOss = 100

#Perturbations
anglePert = 0  #Angular perturbations in Rads
radiusPert = 0  #Radial perturbation in AU



"""
Create class to store information about all bodies
(x position, y position, x velocity, y velocity)
"""
class Bodies:
    def __init__(self,x,y,vx,vy):
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy



"""
Create class to store distances between the different bodies
distance from (origin, sun, jupiter, greek, trojan)
```

```
"""
class Distances:
    def __init__(self,origin, sun, jupiter, greek, trojan):
        self.origin = origin
        self.sun = sun
        self.jupiter = jupiter
        self.greek = greek
        self.trojan = trojan


"""
Function for gravity containing all ODEs to solve
y is an array consisting of all positional and velocity information for all bodies
y= [jupiter.x , jupiter.y , jupiter.vx, jupiter.vy
    sun.x, sun.y , sun.vx, sun.vy,
    greek.x, greek.y, greek.vx, greek.vy
    trojan.x, trojan.y, trojan.vx, trojan.vy ]
"""
def gravity(y,t):
    global G, massSun, massJup
    #Create objects to hold information about each body
    jupiter = Bodies( y[0], y[1], y[2], y[3])
    sun= Bodies( y[4], y[5], y[6], y[7])
    greek= Bodies( y[8], y[9], y[10], y[11] )
    trojan= Bodies( y[12], y[13], y[14], y[15])

    #Calculate relevant distances
    rsj= (sun.x-jupiter.x)**2 + (sun.y-jupiter.y)**2 #Distance from sun to jupiter

    rsg= (sun.x-greek.x)**2 + (sun.y-greek.y)**2 #Distance from sun to Greek asteriods
    rjg= (jupiter.x-greek.x)**2 + (jupiter.y-greek.y)**2 #Distance from jupiter to Greek asteriods

    rst= (sun.x-trojan.x)**2 + (sun.y-trojan.y)**2 #Distance from sun to Trojan asteriods
    rjt= (jupiter.x-trojan.x)**2 + (jupiter.y-trojan.y)**2 #Distance from jupiter to Trojan
        asteriods

    #Sun and Jupiter two body orbit ODEs
    dx1dt, dy1dt = jupiter.vx, jupiter.vy
    dx2dt, dy2dt = sun.vx, sun.vy
    dvx1dt = G * massSun * (sun.x - jupiter.x) / rsj**(3/2)
    dvy1dt = G * massSun * (sun.y - jupiter.y) / rsj**(3/2)
    dvx2dt = G * massJup * (jupiter.x - sun.x) / rsj**(3/2)
    dvy2dt = G * massJup * (jupiter.y - sun.y) / rsj**(3/2)

    #Greek orbit ODEs
    dx3dt, dy3dt= greek.vx, greek.vy
    dvx3dt = G * ( (massSun * (sun.x - greek.x) / rsg**(3/2)) + (massJup * (jupiter.x - greek.x) /
        rjg**(3/2)) )
    dvy3dt = G * ( (massSun * (sun.y - greek.y) / rsg**(3/2)) + (massJup * (jupiter.y - greek.y) /
        rjg**(3/2)) )

    #Trojan Orbit ODEs
    dx4dt, dy4dt = trojan.vx, trojan.vy
    dvx4dt = G * ((massSun * (sun.x - trojan.x) / rst ** (3 / 2)) + (massJup * (jupiter.x -
        trojan.x) / rjt ** (3 / 2)))
    dvy4dt = G * ((massSun * (sun.y - trojan.y) / rst ** (3 / 2)) + (massJup * (jupiter.y -
        trojan.y) / rjt ** (3 / 2)))

    return dx1dt, dy1dt, dvx1dt, dvy1dt, \
          dx2dt, dy2dt, dvx2dt, dvy2dt, \
          dx3dt, dy3dt, dvx3dt, dvy3dt, \
          dx4dt, dy4dt, dvx4dt, dvy4dt
```

```
"""
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Calculate Initial Conditions
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
"""

#Angluar displacement of asteriods from Sun
greekInitialAngle = np.pi / 3
trojanInitialAngle = -np.pi / 3

"""
Jupiter and Sun Initial Conditions
"""
sunInitialdis = radius * massJup / (massSun + massJup) #Distance between Sun and CoM
jupInitialdis = radius * massSun / (massSun + massJup) #Distance between Jupiter and CoM

jupInitialv = np.sqrt(G * massSun**2 / (radius * (massSun + massJup))) #Jupiter initial velocity
sunInitialv = -jupInitialv * massJup / massSun                    #Sun Initial Velocity

#Period for orbits. Should be the same
angularVelocity = jupInitialv / jupInitialdis
jupPeriod = 2*np.pi*jupInitialdis / jupInitialv
sunPeriod = 2*np.pi*sunInitialdis / sunInitialv

"""
Asteroid Initial Conditions
"""

"""
Function to calculate radius and angle from CoM to get initial positions for asteroids
Input = (Distance of Asteroid from sun , Distance of sun from CoM, Distance of Jupiter from CoM,
    Angle of asteroids from Sun)
Output = (Distance of Asteroid from CoM , Angle of Asteroids from CoM)
"""
def fromCom(radiusFromSun, sunComDis, jupComDis, angle):
    comdis = np.sqrt(sunComDis**2 + radiusFromSun**2 - 2 * sunComDis * radiusFromSun *
        np.cos(angle))
    jupdis = np.sqrt(2 * radius**2 * (1 - np.cos(angle)) )
    comangle = np.sign(angle) * np.arccos((comdis**2 + jupComDis**2 - jupdis**2) / (2 * comdis *
        jupComDis))

    return comdis, comangle

"""
Greek asteriod initial conditions
"""
greekComInitial = fromCom(radius, sunInitialdis, jupInitialdis, greekInitialAngle)
#(Initial distance of Greek from CoM , Initial angle of Greek from CoM)
greekComInitial = (greekComInitial[0] +radiusPert, greekComInitial[1] + anglePert) #Apply
    Perturbations. Only greek is perturbed
greekInitialVelocity = jupInitialv * (greekComInitial[0]) / jupInitialdis   #Initial Greek velocity

#Create object to store all initial conditions
greekInitial = Bodies(np.cos(greekComInitial[1] ) * (greekComInitial[0]),
                   np.sin(greekComInitial[1] ) * (greekComInitial[0]),
                   -np.sin(greekComInitial[1] ) * greekInitialVelocity,
                   np.cos(greekComInitial[1] ) * greekInitialVelocity)


"""
Trojan asteriod initial conditions
"""
trojanComInitial = fromCom(radius, sunInitialdis, jupInitialdis, trojanInitialAngle)
#(Initial distance of Trojan from CoM , Initial angle of Trojan from CoM)
trojanInitialVelocity = jupInitialv * trojanComInitial[0]/ jupInitialdis #Initial Trojan velocity
```

```
#Create object to store all initial conditions
trojanInitial = Bodies(np.cos(trojanComInitial[1] ) * trojanComInitial[0] ,
                    np.sin(trojanComInitial[1] ) * trojanComInitial[0],
                    -np.sin(trojanComInitial[1] ) * trojanInitialVelocity,
                    np.cos(trojanComInitial[1] ) * trojanInitialVelocity )




"""
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Solving ODEs
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
"""
#Times to evaluate ODE based on number of ossicillations
t = np.linspace(0, numOss * jupPeriod, 1000*numOss)

#Solve ODEs using odeint
solution = scipy.integrate.odeint(gravity,[jupInitialdis, 0, 0, jupInitialv,
                                     -sunInitialdis, 0, 0, sunInitialv,
                                     greekInitial.x, greekInitial.y, greekInitial.vx,
                                         greekInitial.vy,
                                     trojanInitial.x, trojanInitial.y, trojanInitial.vx,
                                         trojanInitial.vy]
                              ,t)

"""
Create objects to store information for each orbits
Each is an array that holds all x, y, vx, vy information for all times t
for each body
"""
jupiterOrbit = Bodies(solution[:,0],solution[:,1],solution[:,2],solution[:,3])
sunOrbit = Bodies(solution[:,4],solution[:,5],solution[:,6],solution[:,7])
greekOrbit = Bodies(solution[:,8],solution[:,9],solution[:,10],solution[:,11])
trojanOrbit = Bodies(solution[:,12],solution[:,13],solution[:,14],solution[:,15])



"""
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Measurements
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
"""

"""
Create objects to store distance between all bodies
distance from (origin, sun, jupiter, greek, trojan)
"""
disSun = Distances(np.sqrt(sunOrbit.x**2 + sunOrbit.y**2),
                    np.sqrt((sunOrbit.x - sunOrbit.x)**2 + (sunOrbit.y - sunOrbit.y)**2),
                    np.sqrt((sunOrbit.x - jupiterOrbit.x)**2 + (sunOrbit.y - jupiterOrbit.y)**2),
                    np.sqrt((sunOrbit.x - greekOrbit.x)**2 + (sunOrbit.y - greekOrbit.y)**2),
                    np.sqrt((sunOrbit.x - trojanOrbit.x)**2 + (sunOrbit.y - trojanOrbit.y)**2) )

disJupiter = Distances(np.sqrt(jupiterOrbit.x**2 + jupiterOrbit.y**2),
                    np.sqrt((jupiterOrbit.x - sunOrbit.x)**2 + (jupiterOrbit.y - sunOrbit.y)**2),
                    np.sqrt((jupiterOrbit.x - jupiterOrbit.x)**2 + (jupiterOrbit.y -
                        jupiterOrbit.y)**2),
                    np.sqrt((jupiterOrbit.x - greekOrbit.x)**2 + (jupiterOrbit.y -
                        greekOrbit.y)**2),
                    np.sqrt((jupiterOrbit.x - trojanOrbit.x)**2 + (jupiterOrbit.y -
                        trojanOrbit.y)**2) )

disGreek = Distances(np.sqrt(greekOrbit.x**2 + greekOrbit.y**2),
```

```
                    np.sqrt((greekOrbit.x - sunOrbit.x)**2 + (greekOrbit.y - sunOrbit.y)**2),
                    np.sqrt((greekOrbit.x - jupiterOrbit.x)**2 + (greekOrbit.y -
                        jupiterOrbit.y)**2),
                    np.sqrt((greekOrbit.x - greekOrbit.x)**2 + (greekOrbit.y - greekOrbit.y)**2),
                    np.sqrt((greekOrbit.x - trojanOrbit.x)**2 + (greekOrbit.y -
                        trojanOrbit.y)**2) )

disTrojan = Distances(np.sqrt(trojanOrbit.x**2 + trojanOrbit.y**2),
                    np.sqrt((trojanOrbit.x - sunOrbit.x)**2 + (trojanOrbit.y - sunOrbit.y)**2),
                    np.sqrt((trojanOrbit.x - jupiterOrbit.x)**2 + (trojanOrbit.y -
                        jupiterOrbit.y)**2),
                    np.sqrt((trojanOrbit.x - greekOrbit.x)**2 + (trojanOrbit.y -
                        greekOrbit.y)**2),
                    np.sqrt((trojanOrbit.x - trojanOrbit.x)**2 + (trojanOrbit.y -
                        trojanOrbit.y)**2) )


"""
Calculate the angle from x axis
Function to calculate angle as need to apply corrections depending on which quadrant it is on due
    to how arctan function works
"""
def angleCalc(x1,y1):
    #First move origin of coordinates to the sun
    deltax = x1
    deltay = y1
    #Theta calculated seperately in each quadrant due to arctan function
    theta = []
    for i in range(len(jupiterOrbit.x)):
        if deltay[i] > 0 and deltax[i] > 0: # Q1
            theta.append(np.arctan(abs(deltay[i] / deltax[i])))
        elif deltay[i] > 0 and deltax[i] < 0: # Q2
            theta.append(np.pi - np.arctan(abs(deltay[i] / deltax[i])))
        elif deltay[i] < 0 and deltax[i] < 0: # Q3
            theta.append(np.pi + np.arctan(abs(deltay[i] / deltax[i])))
        elif deltay[i] < 0 and deltax[i] > 0: # Q4
            theta.append(2 * np.pi - np.arctan(abs(deltay[i] / deltax[i])))
        else:
            theta.append(np.arctan(deltay[i] / deltax[i]))
    theta = np.array(theta)
    return theta

#Use above function to find angles for each orbits
jupAngle = angleCalc(jupiterOrbit.x, jupiterOrbit.y)
greekAngle = angleCalc(greekOrbit.x, greekOrbit.y)


"""
Calculate "Ideal" Lagrangian Point
"""
greekComIdeal = fromCom(radius, sunInitialdis, jupInitialdis, np.pi/3)
#(Initial distance of Ideal Point from CoM , Initial angle of Ideal Point from CoM)

#Create object to store infromation about Ideal Orbit
#The points are always ideal angle away from Jupiter's Orbit
greekIdealOrbit = Bodies(np.cos(greekComIdeal[1]+jupAngle)*greekComIdeal[0],
                  np.sin(greekComIdeal[1]+jupAngle)*greekComIdeal[0],
                  0,
                  0)

#Create object to store distance infromation for Ideal Orbit
disGreekIdeal = Distances(np.sqrt(greekIdealOrbit.x**2 + greekIdealOrbit.y**2),
                    np.sqrt((greekIdealOrbit.x - sunOrbit.x)**2 + (greekIdealOrbit.y -
                        sunOrbit.y)**2),
                    np.sqrt((greekIdealOrbit.x - jupiterOrbit.x)**2 + (greekIdealOrbit.y -
                        jupiterOrbit.y)**2),
                    np.sqrt((greekIdealOrbit.x - greekOrbit.x)**2 + (greekIdealOrbit.y -
                        greekOrbit.y)**2),
                    np.sqrt((greekIdealOrbit.x - trojanOrbit.x)**2 + (greekIdealOrbit.y -
                        trojanOrbit.y)**2) )

#Difference in x and y from ideal orbit
disFromIdealx = greekOrbit.x - greekIdealOrbit.x
disFromIdealy = greekOrbit.y - greekIdealOrbit.y


#Calculate angle differences between Jupiter and asteriods from Origin
greekJupAngle= []   #List to store values

for i in range(len(jupAngle)):
    if abs(greekAngle[i] - jupAngle[i]) > np.pi:
        #Need to account for when a body finishes orbit and angle is reset to 0
        greekJupAngle.append(greekAngle[i] + 2*np.pi - jupAngle[i])
    else:
        greekJupAngle.append(greekAngle[i] - jupAngle[i])

greekJupAngle = np.array(greekJupAngle)



"""
Transform into rotating reference frame
Create mew bodies to store relevant information
"""
newFrameJupiterOrbit = Bodies(jupiterOrbit.x *np.cos(angularVelocity*t) + jupiterOrbit.y
    *np.sin(angularVelocity*t),
                    -jupiterOrbit.x *np.sin(angularVelocity*t) + jupiterOrbit.y
                        *np.cos(angularVelocity*t),
                    jupiterOrbit.vx *np.cos(angularVelocity*t) - jupiterOrbit.x *
                        angularVelocity * np.sin(angularVelocity*t)
                        + jupiterOrbit.vy*np.sin(angularVelocity*t) + jupiterOrbit.y *
                            angularVelocity * np.cos(angularVelocity*t),
                    -jupiterOrbit.vx *np.sin(angularVelocity*t) - jupiterOrbit.x *
                        angularVelocity *np.cos(angularVelocity*t)
                        + jupiterOrbit.vy*np.cos(angularVelocity*t) - jupiterOrbit.y *
                            angularVelocity * np.sin(angularVelocity*t))

newFrameSunOrbit = Bodies(sunOrbit.x *np.cos(angularVelocity*t) + sunOrbit.y
    *np.sin(angularVelocity*t),
                    -sunOrbit.x *np.sin(angularVelocity*t) + sunOrbit.y
                        *np.cos(angularVelocity*t),
                    sunOrbit.vx *np.cos(angularVelocity*t) - sunOrbit.x * angularVelocity *
                        np.sin(angularVelocity*t)
                        + sunOrbit.vy*np.sin(angularVelocity*t) + sunOrbit.y * angularVelocity *
                            np.cos(angularVelocity*t),
                    -sunOrbit.vx *np.sin(angularVelocity*t) - sunOrbit.x * angularVelocity
                        *np.cos(angularVelocity*t)
                        + sunOrbit.vy*np.cos(angularVelocity*t) - sunOrbit.y * angularVelocity *
                            np.sin(angularVelocity*t))

newframeGreekOrbit = Bodies(greekOrbit.x *np.cos(angularVelocity*t) + greekOrbit.y
    *np.sin(angularVelocity*t),
                    -greekOrbit.x *np.sin(angularVelocity*t) + greekOrbit.y
                        *np.cos(angularVelocity*t),
                    greekOrbit.vx *np.cos(angularVelocity*t) - greekOrbit.x * angularVelocity *
                        np.sin(angularVelocity*t)
                        + greekOrbit.vy*np.sin(angularVelocity*t) + greekOrbit.y *
                            angularVelocity * np.cos(angularVelocity*t),
                     -greekOrbit.vx *np.sin(angularVelocity*t) - greekOrbit.x * angularVelocity
                         *np.cos(angularVelocity*t)
                        + greekOrbit.vy*np.cos(angularVelocity*t) - greekOrbit.y *
                            angularVelocity * np.sin(angularVelocity*t))


"""
Calculate Max and Min values"""

maxAngle = np.max(greekJupAngle) / np.pi
minAngle = np.min(greekJupAngle) / np.pi
midAngle = (maxAngle + minAngle) / 2
maxDis = np.max(disGreek.origin)
minDis = np.min(disGreek.origin)
midDis = (maxDis + minDis) / 2
```