Candidate number: 7000Q

Stability of the Trojan asteroids

Abstract

The circular restricted three body problem is solved numerically using an adaptive fourth order Runge-Kutta technique. Limits are placed on the stability of the Lagrange points, primarily through considerations of the wander in the radial and azimuthal directions. It was determined that a test mass placed at the two triangular Lagrange points, L_4 and L_5, would remain in this position. Small perturbations to this position yielded orbits encompassing the Lagrange point, known as a tadpole orbit, while larger perturbations lead to wander between L_{3-5}, known as a horseshoe orbit. Additionally, a threshold mass around $0.04M_{\odot}$, was found, above which orbits quickly became unstable, and additional resonant behaviour was observed at lower masses.

1. Introduction

The three-body problem has been a significant subject of mathematicians' and scientists' attention for centuries, with perhaps the most significant contributions by Euler and Lagrange **[1]**. While no general solutions exist, they determined that for a corotating system of two massive bodies, five equilibrium positions exist for a third body of negligible mass, known as the Lagrange points **[2]**. The latter two, L_4 and L_5, are of particular interest as a result of their stability, leading to the existence of stable orbits around these points.

Orbits of this nature have been observed around a number of planets, including the Trojan asteroids, which librate about L_4 and L_5 in the Jupiter-Sun corotating frame **[1]**. These asteroids typically have low albedo, making them difficult to detect **[3]**. However, estimates through extrapolation suggest that the Trojan asteroids may be as numerous as half the number of asteroids in the main belt, and so they play an important role in the dynamics of Jupiter and other surrounding bodies **[4]**.

The motion of the Trojan asteroids is therefore a field highly suited to dynamical simulations, allowing studies of properties that are largely inaccessible via physical measurements. This can provide insight into the origins and evolution of the Trojan asteroids, as well as their role in the formation of Jupiter itself, since a number of competing theories exist in both cases **[4]**. These simulations can also readily be extended to a generalised set of three or more bodies, including Saturn's moons, Janus and Epimetheus, who's orbits encompass the L_{3-5} points, unlike any known Jovian body **[1]**. Such studies can also further our understanding of the Earth's Lagrange points, which are exploited in the positioning of satellites **[5]**.

This paper focuses on the stability of the Trojan asteroids with respect to displacement and velocity perturbations from the stable Lagrange points. Analysis is then extended to planets of masses differing to that of Jupiter. Investigation is carried out through the use of an adaptive Runge-Kutta technique of order 4(5) to numerically solve the equations of motion. These

equations primarily correspond to the circular, restricted three-body problem (CR3BP), due to the complexity of general three-body problems, but there is also a limited discussion of motion out of the plane.

The next section summarises the computational analysis of the problem performed to achieve solutions. A description of the implementation of algorithms and performance is given in section 3. Section 4 contains the results of simulations and a discussion of these results, before conclusions are presented in section 5.

2. Analysis

2.1. Analytical Solution

Although the orbit of Jupiter around the Sun is an ellipse, its eccentricity is small, allowing the motion to be approximated as circular while successfully describing the expected behaviour **[1]**. The asteroid mass is regarded as negligible compared to the two massive bodies, and if the motion of the three bodies is taken to be coplanar, as is the case for the focus of analysis to follow, this set of approximations is collectively known as the circular, restricted three-body problem (CR3BP) **[1]**.

The two massive bodies rotate about their centre of mass, known as the barycentre, in circular orbits with angular frequency, ω **[6]**:

$$\omega^2 R^3 = G(M_s + M_j)$$

where R is the separation of the Sun and Jupiter, G is the gravitational constant and M_s and M_j are the masses of the Sun and Jupiter, respectively. If the barycentre is located at the origin, the positions of the Sun, $\vec{\boldsymbol{r}}_s$, and Jupiter, $\vec{\boldsymbol{r}}_j$, defined along the x-axis are given by **[6]**:

$$\vec{\boldsymbol{r}}_s = \left(\frac{-M_j}{M_j + M_s} R, 0, 0\right)$$

$$\vec{\boldsymbol{r}}_j = \left(\frac{M_s}{M_j + M_s} R, 0, 0\right)$$

In the inertial frame of reference, the motion of the asteroid is dominated by rotation about the barycentre, making studies of its motion difficult to resolve **[7]**. Instead, the problem may be solved in corotating frame of reference, where both the Sun and Jupiter remain stationary. In this rotating frame, there exist equilibrium points, known as Lagrange points, where the gravitational attraction of the Sun and Jupiter provides a resultant force on the asteroid towards the centre of mass, such that the centripetal acceleration causes the asteroid to orbit with the same period as Jupiter **[1]**.

Through consideration of the fictitious Coriolis and centrifugal forces, the apparent acceleration of the asteroid can be expressed **[6]**:

$$\frac{d^2\vec{\boldsymbol{r}}}{dt^2} = -\frac{GM_s(\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_s)}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_s|^3} - \frac{GM_j(\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_j)}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_j|^3} - 2\left(\vec{\boldsymbol{\omega}} \times \frac{d\vec{\boldsymbol{r}}}{dt}\right) - \vec{\boldsymbol{\omega}} \times (\vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{r}})$$

where $\vec{\boldsymbol{r}}$ is the position vector of the asteroid confined to the x-y plane, and $\vec{\boldsymbol{\omega}}$ is the (constant) angular frequency vector, in the z-direction. Rather than solving this set of second-order equations of motion, a solution may be obtained through a set of six coupled first-order differential equations:

$$\frac{d\vec{\boldsymbol{r}}}{dt} = \vec{\boldsymbol{s}}$$

$$m\frac{d\vec{\boldsymbol{s}}}{dt} = -\frac{GM_s(\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}_s})}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}_s}|^3} - \frac{GM_j(\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}_j})}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}_j}|^3} - 2m(\omega \times \vec{\boldsymbol{s}}) - m\omega \times (\omega \times \vec{\boldsymbol{r}})$$

In this restricted system, neither energy nor angular momenta is conserved. Instead, it can be shown (Appendix A) that the Hamiltonian, which is closely related to the Jacobi integral, is a constant of motion:

$$\mathcal{H} = \frac{m}{2}\left(\frac{d\vec{\boldsymbol{r}}}{dt}\right)^2 - \frac{m}{2}|\vec{\boldsymbol{\omega}}|^2|\vec{\boldsymbol{r}}|^2 + U(\vec{\boldsymbol{r}})$$

where

$$U(\vec{\boldsymbol{r}}) = -\frac{GM_s}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_s|} - \frac{GM_j}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_j|}$$

Five Lagrange equilibrium points exist in the rotating frame, referred to as L_{1-5} **[1]**. Of particular interest are L_4 and L_5, located at $\pi/3$ radians preceding and trailing the planet respectively. These points, which are equivalent due to the symmetry of the system with respect to reflections in the x-axis, correspond to points of stable equilibrium. L_{1-3}, on the other hand, are unstable **[8]**. With the barycentre at the origin, the Cartesian coordinates of $L_{4,5}$ are given by **[1]**

$$L_{4,5} = \left(\frac{1}{2}\frac{M_s - M_j}{M_s + M_j}R, \pm\frac{\sqrt{3}}{2}R, 0\right)$$

The five Lagrange points are illustrated in figure 1, with the zero velocity curves superimposed.

These curves (or surfaces in 3D) represent points where velocity, v, is 0, and so bound regions where $v^2 < 0$ **[8]**. Since there is no real solution in these regions, they are excluded from the particles motion. As a result, it is common for orbits close to the Lagrange points to resemble these curves (or surfaces), forming "tadpole" orbits that encompass either L_4 or L_5, as well as "horseshoe" orbits that encompass L_{3-5} **[8]**.

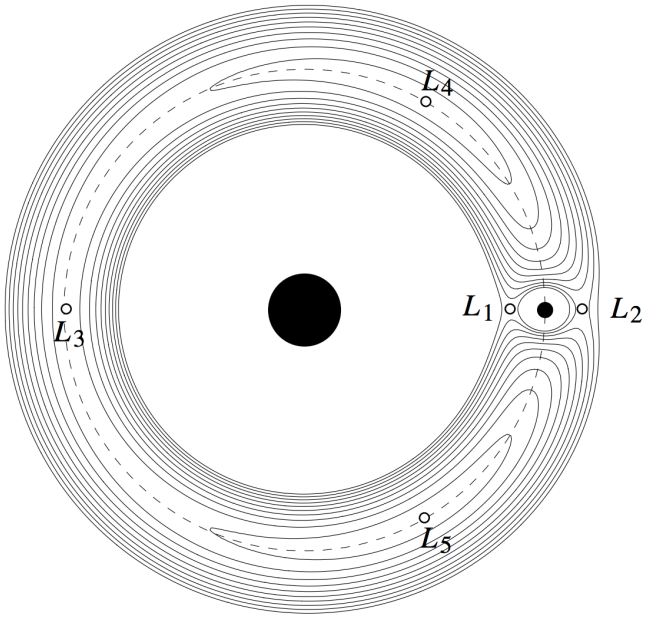


Figure 1: The locations of Lagrange points (open circles) and the zero-velocity curves for a reduced mass of $0.01M_\odot$ [1]. The dashed line represents a circle of radius equal to the separation of the two masses, centred on the larger mass.

2.2. Scaling and choice of routines

Units for this problem were scaled such that they were of order unity, allowing more accurate calculations to be performed by avoiding round-off errors. The units chosen were solar system units, where the mass of the Sun ($M_\odot$) is taken as unit mass, unit distance is the astronomical unit (AU) and unit time is one year. For this system, the gravitational constant is given by $G = 4\pi^2$. The mass of Jupiter, was taken to be $0.001M_\odot$, while 5.2 AU was used as the average distance between Jupiter and Sun.

Integration was performed using the DOPRI5 subroutine through the scipy.integrate.ode interface class. This performs an explicit Runge-Kutta method of order 4(5) based on the Dormand and Prince method **[9]**. This integrator uses an adaptive step size, which is particularly useful for this problem due to the significant variation in the time-scale of the asteroid's motion along its path. Fewer steps can be taken while maintaining the desired precision, and so higher precision calculations can therefore be performed within a reasonable timescale.

Performance of the DOPRI5 subroutine was compared to scipy.integrate.odeint. This function makes use of lsoda from the FORTRAN library odepack **[10]**, a well-established integrator that automatically switches between the Adams method, a non-stiff solver, and the Backward Differentiation Formula (BDF) method, a stiff solver, depending on the behaviour of the problem **[11]**. Results obtained using both integrators were comparable, integrations performed by DOPRI5 are successful.

3. Implementation

3.1. Overall approach

Both the integration and plotting routines were written in python, making use of a number of standard libraries, including NumPy, SciPy and Matplotlib. Where possible, the algorithm was designed to be flexible. For example, massive bodies are defined by a class, providing a foundation for the addition of further bodies to the system. It is also possible to vary multiple parameters simultaneously, such as giving the asteroid a range of initial displacements and velocities.

Due to this flexibility, the algorithm makes use of a large number of parameters. It was therefore decided to hardcode parameters, rather than pass them through a command line. The number of parameters could be reduced through the use of default parameters, but specifying which variables are of interest would become difficult if multiple variables ranges were changing, as is possible in the current implementation.

The integration requires the equation of motion to be cast as six coupled first-order differential equations. Using the set_solout functionality of the integrator, a callable is defined, to be called at every successful integration step. Through this, intermediate time steps were recorded, and integration of unstable orbits was stopped when caught for efficiency.

Wander was measured in three components: radial wander, angular wander and z wander, each measured in AU for direct comparison. Radial wander represents the difference between the maximum and minimum radial distance of the asteroid from the origin, and z wander represents the difference between the maximum and minimum value of z for the asteroid. Angular wander represents the arc length that corresponds to the difference between the asteroid's maximum and minimum angle on a circle of radius equal to the Lagrange point's radius. Angles are defined from $-\pi/3$ to $5\pi/3$, such that the specified Lagrange point had an angle of 0 and the angular discontinuity on the positive x-axis. This meant that angles could be referenced relative to the starting position more easily, and angles were continuous for any orbits of interest.

Limits are placed on the angular wander (2π), more arbitrarily, the radial wander (2.5R). If the integration is stopped, the maximum angular wander is assigned, but the radial wander may not be entirely representative, and so angular wander is typically the most useful measure.

By default, the initial conditions were defined in a consistent manner for both Lagrange points, with positive angles corresponding to anticlockwise motion. However, as L_4 and L_5 are

symmetric, calculations of wanders can be averaged between the two points. In this case, the initial conditions are applied to the selected Lagrange point, before being reflected where necessary to produce the equivalent initial position for the second Lagrange point as the first. In practise, this corresponds to multiplying each element of the angular displacement and velocity arrays by -1.

3.2. Performance

Non-integer data are stored as double-precision floating point numbers, allowing sufficient accuracy without taking up excessive memory. Where possible, explicit loops were avoided, and instead problems were vectorised, making use the NumPy library to greatly improve the efficiency of calculations. However, while this was feasible for almost all calculations, loops over sets of different initial conditions were unavoidable.

The most computationally expensive feature is the integrator itself, which dominates the total time taken. An example is a total run time of 1.4353s, where 1.4330s was taken up by the integration function alone. Although the break condition can save time in certain cases, run times of the order of 1-2s per loop are typical for the given integration parameters: tmax = 1000 years, atol = rtol = 10^{-6} and N = 50,000, where tmax is the maximum time, atol and rol are the relative and absolute tolerances respectively, and N is the maximum number of (internally defined) steps during a call to the solver. These conditions were chosen to provide solutions of the desired accuracy, while allowing calculations to be performed in a reasonable time.

4. Results and Discussion

4.1. Errors

The two primary sources of error in the algorithm are roundoff errors, due to the finite number of bits used to represent floating point numbers, and truncation errors, which result from the finite step size of the integration **[12]**. Since roundoff errors can occur during any mathematical operation involving floating point numbers, this error is expected to accumulate, increasing as the number of integration steps is increased. Truncation errors, on the other hand, are expected to decrease with the number of integration steps. This error tends to zero as the number of steps tends to infinity, since this corresponds to the function being continuously sampled **[12]**.

It is therefore necessary to consider these competing effects when determining an appropriate step size, as while initially the truncation error dominates, the roundoff error eventually dominates as step size is decreased. All floating-point numbers were defined with double precision, resulting in errors of the order of 10^{-16} for each instance of a floating-point number **[12]**. Truncation errors are estimated by the integrator, which adapts the step size such that its estimates are consistent with the input relative and absolute tolerances, known as rtol and atol respectively.

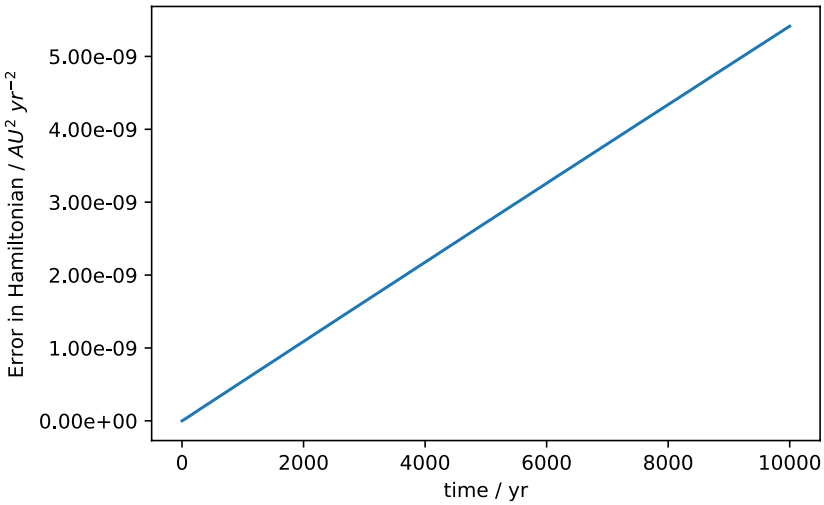
In order to maintain feasible runtimes for large number of conditions, rtol and atol were typically relatively large. rtol = atol = 10^{-6} was selected as a default, as results with this tolerance appeared consistent with higher precision measurements for stable orbits. It is expected that truncation errors of this magnitude would dominate roundoff errors, but these errors were considered acceptable in most cases.

Although the integrator estimates the dominant error, it is useful to quantify errors in a method independent of the solver. Estimates of the total error accumulated were therefore made using the Hamiltonian, $\mathcal{H}$. The Hamiltonian is a constant of motion, and so its variation can be used to measure the propagation of errors. As the asteroid is of constant but negligible mass, $\mathcal{H}$ is calculated per unit mass at every time step. This allows the range of values it takes to be determined through calculation of the difference between the maximum and minimum.

Examples of the time evolution of the calculated Hamiltonian's error are illustrated in figures 2a and 2b, based on the assumption that the initial value calculated is the true value. A clear increase in accuracy is obtained through decreasing the tolerance, but despite the relatively low tolerance of the former graph, the simulated solution is still highly accurate, with the maximum variation corresponding to less than $10^{-7}\ AU^2\ yr^{-2}$ over an integration time of 10,000 years.

The Hamiltonian is typically of the order of $10\ AU^2\ yr^{-2}$, while quantities such as distances were chosen to be of order unity, and so this variation represents a negligible error in stable situations. When plotting over a number of initial conditions, a warning message is given if the variation is greater than $10^{-4}\ AU^2\ yr^{-2}$.

a)



b)

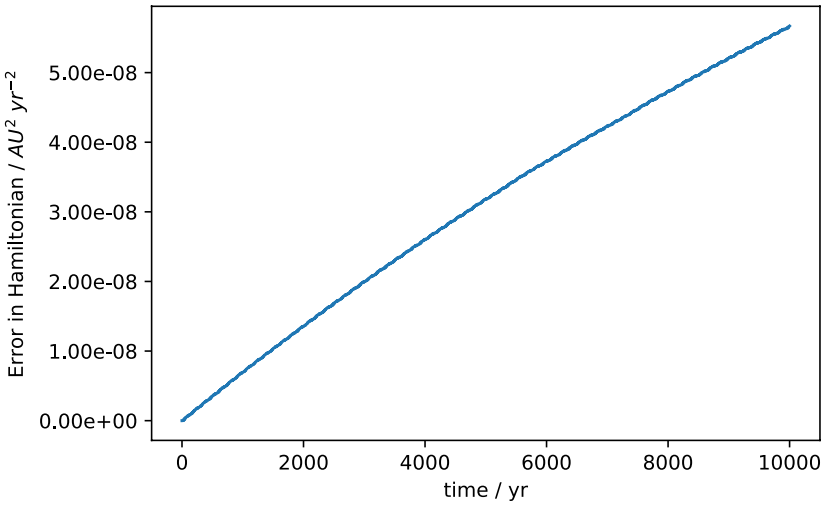


Figure 2: Time evolution of the absolute error in the Hamiltonian relative to its initial value, over 10,000 years. (a) is for atol = rtol = 10^{-8}, while (b) has atol = rtol = 10^{-6}, and both are for an initial azimuthal displacement of 0.25 AU anticlockwise from L_4.

4.2. Preliminary checks

Initial checks focused on the stability an asteroid placed at either Lagrange point. Wander was measured over 100,000 years, corresponding to thousands of orbits of Jupiter, yielding radial wander and angular wanders on the order of 0.001 AU, as well as variation in the Hamiltonian of the order of $10^{-7}\ AU^2\ yr^{-2}$. As expected, these values are negligible, confirming the analytical solution of the equations of motion, and showing that the program is operating correctly with no numerical instabilities being propagated during the computation.

Following conformation of the correct behaviour at L_4/L_5, perturbations were examined to ensure equivalence of the two points, as well as to confirm that tadpole and horseshoe orbits could be reproduced. Figures 3-5 illustrate examples of tadpole, horseshoe and chaotic solutions respectively. In the case of tadpole and horseshoe orbits, the solutions for both Lagrange points are equivalent, as expected, whereas the chaotic motions do not correspond. This is because chaotic motion is highly sensitive to initial conditions, so even very small deviations due to errors lead to significantly different behaviour. As this analysis focuses on stable orbits, this behaviour does not pose a problem.

Motion of the asteroid in the inertial frame was also checked, but as expected this revealed no clear details, as a result of the dominance of the orbital motion.

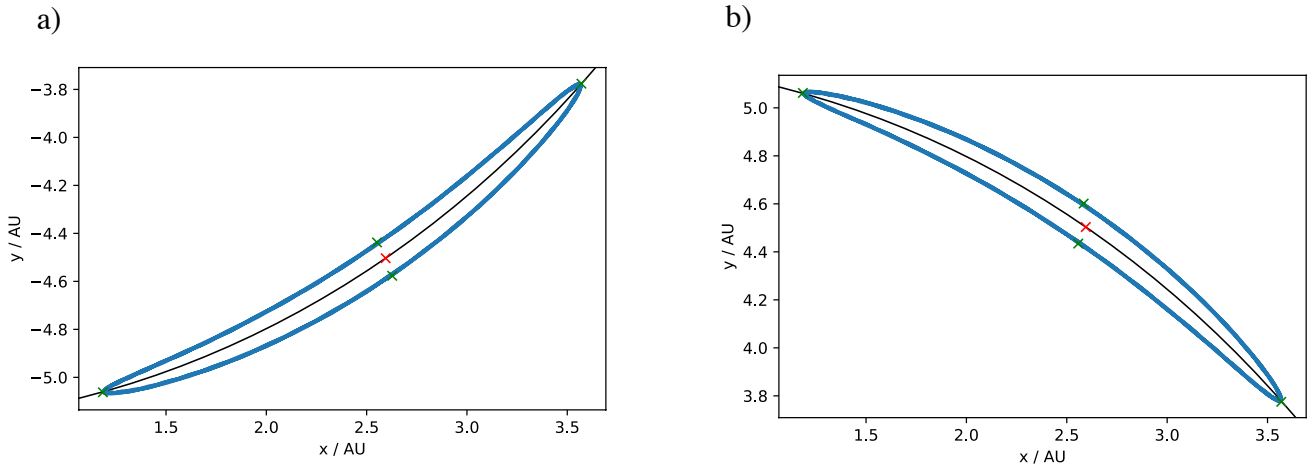


Figure 3: Examples of tadpole orbits enclosing (a) L_5 and (b) L_4, produced by symmetric angular displacements for an integration time of 1000 years. Lagrange points are indicated by red crosses, while green crosses indicate positions on minimum and maximum angle or radius. A circle of centred on the origin and passing through the Lagrange points is drawn.

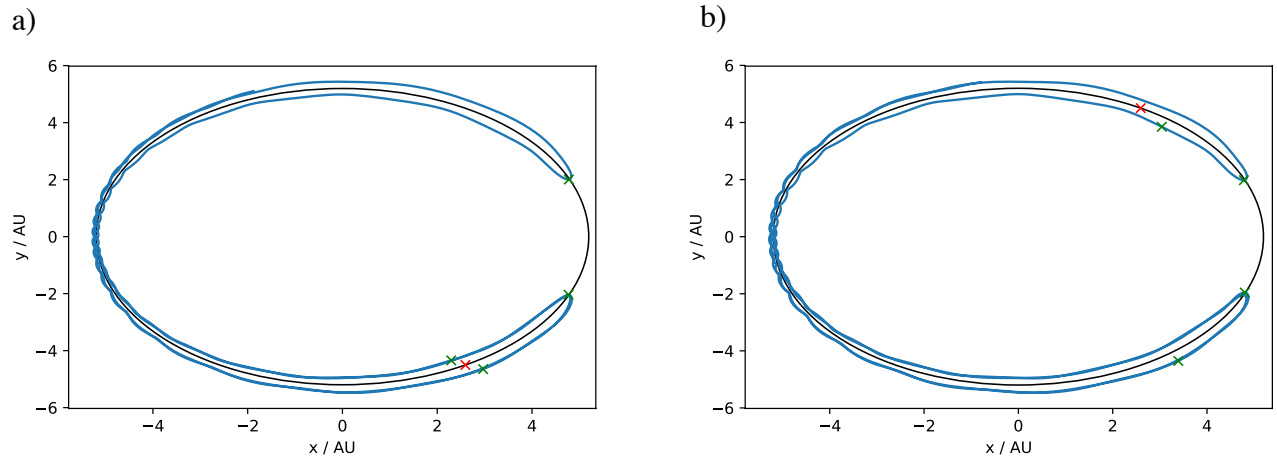


Figure 4: Examples of horseshoe orbits enclosing L_{3-5}, originating relative to (a) L_5 and (b) L_4. Both plots were produced using symmetric angular displacements and integration was run for 1000 years. Lagrange points are indicated by red crosses, while green crosses indicate positions on minimum and maximum angle or radius. A circle of centred on the origin and passing through the Lagrange points is drawn.

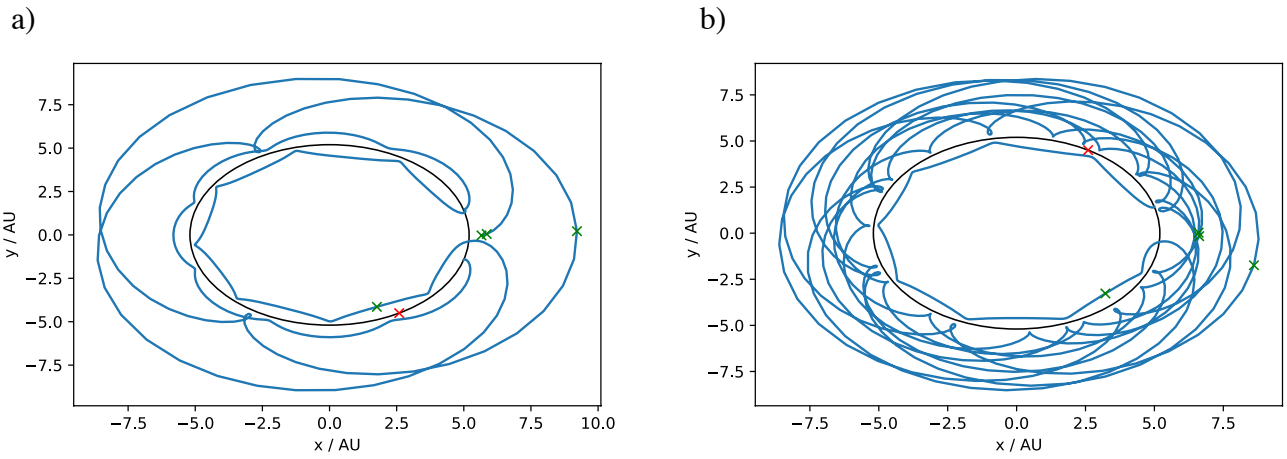


Figure 5: Examples of chaotic orbits resulting from displacements from (a) L_5 and (b) L_4. Both plots were produced using symmetric angular displacements and integration was run for 1000 years. Lagrange points are indicated by red crosses, while green crosses indicate positions on minimum and maximum angle or radius. A circle of centred on the origin and passing through the Lagrange points is drawn.

4.3. Initial displacement variation

Having confirmed that the integrator was performing as required, the effect of displacement from Lagrange points was investigated. The angular and radial wanders resulting from initial angular displacements are presented in figures 6 and 7. This wander, and subsequent wanders, are given by the average value calculated for the two Lagrange points, unless otherwise specified, as the plots from the individual Lagrange points are very similar in form.

These figures demonstrate the symmetry of the system, as the negative x-axis corresponds to an angular displacement $5.2 \times 2\pi/3 \approx 10.9\ AU$ anticlockwise from L_4. The graph's symmetry about this point is therefore further evidence of the equivalence of L_4 and L_5, as angular displacements greater than this value correspond to the asteroid being captured by the opposite Lagrange point.

Asteroids displaced angularly appear to be remarkably stable, with only a limited region of displacements that are unstable. These regions of instability appear to correspond to angular displacements greater than $(25.175 \pm 0.001)AU$, or less than $(-3.409 \pm 0.001)AU$, both corresponding to close proximity with Jupiter. The majority of orbits appear to be stable tadpole orbits, with both their radial and angular wanders increasing with angular displacement. In the region of displacements near to 10.9 AU, however, the asteroids appear to be able to form horseshoe orbits, encompassing L_{3-5}, resulting in very large angular wanders, while the radial wander is much smaller.

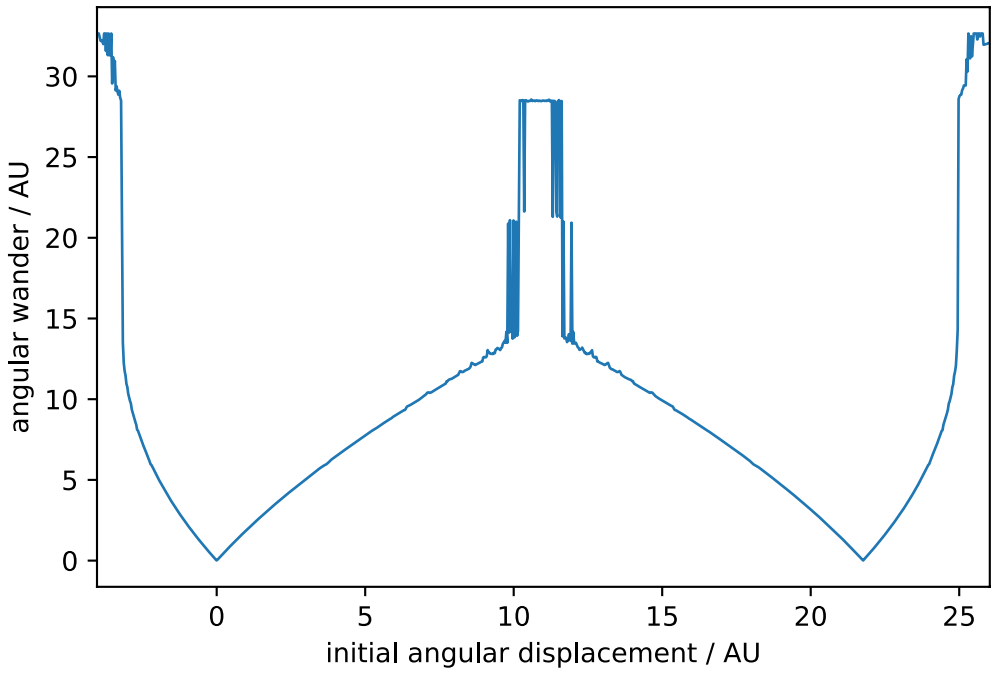


Figure 6: Plot of averaged angular wander of an asteroid against initial angular displacement. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

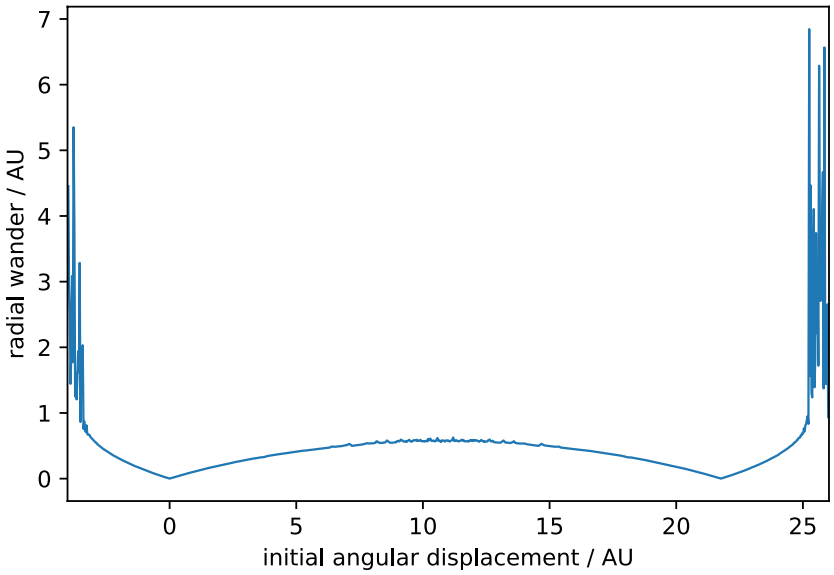


Figure 7: Plot of averaged radial wander of an asteroid against initial angular displacement. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

The effects of perturbations in the form of radial displacements relative to Lagrange points are illustrated in figures 8 and 9, which correspond to the angular and radial wanders respectively. Both graphs suggest these displacements are symmetric towards and away from the barycentre, but the origins of this were not clear. A relatively linear increase in wander with initial displacement is observed until around $(-0.065 \pm 0.001)AU$ and $(0.065 \pm 0.001)AU$ where the orbit rapidly becomes unstable. These limits are much more stringent than the those imposed on angular displacements for instability.

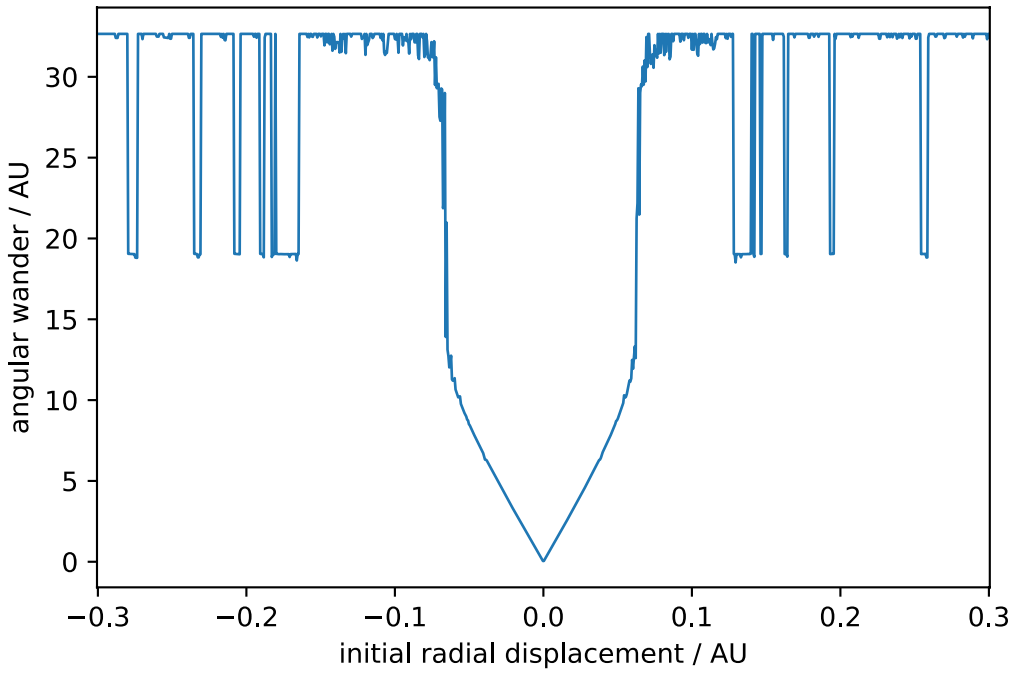


Figure 8: Plot of averaged angular wander of an asteroid against initial radial displacement. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

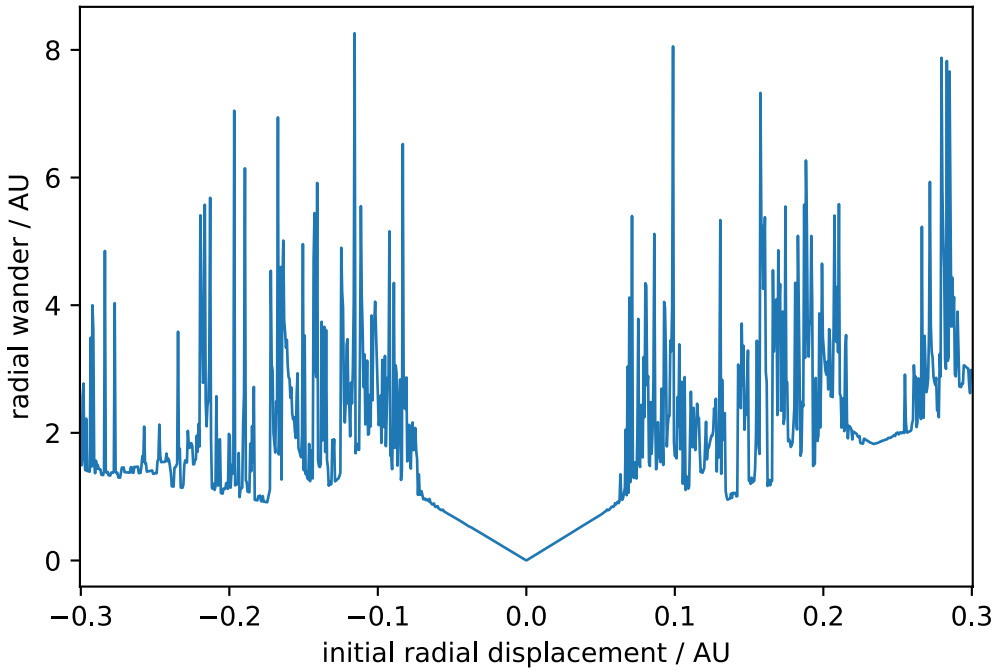


Figure 9: Plot of averaged radial wander of an asteroid against initial radial displacement. Integration was performed over 1000 years, for a planet mass of $0.001 M_{\odot}$.

4.4. Initial velocity variation

In addition to displacement perturbations, the impact of giving velocity to asteroids placed at the Lagrange points was also studied. Graphs demonstrating wander as a result of initial angular velocity (Figures 10 and 11) and initial radial velocity (Figures 12 and 13) are presented. From these graphs, it can be seen that the asteroids are more stable when given an angular velocity than a radial velocity. Instability is observed for angular velocities less than $(-0.073 \pm 0.001) AU\ yr^{-1}$ or greater than $(0.062 \pm 0.001) AU\ yr^{-1}$, compared to radial velocities less than $(-0.510 \pm 0.001) AU\ yr^{-1}$ or greater than $(0.457 \pm 0.001) AU\ yr^{-1}$.

While less extreme, this is the same relationship as between angular and radial displacements.

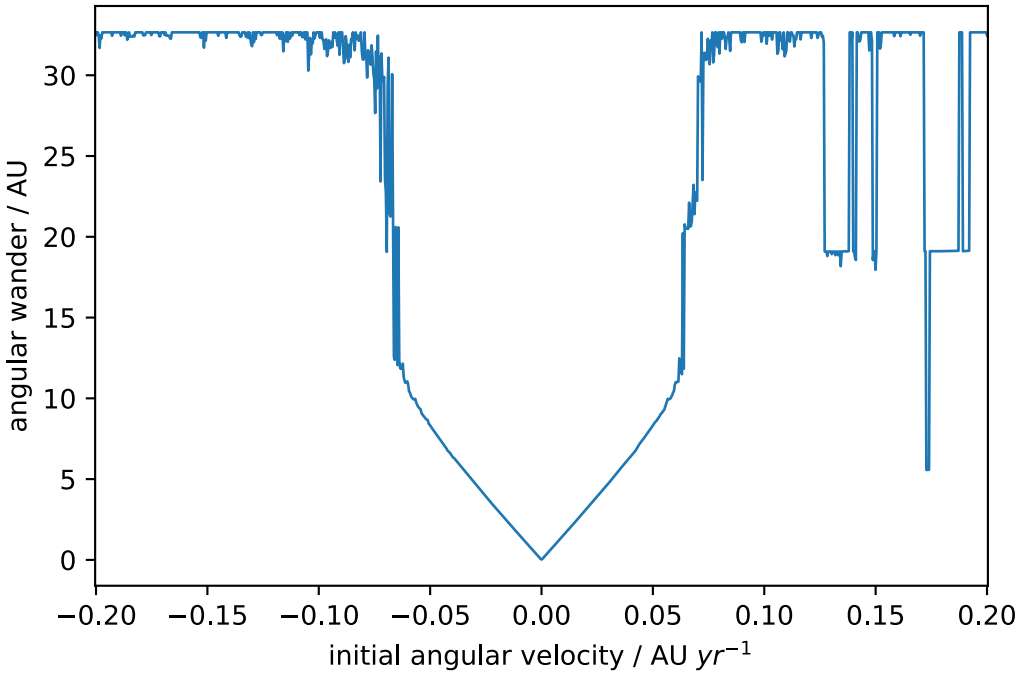


Figure 10: Plot of averaged angular wander of an asteroid against initial angular velocity. Integration was performed over 1000 years, for a planet mass of $0.001 M_{\odot}$.

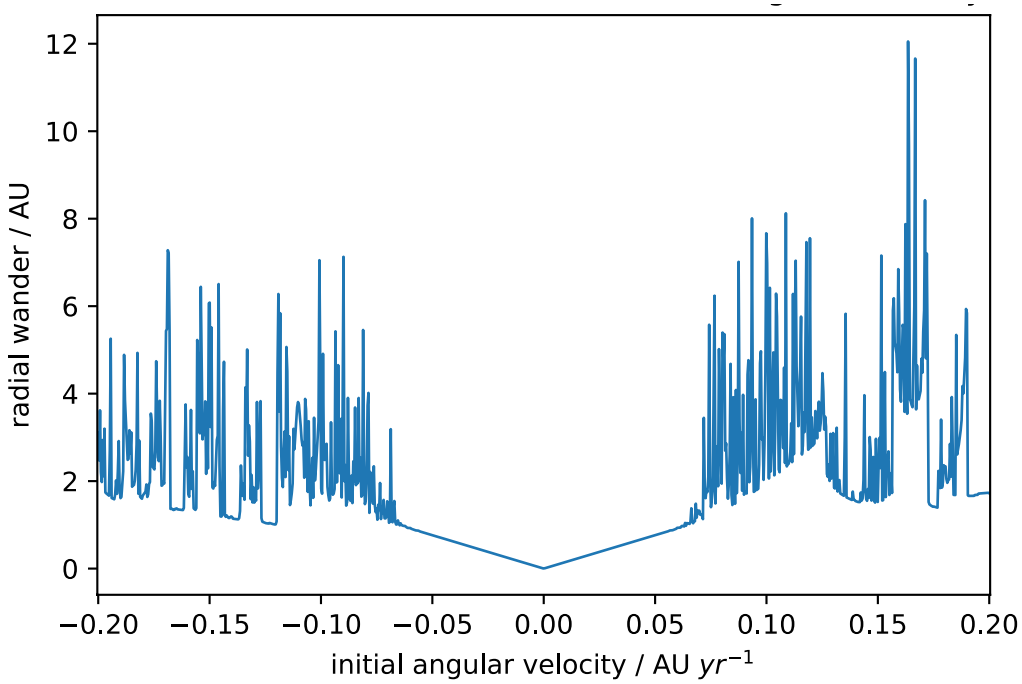


Figure 11: Plot of averaged radial wander of an asteroid against initial angular velocity. Integration was performed over 1000 years, for a planet mass of $0.001 M_{\odot}$.

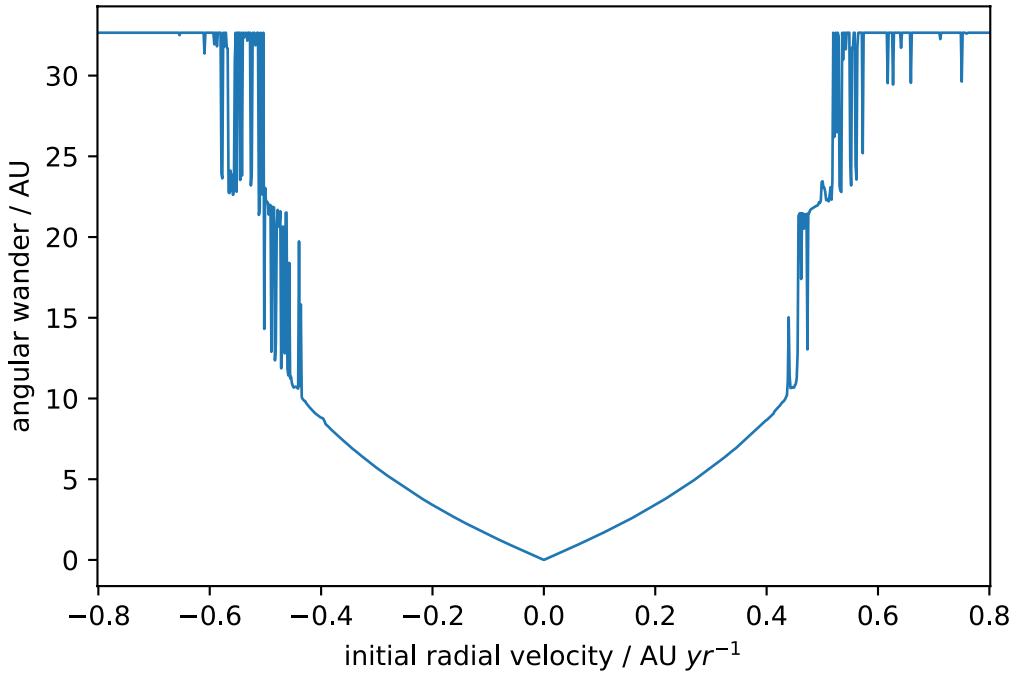


Figure 12: Plot of averaged angular wander of an asteroid against initial radial velocity. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

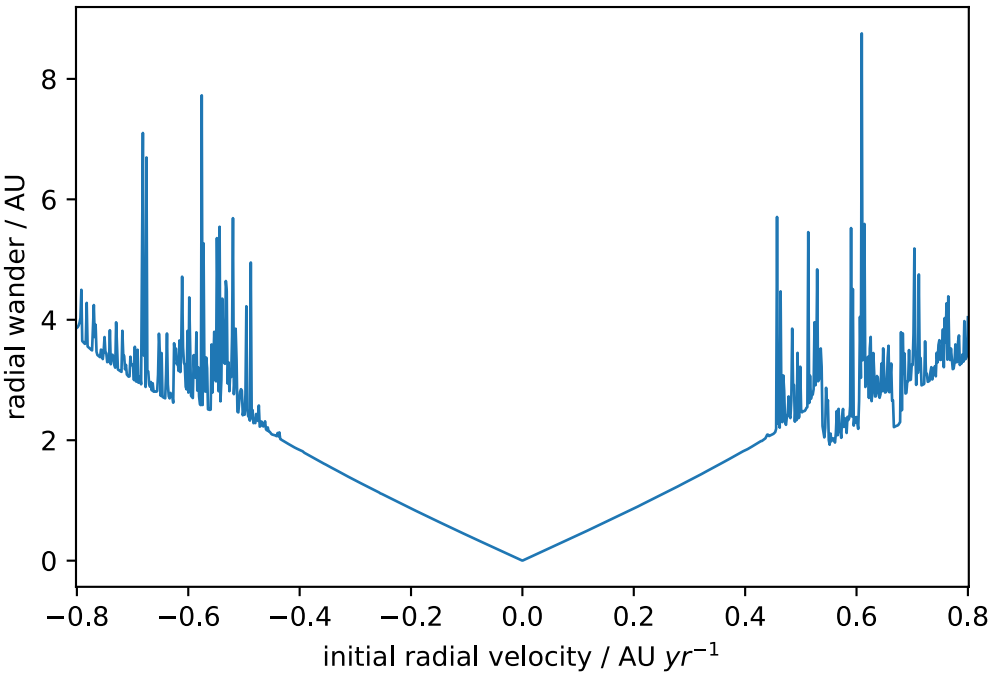


Figure 13: Plot of averaged radial wander of an asteroid against initial radial velocity. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

4.5. Variation of planet's mass

As well as studying the Trojan asteroids of Jupiter, the stability of asteroids was investigated in the context of planets with different masses, focussing on those within one or two orders of magnitude of Jupiter. Graphs demonstrating the angular and radial wander for a large mass range are presented in figures 14 and 15. From these, it is clear a threshold mass exists, around $0.04M_{\odot}$, where orbits quickly become unstable. Additional resonant behaviour is seen between 0.02 and $0.03M_{\odot}$, as well as around $0.013M_{\odot}$.

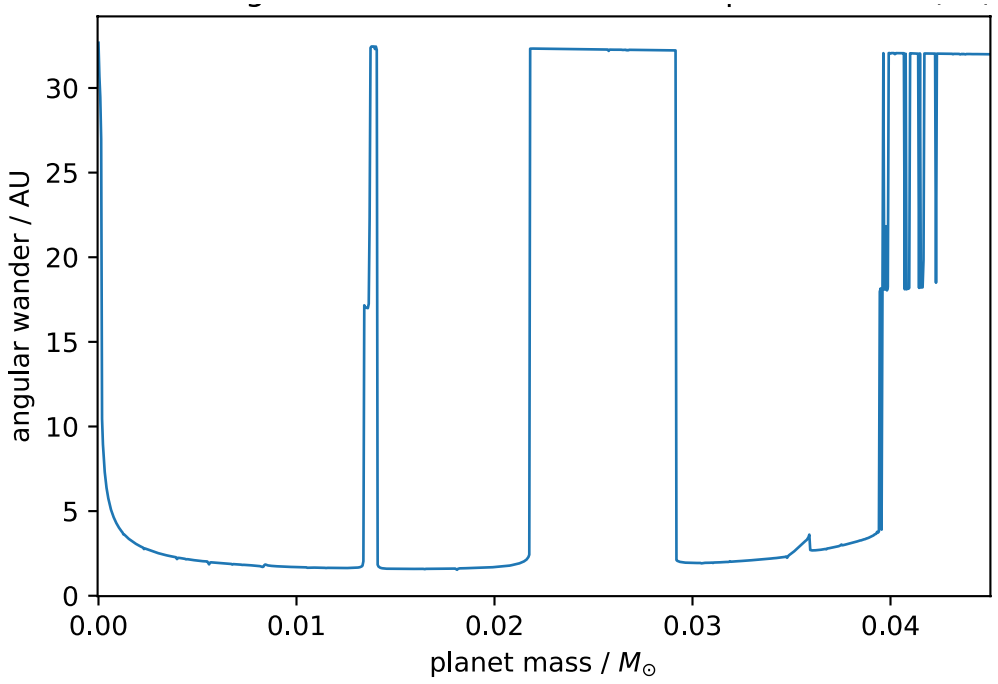


Figure 14: Plot of averaged angular wander of an asteroid against initial planet mass. Integration was performed over 1000 years, and the asteroid was given an initial radial displacement of 0.025AU.

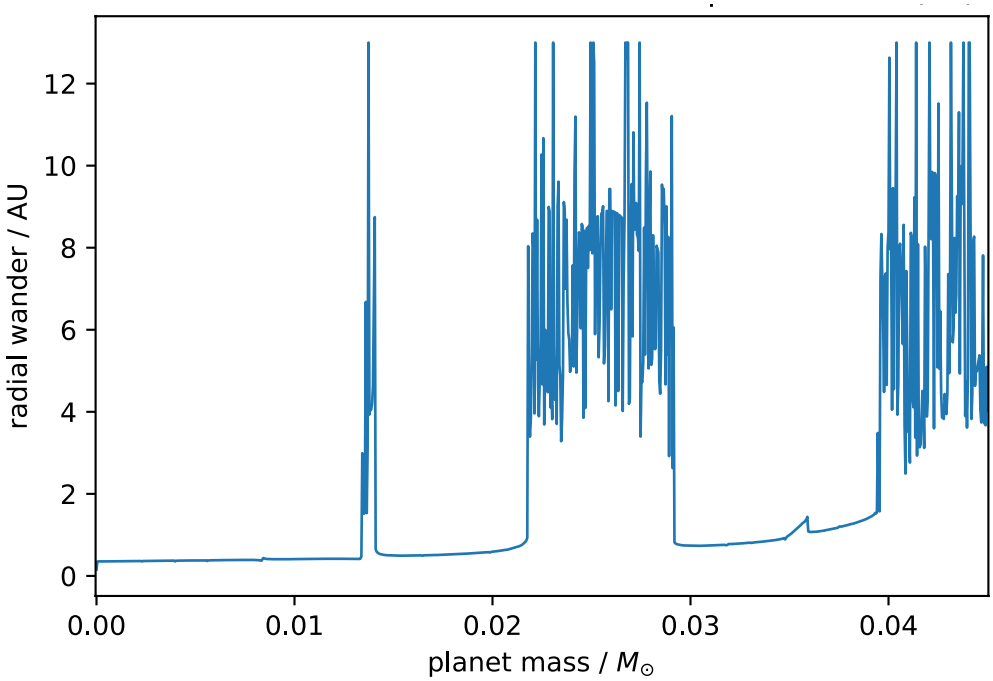


Figure 15: Plot of averaged radial wander of an asteroid against initial planet mass. Integration was performed over 1000 years, and the asteroid was given an initial radial displacement of 0.025AU.

For small displacements from the Lagrange points, it is predicted that the tadpole orbits will form ellipses. If the angular and radial wanders are small enough, they can then be used as estimates of the semi-major and semi-minor axes respectively, and these can be related to the reduced mass (approximately equal to the mass of the planet, since it is so small), by [1]:

$$\frac{b}{a} = \sqrt{3} * \mu$$

Plots of ln(radial wander / angular wander) against ln(mass) (Figure 16) and (radial wander / angular wander) against sqrt(m) (Figure 17) were produced, yielding a gradient of 0.48 for the former, while a clear linear relationship is seen from the latter, verifying this relationship.

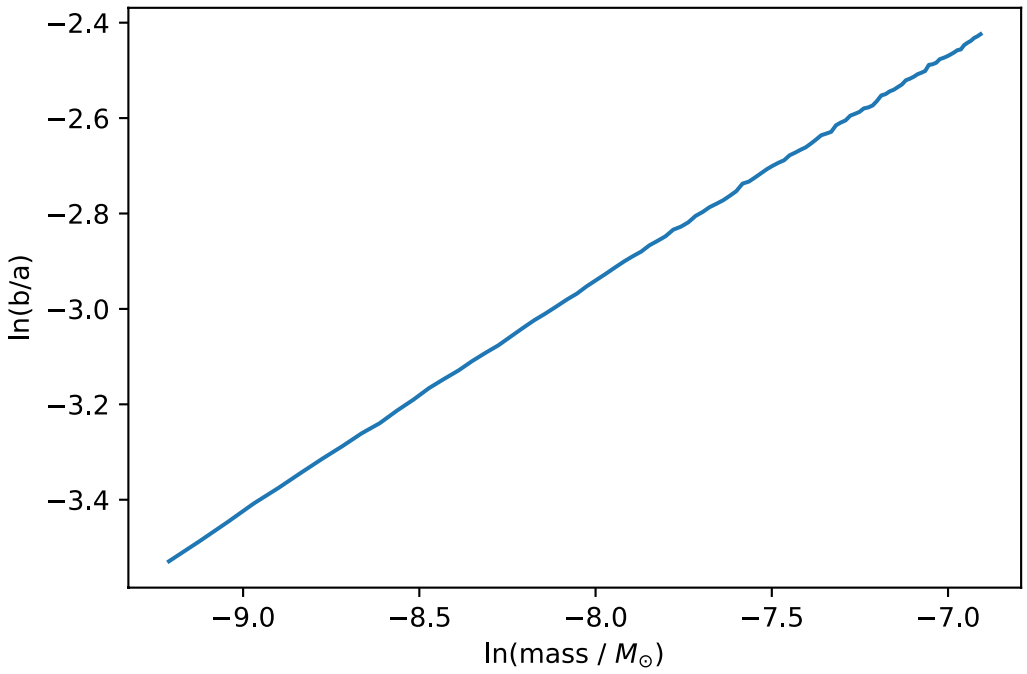


Figure 16: Plot of ln(radial wander / angular wander) against ln(mass), yielding a linear relationship with a gradient of approximately 0.5.

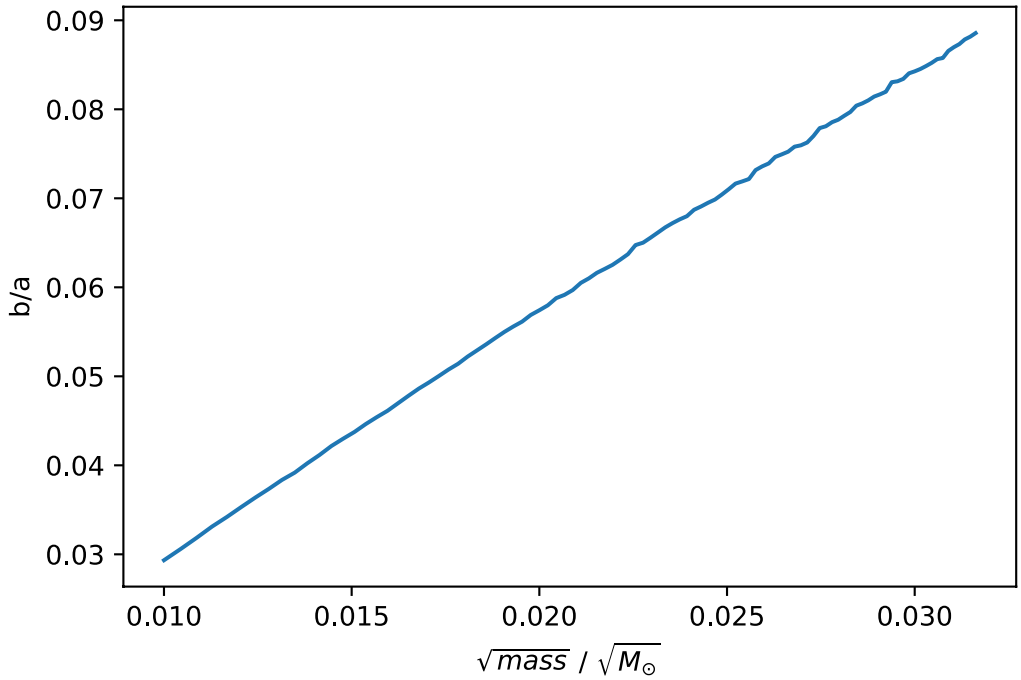


Figure 17: Plot of (radial wander / angular wander) against square root of mass, yielding a linear relationship.

4.6. Motion out of the plane

In motion not confined to the x-y plane, the Hamiltonian is no longer a constant, and so verifying the accuracy of solutions is difficult. Nonetheless, a limited study of wander due to displacements and velocities in the z-direction was undertaken, with the resulting plots illustrated in figures 18-23. These plots all demonstrate symmetry about z = 0, respecting the symmetry of the plane as desired. As in motion within the plane, extremities resulted in chaotic behaviour, with limiting values of $(-1.336 \pm 0.001)AU$ and $(1.336 \pm 0.001)AU$ for the z displacements, as well as $(\pm 0.001)AU\ yr^{-1}$ and $(0.665 \pm 0.001)AU\ yr^{-1}$ for the z velocities.

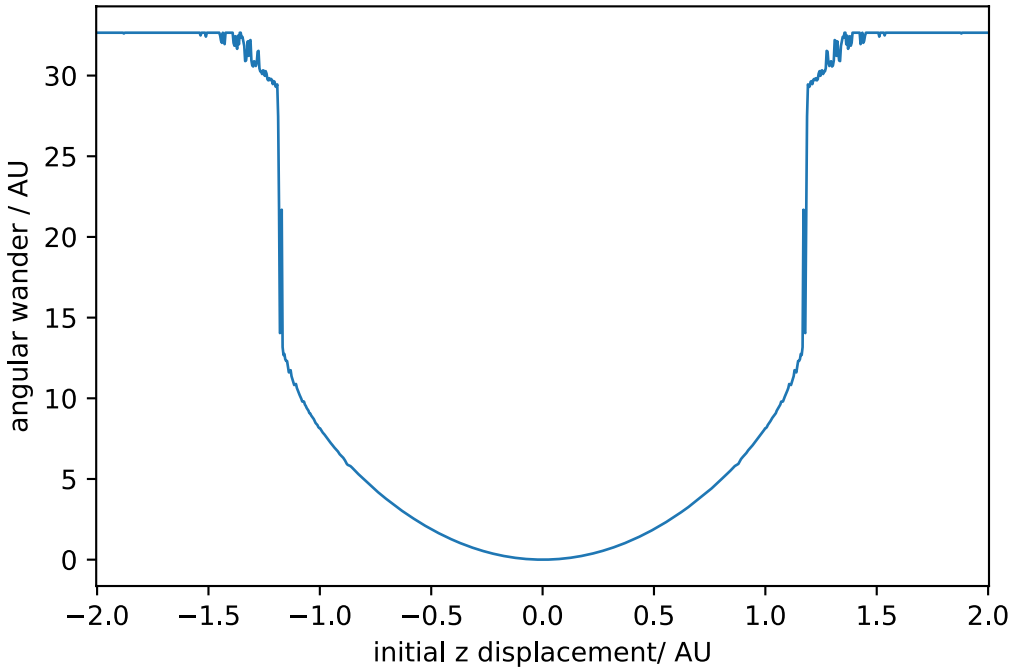


Figure 18: Plot of averaged angular wander of an asteroid against initial z displacement. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

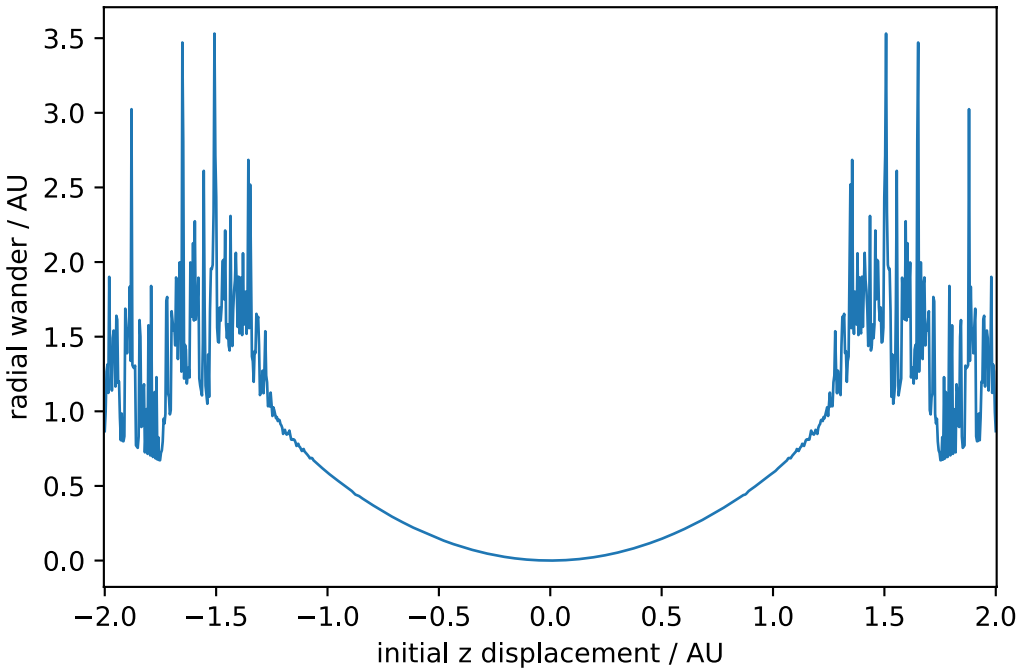


Figure 19: Plot of averaged radial wander of an asteroid against initial z displacement. Integration was performed over 1000 years, for a planet mass of $0.001 M_{\odot}$.

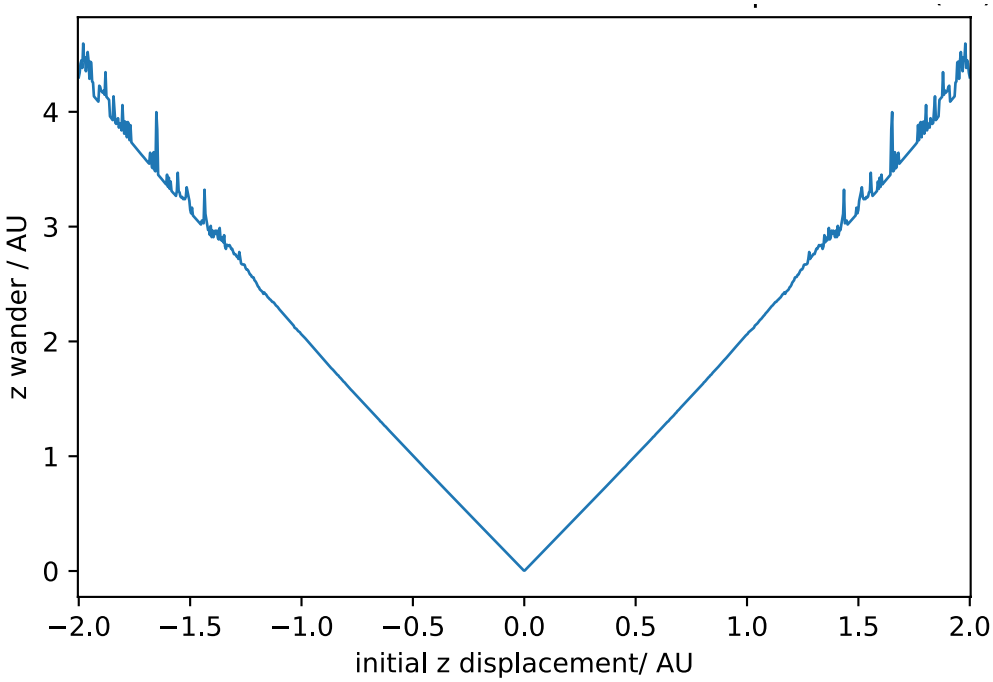


Figure 20: Plot of averaged z wander of an asteroid against initial z displacement. Integration was performed over 1000 years, for a planet mass of $0.001 M_{\odot}$.

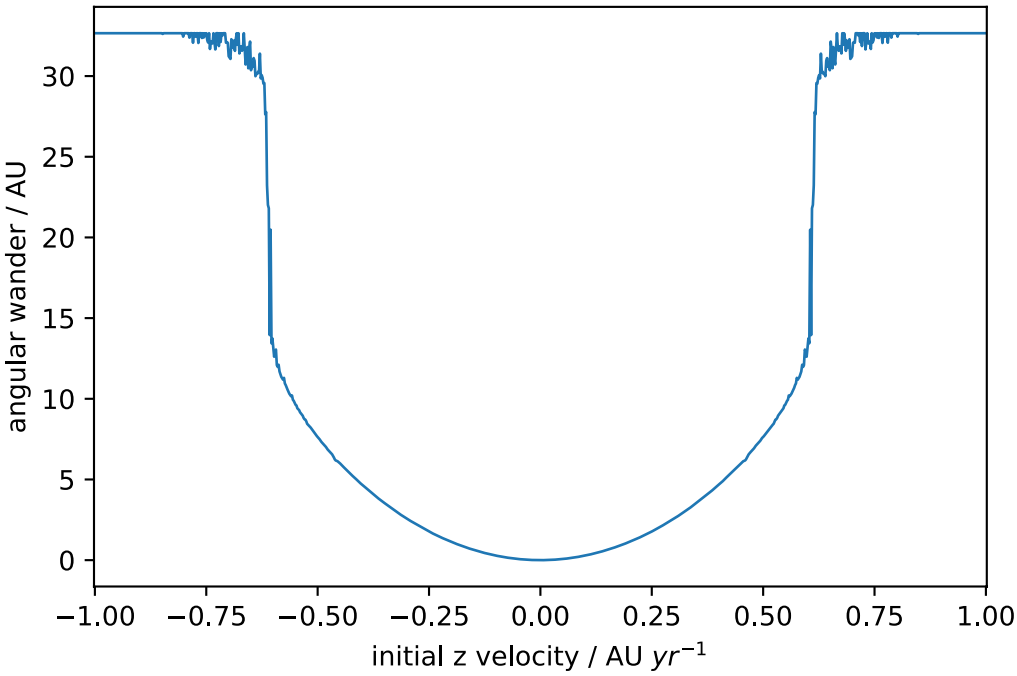


Figure 21: Plot of averaged angular wander of an asteroid against initial z velocity. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

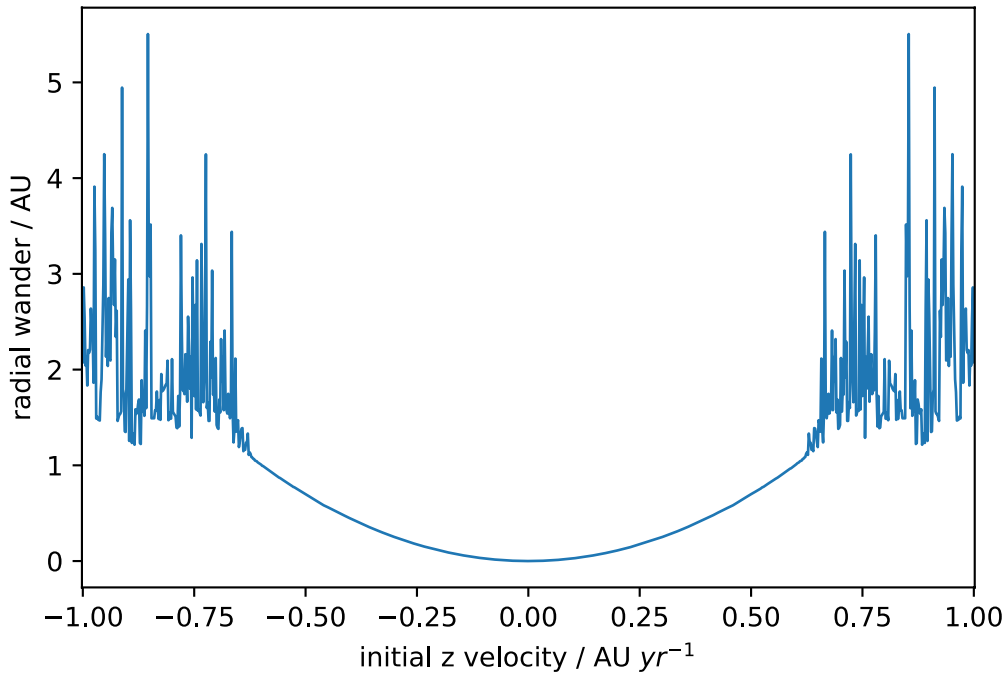


Figure 22: Plot of averaged radial wander of an asteroid against initial z velocity. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

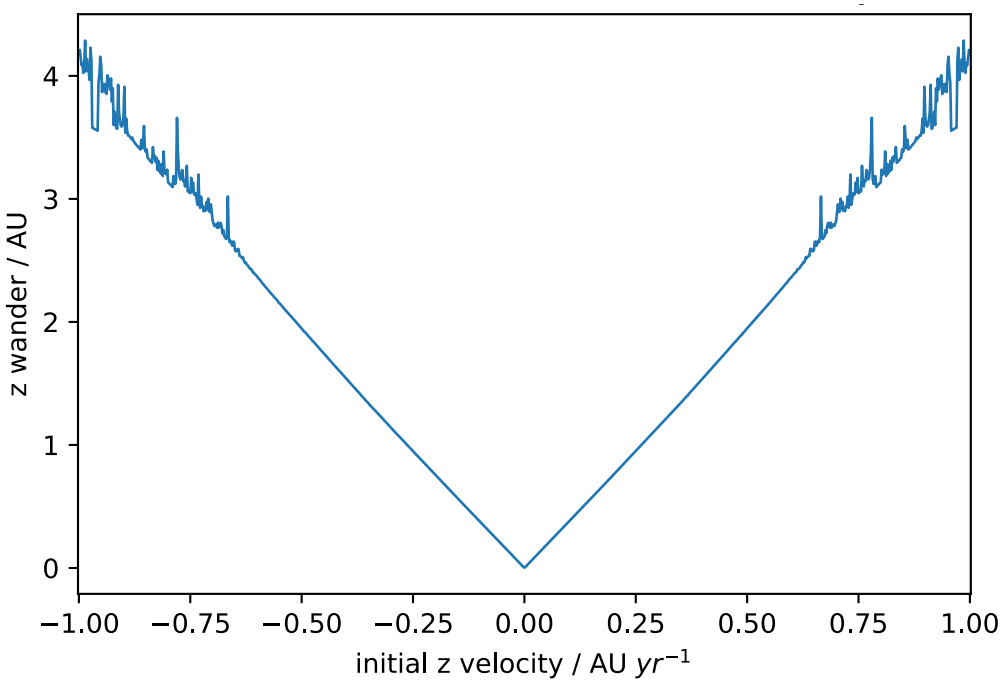


Figure 23: Plot of averaged z wander of an asteroid against initial z velocity. Integration was performed over 1000 years, for a planet mass of $0.001M_{\odot}$.

5. Conclusions

The circular restricted three body problem is solved numerically using an adaptive fourth order Runge-Kutta technique. Limits are placed on the stability of the Lagrange points, primarily through considerations of the wander in the radial and azimuthal directions. It was determined that a test mass placed at the two triangular Lagrange points, L_4 and L_5, would remain in this position. Small perturbations to this position yielded orbits encompassing the Lagrange point, known as a tadpole orbit, while larger perturbations lead to wander between L_{3-5}, known as a horseshoe orbit. Additionally, a threshold mass around $0.04M_{\odot}$, was found, above which orbits quickly became unstable, and additional resonant behaviour was observed at lower masses.

Due to the number of approximations used in this analysis, there is significant scope to improve the correspondence to real physics systems. N-body interactions, for example, may be considered, and additional forces such as drag, which results from radiation pressure, may be taken into account **[1]**. As well as this, the inclined, elliptical nature of real orbits can be considered. Such extensions can be used to probe numerous unanswered questions in astrophysics, including as the effect of Trojans on planets mass accretion.

Word count, excluding summary, captions, references and appendices: ~3300.

References

1. Murray C.D., Dermott, S.F., 1999. *Solar System Dynamics*. Cambridge University Press, Cambridge
2. Lagrange, J. L., 1772. "Essai sur le problème des trois corps (Essay on the problem of three bodies)". Prix de l'Acadèmie Royale des Sciences de Paris, tome IX, in volume 6 of œuvres, pp 292
3. Fleming H. J., Hamilton D. P., 2000. "On the Origin of the Trojan Asteroids: Effects of Jupiter's Mass Accretion and Radial Migration". Icarus 148, 2, pp 479-493
4. Shoemaker E. M., Shoemaker C. S., Wolfe R. F., 1989. "Trojan asteroids: Population, dynamical structure, and origin of the L4 and L5 swarms." In Binzel R. P., Gehrels T., Matthews M. S. (eds.) *Asteroids II*. Univ. of Arizona Press, Tucson, 487-523.
5. The Gaia Collaboration, 2016. "The Gaia mission." Astronomy and Astrophysics, 595, A1
6. Green D. A., 2017. *Classical dynamics*. Course handout: Department of Physics, University of Cambridge
7. Wyatt M., 2017. *Planetary system dynamics*. Course handout: Department of Physics, University of Cambridge
8. Borcherds P.H., 1996. "The restricted gravitational three-body problem trajectories associated with Lagrange fixed points". Eur. J. Phys. 17 63–70
9. Hairer E., Norsett S.P., Wanner G., 1993. *Solving Ordinary Differential Equations I: Nonstiff Problems (2nd edition)*. Springer-Verlag, Berlin
10. Hindmarsh A. C., 1983. "ODEPACK, A Systematized Collection of ODE Solvers". In R. S. Stepleman et al. (eds.), *Scientific computing: Vol. 1. IMACS transaction on scientific computation (pp 55-64)*. Amsterdam: IMACS/North-Holland
11. Petzold L., 1983. "Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations". SIAM Journal on Scientific and Statistical Computing, Vol. 4, No. 1, pp. 136-148
12. Buscher, D., 2018. *Computational Physics*. Course handout: Department of Physics, University of Cambridge

A Derivation of the Hamiltonian/equation of motion with the Lagrangian

The Lagrangian, $\mathcal{L}$, in the restricted three-body problem is given by

$$\mathcal{L} = \frac{m}{2}\left|\frac{d\vec{\boldsymbol{r}}}{dt} + \vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{r}}\right|^2 - U(\vec{\boldsymbol{r}})$$

where

$$U(\vec{\boldsymbol{r}}) = -\frac{GM_s}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_s|} - \frac{GM_j}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_j|}$$

The canonical momentum, $\vec{\boldsymbol{p}}$, is given by

$$\vec{\boldsymbol{p}} = \frac{\partial \mathcal{L}}{\partial\left(\frac{d\vec{\boldsymbol{r}}}{dt}\right)}$$

The Lagrangian may be expressed using the identity

$$\left|\frac{d\vec{\boldsymbol{r}}}{dt} + \vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{r}}\right|^2 = \left(\frac{d\vec{\boldsymbol{r}}}{dt}\right)^2 + 2\vec{\boldsymbol{\omega}} \cdot \left(\vec{\boldsymbol{r}} \times \frac{d\vec{\boldsymbol{r}}}{dt}\right) + |\vec{\boldsymbol{\omega}}|^2|\vec{\boldsymbol{r}}|^2 - (\vec{\boldsymbol{\omega}} \cdot \vec{\boldsymbol{r}})^2 \qquad (1)$$

allowing the canonical momentum to then be expressed as

$$\vec{\boldsymbol{p}} = m\left(\frac{d\vec{\boldsymbol{r}}}{dt} + \vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{r}}\right)$$

From here the equation of motion, found through the consideration of forces, may be derived from the Euler-Lagrange equations, since

$$\frac{d\vec{\boldsymbol{p}}}{dt} = \frac{\partial \mathcal{L}}{\partial \vec{\boldsymbol{r}}}$$

where

$$\frac{\partial \mathcal{L}}{\partial \vec{\boldsymbol{r}}} = -\nabla U(\vec{\boldsymbol{r}}) - m(\vec{\boldsymbol{\omega}} \times \frac{d\vec{\boldsymbol{r}}}{dt} + \vec{\boldsymbol{\omega}} \times (\vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{r}})$$

and

$$\frac{d\vec{\boldsymbol{p}}}{dt} = m\left(\frac{d^2\vec{\boldsymbol{r}}}{dt^2} + \frac{d\vec{\boldsymbol{\omega}}}{dt} \times \vec{\mathbf{r}} + \vec{\boldsymbol{\omega}} \times \frac{d\vec{\boldsymbol{r}}}{dt}\right)$$

yielding:

$$m\frac{d^2\vec{\boldsymbol{r}}}{dt^2} = -\nabla U(\vec{\boldsymbol{r}}) - m\left(\frac{d\vec{\boldsymbol{\omega}}}{dt} \times \vec{\boldsymbol{r}} + 2\vec{\boldsymbol{\omega}} \times \frac{d\vec{\boldsymbol{r}}}{dt} + \vec{\boldsymbol{\omega}} \times (\vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{r}})\right)$$

which simplifies for constant $\vec{\boldsymbol{\omega}}$ to the equation of motion stated in section 2.1.

The Hamiltonian, $\mathcal{H}$, is defined as:

$$\mathcal{H} \equiv \vec{\boldsymbol{p}} \cdot \frac{d\vec{\boldsymbol{r}}}{dt} - \mathcal{L}$$

which can then be expressed as

$$\mathcal{H} = m\left(\left(\frac{d\vec{\boldsymbol{r}}}{dt}\right)^2 + \frac{d\vec{\boldsymbol{r}}}{dt} \cdot (\vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{r}})\right) - \frac{m}{2}\left|\frac{d\vec{\boldsymbol{r}}}{dt} + \vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{r}}\right|^2 + U(\vec{\boldsymbol{r}})$$

or, using equation (1),

$$\mathcal{H} = \frac{m}{2}\left(\frac{d\vec{\boldsymbol{r}}}{dt}\right)^2 - \frac{m}{2}|\vec{\boldsymbol{\omega}}|^2|\vec{\boldsymbol{r}}|^2 + (\vec{\boldsymbol{\omega}} \cdot \vec{\boldsymbol{r}})^2 + U(\vec{\boldsymbol{r}})$$

Since $\vec{\boldsymbol{\omega}}$ is in the z-direction, $(\vec{\boldsymbol{\omega}} \cdot \vec{\boldsymbol{r}})^2 = 0$ for motion comfined to the x-y plane, yielding:

$$\mathcal{H} = \frac{m}{2}\left(\frac{d\vec{\boldsymbol{r}}}{dt}\right)^2 - \frac{m}{2}|\vec{\boldsymbol{\omega}}|^2|\vec{\boldsymbol{r}}|^2 + U(\vec{\boldsymbol{r}})$$

This is a constant of motion, as there is no explicit time dependence in the Lagrangian.

B Python code

```
#Import relevant libraries
#Note: Axes3D not explicitly used in code but still needed for 3D plot
import numpy as np
from numpy import pi, sqrt
from scipy.integrate import ode
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time
import matplotlib.ticker as mtick


#Define class for massive bodies:
class body:

    def __init__(self):
        self.mass = np.float64(0.0)
        self.position = np.array([0.0,0.0,0.0], dtype = np.float64)
        self.velocity = np.array([0.0,0.0,0.0], dtype = np.float64)


#Plot whole solar system, i.e. all bodies, L4, L5 and asteroid trajectory:
def plot_sol(R, x, jupiter, sun, max_ang_index, min_ang_index, max_rad_index, min_rad_index,
L_point, L4, L5):

    #Note radii of Sun/Jupiter increased by a factor of 100 in plot
    #Sun radius = ~0.005
    #Jupiter radius = ~0.0005

    jup_rad = 0.05
    sun_rad = 0.5
    L_rad = np.linalg.norm(L_point)

    #Create circles for Jupiter, the Sun and a circle passing through the Lagrange points
    jup_circle = plt.Circle((jupiter.position[0], jupiter.position[1]), jup_rad, color='g')
    sun_circle = plt.Circle((sun.position[0], sun.position[1]), sun_rad, color='r')
    lagrange_circle = plt.Circle((0, 0), L_rad, color='k', fill=False)

    #Create square plot
    fig, ax = plt.subplots()
    ax.cla()
    ax.set_aspect('equal')
    ax.set_xlim((-6, 6))
    ax.set_ylim((-6, 6))
```

```
    #Plot x against y, as well as marks for L4, L5 (red) and the max/min coordinate positions
(green)
    ax.plot(x[:,0],x[:,1])
    ax.plot(L4[0], L4[1], 'x', color='r')
    ax.plot(L5[0], L5[1], 'x', color='r')

    ax.plot(x[max_ang_index,0],x[max_ang_index,1], 'x', color='g')
    ax.plot(x[min_ang_index,0],x[min_ang_index,1], 'x', color='g')
    ax.plot(x[max_rad_index,0],x[max_rad_index,1], 'x', color='g')
    ax.plot(x[min_rad_index,0],x[min_rad_index,1], 'x', color='g')

    #Draw circles for Jupiter/Sun/L point and label axes
    ax.add_artist(sun_circle)
    ax.add_artist(jup_circle)
    ax.add_artist(lagrange_circle)
    ax.set_xlabel("x / AU")
    ax.set_ylabel("y / AU")

    plt.savefig('sol.eps', format='eps', dpi=1000)

    return 0


#Plot just relevant trajectory and L point:
def plot_traj(R, x, jupiter, sun, max_ang_index, min_ang_index, max_rad_index,
min_rad_index, L_point):

    #Create circle passing through the Lagrange point, to be compared to orbit
    lagrange_circle = plt.Circle((0, 0), np.linalg.norm(L_point), color='k', fill=False)

    fig, ax = plt.subplots()
    ax.cla()

    #Plot x against y, as well as marks for the Lagrange point and the max/min coordinate
positions
    ax.plot(x[:,0],x[:,1])

    ax.plot(L_point[0], L_point[1], 'x', color='r')

    ax.plot(x[max_ang_index,0],x[max_ang_index,1], 'x', color='g')
    ax.plot(x[min_ang_index,0],x[min_ang_index,1], 'x', color='g')
    ax.plot(x[max_rad_index,0],x[max_rad_index,1], 'x', color='g')
    ax.plot(x[min_rad_index,0],x[min_rad_index,1], 'x', color='g')

    ax.add_artist(lagrange_circle)
    ax.set_xlabel("x / AU")
```

```
    ax.set_ylabel("y / AU")

    plt.savefig('traj.eps', format='eps', dpi=1000)

    return 0


#Plot x against y:
def plot_xy(x, y, xlabel, ylabel, doctitle):

    #Set up figure and plot
    fig, ax = plt.subplots()
    ax= plt.gca()
    ax.cla()
    ax.plot(x, y)
    ax.yaxis.set_major_formatter(mtick.FormatStrFormatter('%.2e'))
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    plt.savefig(doctitle +'.eps', format='eps', dpi=1000, bbox_inches="tight")

    return 0


#Plot x against y and z (3D)
def plot_3D(x, y, z):

    #Set up figure and plot
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(x, y, z)
    ax.set_xlabel("x / AU")
    ax.set_ylabel("y / AU")
    ax.set_zlabel("z / AU")

    plt.savefig('3D.eps', format='eps', dpi=1000)

    return 0


#Plot angular/radial wander against variable:
def plot_wander(var_arr, wander_arr, wander_text, var_text, var_unit, label):

    #Estimate suitable max/min for variable (requires large number to work well)
    var_max = np.max(var_arr)
    var_min = np.min(var_arr)
    plot_min = var_min - (var_max - var_min)/len(wander_arr)
```

```
    plot_max = var_max + (var_max - var_min)/len(wander_arr)

    #Create labels for save file, axes and title
    smax = str(var_max)
    smin = str(var_min)
    snum = str(len(var_arr))
    xlabel = var_text + var_unit
    ylabel = wander_text +  " wander / AU"
    title = "Plot of " +  wander_text + " wander as a function of " + var_text + " (" + label + ")"
    eps_title = wander_text + var_text + smin + "_" + smax + "_" + snum + label + ".eps"

    #Set up figure and plot
    fig, ax = plt.subplots()
    ax.cla()
    ax.set_xlim(plot_min, plot_max)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)
    ax.plot(var_arr, wander_arr, linewidth=1)

    plt.savefig(eps_title, format='eps', dpi=1000)

    return 0

#Designed for investigation of b/a relationship with m
#Plots b against a, sqrt(m) against b/a and ln(m) against b/a
def plot_mass_wander(mass_arr, rad_wander_arr, ang_wander_arr):

    #Calculate radial wanders / angular wanders, and square roots of masses
    div_arr = np.divide(rad_wander_arr, ang_wander_arr)
    mass_sqrt = np.sqrt(mass_arr)

    #Plot radial wander against radial wander
    fig1, ax1 = plt.subplots()
    ax1.cla()
    ax1.plot(rad_wander_arr, ang_wander_arr)
    ax1.set_xlabel("angular wander / AU")
    ax1.set_ylabel("radial wander / AU")
    fig1.savefig("rad_against_ang", format='eps', dpi=1000)

    #Plot radial wander / angular wander against square root of mass
    fig2, ax2 = plt.subplots()
    ax2.cla()
    ax2.plot(mass_sqrt, div_arr)
    ax2.set_xlabel(r'$\sqrt{mass}$ / $\sqrt{M_{\odot}}$')
    ax2.set_ylabel("b/a")
```

```
    fig2.savefig("rad_ang_sqrt_mass", format='eps', dpi=1000)

    #Plot ln(radial wander/angular wander) against ln(mass)
    fig3, ax3 = plt.subplots()
    ax3.cla()
    ax3.plot(np.log(mass_arr), np.log(div_arr))
    ax3.set_xlabel("ln(mass / $M_{\odot}$)")
    ax3.set_ylabel("ln(b/a)")
    fig3.savefig("ln_rad_ang_ln_mass", format='eps', dpi=1000)

    #Calculate slopes of the graphs and print results
    p1, V1 = np.polyfit(rad_wander_arr, ang_wander_arr, 1, cov=True)
    p2, V2 = np.polyfit(mass_sqrt, div_arr, 1, cov=True)
    p3, V3 = np.polyfit(np.log(mass_arr), np.log(div_arr), 1, cov=True)

    print("rad/ang", div_arr)
    print("Average of rad/ang", np.average(div_arr))
    print("Gradient 1: {} +/- {}".format(p1[0], np.sqrt(V1[0][0])))
    print("Intercept 1: {} +/- {}".format(p1[1], np.sqrt(V1[1][1])))
    print("Gradient 2: {} +/- {}".format(p2[0], np.sqrt(V2[0][0])))
    print("Intercept 2: {} +/- {}".format(p2[1], np.sqrt(V2[1][1])))
    print("Gradient 3: {} +/- {}".format(p3[0], np.sqrt(V3[0][0])))
    print("Intercept 3: {} +/- {}".format(p3[1], np.sqrt(V3[1][1])))


#   print("Slope of rad against ang", slope1)
#   print("Intercept of rad against ang", intercept1)
#   print("Slope of rad/ang against sqrt mass", slope2)
#   print("Inercept of of rad/ang against sqrt mass", intercept2)
#   print("Slope of ln(rad/ang) against mass", slope3)
#   print("Intercept of ln(rad/ang) against mass", intercept3)

    return 0


#Function to dermine which plots to produce for a given set of initial conditions:
def plot_set_cond(R, x, t, ham_error, rot_x, rot_y, jupiter, sun, L_point, L4, L5, max_ang_index,
min_ang_index, max_rad_index, min_rad_index, ham_plt, rot_plt, sol_plt, traj_plt, xyz_plt):

    #Plots of the solar system, trajectory, Hamiltonian, inertial frame and 3D:
    #Selection of graphs plotted through parameters from main()
    if sol_plt == 1:
        plot_sol(R, x, jupiter, sun, max_ang_index, min_ang_index, max_rad_index,
min_rad_index, L_point, L4, L5)

    if traj_plt == 1:
```

```
        plot_traj(R, x, jupiter, sun, max_ang_index, min_ang_index, max_rad_index,
min_rad_index, L_point)

    #Plot Hamiltonian and log(dH) against t
    if ham_plt == 1:
        plot_xy(t, ham_error, "time / yr", "Error in Hamiltonian / $AU^{2}$ $yr^{-2}$",
"ham_err")

    if rot_plt == 1:
        plot_xy(rot_x, rot_y, "x / AU", "y / AU", "rotxy")

    if xyz_plt == 1:
        plot_3D(x[:,0],x[:,1], x[:,2])

    return 0


#Determine which plots to produce, involving plotting wander against another variable:
def plot_var_cond(mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr, ang_vel_arr, rad_vel_arr,
z_vel_arr, ang_wander_arr, rad_wander_arr, z_wander_arr, ang_wander_plt, rad_wander_plt,
z_wander_plt, vary_mass, vary_ang_disp, vary_rad_disp, vary_z_disp, vary_ang_vel,
vary_rad_vel, vary_z_vel, L_ref):

    #Angular, radial and z wander plots
    #For each, all other variables may be plotted on the x axis
    #Selection of graphs plotted through parameters from main()
    #Text, labels etc for titles, axes and file names

    if L_ref == 0:
        label = "L4"
    elif L_ref == 1:
        label = "L5"
    else:
        label = "av"

    if ang_wander_plt == 1:
        wander_text = "angular"
        if vary_mass == 1:
            plot_wander(mass_arr, ang_wander_arr, wander_text, "planet mass", " / $M_{\odot}$",
label)
        elif vary_ang_disp == 1:
            plot_wander(ang_disp_arr, ang_wander_arr, wander_text, "initial angular displacement",
" / AU", label)
        elif vary_rad_disp == 1:
            plot_wander(rad_disp_arr, ang_wander_arr, wander_text, "initial radial displacement", " /
AU", label)
```

```
    elif vary_z_disp == 1:
        plot_wander(z_disp_arr, ang_wander_arr, wander_text, "initial z displacement", "/ AU",
label)
    elif vary_ang_vel == 1:
        plot_wander(ang_vel_arr, ang_wander_arr, wander_text, "initial angular velocity", " /
AU $yr^{-1}$", label)
    elif vary_rad_vel == 1:
        plot_wander(rad_vel_arr, ang_wander_arr, wander_text, "initial radial velocity", " / AU
$yr^{-1}$", label)
    elif vary_z_vel == 1:
        plot_wander(z_vel_arr, ang_wander_arr, wander_text, "initial z velocity", " / AU $yr^{-
1}$", label)

  if rad_wander_plt == 1:
    wander_text = "radial"
    if vary_mass == 1:
        plot_wander(mass_arr, rad_wander_arr, wander_text, "planet mass", " / $M_{\odot}$",
label)
    elif vary_ang_disp == 1:
        plot_wander(ang_disp_arr, rad_wander_arr, wander_text, "initial angular displacement",
" / AU", label)
    elif vary_rad_disp == 1:
        plot_wander(rad_disp_arr, rad_wander_arr, wander_text, "initial radial displacement", " /
AU", label)
    elif vary_z_disp == 1:
        plot_wander(z_disp_arr, rad_wander_arr, wander_text, "initial z displacement", " / AU",
label)
    elif vary_ang_vel == 1:
        plot_wander(ang_vel_arr, rad_wander_arr, wander_text, "initial angular velocity", " / AU
$yr^{-1}$", label)
    elif vary_rad_vel == 1:
        plot_wander(rad_vel_arr, rad_wander_arr, wander_text, "initial radial velocity", " / AU
$yr^{-1}$", label)
    elif vary_z_vel == 1:
        plot_wander(z_vel_arr, rad_wander_arr, wander_text, "initial z velocity", " / AU $yr^{-
1}$", label)

  if z_wander_plt == 1:
    wander_text = "z"
    if vary_mass == 1:
        plot_wander(mass_arr, z_wander_arr, wander_text, "planet mass", " / $M_{\odot}$",
label)
    elif vary_ang_disp == 1:
        plot_wander(ang_disp_arr, z_wander_arr, wander_text, "initial angular displacement", "/
AU", label)
    elif vary_rad_disp == 1:
```

```
        plot_wander(rad_disp_arr, z_wander_arr, wander_text, "initial radial displacement", "/
AU", label)
    elif vary_z_disp == 1:
        plot_wander(z_disp_arr, z_wander_arr, wander_text, "initial z displacement", "/ AU",
label)
    elif vary_ang_vel == 1:
        plot_wander(ang_vel_arr, z_wander_arr, wander_text, "initial angular velocity", " / AU
$yr^{-1}$", label)
    elif vary_rad_vel == 1:
        plot_wander(rad_vel_arr, z_wander_arr, wander_text, "initial radial velocity", " / AU
$yr^{-1}$", label)
    elif vary_z_vel == 1:
        plot_wander(z_vel_arr, z_wander_arr, wander_text, "initial z velocity", " / AU $yr^{-
1}$", label)

  return 0


#Swap order of max and min if necessary, and determine if the parameter is changing:
def minmax_order(max, min):

  #Order check, swap if necessary
  if min > max:
    temp = min
    min = max
    max = temp

  #Flag if the parameter is varying or not
  if min == max:
    vary_flag = np.bool(0)
  else:
    vary_flag = np.bool(1)

  return max, min, vary_flag


#Take unweighted average of two scalars or arrays
def average(a, b):

  if np.isscalar(a) == 1:
    average = np.float64(0)
  else:
    average = np.zeros(len(a))

  average = np.add(a, b)
  average = np.divide(average, 2)
```

```
    return average


#Calculate (integer) number of orbits of Jupiter for a given time, and the time period:
def calc_orbits(tmax, omega):

    T_period = np.float64(0)
    orbits_no = np.float64(0)
    int_orbits_no = np.int(0)

    T_period = 2 * pi / omega
    orbits_no = tmax / T_period
    int_orbits_no = round(orbits_no)

    return int_orbits_no, T_period


#Add three vectors in quadrature:
def quad_add(x, y, z):

    arr_size = len(x)
    tot_sqr, tot = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype = np.float64)
    x_sqr, y_sqr, z_sqr = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype =
np.float64), np.zeros(arr_size, dtype = np.float64)

    #Square each component and sum
    x_sqr = np.square(x)
    y_sqr = np.square(y)
    z_sqr = np.square(z)

    tot_sqr = np.add(x_sqr,np.add(y_sqr,z_sqr))
    tot = np.sqrt(tot_sqr)

    return tot_sqr, tot


#Calculate angular frequency, omega, for two bodies rotating about the z-axis:
def ang_freq(G, R, body1, body2):

    mass_sum = np.float64(0)
    omega = np.float64(0)
    omega_vector = np.zeros(3, dtype = np.float64)

    mass_sum = body1.mass + body2.mass
    omega = sqrt((G * mass_sum / R**3))
```

```
    omega_vector = [0, 0, omega]

    return omega_vector


#Calculate the positions of the Sun and Jupiter relative to the COM:
def body_positions(R, body1, body2, L_point):

    mass_sum = np.float64(0)
    COM_dist = np.float64(0)
    pos_1, pos_2 = np.zeros(3, dtype = np.float64), np.zeros(3, dtype = np.float64)

    #Calculate the sum of the two masses and the distance from the larger mass to the COM
    mass_sum = body1.mass + body2.mass
    COM_dist = body1.mass * R / mass_sum

    #Calculate the position vectors of both masses
    pos_1 = [R-COM_dist, 0.0, 0.0]
    pos_2 = [-COM_dist, 0.0, 0.0]

    return pos_1, pos_2


#Calculates the (absolute) distance between asteroid and another body:
def body_dist(body, x):

    arr_size = len(x[:,0])
    x_body, x_body_sqr = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype =
np.float64)
    y_body, y_body_sqr = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype =
np.float64)
    z_body, z_body_sqr = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype =
np.float64)
    r_body, r_body_sqr = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype =
np.float64)

    #Calculate the distance beteeen the body and x in each Cartesian direction
    x_body = np.subtract(x[:,0], body.position[0])
    x_body_sqr = np.square(x_body)
    y_body = np.subtract(x[:,1], body.position[1])
    y_body_sqr = np.square(y_body)
    z_body = np.subtract(x[:,2], body.position[2])
    z_body_sqr = np.square(z_body)

    #Calculate r and r^2 from components
    r_body_sqr = np.add(x_body_sqr, np.add(y_body_sqr, z_body_sqr))
```

```
    r_body = np.sqrt(r_body_sqr)

    return r_body


#Calculate the L4 and L5 Langrange points for two massive bodies:
def L_points(R, body1, body2):

    L4_x, L4_y  = np.float64(0), np.float64(0)
    L5_x, L5_y  = np.float64(0), np.float64(0)

    #Calculate components of L4 and L5
    L4_x = 0.5 * (body2.mass - body1.mass) * R / (body1.mass + body2.mass)
    L4_y = sqrt(3) * R /2

    L5_x = L4_x
    L5_y = - L4_y

    return [L4_x, L4_y, 0], [L5_x, L5_y, 0]


#Rotate coordinates about z-axis:
#Note: t input can be both a scalar and an array
def coord_rot(x, t, omega):

    arr_size = len(x[:,0])

    #Initialise, taking into account t can be a scalar and an array
    if np.isscalar(t) == 1:
        theta = np.float64(0)
        cos = np.float64(0)
        sin = np.float64(0)
    else:
        theta = np.zeros(arr_size, dtype = np.float64)
        cos = np.zeros(arr_size, dtype = np.float64)
        sin = np.zeros(arr_size, dtype = np.float64)

    cos_x = np.zeros(arr_size, dtype = np.float64)
    cos_y = np.zeros(arr_size, dtype = np.float64)
    sin_x = np.zeros(arr_size, dtype = np.float64)
    sin_y = np.zeros(arr_size, dtype = np.float64)
    rot_x = np.zeros(arr_size, dtype = np.float64)
    rot_y = np.zeros(arr_size, dtype = np.float64)

    #Array/scalar (depending on input t) to represent angle(s) to rotate by
    theta = np.multiply(t, omega)
```

```
    #Calculate (x * cos - y * sin) and (x * sin + y * sin)
    cos = np.cos(theta)
    sin = np.sin(theta)

    cos_x = np.multiply(cos, x[:,0])
    cos_y = np.multiply(cos, x[:,1])
    sin_x = np.multiply(sin, x[:,0])
    sin_y = np.multiply(sin,x[:,1])

    rot_x = np.subtract(cos_x,sin_y)
    rot_y = np.add(sin_x,cos_y)

    return rot_x, rot_y


#Calculate sum of potential energies due to two bodies:
def pot_en(G, x, body1, body2):

    arr_size = len(x[:,0])
    dist1, dist2 = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype = np.float64)
    U1, U2, U_tot = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype = np.float64),
np.zeros(arr_size, dtype = np.float64)

    #Calculate the disance from x to two different bodies
    dist1 = body_dist(body1, x)
    dist2 = body_dist(body2, x)

    #Calculate the potential energy as a result of each of the two bodies, and their sum
    U1 = - np.divide(G * body1.mass, dist1)
    U2 = - np.divide(G * body2.mass, dist2)
    U_tot = np.add(U1,U2)

    return U_tot


#Calcualte kinetic energies per unit mass:
def kin_en(x, omega):

    arr_size = len(x[:,0])
    KE1, KE2 = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype = np.float64)
    vel_sqr, vel = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype = np.float64)
    dist_sqr, dist = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype = np.float64)

    #Caluclate v^2 and r^2
    vel_sqr, vel = quad_add(x[:,3],x[:,4],x[:,5])
```

```
    dist_sqr, dist = quad_add(x[:,0],x[:,1],x[:,2])

    KE1 = np.multiply(vel_sqr, 0.5)
    KE2 = np.multiply(0.5 * omega**2, dist_sqr)

    return KE1, KE2


#Caclulate the Hamiltonian per unit mass from the kinetic and potential energies:
def hamiltonian(G, x, jupiter, sun, omega):

    arr_size = len(x[:,0])
    U_tot, KE1, KE2 = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype =
np.float64), np.zeros(arr_size, dtype = np.float64)
    ham = np.zeros(arr_size, dtype = np.float64)
    ham_error = np.zeros(arr_size, dtype = np.float64)

    #Calculate potential and kinetic energies
    U_tot = pot_en(G, x, jupiter, sun)
    KE1, KE2 = kin_en(x, omega)

    #Combine energies to give the Hamiltonian per unit mass
    ham = np.add(KE1, np.subtract(U_tot, KE2))
    ham_error = np.abs(np.subtract(ham, ham[0]))

    return ham, ham_error


#Calculate maximum wander angle:
def calc_ang_wander(R, x, L_point, conditions_no, interupt):

    L_angle = np.zeros(2)

    ast_angle = np.zeros([conditions_no, 3])
    ast_angle_sign = np.zeros([conditions_no, 3])
    ast_angle_negatives = np.zeros([conditions_no, 3])
    ast_angle_correction = np.zeros([conditions_no, 3])
    wander_angle = np.zeros([conditions_no, 3])

    max_ang_wander, min_ang_wander, tot_ang_wander = np.float64(0.0), np.float64(0.0),
np.float64(0.0)
    max_ang_index, min_ang_index = np.int(0), np.int(0)
    rad_mag = np.float64(0)

    #Angles of the Lagrange reference point and the asteroid at each point in time
    #Defined from -pi to pi
```

```
    L_angle = np.arctan2(L_point[1], L_point[0])
    ast_angle = np.arctan2(x[:,1], x[:,0])

    #Find the angle wrt the relevant Lagrange point
    #add/subtract 2pi to angles for angles in two quadrants above/below L5/L4
    #Difference to L point taken, so angles are positive away from Jupiter, positive towards it

    if L_angle > 0:
        ast_angle_sign = np.sign(ast_angle)
        ast_angle_negatives = np.subtract(ast_angle_sign, 1)
        ast_angle_correction = np.multiply(ast_angle_negatives, -pi)
        ast_angle = np.add(ast_angle, ast_angle_correction)
        wander_angle = np.subtract(ast_angle, L_angle)
    else:
        ast_angle_sign = np.sign(ast_angle)
        ast_angle_positives = np.add(ast_angle_sign, 1)
        ast_angle_correction = np.multiply(ast_angle_positives, pi)
        ast_angle = np.subtract(ast_angle, ast_angle_correction)
        wander_angle = np.subtract(L_angle, ast_angle)

    #Find the max-min of the angle, and get index for the time/position this occurs
    max_ang_wander = np.max(wander_angle)
    min_ang_wander = np.min(wander_angle)
    max_ang_index = np.argmax(wander_angle)
    min_ang_index = np.argmin(wander_angle)

    tot_ang_wander = max_ang_wander - min_ang_wander

    if interupt == 1:
        tot_ang_wander = 2 * pi

    #Convert to AU
    rad_mag  = np.linalg.norm(L_point)
    tot_ang_wander *= rad_mag

    return tot_ang_wander, max_ang_index, min_ang_index


#Calculate maximum radial wander, based on radial distance from COM:
def calc_rad_wander(x, R, conditions_no):

    max_rad_wander, min_rad_wander, tot_rad_wander = np.float64(0.0), np.float64(0.0),
np.float64(0.0)
    max_rad_index, min_rad_index = np.int(0), np.int(0)
    ast_mag_sq = np.zeros(conditions_no, dtype = np.float64)
    ast_mag = np.zeros(conditions_no, dtype = np.float64)
```

```
    #calculate the asteroids radius at all times, and find the max-min
    ast_mag_sq, ast_mag = quad_add(x[:,0],x[:,1],x[:,2])

    max_rad_wander = np.max(ast_mag)
    min_rad_wander = np.min(ast_mag)
    max_rad_index = np.argmax(ast_mag)
    min_rad_index = np.argmin(ast_mag)

    tot_rad_wander = max_rad_wander - min_rad_wander

    #Limit to 2.5 * the radius of Jupiter's orbit
    if tot_rad_wander > 2.5 * R:
        tot_rad_wander = 2.5 * R

    return tot_rad_wander, max_rad_index, min_rad_index


#Calculate maximum wander in the z direction:
def calc_z_wander(x):

    max_z_wander, min_z_wander, tot_z_wander = np.float64(0.0), np.float64(0.0),
np.float64(0.0)

    #Calculate max - min of the asteroids z position
    max_z_wander = np.max(x[:,2])
    min_z_wander = np.min(x[:,2])

    tot_z_wander = max_z_wander - min_z_wander

    return tot_z_wander


#Create arrays of x and y positions of the asteroid, displaced in the azimuthal direction:
def calc_ang_disp(position, ang_disp_arr, conditions_no):

    rad_mag = np.float64(0)
    omega = np.float64(0)
    x = np.zeros((1,2))
    ang_disp_x_arr, ang_disp_y_arr = np.zeros(conditions_no, dtype = np.float64),
np.zeros(conditions_no, dtype = np.float64)

    #Definte angular frequency, omega, such that angular displacement is in AU
    rad_mag  = np.linalg.norm(position)
    omega = 1 / rad_mag
```

```
    #Location of the point to be rotated, cast in the form 'coord_rot' takes
    x[0,0] = position[0]
    x[0,1] = position[1]

    #Rotate coordinates  of L point to produce angular displacement
    ang_disp_x_arr, ang_disp_y_arr = coord_rot(x, ang_disp_arr, omega)

    return ang_disp_x_arr, ang_disp_y_arr


#Create array of vectors in the radial direction of varying length:
def calc_rad_arr(x_pos_arr, y_pos_arr, var_arr):

    arr_size = len(x_pos_arr)
    pos_arr_sq, pos_arr = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype =
np.float64)
    norm_x_pos_arr, norm_y_pos_arr = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size,
dtype = np.float64)
    rad_x_arr, rad_y_arr = np.zeros(arr_size, dtype = np.float64), np.zeros(arr_size, dtype =
np.float64)

    #Calculte normalised radial vector, split into x and y components
    pos_arr_sq, pos_arr = quad_add(x_pos_arr, y_pos_arr, np.zeros(arr_size))
    norm_x_pos_arr = np.divide(x_pos_arr, pos_arr)
    norm_y_pos_arr = np.divide(y_pos_arr, pos_arr)

    #Vary length of r vectors, based on var_arr
    rad_x_arr = np.multiply(norm_x_pos_arr, var_arr)
    rad_y_arr = np.multiply(norm_y_pos_arr, var_arr)

    return rad_x_arr, rad_y_arr


#Calculate initial position and velocity of asteroid:
def initial_cond(L_point, ang_disp_arr, rad_disp_arr, ang_vel_arr, rad_vel_arr, conditions_no):

    ang_x_pos = np.zeros(conditions_no, dtype = np.float64)
    ang_y_pos = np.zeros(conditions_no, dtype = np.float64)
    rad_x_disp = np.zeros(conditions_no, dtype = np.float64)
    rad_y_disp = np.zeros(conditions_no, dtype = np.float64)
    ang_x_vel = np.zeros(conditions_no, dtype = np.float64)
    ang_y_vel = np.zeros(conditions_no, dtype = np.float64)
    rad_x_vel = np.zeros(conditions_no, dtype = np.float64)
    rad_y_vel = np.zeros(conditions_no, dtype = np.float64)
    tot_x_pos = np.zeros(conditions_no, dtype = np.float64)
    tot_y_pos = np.zeros(conditions_no, dtype = np.float64)
```

```
    x_vel_tot = np.zeros(conditions_no, dtype = np.float64)
    y_vel_tot = np.zeros(conditions_no, dtype = np.float64)


    #If no angular displacement, initially at L point, else calculate angular displacements
    #Output is final set of displaced positions, not displacements
    if np.max(ang_disp_arr) == np.min(ang_disp_arr) == 0.0:
        ang_x_pos = np.add(ang_x_pos, L_point[0])
        ang_y_pos = np.add(ang_y_pos, L_point[1])
    else:
        ang_x_pos, ang_y_pos = calc_ang_disp(L_point, ang_disp_arr, conditions_no)

    #Calculate radial displacements. Note: radial direction defined from angularly displaced
position
    if not (np.max(rad_disp_arr) == np.min(rad_disp_arr) == 0.0):
        rad_x_disp, rad_y_disp = calc_rad_arr(ang_x_pos, ang_y_pos, rad_disp_arr)

    tot_x_pos = np.add(rad_x_disp, ang_x_pos)
    tot_y_pos = np.add(ang_y_pos, rad_y_disp)


    #Calculate velocities in the radial and azimuthal (tangential) directions
    if not (np.max(ang_vel_arr) == np.min(ang_vel_arr) == 0.0):
        ang_x_vel, ang_y_vel = calc_rad_arr(tot_x_pos, tot_y_pos, ang_vel_arr)

    if not (np.max(rad_vel_arr) == np.min(rad_vel_arr) == 0.0):
        rad_x_vel, rad_y_vel = calc_rad_arr(tot_x_pos, tot_y_pos, rad_vel_arr)

    x_vel_tot = np.subtract(rad_x_vel, ang_y_vel)
    y_vel_tot = np.add(rad_y_vel, ang_x_vel)

    return tot_x_pos, tot_y_pos, x_vel_tot, y_vel_tot


#Calculate angular frequency (omega), total number of orbits of Jupiter, Lagrange points inc.
#which is the reference point, and the positions of Jupiter and the Sun relative to the COM
def system_consts(G, R, jupiter, sun, L4_L5, tmax, orbit_info):

    omega_vector = np.zeros(3, dtype = np.float64)
    L4, L5 = np.zeros(3, dtype = np.float64), np.zeros(3, dtype = np.float64)
    L_point = np.zeros(3, dtype = np.float64)

    #Calculate angular frequency as a vector (in the z direction)
    omega_vector  = ang_freq(G, R, jupiter, sun)

    #Print time period and number of orbits of Jupiter, if requested
```

```
    if orbit_info == 1:
        orbits_no, T_period = calc_orbits(tmax, omega_vector[2])
        print("Number of orbits:", orbits_no)
        print("Time period:", T_period)

    #Calculate Lagrange points and selects the starting point
    L4, L5 = L_points(R, jupiter, sun)

    if L4_L5 == 0:
        L_point = L4
    else:
        L_point = L5

    jupiter.position, sun.position = body_positions(R, jupiter, sun, L_point)

    return jupiter, sun, omega_vector, L_point, L4, L5


#Evaluates the derivatives for the equation of motion of the asteroid, split into Cartesian
components
#x[0:2] = displacements (x) of asteroid, x[3:5] = velocities (dx/dt) of asteroid
#Returns y (= dx/dt), dy/dt (= d^2 x/dt^2 )
def derivatives(t, x, G, jupiter, sun, omega):

    rel_pos_jup, rel_pos_sun = np.zeros(3, dtype = np.float64), np.zeros(3, dtype = np.float64)
    r_jup, r_sun = np.float64(0.0), np.float64(0.0)
    corr_force, cent_force,  fic_forces = np.zeros(3, dtype = np.float64), np.zeros(3, dtype =
np.float64), np.zeros(3, dtype = np.float64)
    jup_grav, sun_grav = np.zeros(3, dtype = np.float64), np.zeros(3, dtype = np.float64)
    y = np.zeros(3, dtype = np.float64)

    #Calculate distance between the asteroid and the Sun/Jupiter
    rel_pos_jup = np.subtract([x[0], x[1], x[2]], jupiter.position)
    rel_pos_sun = np.subtract([x[0], x[1], x[2]], sun.position)
    r_jup = np.linalg.norm(rel_pos_jup)
    r_sun = np.linalg.norm(rel_pos_sun)

    #Calculate forces on asteroid (fictitious, then gravitatonal)
    corr_force = - 2 * np.cross(omega, x[3:])
    cent_force = - np.cross(omega, np.cross(omega, x[:3]))

    fic_forces = np.add(corr_force, cent_force)

    jup_grav = - np.multiply(G * jupiter.mass / r_jup**3, rel_pos_jup)
    sun_grav = - np.multiply(G * sun.mass / r_sun**3, rel_pos_sun)
```

```
    y = np.add(np.add(fic_forces, sun_grav), jup_grav)

    return [x[3], x[4], x[5], y[0], y[1], y[2]]


#Solve ODE for initial displacements and velocities, using:
def dopri5_integrator(G, asteroid, jupiter, sun, omega_vector, hmax, N, tmax, tol, y_excl_rng,
L_point, warning_supp):

    interupt = np.bool(0)

    #Create ode instance to solve the system of differential equations,
    #defined by `derivatives`, solver method set to 'dorpi5'.
    backend = 'dopri5'
    solver = ode(derivatives).set_integrator(backend, nsteps = N, rtol = tol, atol = tol, max_step =
hmax)

    #Parameters
    solver.set_f_params(G, jupiter, sun, omega_vector)

    #Initial values for displacements, then velocities
    x0 = [asteroid.position[0],asteroid.position[1], asteroid.position[2], asteroid.velocity[0],
asteroid.velocity[1],asteroid.velocity[2]]
    t0 = np.float64(0.0)
    solver.set_initial_value(x0, t0)

    soln = []
    t_arr = []

    #Single integration called, solout used to output variables for each step taken
    def solout(t, y):
        soln.append([*y])
        t_arr.append(t)

        #Stop integration if asteroid passes Jupiter
        if -y_excl_rng < y[1] < y_excl_rng and y[0] > 0:
            print("Integration interupted: asteroid passing Jupiter")
            return -1
        else:
            return 0

    solver.set_solout(solout)
    solver.integrate(tmax)
    soln = np.array(soln)

    #Flag if the integration was interupted i.e. the asteroid passed Jupiter
```

```
    if np.max(t_arr) < tmax:
        interupt = 1

    return soln, t_arr, interupt


#Calculate solution for a given set of conditions
#Calls function to perform integration, calculates wander and calls function for plotting solution
def single_cond_soln(G, R, asteroid, jupiter, sun, i, L_point, L4, L5, omega_vector, tmax, N, tol,
y_excl_rng, max_step, ham_info, warning_supp, conditions_no, ham_plt, rot_plt, sol_plt,
traj_plt, xyz_plt):

    x = np.zeros(conditions_no, dtype = np.float64)
    t = np.zeros(conditions_no, dtype = np.float64)
    ham = np.zeros(conditions_no, dtype = np.float64)
    ham_error = np.zeros(conditions_no, dtype = np.float64)

    #Calculate displacements and velocities in rotating frame of reference
    x, t, interupt = dopri5_integrator(G, asteroid, jupiter, sun, omega_vector, max_step, N, tmax,
tol, y_excl_rng, L_point, warning_supp)

    #Inertial coordinates and plotting by consiering a rotation
    rot_x, rot_y = np.zeros(conditions_no, dtype = np.float64), np.zeros(conditions_no, dtype =
np.float64)

    if rot_plt == 1:
        rot_x, rot_y = coord_rot(x, t, omega_vector[2])

    #Calculate Hamiltonian for all points in time
    ham, ham_error = hamiltonian(G, x, jupiter, sun, omega_vector[2])
    if (np.max(ham) - np.min(ham)) > (10**-4) and warning_supp == 0:
        print("Warning: larger than usual error in Hamiltonian for condition number = ", i)
        print("Asteroid position = ", asteroid.position)
        print("Asteroid velocity = ", asteroid.velocity)
        print("Planet mass = ", jupiter.mass)
        print("Error =", np.max(ham) - np.min(ham))

    if ham_info == 1:
        print("Min Hamiltonian:", np.min(ham))
        print("Max Hamiltonian:", np.max(ham))
        print("Hamiltonian variation:", np.max(ham) - np.min(ham))


    #Calculate angular, radial and z wander and their index for reference
    ang_wander, max_ang_index, min_ang_index = np.float64(0.0), np.int(0), np.int(0)
    rad_wander, max_rad_index, min_rad_index = np.float64(0.0), np.int(0), np.int(0)
```

```
    z_wander = np.float64(0.0)

    ang_wander, max_ang_index, min_ang_index = calc_ang_wander(R, x, L_point,
conditions_no, interupt)
    rad_wander, max_rad_index, min_rad_index = calc_rad_wander(x, R, conditions_no)
    z_wander = calc_z_wander(x)

    #Warning for very large radial wander
    if rad_wander == 2.5 * R and warning_supp == 0:
        print("Warning: asteroid has very large radial displacement for value = ", i)
        print("Asteroid position = ", asteroid.position)
        print("Asteroid velocity = ", asteroid.velocity)
        print("Planet mass = ", jupiter.mass)
        print("error =", np.max(ham) - np.min(ham))

    #Plotting functions for given set of conditions
    if ham_plt == 1 or rot_plt == 1 or sol_plt ==1 or traj_plt == 1 or xyz_plt == 1:
        plot_set_cond(R, x, t, ham_error, rot_x, rot_y, jupiter, sun, L_point, L4, L5,
max_ang_index, min_ang_index, max_rad_index, min_rad_index, ham_plt, rot_plt, sol_plt,
traj_plt, xyz_plt)

    return ang_wander, rad_wander, z_wander


#Single set of system values calculated
#Initial conditions calculated for variable (mass is const), then asteroid motion solved for each
case
def const_mass_calc(G, R, asteroid, jupiter, sun, mass_arr, ang_disp_arr, rad_disp_arr,
z_disp_arr, ang_vel_arr, rad_vel_arr, z_vel_arr, max_step, N, tmax, tol, y_excl_rng, L4_L5,
ham_info, orbit_info, warning_supp, conditions_no, mass_max, mass_min, ang_disp_max,
ang_disp_min, rad_disp_max, rad_disp_min, z_disp_max, z_disp_min, ang_vel_max,
ang_vel_min, rad_vel_max, rad_vel_min, z_vel_max, z_vel_min, vary_mass, vary_ang_disp,
vary_rad_disp, vary_z_disp, vary_ang_vel, vary_rad_vel, vary_z_vel, ang_wander_plt,
rad_wander_plt, z_wander_plt, ham_plt, rot_plt, sol_plt, traj_plt, xyz_plt):

    omega_vector = np.zeros(3, dtype = np.float64)
    L4, L5 = np.zeros(3, dtype = np.float64), np.zeros(3, dtype = np.float64)
    L_point = np.zeros(3, dtype = np.float64)

    tot_x_pos = np.zeros(conditions_no, dtype = np.float64)
    tot_y_pos = np.zeros(conditions_no, dtype = np.float64)
    x_vel_tot = np.zeros(conditions_no, dtype = np.float64)
    y_vel_tot = np.zeros(conditions_no, dtype = np.float64)
    temp_ang_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    temp_rad_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    temp_z_wander_arr = np.zeros(conditions_no, dtype = np.float64)
```

```
    jupiter, sun, omega_vector, L_point, L4, L5 = system_consts(G, R, jupiter, sun, L4_L5, tmax,
orbit_info)
    tot_x_pos, tot_y_pos, x_vel_tot, y_vel_tot = initial_cond(L_point, ang_disp_arr, rad_disp_arr,
ang_vel_arr, rad_vel_arr, conditions_no)

    #Vary initial conditions, then calculate wanders for each
    for i in range(conditions_no):

        asteroid.position = [tot_x_pos[i], tot_y_pos[i], z_disp_arr[i]]
        asteroid.velocity = [x_vel_tot[i], y_vel_tot[i], z_vel_arr[i]]

        temp_ang_wander_arr[i], temp_rad_wander_arr[i], temp_z_wander_arr[i] =
single_cond_soln(G, R, asteroid, jupiter, sun, i, L_point, L4, L5, omega_vector, tmax, N, tol,
y_excl_rng, max_step, ham_info, warning_supp, conditions_no, ham_plt, rot_plt, sol_plt,
traj_plt, xyz_plt)

        print(i)

    return temp_ang_wander_arr, temp_rad_wander_arr, temp_z_wander_arr


#For each mass, Lagrange point, positions etc determinded,
#Initial conditions calculated, then asteroid motion solved for each mass of Jupiter
def var_mass_calc(G, R, asteroid, jupiter, sun, mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr,
ang_vel_arr, rad_vel_arr, z_vel_arr, max_step, N, tmax, tol, y_excl_rng, L4_L5, ham_info,
orbit_info, warning_supp, conditions_no, mass_max, mass_min, ang_disp_max, ang_disp_min,
rad_disp_max, rad_disp_min, z_disp_max, z_disp_min, ang_vel_max, ang_vel_min,
rad_vel_max, rad_vel_min, z_vel_max, z_vel_min, vary_mass, vary_ang_disp, vary_rad_disp,
vary_z_disp, vary_ang_vel, vary_rad_vel, vary_z_vel, ang_wander_plt, rad_wander_plt,
z_wander_plt, ham_plt, rot_plt, sol_plt, traj_plt, xyz_plt):

    omega_vector = np.zeros(3, dtype = np.float64)
    L4, L5 = np.zeros(3, dtype = np.float64), np.zeros(3, dtype = np.float64)
    L_point = np.zeros(3, dtype = np.float64)

    tot_x_pos = np.zeros(conditions_no, dtype = np.float64)
    tot_y_pos = np.zeros(conditions_no, dtype = np.float64)
    x_vel_tot = np.zeros(conditions_no, dtype = np.float64)
    y_vel_tot = np.zeros(conditions_no, dtype = np.float64)
    temp_ang_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    temp_rad_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    temp_z_wander_arr = np.zeros(conditions_no, dtype = np.float64)

    #Vary mass of planet, calculating new system coordinates, initial conditions, then wanders
    for i in range(conditions_no):
```

```
        jupiter.mass = mass_arr[i]
        jupiter, sun, omega_vector, L_point, L4, L5 = system_consts(G, R, jupiter, sun, L4_L5,
tmax, orbit_info)
        tot_x_pos, tot_y_pos, x_vel_tot, y_vel_tot = initial_cond(L_point, ang_disp_arr[i],
rad_disp_arr[i], ang_vel_arr[i], rad_vel_arr[i], conditions_no)

        asteroid.position = [tot_x_pos[0], tot_y_pos[0], z_disp_arr[i]]
        asteroid.velocity = [x_vel_tot[0], y_vel_tot[0], z_vel_arr[i]]

        temp_ang_wander_arr[i], temp_rad_wander_arr[i], temp_z_wander_arr[i] =
single_cond_soln(G, R, asteroid, jupiter, sun, mass_arr[i], L_point, L4, L5, omega_vector, tmax,
N, tol, y_excl_rng, max_step, ham_info, warning_supp, conditions_no, ham_plt, rot_plt, sol_plt,
traj_plt, xyz_plt)

        print(i)

    return temp_ang_wander_arr, temp_rad_wander_arr, temp_z_wander_arr


#Generate variable initial conditions, call function to calculate solns for each set, and plot
wander/variation
def tot_wander(G, M_sol, R, max_step, N, tmax, tol, y_excl_rng, L4_L5, L_point_av, ham_info,
orbit_info, warning_supp, conditions_no, mass_max, mass_min, ang_disp_max, ang_disp_min,
rad_disp_max, rad_disp_min, z_disp_max, z_disp_min, ang_vel_max, ang_vel_min,
rad_vel_max, rad_vel_min, z_vel_max, z_vel_min, ang_wander_plt, rad_wander_plt,
z_wander_plt, ham_plt, mass_wander_plt, rot_plt, sol_plt, traj_plt, xyz_plt):

    #Create massive bodies
    jupiter = body()
    sun = body()
    asteroid = body()

    sun.mass = M_sol
    jupiter.mass = mass_min

    #Re-order min/max if necessary and determine if there is any variation
    mass_max, mass_min, vary_mass = minmax_order(mass_max, mass_min)
    ang_disp_max, ang_disp_min, vary_ang_disp = minmax_order(ang_disp_max,
ang_disp_min)
    rad_disp_max, rad_disp_min, vary_rad_disp = minmax_order(rad_disp_max, rad_disp_min)
    z_disp_max, z_disp_min, vary_z_disp = minmax_order(z_disp_max, z_disp_min)
    ang_vel_max, ang_vel_min, vary_ang_vel = minmax_order(ang_vel_max, ang_vel_min)
    rad_vel_max, rad_vel_min, vary_rad_vel = minmax_order(rad_vel_max, rad_vel_min)
    z_vel_max, z_vel_min, vary_z_vel = minmax_order(z_vel_max, z_vel_min)
```

```
    #If no variation, only one condition
    if vary_mass == vary_ang_disp == vary_rad_disp  == vary_z_disp == vary_ang_vel  ==
vary_rad_vel == vary_z_vel == 0:
        conditions_no = 1

    #Create arrays for variables and wander
    mass_arr = np.linspace(mass_min, mass_max, conditions_no, dtype = np.float64)
    ang_disp_arr = np.linspace(ang_disp_min, ang_disp_max, conditions_no, dtype = np.float64)
    rad_disp_arr = np.linspace(rad_disp_min, rad_disp_max, conditions_no, dtype = np.float64)
    z_disp_arr = np.linspace(z_disp_min, z_disp_max, conditions_no, dtype = np.float64)
    ang_vel_arr = np.linspace(ang_vel_min, ang_vel_max, conditions_no, dtype = np.float64)
    rad_vel_arr = np.linspace(rad_vel_min, rad_vel_max, conditions_no, dtype = np.float64)
    z_vel_arr = np.linspace(z_vel_min, z_vel_max, conditions_no, dtype = np.float64)

    ang_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    rad_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    z_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    ang_wander_arr_2 = np.zeros(conditions_no, dtype = np.float64)
    rad_wander_arr_2 = np.zeros(conditions_no, dtype = np.float64)
    z_wander_arr_2 = np.zeros(conditions_no, dtype = np.float64)
    av_ang_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    rad_wander_arr = np.zeros(conditions_no, dtype = np.float64)
    z_wander_arr = np.zeros(conditions_no, dtype = np.float64)

    #Calculate wanders for sets of initial conditions. Mass is separate as the entire system changes
for each mass
    if vary_mass == 0:
        ang_wander_arr, rad_wander_arr, z_wander_arr = const_mass_calc(G, R, asteroid, jupiter,
sun, mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr, ang_vel_arr, rad_vel_arr, z_vel_arr,
max_step, N, tmax, tol, y_excl_rng, L4_L5, ham_info, orbit_info, warning_supp, conditions_no,
mass_max, mass_min, ang_disp_max, ang_disp_min, rad_disp_max, rad_disp_min,
z_disp_max, z_disp_min, ang_vel_max, ang_vel_min, rad_vel_max, rad_vel_min, z_vel_max,
z_vel_min, vary_mass, vary_ang_disp, vary_rad_disp, vary_z_disp, vary_ang_vel, vary_rad_vel,
vary_z_vel, ang_wander_plt, rad_wander_plt, z_wander_plt, ham_plt, rot_plt, sol_plt, traj_plt,
xyz_plt)
    else:
        ang_wander_arr, rad_wander_arr, z_wander_arr = var_mass_calc(G, R, asteroid, jupiter,
sun, mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr, ang_vel_arr, rad_vel_arr, z_vel_arr,
max_step, N, tmax, tol, y_excl_rng, L4_L5, ham_info, orbit_info, warning_supp, conditions_no,
mass_max, mass_min, ang_disp_max, ang_disp_min, rad_disp_max, rad_disp_min,
z_disp_max, z_disp_min, ang_vel_max, ang_vel_min, rad_vel_max, rad_vel_min, z_vel_max,
z_vel_min, vary_mass, vary_ang_disp, vary_rad_disp, vary_z_disp, vary_ang_vel, vary_rad_vel,
vary_z_vel, ang_wander_plt, rad_wander_plt, z_wander_plt, ham_plt, rot_plt, sol_plt, traj_plt,
xyz_plt)

    #Average over L points if necessary, reversing relevant variables (temporarily) for symmetry
```

```
    if L_point_av == 1:

        #Swap L points, and reflect required variables
        L4_L5 = not L4_L5
        ang_disp_arr = np.multiply(ang_disp_arr, -1)
        ang_vel_arr = np.multiply(ang_vel_arr, -1)

        #Calculate wanders from alt. L point
        if vary_mass == 0:
            ang_wander_arr_2, rad_wander_arr_2, z_wander_arr_2 = const_mass_calc(G, R,
asteroid, jupiter, sun, mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr, ang_vel_arr, rad_vel_arr,
z_vel_arr, max_step, N, tmax, tol, y_excl_rng, L4_L5, ham_info, orbit_info, warning_supp,
conditions_no, mass_max, mass_min, ang_disp_max, ang_disp_min, rad_disp_max,
rad_disp_min, z_disp_max, z_disp_min, ang_vel_max, ang_vel_min, rad_vel_max,
rad_vel_min, z_vel_max, z_vel_min, vary_mass, vary_ang_disp, vary_rad_disp, vary_z_disp,
vary_ang_vel, vary_rad_vel, vary_z_vel, ang_wander_plt, rad_wander_plt, z_wander_plt,
ham_plt, rot_plt, sol_plt, traj_plt, xyz_plt)
        else:
            ang_wander_arr_2, rad_wander_arr_2, z_wander_arr_2 = var_mass_calc(G, R, asteroid,
jupiter, sun, mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr, ang_vel_arr, rad_vel_arr,
z_vel_arr, max_step, N, tmax, tol, y_excl_rng, L4_L5, ham_info, orbit_info, warning_supp,
conditions_no, mass_max, mass_min, ang_disp_max, ang_disp_min, rad_disp_max,
rad_disp_min, z_disp_max, z_disp_min, ang_vel_max, ang_vel_min, rad_vel_max,
rad_vel_min, z_vel_max, z_vel_min, vary_mass, vary_ang_disp, vary_rad_disp, vary_z_disp,
vary_ang_vel, vary_rad_vel, vary_z_vel, ang_wander_plt, rad_wander_plt, z_wander_plt,
ham_plt, rot_plt, sol_plt, traj_plt, xyz_plt)

        #Calculate average wanders
        av_ang_wander_arr = average(ang_wander_arr, ang_wander_arr_2)
        av_rad_wander_arr = average(rad_wander_arr, rad_wander_arr_2)
        av_z_wander_arr = average(z_wander_arr, z_wander_arr_2)

        #Reflect variables back for plotting
        ang_disp_arr = np.multiply(ang_disp_arr, -1)
        ang_vel_arr = np.multiply(ang_vel_arr, -1)
        L4_L5 = not L4_L5

        #Plotting functions for given variable of conditions for chosen L point
        if (ang_wander_plt == 1 or rad_wander_plt == 1 or z_wander_plt == 1) and conditions_no
> 1:
            plot_var_cond(mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr, ang_vel_arr,
rad_vel_arr, z_vel_arr, ang_wander_arr_2, rad_wander_arr_2, z_wander_arr_2, ang_wander_plt,
rad_wander_plt, z_wander_plt, vary_mass, vary_ang_disp, vary_rad_disp, vary_z_disp,
vary_ang_vel, vary_rad_vel, vary_z_vel, not L4_L5)
            plot_var_cond(mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr, ang_vel_arr,
rad_vel_arr, z_vel_arr, av_ang_wander_arr, av_rad_wander_arr, av_z_wander_arr,
```

```
ang_wander_plt, rad_wander_plt, z_wander_plt, vary_mass, vary_ang_disp, vary_rad_disp,
vary_z_disp, vary_ang_vel, vary_rad_vel, vary_z_vel, 2)


    #Plotting functions for given variable of conditions for chosen L point
    if (ang_wander_plt == 1 or rad_wander_plt == 1 or z_wander_plt == 1) and conditions_no >
1:
        plot_var_cond(mass_arr, ang_disp_arr, rad_disp_arr, z_disp_arr, ang_vel_arr, rad_vel_arr,
z_vel_arr, ang_wander_arr, rad_wander_arr, z_wander_arr, ang_wander_plt, rad_wander_plt,
z_wander_plt, vary_mass, vary_ang_disp, vary_rad_disp, vary_z_disp, vary_ang_vel,
vary_rad_vel, vary_z_vel, L4_L5)

    #Plot mass variation related to the radial and angular wander (b/a for small orbits)
    if mass_wander_plt == 1:
        plot_mass_wander(mass_arr, rad_wander_arr, ang_wander_arr)

    return 0

#Defines constants, controls and limits of parameters and calls function to implement all other
functions
def main():

    #Constants:
    G = np.float64(4 * pi**2)   #gravitational constant (4 * PI**2)
    R = np.float64(5.2)    #radius of Jupiter's orbit (AU) (5.2 AU)
    M_sol = np.int(1)  #mass of sun (M_sol)

    #Integration control
    tmax = np.float64(1 * 10**3)  #maximum time (years)
    N = np.int(tmax * 500)  #maximum number of time steps
    tol = np.float64(10**-6)  #tolerance of integrator (rtol = atol = tol)
    max_step = 0.0  #maximum step size of integrator
    y_excl_rng = 5e-2  #range of y values (+ to -) where integration stopped when x is positive

    #Misc. controls:
    L4_L5 = np.bool(0)  #Start relative to L4 (=0) or L5 (=1)
    L_point_av = np.bool(0)  #Average wander over both L4 and L5 (=1)
    ham_info = np.bool(0)  #Print Hamiltonian variation (=1)
    orbit_info = np.bool(0)  #Print number of orbits and timer period (=1)
    warning_supp = np.bool(0)  #Supress warnings

    #Plots for each soltuion. Set equal to 1 to plot:
    ham_plt = np.bool(0)  #Plot Hamiltonian time evolution, a constant of motion
    rot_plt = np.bool(0)  #Calculate and plot coordinates in inertial frame, rarely needed
    sol_plt = np.bool(0)  #Plot whole system view
    traj_plt = np.bool(0)  #Plot trajectory
```

```
    xyz_plt = np.bool(0)  #Plot in 3D

    #Plots for many solutions. Set equal to 1 to plot:
    ang_wander_plt = np.bool(1)     #Plot angular wander as a function of a variable (=1)
    rad_wander_plt = np.bool(1)     #Plot radial wander as a function of a variable (=1)
    z_wander_plt = np.bool(0)   #Plot z wander as a function of a variable (=1)
    mass_wander_plt = np.bool(1)  #Plots comparing radial wander/angle angular wander and
mass

    #Initial conditions:
    conditions_no = np.int(100)   #Number of sets of initial conditions

    #Note: The following variables are listed in order of priority
    #If more than one variable changes, the highest listed variation will be plotted (if any)
    #If averaged over L points, variables defined wrt point specified by L4_L5, then changed to be
symmetric for second L point
    #Any variation defined is taken into account in initial conditions, even if not plotted

    mass_min = np.float64(0.0001)   #Min ass of planet in M_sol (Jupiter = 0.001)
    mass_max = np.float64(0.001)   #Max mass of planet in M_sol (Jupiter = 0.001)

    ang_disp_min = np.float64(0)    #Min anticlockwise arc displacement in AU e.g. -0.5
    ang_disp_max = np.float64(0)  #Max anticlockwise arc displacement in AU e.g. 0.5

    rad_disp_min = np.float64(0.0025)    #Min radial displacement in AU, e.g. -0.05
    rad_disp_max = np.float64(0.0025)  #Max displacement displacement in AU, e.g. 0.05

    z_disp_min = np.float64(0.0)    #Min displacement in the z directionin AU
    z_disp_max = np.float64(0.0)  #Min displacement in the z directionin AU

    ang_vel_min = np.float64(0.0)    #Min anticlockwise angular velocity in AU/yr, e.g.-0.05
    ang_vel_max = np.float64(0.0)  #Max anticlockwise angular velocity in AU/yr, e.g. 0.05

    rad_vel_min = np.float64(0.0)    #Min radial velocity in AU/yr, e.g. -0.05
    rad_vel_max = np.float64(0.0)  #Max radial velocity in AU/yr, e.g. 0.05

    z_vel_min = np.float64(0.0)    #Min velocity in the z directionin AU/yr
    z_vel_max = np.float64(0.0)  #Min velocity in the z directionin AU/yr

    #Use initial conditions to find solutions and plot relevant results
    tot_wander(G, M_sol, R, max_step, N, tmax, tol, y_excl_rng, L4_L5, L_point_av, ham_info,
orbit_info, warning_supp, conditions_no, mass_max, mass_min, ang_disp_max, ang_disp_min,
rad_disp_max, rad_disp_min, z_disp_max, z_disp_min, ang_vel_max, ang_vel_min,
rad_vel_max, rad_vel_min, z_vel_max, z_vel_min, ang_wander_plt, rad_wander_plt,
z_wander_plt, ham_plt, mass_wander_plt, rot_plt, sol_plt, traj_plt, xyz_plt)
```

```
    #L_rad = 5.197404544480642

    return 0

#Time total run
t_init = time.time()
main()
t_final = time.time()
print("Total time = ", t_final - t_init)
```