Project B:
Using the Runge-Kutta Method to Investigate the Stability of Jupiter's Lagrange Points

December 4, 2019

Word count = 2989

Abstract

In the orbit of Jupiter there are two stable Lagrange points, L_4 and L_5, at $\pm\pi/3$ radians to the Sun-Jupiter axis. At these points there are two separate clusters of asteroids named the Greeks and the Trojans. By treating the Sun and Jupiter as point masses and using Newton's law of gravity, the Runge-Kutta method was used to solve the coupled odes for the motion of the bodies numerically. The programme demonstrated, in the absence of perturbations, the asteroids stay fixed at the Lagrange points to within 10^{-12} AU; and that, for small perturbations, the asteroids oscillate about the Lagrange points. The wander was derived quantitatively to vary linearly with small perturbations, and the constant of proportionality was found semi-empirically to depend on the masses of Jupiter and the Sun. Different perturbation types had different formulae for the constant of proportionality.

Contents

1 Introduction

The task was to investigate the stability of L_4 and L_5 in Jupiter's orbit. This consisted of:

1. Demonstrating the Lagrange points are stable.
2. Demonstrating the asteroids will oscillate about these points under small perturbations.
3. Deriving quantitative measures for how far the orbits wander[1] from the Lagrange points.
4. Deriving a relationship between the mass of Jupiter and the range of wander.
5. Deriving a relationship between the mass of the Sun and the range of wander.

Rescaled units were used to avoid excessively small/large values. Unit distance = 1 AU, unit time = 1 Earth year, unit mass = the mass of the Sun. Four types of perturbation were defined: position or velocity in the radial or tangential direction.

A programme to solve the problem was written in python3. Treating the Sun and Jupiter as point masses, Newton's law of gravity was used to find coupled ordinary differential equations for the motion. These were solved numerically using the Runge-Kutta method (from the SciPy library), for its accuracy and speed. By plotting the wander as a function of time, it was demonstrated that the asteroids stay fixed at the Lagrange points with deviations on the order of 10^{-12} AU. By looking at the wander and animations[2] of the motion, the asteroids were shown to oscillate about the Lagrange points under small position/velocity perturbations. A first order, linear relationship between the wander and perturbation strength was derived for each perturbation type, using linear regression from the SciPy library. Rather arbitrary relationships for the range of wander and the masses of Jupiter and the Sun were also derived, using a combination of trial functions and linear regression, comparing the actual range of wander to the proposed relationship. The programme was also able to investigate two perturbations at once and plot contours and surfaces of the wander.

The performance of the programme vastly depended on the task it was completing. To look at specific motion it was fast, finishing within a few seconds. However, when quantifying wander, the motion had to be generated n times, so time scaled with n. This then scaled with n^2 when generating the wander for two perturbations, and in general for k independent variables, the run time scaled with n^k.

The remainder of this report consists of the theoretical analysis in section 2; the implementation and performance in section 3; testing the programme in section 4 and quantifying the wander in section 5.

2 Theoretical Analysis

Within the three body problem, in the absence of perturbations, a Lagrange point, star and planet system will rotate in a plane around their centre of mass; at constant angular frequency[3] ω and maintain their relative positions[1]. Therefore all positions and velocities were given as 2-component vectors [x,y] with respect to the centre of mass of the system.

[1] Wander was assumed to mean the absolute distance between the asteroids and the Lagrange point.
[2] See pidgeon hole.
[3] Provided the Sun's mass is greater than 25 times the mass of planet.

When there is a small perturbation of the asteroid, it will oscillate around $L_{4/5}$. This provides two tests of the code: in the absence of perturbations asteroids should stay fixed at Lagrange points and in the presence of small perturbations they should oscillate about them.

2.1 Shifting to the COM Frame

All distances and angles were quoted in the Sun's frame, they needed to be calculated in the COM frame.

The masses of the Greeks and Trojans can be treated as negligible[2], therefore, the angular frequency of the system[4] is:

$$\omega = \sqrt{\frac{G(m_{Sun} + m_{Jup})}{a^3}} \tag{1}$$

Where a is the average distance between Jupiter and the Sun. Initially the Sun and Jupiter were on the x axis as in figure 1. The angles and positions of the asteroids were found using trigonometry.

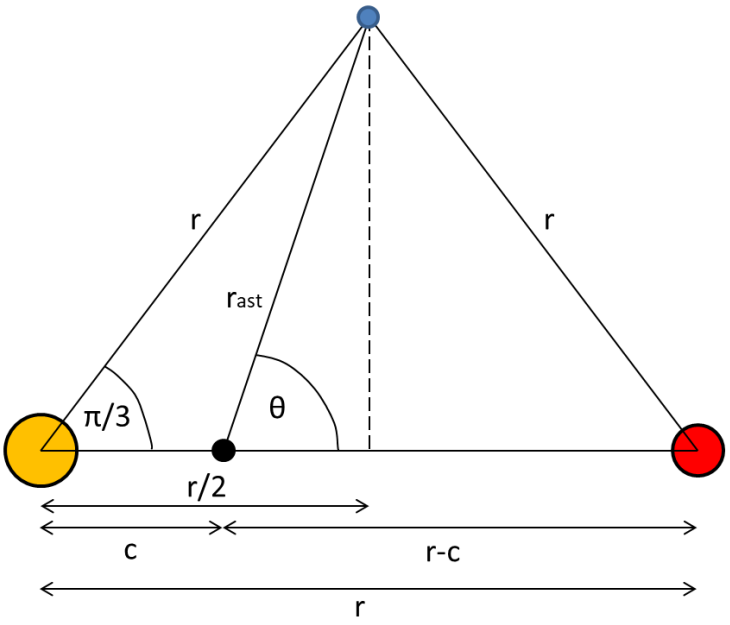

Figure 1: Diagram NOT TO SCALE, of the initial system, the COM is the black dot at c, clearly neither θ nor r_{ast} are equal to $\pi/3$ or r respectively. They are found using trigonometry.

By equating the two expressions for the height of the triangle, θ can be found:

$$\tan\theta = \frac{r\sin\pi/3}{r/2 - c} \tag{2}$$

Using the sine rule, r_{ast} can be calculated:

$$r_{ast} = r\frac{\sin(\pi/3)}{\sin\theta} \tag{3}$$

[4]In the COM frame.

With the COM at the centre, the initial conditions[5] are calculated for body i:

$$\mathbf{r_i} = r_i \begin{pmatrix} \cos\theta_i \\ sin\theta_i \end{pmatrix} \tag{4}$$

$$\mathbf{v_i} = \omega r_i \begin{pmatrix} -\sin\theta_i \\ cos\theta_i \end{pmatrix} \tag{5}$$

The specific initial radii and angles are below:

Table 1: Table of initial radii and angles for all bodies in the COM frame.

Body	r (AU)	θ (rad)
Sun	0.005194805194805196	π
Jupiter	5.194805194805195	0
Greeks	5.197404544480642	1.0480631435878964
Trojans	5.197404544480642	-1.0480631435878964

2.2 Isotropy of Perturbations

The acceleration for all bodies is radially inwards, towards the COM. Therefore the direction of a perturbation affects the subsequent motion. It is expected that for velocity perturbations, tangential have a greater effect on the wander than radial; due to shifting the asteroids to a point of maximum/minimum velocity or to a semi-latus rectum. These effects are shown in figures 2 and 3. Similarly radial and tangential position perturbations will affect the wander to different degrees. The wander is expected to depend on the direction of perturbation, justifying having four different types.

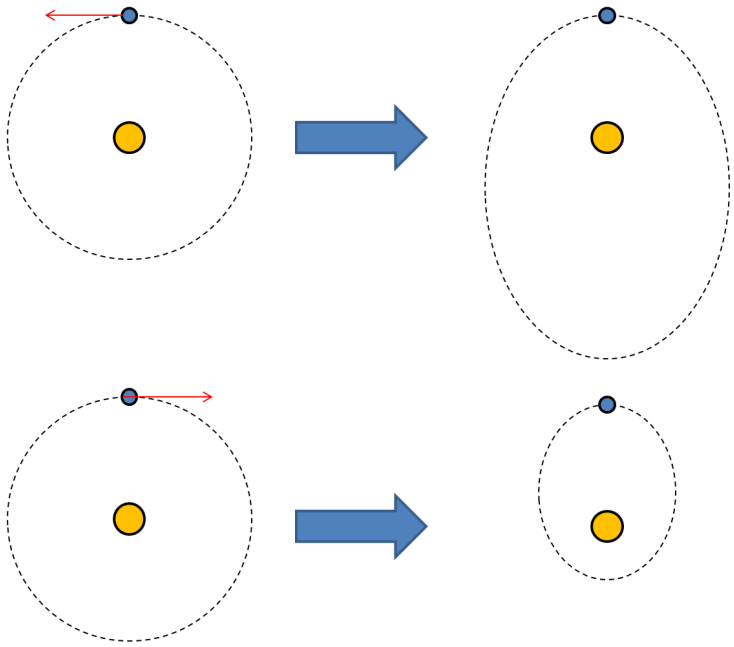

Figure 2: Schematic NOT TO SCALE, of the effects of a tangential velocity perturbation/boost, for a counter-clockwise orbit.

[5]The initial conditions have all been calculated assuming initially circular orbits.

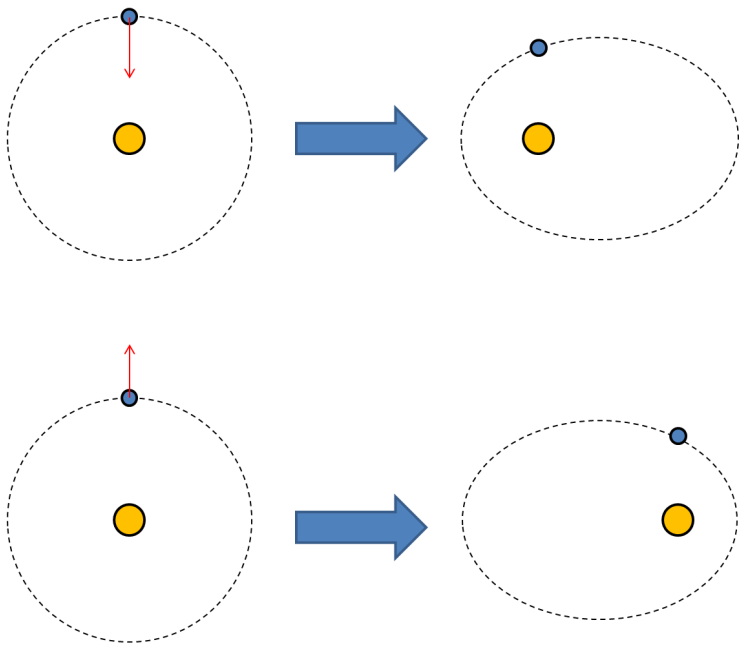

Figure 3: Schematic NOT TO SCALE, of the effects of a radial velocity perturbation/-boost, for a counter-clockwise orbit.

Additionally this provides another test; by setting m_{Jup} to 0, and introducing tangential/radial velocity perturbations, the subsequent Greeks' orbits are expected to shift according to figures 2 and 3.

2.3 The Wander

Due to the anisotropy of the perturbations there will be a radial/tangential dependence. Trivially there is also a position/velocity dependence. Hence the four separate types of perturbation were investigated. The form of the wander is (expanding as a Taylor Series):

$$W = \sum_{k=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=0}^{\infty}\sum_{n=0}^{\infty} C_{k,l,m,n}|\Delta r_{rad}|^k|\Delta r_{tan}|^l|\Delta v_{rad}|^m|\Delta v_{tan}|^n \tag{6}$$

When there are no perturbations the wander vanishes, therefore, $C_{0,0,0,0} = 0$. In the limit of small perturbations the second order terms and higher vanish, leading to the new form of the wander[6]:

$$W \approx A|\Delta r_{rad}| + B|\Delta r_{tan}| + C|\Delta v_{rad}| + D|\Delta v_{tan}| \tag{7}$$

Where $A = C_{1,0,0,0}$, $B = C_{0,1,0,0}$, $C = C_{0,0,1,0}$ and $D = C_{0,0,0,1}$. There will also be mass dependence; this will not be a new term in the equation, because the mass can not be "perturbed" in this system. However, the range of wander will change if the masses are different. The mass dependence is present in the constants A, B, C and D; these constants will depend on both m_{Jup} and m_{Sun}.

Note: the above equation and analysis is only a hypothesis for the programme to test.

[6]This assumes that the perturbations are small enough so that they depend on the absolute value only. E.g. a radial position perturbation outwards would have different effects to one inwards. However, these effects can be assumed to be negligible for small enough perturbations. See Appendix A.

2.4 Symmetries of the System

There are multiple symmetries that were exploited to simplify the problem:

Clockwise/counter-clockwise: The initial conditions have been set up so that the orbit is counter-clockwise. However, the exact same phenomena will occur if observing from "above" or "below", and so it doesn't matter which direction the orbit is in.

Time with rotation: In the absence of perturbations, at t = τ, the system will be identical to the system at t = 0, but will be rotated through an angle $\omega\tau$. Due to the choice of coordinate angle being arbitrary, a perturbation introduced at t = 0, will produce the same motion as a rotated perturbation introduced at t = τ.

Greeks and Trojans: There is a mirror flip in the Sun-Jupiter axis, between the Greeks and Trojans. The result of this symmetry is that any effects perturbations of the Greeks have, would lead to the same effects on the Trojans; provided there was a sufficient reflection of coordinates.

The combination of these symmetries allows the problem to be simplified. Such that only the Greeks, orbiting counter-clockwise, with perturbations at t = 0, need to be investigated.

3 Implementation and Performance

3.1 Implementation

The problem consisted of eight coupled first order vector differential equations. One for the position and velocity of each body in the problem (the Sun, Jupiter, Greeks and Trojans). These were solved using the Runge-Kutta method from the SciPy library; which, written as a function, created the arrays of all bodies' positions and velocities, and the Greeks' theoretical position/velocity. This was the foundation of the programme, with the rest of the methods being built around this. The programme was written so that the main programme defines all of the functions to be used, all of which can be used for specific tasks. Then the ones that are needed are entered at the bottom of the programme and run in the terminal. It is important to note that the programme and functions created were made for this task, and would not be used for a general gravitational problem.

The theoretical position of L_4 was found by taking the position of Jupiter, rotating it through θ and rescaling by the ratio of their radii:

$$\mathbf{r_{Gre}} = \frac{r_{Gre}}{r_{Jup}} \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \mathbf{r_{Jup}} \tag{8}$$

The theoretical velocity was calculated by rotating the position vector 90° and multiplying by ω:

$$\mathbf{v_{Gre}} = \omega \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \mathbf{r_{Gre}} \tag{9}$$

The other key task was finding the wander. Each time the range of wander was found, the programme ran the entire motion of all four bodies, and the Greeks' theoretical position/velocity for 500 years. The output was then the maximum difference between theoretical and actual position. This was written in a function, which took perturbations as its input, ran the motion and output the wander. This was called in various other functions multiple times, to derive relationships between the strength of perturbation and range of wander.

3.2 Performance

To look at one set of the bodies' motion, the programme performed well. Running in typically less than 5 seconds. However when deriving mass relationships, it slowed down significantly, taking on the order of minutes to run. This is due to the method. The range of wander for specific conditions is found in t seconds. This is then repeated m times to get a slope. And then looking at n different masses, n slopes need to be calculated. Run time is approximately tmn, plus negligible time it takes to plot the data and run the statistical analysis. Looking at multiple perturbations in a contour or surface plot has the same scaling problem. When looking at m points for perturbation 1 and n points for perturbation 2, run time is approximately tmn. For each new independent variable, the run time multiplies by the number of points.

3.3 Improvements

The majority of the run time was in the multiple perturbation plots and mass variations where run time was approximately tmn. The time could be reduced by minimising t. The majority of the computation was in calculating the motion of all four bodies. It was stated that the masses of the asteroids are negligible[2], so they cannot affect each other. Therefore the Trojan's motion was being calculated unnecessarily, calculating the motion for three bodies and not four would reduce the computation to 75%. Similarly the theoretical velocity of the Greeks is being found unnecessarily, since it is not used in the wander. Another possible solution would be to change the function that finds the wander; to run for 250 years rather than 500. This would reduce the computation to 50% at risk of compromising the accuracy. Another way to speed up the programme would be to save generated data as a .npy file. This would allow statistical analysis and plotting figures to be completed without generating the data each time.

4 Testing

4.1 Stability of Lagrange Points

The differences between the theoretical and actual arrays of the Greeks' position and velocity were taken. Plots and animations[7] were made of the motion and the deviation to determine whether the asteroids were still bound to L_4. This was run for 5000 years with 50000 points, approximately 420 orbits. The results are plotted in figures 4 and 5. Looking at the figures and watching the animations it was evident the Greeks are fixed at L_4 in agreement with theory.

[7] See pigeon hole.

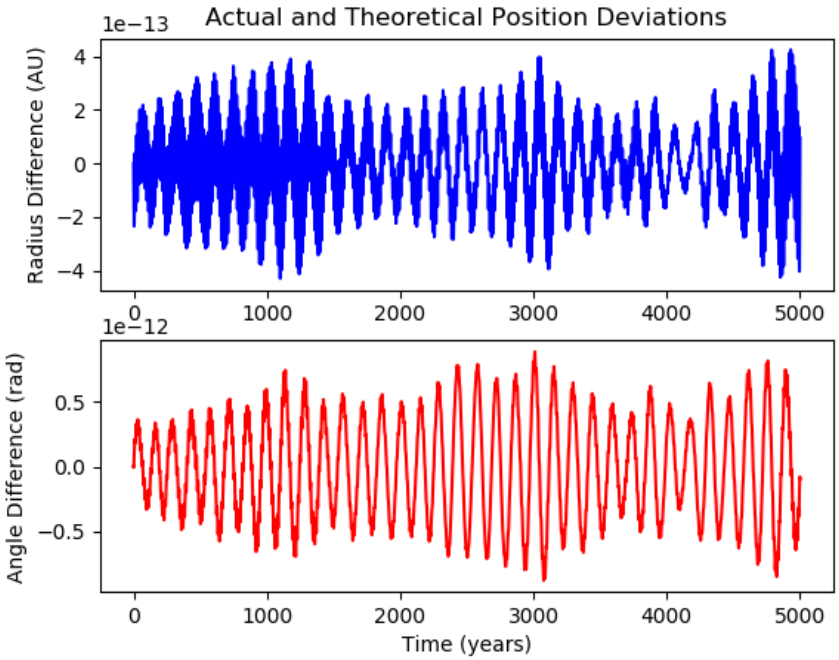

Figure 4: Radial and angular deviations against time, radial deviations are on the order of 10^{-13} AU and angular deviations on the order of 10^{-12} radians.

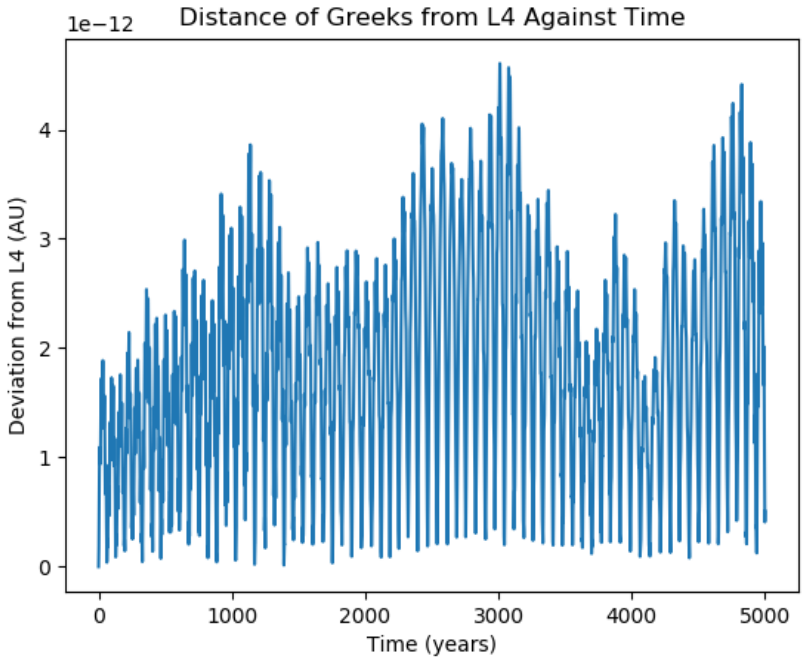

Figure 5: Absolute deviation from L_4 against time. The deviations are on the order of 10^{-12} AU, confirming that the Greeks are fixed at L_4.

4.2 Small Perturbations

Theoretically, for small perturbations the asteroids will oscillate around the Lagrange points; the Trojans can deviate by approximately 20° from L_5[1]. This was tested over 5000 years, the plots and animations[8] demonstrated this to be the case. This is shown for a specific perturbation: tangential position of strength 0.05 AU, in figures 6, 7 and 8.

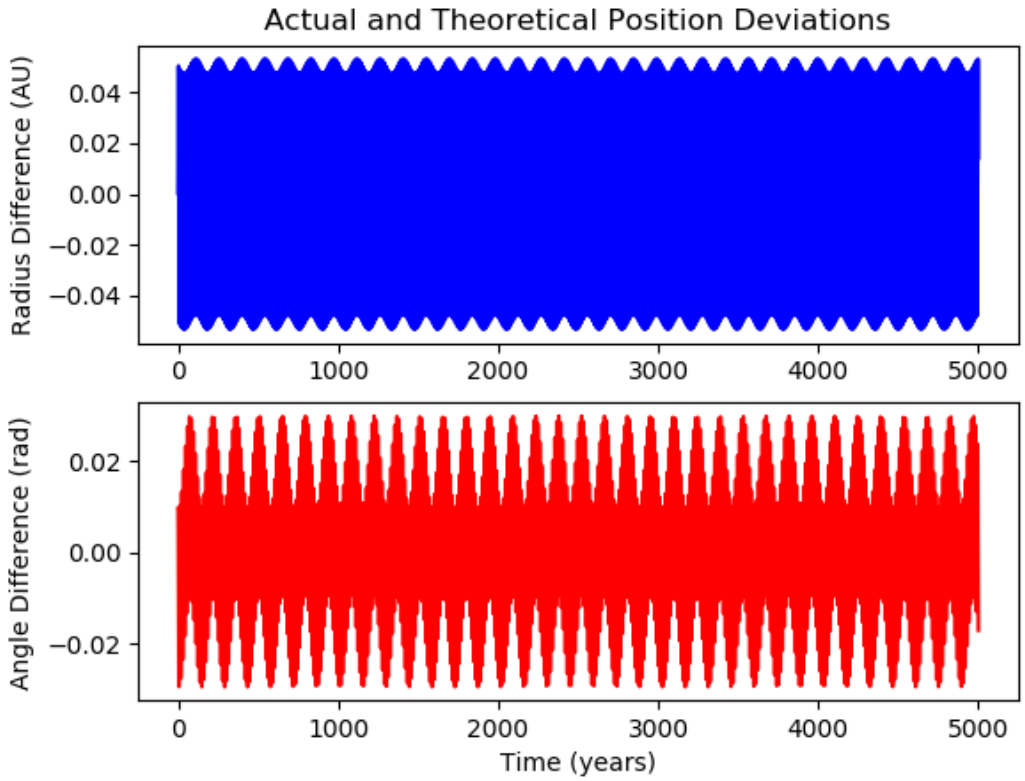

Figure 6: The difference in radius and angle between the Greeks and L_4, after a tangential position perturbation 0.05 AU over 5000 years. The deviations are very small and oscillate about zero, showing the Greeks oscillate about the Lagrange point under small perturbations.

[8]See pigeon hole.

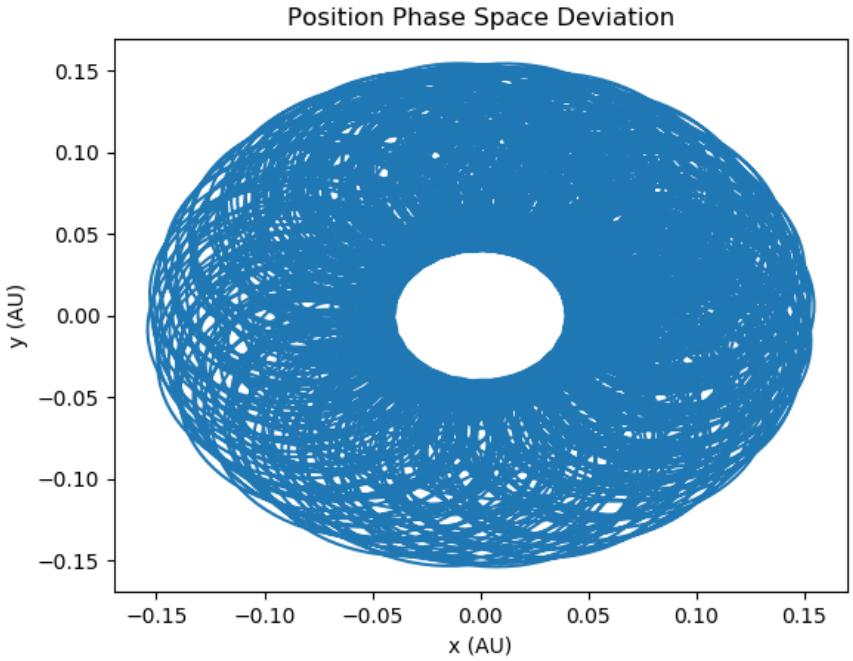

Figure 7: The wander in position phase space, after a tangential position perturbation 0.05 AU over 5000 years. The deviations are very small at 0.15 AU, showing the Greeks oscillate about the Lagrange point.

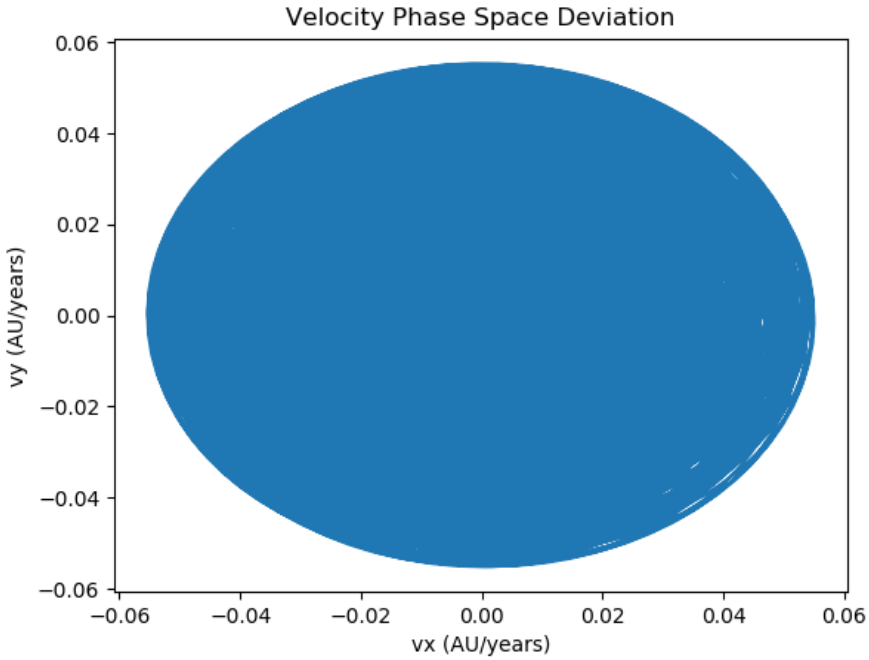

Figure 8: The wander in velocity phase space, after a tangential position perturbation 0.05 AU over 5000 years. The deviations are very small at 0.06 AU/years, showing the Greeks oscillate about the Lagrange point.

4.3 Energy Conservation

The energy of Jupiter is expected to be conserved and negative[9]. The energy was investigated in the Sun's frame by using a Galilean transformation[10]. The plot of its energy over 5000 years can be seen in figure 9. The energy behaves as expected, staying approximately constant (only 0.018% deviation over 420 orbits) and negative.

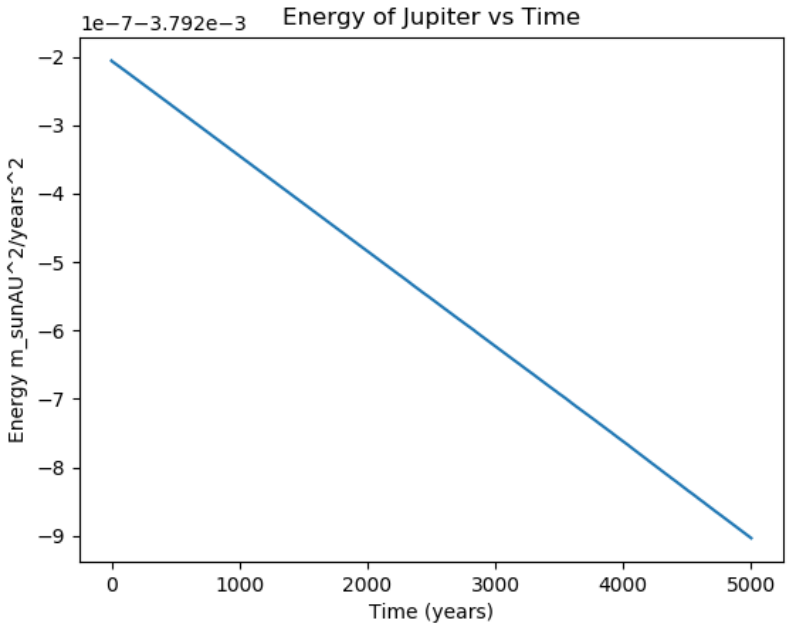

Figure 9: The energy of Jupiter in the Sun's frame over 5000 years. Over 420 orbits it has only lost 0.018% of its energy as expected. The energy is also negative as expected.

4.4 Velocity Boosts

The mass of Jupiter was set to 0 and velocity boosts of strength 0.6 AU/years were introduced in the tangential and radial directions. The results are presented in figure 10 and are in good agreement with theory.

[9] In order to remain gravitationally bound to the sun.
[10] Assuming non-relativistic speeds or gravitational fields.

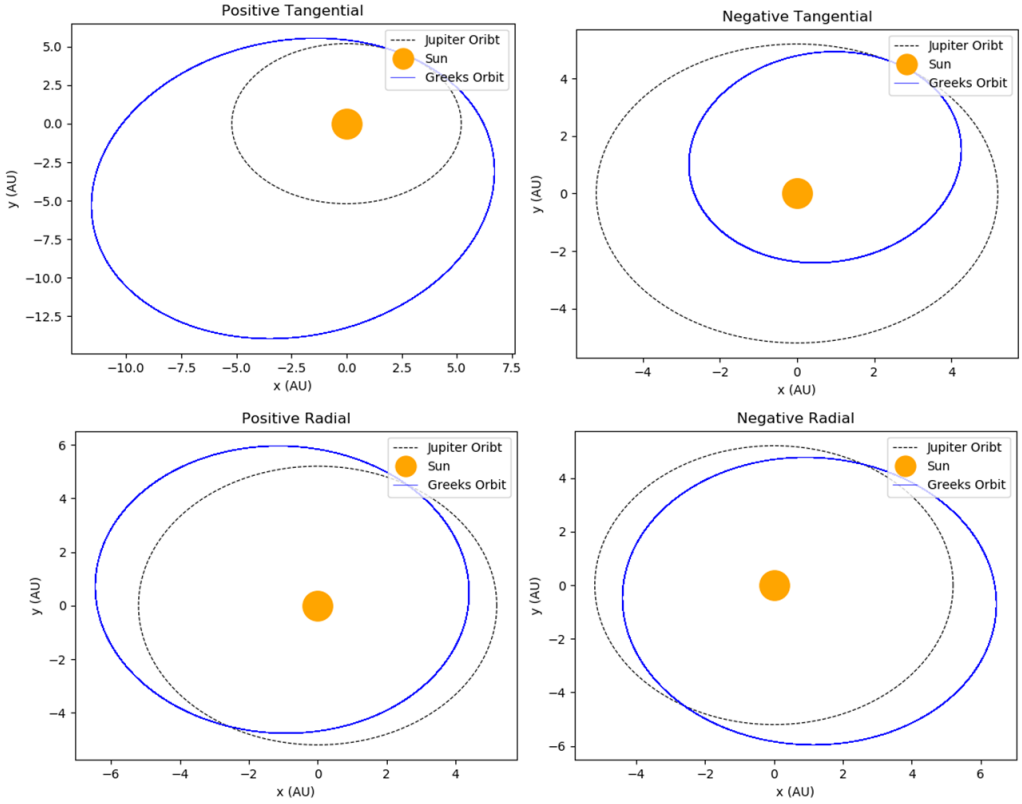

Figure 10: The subsequent motion of the Greeks for 5000 years after different velocity boosts. These are in good agreement with figures 2 and 3.

5 Quantifying Wander

The hypothesised form of the wander is given in eq. 7. This has been investigated by looking at individual and mixed perturbations.

5.1 Individual Perturbations

A function was made that collected data for how the wander is affected by each perturbation type. It then runs a linear regression, returning the slope, the intercept and the R^2 value. It was also able to add constant perturbations to determine whether the intercept and slope would be affected. All four perturbation types, of strength -0.001 to 0.001 AU or AU/years were investigated and the results are below:

Table 2: Table of results from the linear regression on the wander vs individual perturbations, slopes and intercepts given to 3 d.p.

Perturbation	Slope	Intercept	R^2
Radial Position	38.829	8.471e-08	0.9999805271073561
Tangential Position	3.029	-2.364e-08	0.9999998888055677
Radial Velocity	7.624	-3.903e-09	0.9999987075699418
Tangential Velocity	77.358	-3.871e-06	0.9999235465956227

The wander for tangential velocity perturbations is plotted in figure 11. As hypothesised, and evident from the graph and table, for small enough perturbations the wander obeys eq. 7 with a linear relationship and no intercept. The R^2 values support this hypothesis all of which are very close to one.

Additionally figures 11, 12, 13 and 14 are all invariant to reflections in the axes; which justifies using the absolute value of each perturbation type in eq. 7. It is shown graphically in Appendix A, that for large perturbations the linear approximation stops working, and that the absolute value of a perturbation can't be used.

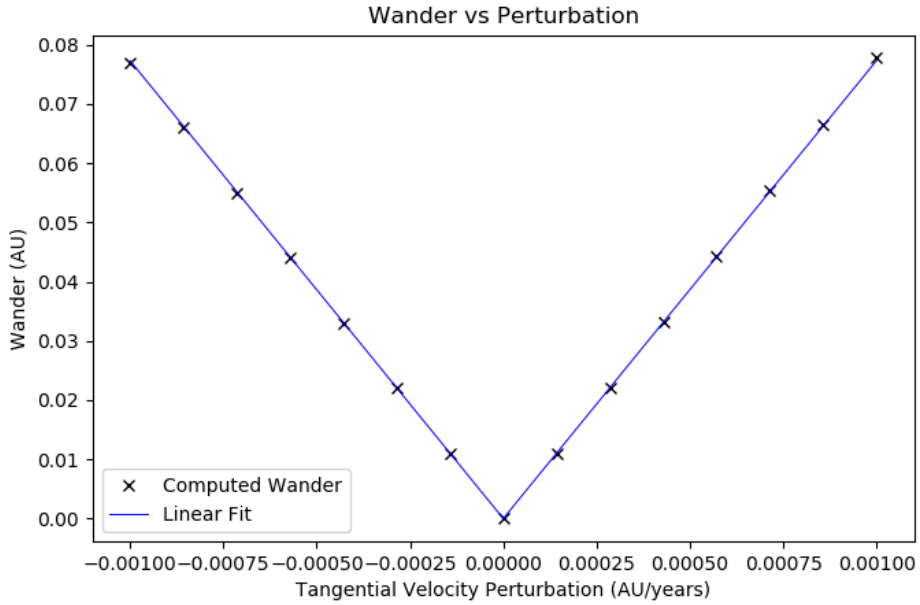

Figure 11: This figure shows the wander against tangential velocity perturbations. As hypothesised this is a linear relationship with no intercept. The graphs for the other three perturbations are linear relationships as well, but have different constants of proportionality.

5.2 Mixed Perturbations

In the limit of small perturbations it was hypothesised in eq. 7 that when a second perturbation was added, that the wander would be a linear sum of them. Mixed perturbations were looked at and the radial and tangential positions are plotted in figures 12 and 13. For small enough perturbations eq. 7 does describe the wander. However the linear relationship is starting to fail, higher order terms from the Taylor expansion start to become significant. A slice of the surface plot in figure 13 has been taken at tangential position perturbation = 0.0006 AU. This is shown in figure 14, the linear approximation works reasonably well, but the quadratic terms are clearly becoming significant in agreement with eq. 6.

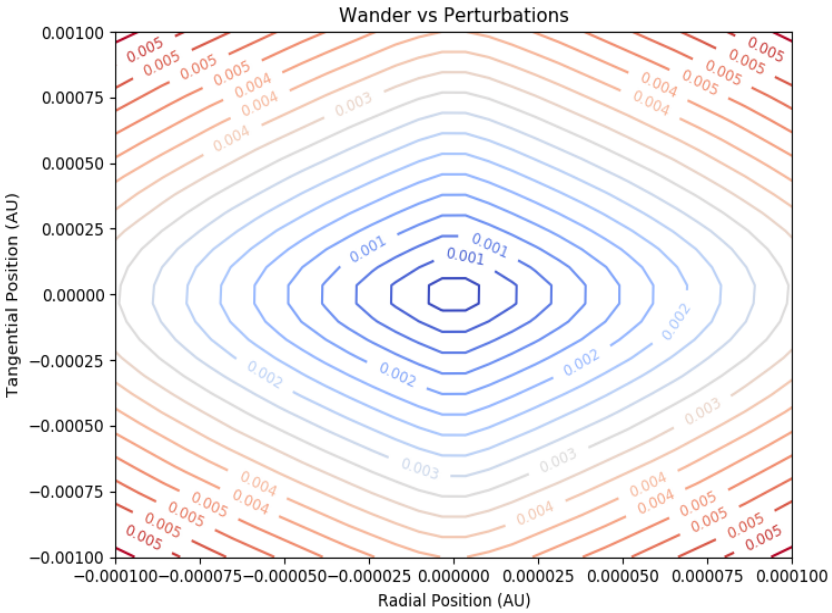

Figure 12: The contour plot of wander vs small position perturbations. The contours of constant wander are expected to be diamonds, if eq. 7 is correct. The contours are approximately diamonds, so the wander is approximately a linear sum of the perturbations as hypothesised. However, higher order terms from eq. 6 are required to describe the wander more accurately.

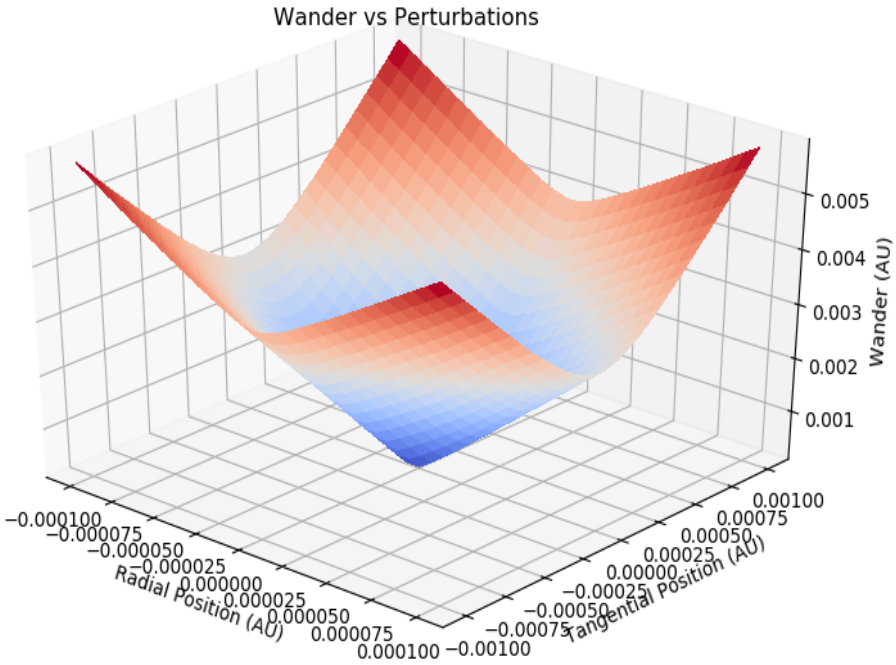

Figure 13: The surface plot of wander vs small position perturbations. It is approximately a sum of the linear perturbations. However, the effects of higher order terms are beginning to become significant.

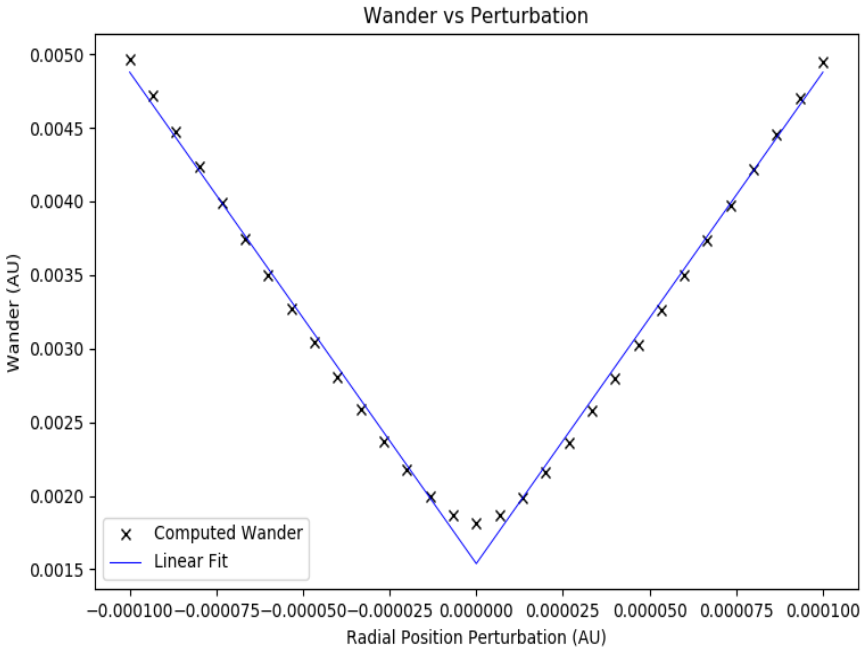

Figure 14: The plot of wander vs radial position perturbations, with a constant tangential position perturbation 0.0006 Au. The linear approximation works quite well, however, the second order terms are clearly becoming significant.

5.3 Varying Mass

The mass dependence was looked at by varying both m_{Jup} and m_{Sun}, by an order of magnitude in each direction. Then looking at each of the four perturbation types individually, and how the mass affected the slopes and intercepts of the wander. The intercepts were unaffected and still approximately zero. The slopes were found to vary differently, which would be expected due to the anisotropy of the system. The slopes approximately obey the below equations (all numbers given to 3 d.p.). In the case of D, this was accurate for masses varying between 0.7 and 10 times the original only.

$$A \approx 1.850 \left(\frac{m_{Sun}}{m_{Jup}} \right)^{0.449} \tag{10}$$

$$B \approx 44.363 \left(\frac{m_{Jup}}{m_{Sun}} \right) + 2.977 \tag{11}$$

$$C \approx \frac{119.729 m_{Jup} + 7.492}{{m_{Sun}}^{0.524}} \tag{12}$$

$$D \approx \frac{0.325({m_{Sun}}^{-1.187} + 12.927)}{{m_{Jup}}^{0.406}} \tag{13}$$

These were only semi-empirical estimates, and there is not a clear theoretical reason why the constants would vary in this way. The relationships were tested by subbing in $m_{Jup} = 0.001$ solar masses, or $m_{Sun} = 1$ solar mass as constant. Then running a linear

regression on computed slopes against the proposed relationship[11] for 20 masses. The expected result was y = x. The results are in the two tables below, and plotted for B in figure 15:

Table 3: Linear Regression on proposed relationship vs actual data, where m_{Jup} is independent and m_{Sun} = 1 solar mass.

Constant	Linear Relationship	R^2
A	y = 1.02x - 1.07	0.99767
B	y = 1.00x + 1.20e-03	0.99092
C	y = 1.00x - 2.11e-03	0.99210
D	y = 0.99x + 3.03	0.99354

Table 4: Linear Regression on proposed relationship vs actual data, where m_{Sun} is independent and m_{Jup} = 0.001 solar masses.

Constant	Linear Relationship	R^2
A	y = 1.05x - 3.81	0.99843
B	y = 1.05x - 0.13	0.93823
C	y = 1.08x - 0.24	0.99860
D	y = 1.03x + 0.48	0.93834

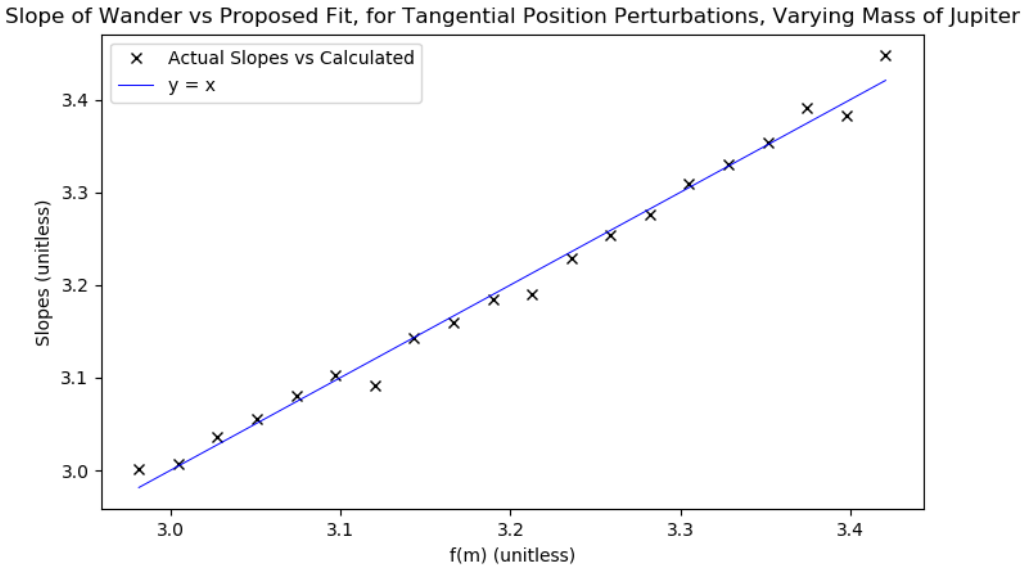

Figure 15: A plot of the actual slopes vs the proposed function, for tangential position perturbations, varying the mass of Jupiter. The blue line is y = x, which the data should lie on. Here the data fits quite well to the line, showing B approximately obeys eq. 11.

The results from the regressions aren't perfect, however they are reasonable estimates for this complex system. A Taylor series could have also been derived using numpy.polyfit.

[11] Which depended on the independent mass.

It would have likely been more accurate, however it was unclear how many terms to test, and how quickly the series would converge.

6 Conclusion

The problem at hand was to write a programme to investigate the orbit of the Greek and Trojan asteroids, and how they responded to perturbations from the Lagrange points. Newton's law of gravity, treating the Sun and Jupiter as point masses, was used, to generate eight coupled ordinary differential equations for the motion of the four bodies. Then the Runge-Kutta method, with adaptive stepping, was used to solve them. Looking at specific motion this method worked well (for 5000 years it took on the order of 5 seconds to complete). When looking at multiple scenarios, (e.g. when scaling the perturbation linearly to derive a relationship), the programme ran a lot slower. One of the key ways to fix this would be to remove the Trojans from the problem reducing the computation to 75%. Additionally rather than looking over 500 years to find the range of wander, the programme could have run over 250 years, decreasing the computation to 50% but possibly affecting the accuracy. The stability of the Lagrange points was confirmed to an accuracy of 10^{-12} AU, and that the asteroids oscillate about the Lagrange points under small perturbations. It was found that for small perturbations, the wander varied linearly with each of the four types of perturbation (as proposed in eq. 7), and that the constants of proportionality were functions of m_{Jup} and m_{Sun} (described by equations 10, 11, 12 and 13). These functions were only accurate for masses varying by an order of magnitude, which is acceptable, because the Lagrange points are only stable if the Sun is more than 25 times the mass of Jupiter. This programme, given more time and development, has potential to investigate the Lagrange points further. It could run a multivariate analysis, on the types of perturbations, and the masses of Jupiter and the Sun. This would allow the wander to be more accurately determined as perturbations got larger. The coefficients for higher order terms in the Taylor expansion could be determined, also the existing ones, being functions of m_{Sun} and m_{Jup} could be found to higher accuracy.

References

[1] Roy, A.E., 2004. *Orbital motion Fourth Edition.* CRC Press. pp. 117

[2] Buscher, D., 2019. *Part II Computational Physics Projects.* Cambridge University. pp. 7

Appendix A: Non-linear Regime of Wander

Another demonstration of the linear/non-linear behaviour of the wander is below, in figure 16. The wander vs radial velocity perturbations has been plotted, up to ±0.15 AU/years. The linear fit y = 7.624$|x|$ - 3.903e-09 has also been plotted. It is evident that as the perturbation gets larger the wander deviates from the linear regime, requiring more terms from the Taylor expansion. Additionally the +0.15 AU/years perturbation produces a larger wander than the -0.15 AU/years one. This shows that for large enough perturbations the absolute value cannot be used, as hypothesised.

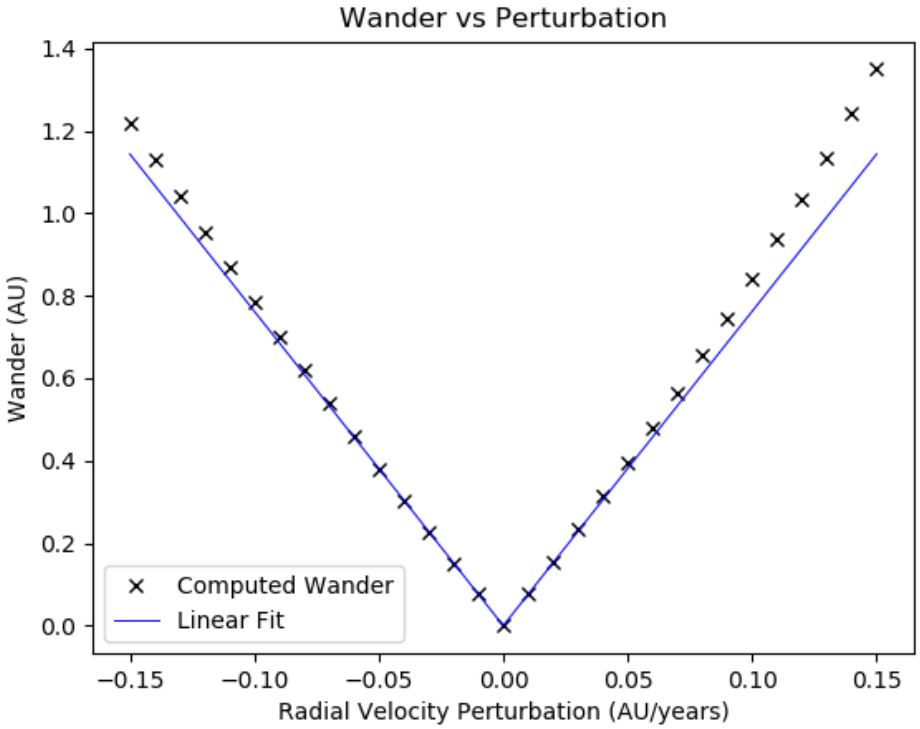

Figure 16: A plot of wander vs radial velocity perturbations (with no other perturbations). For large perturbations the wander deviates from the linear fit, requiring more terms from the Taylor expansion as expected. Also the wander is different for perturbations of +0.15 AU/years and -0.15 AU/years, this demonstrates that the absolute value of the perturbation cannot be used for large enough perturbations, as hypothesised.

Appendix B: Source Code

The source code is below, the commands at the bottom have been included to show slightly perturbed motion over 500 years. As well as the motion in phase space. It will also derive the relationship between wander and radial position perturbations and plot it. Other commands have been commented out, but can be uncommented to see them run. Note that on different devices the programme ran differently/produced slightly different figures. For example the MCS computers could save mp4 animations, but was unable to show them.

```
#Project B, Trojan Asteroids
#Treat Sun and Jupiter as point masses, and use Newton's law
#upiter's orbit is a circle, in a plane, (x,y) needed only
#Units are rescaled: unit mass = Msun, unit time = 1year, unit distance = 1AU
#Mjupiter = 0.001Msun, orbit radius = 5.2AU, G = 4pi^2
#The COM of the system is at the centre

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import FuncAnimation
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

```
import scipy.integrate as integrate
from scipy import stats

#mp4 writers
Writer = animation.writers['ffmpeg']
writer = Writer(fps=25, metadata=dict(artist='Me'), bitrate=1800)


#These are the constants, including masses and initial conditions

#Fundamental constants, masses of the bodies, gravitational constant, radius of
    jupiter's orbit
#Centre of mass of jupiter and sun, angular frequency and velocity of jup
G = 4*((np.pi)**2)
radius = 5.2
masses = [0.001,1]


#Makes the initial conditions after changing one of the masses
def make_parameters(jup_or_sun,m):
        global m_jup, m_sun, w, rad_jup, angle_ast, rad_ast
        global cos, sin
        global z0
        masses[jup_or_sun] = m
        m_jup = masses[0]
        m_sun = masses[1]

        com = radius*m_jup/(m_sun+m_jup)
        w = (G*(m_sun+m_jup)/radius**3)**0.5

        rad_jup = radius - com
        angle_ast = np.arctan((radius*np.sin(np.pi/3))/(radius/2-com))
        rad_ast = radius*np.sin(np.pi/3)/np.sin(angle_ast)
        v_ast = w*(rad_ast)

        #Initial conditions, this can be changed in make_z()
        #The default is the perfect set up, with no perturbations
        cos, sin = np.cos(angle_ast), np.sin(angle_ast)
        rcos, rsin = rad_ast*cos, rad_ast*sin
        vcos, vsin = v_ast*cos, v_ast*sin

        r_sun = np.array([-com,0])
        r_jup = np.array([rad_jup,0])
        r_gre = np.array([rcos,rsin])
        r_tro = np.array([rcos,-rsin])
        v_sun = np.array([0,-w*com])
        v_jup = np.array([0,w*rad_jup])
        v_gre = np.array([-vsin,vcos])
        v_tro = np.array([vsin,vcos])
        z0 = np.concatenate((r_sun,r_jup,r_gre,r_tro,v_sun,v_jup,v_gre,v_tro))
make_parameters(0,0.001)     #The function has been used here to set the default



#These functions are used to solve the coupled odes
```

```
#This function finds the acceleration of a body at position r (2d vector)
#due to a massive body at r_body with mass = mass
#returns the acceleration as an array
def acc(r,mass,r_body):
       r = (np.asarray(r) - np.asarray(r_body))
       a = -G*mass/(np.linalg.norm(r))**(3)
       a = np.full_like(r,a)*r
       return np.asarray(a)


#This function finds all of the derivatives, i.e. 2d velocities
#and accelerations of the 4 moving bodies
#Slices the last 8, and moves to front, i.e. the velocities
#Finds the accelerations by summing the vector quantities
#of acceleration from the sun or Jupiter or both
#Order in the z array: sun,jupiter,greeks,trojans, those four positions then
    their velocities
def derivatives(z,t):
       velocities = z[8:16]
       r_sun, r_jup, r_gre, r_tro = z[0:2], z[2:4], z[4:6], z[6:8]
       a_sun = acc(r_sun,m_jup,r_jup)
       a_jup = acc(r_jup,m_sun,r_sun)
       a_gre = acc(r_gre,m_sun,r_sun)+acc(r_gre,m_jup,r_jup)
       a_tro = acc(r_tro,m_sun,r_sun)+acc(r_tro,m_jup,r_jup)
       return np.concatenate((velocities,a_sun,a_jup,a_gre,a_tro),axis = None)


#This function takes end time, number of points and pertubations and makes the z
    array of all positions of the bodies
#Also creates the theoretical values of the positions of Greeks based on its
    relative position to Jupiter
def make_z(t_end,number_of_points,gre_r_perturb,gre_v_perturb):
       global no_points, t
       global z, jup_r, tro_r
       global theory_gre_r, gre_r, gre_r_dev, mod_gre_r_dev
       global gre_v, gre_v_dev
       no_points = number_of_points
       t = np.linspace(0,t_end,number_of_points)
       z1 = z0 + np.concatenate((np.zeros(4),gre_r_perturb,np.zeros(6),
           gre_v_perturb,np.zeros(2)),axis = None)
       z = integrate.odeint(derivatives,z1,t)
       #The theoretical positions of greeks and trojans is calculated by rotating
            the Jupiter vector the angle and scaling appropriately
       jup_r = np.array([z[:,2],z[:,3]])
       tro_r = np.array([z[:,6],z[:,7]])
       gre_r = np.array([z[:,4],z[:,5]])
       gre_v = np.array([z[:,12],z[:,13]])

       c_array = np.full_like(t,rad_ast*np.cos(angle_ast)/rad_jup)
       s_array = np.full_like(t,rad_ast*np.sin(angle_ast)/rad_jup)

       theory_gre_r = np.array([c_array*jup_r[0]-s_array*jup_r[1],s_array*jup_r
           [0]+c_array*jup_r[1]])
       gre_r_dev = theory_gre_r - gre_r
       mod_gre_r_dev = np.linalg.norm(gre_r_dev,axis = 0)
```

```
        theory_gre_v = np.full_like(theory_gre_r,w)*[-theory_gre_r[1],theory_gre_r
            [0]]
        gre_v_dev = theory_gre_v - gre_v



#This function looks at the energy of Jupiter in the suns frame, is used to test
    how effective the code is
def test_energy():
        sun_r = np.array([z[:,0],z[:,1]])
        sun_v = np.array([z[:,8],z[:,9]])
        jup_v = np.array([z[:,10],z[:,11]])
        r_sun2jup = sun_r - jup_r
        v_jup_sunframe = jup_v-sun_v
        ke = np.array(np.full_like(t,m_jup/2)*np.linalg.norm(v_jup_sunframe,axis =
             0)**2)
        pe = np.array(np.full_like(t,-G*m_sun*m_jup)*1/(np.linalg.norm(r_sun2jup,
            axis = 0)))
        e = ke + pe
        plt.plot(t,e)
        plt.xlabel('Time (years)')
        plt.ylabel('Energy m_sunAU^2/years^2')
        plt.title('Energy of Jupiter vs Time')
        plt.show()



#These functions are used to make animations

#This function is used to watch the animation of the motion of the 3 bodies
    around the sun
#Input is 0, or 1, 0 animates the 4 bodies: sun, jupiter, greeks, trojans
#1 animates the sun, jupiter, greeks and the theoretical position of greeks
#It has nested functions inside, which are only used in the animation
#jan = jupiter animation, gan = greeks animation, tan = trojans animation/
    theoretical greeks animation
def watch_real_animation(deviation):
        fig1, ax = plt.subplots()
        #Makes three moving dots, Jupiter = red, Greeks = green, Trojans/
            Theoretical Greeks = blue/black cross
        if deviation == 0:
                name = 'Trojans'
                point = 'bo'
        else:
                name = 'L4'
                point = 'kx'

        jup_an, = plt.plot([],[],'ro',label = 'Jupiter')
        gre_an, = plt.plot([],[],'go',label = 'Greeks')
        tro_an, = plt.plot([],[],point,label = name)

        def init():
                half_length = radius+1
                ax.set_xlim(-half_length, half_length)
                ax.set_ylim(-half_length, half_length)
```

```
        return jup_an, gre_an, tro_an,

    def update(frame):
        x_j = jup_r[0][frame]
        y_j = jup_r[1][frame]
        jup_an.set_data(x_j,y_j)
        x_g = gre_r[0][frame]
        y_g = gre_r[1][frame]
        gre_an.set_data(x_g, y_g)
        if deviation == 0:
            x_t = tro_r[0][frame]
            y_t = tro_r[1][frame]
        else:
            x_t = theory_gre_r[0][frame]
            y_t = theory_gre_r[1][frame]
        tro_an.set_data(x_t, y_t)
        return jup_an, gre_an, tro_an,

    #Make a circle and the sun to show what the orbit should be
    theta = np.linspace(0,2*np.pi,128)
    circlex = np.asarray(np.full_like(theta,rad_jup)*np.cos(theta))
    circley = np.asarray(np.full_like(theta,rad_jup)*np.sin(theta))
    plt.plot(circlex,circley,'k--',linewidth=0.8)

    #Sun, is an orange marker at centre, as it hardly moves around the COM
    plt.plot(0,0,linewidth=0,markerfacecolor='orange',marker='o',markersize
        =25,markeredgewidth=0,label = 'Sun')

    #making and showing the animation
    ani = FuncAnimation(fig1, update, frames=np.arange(no_points),interval =
        10 ,init_func=init)
    plt.legend(loc = 'upper right', markerscale = 0.7)
    plt.title('Animation of the Motion')
    plt.xlabel('x (AU)')
    plt.ylabel('y (AU)')
    ani.save('real_animation.mp4', writer=writer)
    plt.show()


#This function will plot the evolution in phase space of the position and
#velocity of Greeks relative to the theoretical position
#Input is 0 or 1, 0 animates the evolution in position phase space, 1 plots
#the evolution in velocity space
def watch_phase_animation(r_or_v):
    fig1, ax = plt.subplots()
    if r_or_v == 0:
        name = 'Position'
        prefix = ''
        unit = '(AU)'
    else:
        name = 'Velocity'
        prefix = 'v'
        unit = '(AU/years)'

    an, = plt.plot([],[],'ro',label = name)
```

```
        def init():
                half_length = 1
                ax.set_xlim(-half_length, half_length)
                ax.set_ylim(-half_length, half_length)
                return an,

        def update(frame):
                if r_or_v == 0:
                        x_t = gre_r_dev[0][frame]
                        y_t = gre_r_dev[1][frame]
                else:
                        x_t = gre_v_dev[0][frame]
                        y_t = gre_v_dev[1][frame]
                an.set_data(x_t, y_t)
                return an,

        #Placing a point at the centre to compare
        plt.plot(0,0,'kx',label = 'Centre')

        #making and showing the animation
        ani = FuncAnimation(fig1, update, frames=np.arange(no_points),interval =
            10,init_func=init)
        plt.legend(loc = 'upper right', markerscale = 0.7)
        plt.title('Animation of ' +name+ ' Phase Space Deviation')
        plt.xlabel(prefix +'x '+unit)
        plt.ylabel(prefix +'y '+unit)
        ani.save(name+'_phase_animation.mp4', writer=writer)
        plt.show()



#These functions are used to plot deviations, and how the positions 'wanders'

#This gives the angle between two vectors, using the dot product
def angle(v1,v2):
        return np.arccos(np.dot(v1,v2)/(np.linalg.norm(v1)*np.linalg.norm(v2)))


#This function is used after make_z(), it plots the difference between the actual
     radius and the theoretical radius, as well as
#The difference between the actual angle and the theoretical angle, plots them
    side by side to determine if state is still bound
def pos_dev_plots():
        radius_difference = np.linalg.norm(gre_r,axis = 0)-np.linalg.norm(
            theory_gre_r,axis = 0)
        angle_difference = [angle([gre_r[0][i],gre_r[1][i]],[jup_r[0][i],jup_r[1][
            i]])-angle_ast for i in range(no_points)]

        plt.figure(1)
        plt.subplot(211)
        plt.plot(t, radius_difference, 'b-')
        plt.title('Actual and Theoretical Position Deviations')
        plt.ylabel('Radius Difference (AU)')

        plt.subplot(212)
```

```
        plt.plot(t, angle_difference, 'r-')
        plt.xlabel('Time (years)')
        plt.ylabel('Angle Difference (rad)')
        plt.show()


#This function takes the theoretical values of where the asteroids should be and
    finds the absolute distance from the actual asteroid
#Inputs are 0, 1 or 2, 0 plots the real motion, 1: position and velocity
    deviations with time, 2: motion in phase space for position and velocity
def wander_plots(x):
        if x == 0:
                #Make a circle and the sun to show what the orbit should be
                theta = np.linspace(0,2*np.pi,128)
                circlex = np.asarray(np.full_like(theta,rad_jup)*np.cos(theta))
                circley = np.asarray(np.full_like(theta,rad_jup)*np.sin(theta))
                plt.plot(circlex,circley,'k--',linewidth=0.8,label='Jupiter Oribt
                    ')

                #Sun, is an orange marker at the centre
                plt.plot(0,0,linewidth=0,markerfacecolor='orange',marker='o',
                    markersize=25,markeredgewidth=0,label = 'Sun')

                #Motion of Greeks
                plt.plot(gre_r[0],gre_r[1],'b-',linewidth=0.5,label='Greeks Orbit
                    ')
                plt.xlabel('x (AU)')
                plt.ylabel('y (AU)')
                plt.legend(loc = 'upper right',markerscale=0.7)
                plt.title('Motion of Greeks After Perturbation')
                plt.show()

        elif x == 1:
                plt.plot(t,mod_gre_r_dev)
                plt.title('Distance of Greeks from L4 Against Time')
                plt.xlabel('Time (years)')
                plt.ylabel('Deviation from L4 (AU)')
                plt.show()

                mod_gre_v_dev = np.linalg.norm(gre_v_dev,axis = 0)
                plt.plot(t,mod_gre_v_dev)
                plt.title('Velocity Deviation of Greeks Against Time')
                plt.xlabel('Time (years)')
                plt.ylabel('Deviation from L4 Velocity (AU/Years)')
                plt.show()

        else:
                plt.plot(gre_r_dev[0],gre_r_dev[1])
                plt.xlabel('x (AU)')
                plt.ylabel('y (AU)')
                plt.title('Position Phase Space Deviation')
                plt.show()

                plt.plot(gre_v_dev[0],gre_v_dev[1])
                plt.xlabel('vx (AU/years)')
                plt.ylabel('vy (AU/years)')
```

```
            plt.title('Velocity Phase Space Deviation')
            plt.show()



#These functions deal with perturbations, make the perturbations, find the
    maximum wander due to perturbations

#This function makes a radial perturbation to be put in can be used for position
    or velocity
def radial_perturb(x):
       return np.array([x*cos,x*sin])


#This function makes a tangential perturbation to be put in make_z can be used
    for position or velocity
def tangential_perturb(x):
       return np.array([-x*sin,x*cos])

#Makes a perturbation for the quantify wander function
def make_perturbation(rad_or_tan,r_or_v,pert_strength):
       if rad_or_tan == 0:
              perts = [radial_perturb(pert_strength[i]) for i in range(len(
                  pert_strength))]
       else:
              perts = [tangential_perturb(pert_strength[i]) for i in range(len(
                  pert_strength))]
       zeros = np.full_like(perts,0)
       if r_or_v == 0:
              return perts, zeros
       else:
              return zeros, perts


#This returns the maximum positional deviation, over 500 years
#Inputs are the position perturbation and velocity perturbation
def perturb_wander(r_perturb,v_perturb):
       make_z(500,500,r_perturb,v_perturb)
       return max(mod_gre_r_dev)


#This function uses the perturb vectors above to plot the maximum wander as a
    function of the perturbation
#Inputs: max_perturbation, half_number_of_points
#rad_or_tan = 0 or 1, 0 for radial 1 for tangential perturbations
#r_or_v = 0 or 1, 0 for positional, 1 for velocity perturbations
#show_plot, can be 0 to not show, or 1 to show
#Can also input constant perturbations, to see how the plots change
#Prints the linear fit and the r squared value, also gives the plot to be looked
    at.
def quantify_wander(max_perturb,no_points,rad_or_tan,r_or_v,const_r_perturb,
    const_v_perturb,show_plot):
       if rad_or_tan == 0:
              name0 = 'Radial'
       else:
```

```
            name0 = 'Tangential'
        if r_or_v == 0:
            name1 = 'Position'
            unit = '(AU)'
        else:
            name1 = 'Velocity'
            unit = '(AU/years)'

        perts = np.linspace(-max_perturb,max_perturb,no_points)
        perts_0 = make_perturbation(rad_or_tan,r_or_v,perts)
        r_perts = [perts_0[0]+np.full_like(perts_0[0],const_r_perturb)]
        v_perts = [perts_0[1]+np.full_like(perts_0[1],const_v_perturb)]
        wander = [perturb_wander(r_perts[0][i],v_perts[0][i]) for i in range(
            no_points)]

        slope, intercept, r_value= stats.linregress(abs(perts),wander)[:3]
        #This part just plots the graph
        for i in range(show_plot):
            plt.plot(perts,wander,'kx',label = 'Computed Wander')
            plt.plot(perts,np.full_like(perts,slope)*abs(perts) + np.full_like
                (perts,intercept),'b-',linewidth=0.7,label = 'Linear Fit')
            plt.legend(loc ='lower left')
            plt.xlabel(name0 + ' ' + name1 + ' Perturbation ' + unit)
            plt.ylabel('Wander (AU)')
            plt.title('Wander vs Perturbation')
            plt.show()
        return slope, intercept, r_value**2


#This makes 3d plots for the wander when two variable are looked at
#Number of points, x_limit and y_limit are trivial
#x_param and y_param, 0,1,2,3 0 = radial_pos
#1 = tangential_pos, 2 = radial_velocity, 3 = tangential_velocity
def multi_variable_plots(number_of_points,x_limit,y_limit,x_param,y_param):

        x = np.linspace(-x_limit,x_limit,number_of_points)
        y = np.linspace(-y_limit,y_limit,number_of_points)
        wander = []
        p = np.zeros(4)
        p[x_param] = 1
        p[y_param] = 1
        names = ['Radial Position (AU)','Tangential Position (AU)',
            'Radial Velocity (AU/years)', 'Tangential Velocity (AU/years)']

        #Selecting what data to collect based on the selection of variables
        if p[0] == 1:
            if p[1] == 1:
                wander_func = lambda i,j: perturb_wander(radial_perturb(x[i
                    ])+tangential_perturb(y[j]),[0,0])
            elif p[2] == 1:
                wander_func = lambda i,j: perturb_wander(radial_perturb(x[i
                    ]),radial_perturb(y[j]))
            else:
                wander_func = lambda i,j: perturb_wander(radial_perturb(x[i
                    ]),tangential_perturb(y[j]))
        elif p[1] == 1:
```

```
            if p[2] == 1:
                wander_func = lambda i,j: perturb_wander(tangential_perturb
                    (x[i]),radial_perturb(y[j]))
            else:
                wander_func = lambda i,j: perturb_wander(tangential_perturb
                    (x[i]),tangential_perturb(y[j]))
        else:
            wander_func = lambda i,j: perturb_wander([0,0],radial_perturb(x[i
                ])+tangential_perturb(y[j]))

        #Making data
        for i in range(len(x)):
            for j in range(len(y)):
                wander += [wander_func(i,j)]

        wander = np.asarray(wander)
        wander = np.reshape(wander, (number_of_points,number_of_points))

        fig = plt.figure()
        ax = fig.add_subplot(111)
        cset = ax.contour(x, y, wander, 20,cmap=cm.coolwarm)
        ax.clabel(cset, fontsize=9, inline=1)
        plt.xlabel(names[x_param])
        plt.ylabel(names[y_param])
        plt.title('Wander vs Perturbations')
        plt.show()

        X, Y = np.meshgrid(x,y)
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        surf = ax.plot_surface(X, Y, wander, cmap=cm.coolwarm, linewidth=0,
            antialiased=False)
        ax.set_xlabel(names[x_param])
        ax.set_ylabel(names[y_param])
        ax.set_zlabel('Wander (AU)')
        plt.title('Wander vs Perturbations')
        plt.show()


#Gives the slope and intercept for given masses
def look_at_point(jup_or_sun,m,rad_or_tan,r_or_v):
        make_parameters(jup_or_sun,m)
        return quantify_wander(0.001,7,rad_or_tan,r_or_v,[0,0],[0,0],0)[:2]


#This function plots the log log of the slopes and masses of a given body and
    perturbation
#Choose whether to change the mass of sun or jupiter, choosing radial/tangential,
     or position/velocity
def mass_dependence(jup_or_sun,rad_or_tan,r_or_v,no_of_points):
        if rad_or_tan == 0:
            name0 = 'Radial '
        else:
            name0 = 'Tangential '
        if r_or_v == 0:
            name1 = 'Position '
```

```
            units = '(unitless)'
        else:
            name1 = 'Velocity '
            units = '(years)'
        if jup_or_sun == 0:
            name2 = 'Jupiter'
        else:
            name2 = 'Sun'

        m = np.linspace(masses[jup_or_sun]*0.7,masses[jup_or_sun]*10,no_of_points)
        p = np.array([look_at_point(jup_or_sun,m[i],rad_or_tan,r_or_v) for i in
            range(no_of_points)]).T
        slopes = p[0]

        proposed_fit_vector = np.vectorize(proposed_fit)
        f_of_m = proposed_fit_vector(m)
        slope, intercept, r_value= stats.linregress(f_of_m,slopes)[:3]

        plt.plot(f_of_m,slopes,'kx',label = 'Actual Slopes vs Calculated')
        plt.plot(f_of_m,f_of_m,'b-',linewidth=0.7,label = 'y = x')
        plt.legend(loc ='upper left')
        plt.xlabel('f(m) ' + units)
        plt.ylabel('Slopes ' + units)
        plt.title('Slope of Wander vs Proposed Fit, for ' + name0 + name1 + '
            Perturbations, Varying Mass of ' + name2)
        plt.show()
        print('Intercepts = ' + str(p[1]))
        print(slope, intercept, r_value**2)



#Enter functions to run here:

make_z(500,5000,tangential_perturb(0.05),[0,0])
#test_energy()
pos_dev_plots()
wander_plots(2)
#watch_real_animation(1)
#watch_phase_animation(0)
print(quantify_wander(0.001,15,0,0,[0,0],[0,0],1))
#multi_variable_plots(30,0.0001,0.001,0,1)
#proposed_fit = lambda x: 1.85*x**0.449/0.001**0.449
#mass_dependence(1,0,0,20)
```