# Computer Simulation to Investigate the Stability of Jupiter's Trojan Asteroids

Candidate number: 7022R

June 1, 2020

**Abstract**

Asteroids at Jupiter's L4 Lagrange point were simulated in *Python* and their oscillations about L4 observed. The initial conditions in displacement and velocity space were perturbed and the effect on the stability and range of wander from L4 was measured. The range of wander was found to be proportional to $M^{-0.497}$, where $M$ is the planetary mass, and no stability was observed for $M/M_\odot > 0.0415$, in good agreement with the analytic critical value of 0.04006. This report contains 2400 words.

## 1 Introduction and Background

The "Lagrange points" are points of equilibrium, shown in fig. 1, where the gravitational forces from a sun and orbiting planet precisely balance the centrifugal force on an object placed there. The Lagrange points are of interest to astronomers because they can serve as "parking spots" for satellites. For example, the James Webb Space Telescope will rest at L2, which allows its heat shield to block radiation from the sun and earth simultaneously so that it can take accurate infrared readings. [1]

When Coriolis forces are accounted for, only L4 and L5 are points of stable equilibrium. Jupiter has a large collection of "Trojan" asteroids orbiting these points; 7039 have been discovered as of April 2019 [2]. In this report, I investigate the stability of the L4 and L5 points. Section 2 gives an overview of the mathematical analysis of the problem and section 3 summarises the computer implementation and performance. Section 4 presents and discusses my results.

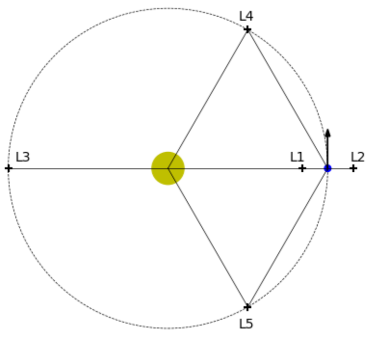

Figure 1: The approximate locations of the five Lagrange points.

## 2 Analysis of the Problem

There are four forces acting on a test mass in this system when viewed in a co-rotating frame: The gravitational forces from the planet and sun (1), the centrifugal force (2) and the Coriolis force (3).

$$\boldsymbol{F}_g = \frac{GM_\odot M}{r^3}\boldsymbol{r} \tag{1}$$

$$\boldsymbol{F}_{ce} = -M\boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{r}) \tag{2}$$

$$\boldsymbol{F}_{co} = -2M\boldsymbol{\omega} \times \dot{\boldsymbol{r}} \tag{3}$$

where $M$ is the planetary mass and $M_\odot$ is the solar mass.

The gravitational and centrifugal forces can be combined to give an effective potential, shown in fig. 2, which has the five equilibrium Lagrange points. L4 and L5 are maxima in this effective potential, but the Coriolis force actually makes them stable. Since L4 and L5 are physically very similar, I focused only on L4.

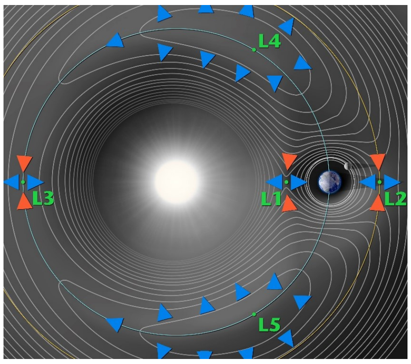

Figure 2: A contour plot of the effective potential around a sun and a planet. The arrows show the direction of decreasing potential. [3]

Rather than inputting the centrifugal and Coriolis forces into the program, I opted to work from more fundamental principles, using an an inertial frame so only the gravitational forces were required. However, this raised a problem; to obtain accurate results, I could not assume that the gravitational force on the sun from Jupiter was negligible such that the sun was stationary [1]. The motion of Jupiter was therefore not a simple circular orbit around a fixed mass, but a “two body problem” as shown in fig. 3. The distances of the sun and Jupiter from the centre of mass are given by:

$$r_s = M/(M_\odot + M) \tag{4}$$

$$r_J = M_\odot/(M_\odot + M) \tag{5}$$

and the angular velocity of the system is given by:

$$\omega^2 = \frac{G(M_\odot + M)}{r^3} \tag{6}$$

[1]Assuming the sun was fixed caused an asteroid placed at L4 to oscillate L4 and L5 in a large crescent when viewed in a co-rotating frame

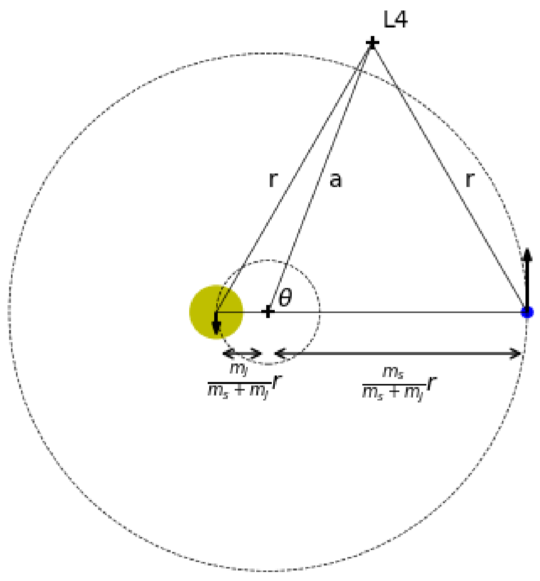

Figure 3: The location of L4 in the two-body system.

The differential equations to solve are:

$$\ddot{\boldsymbol{r}}_J = -\frac{GM_\odot}{|\boldsymbol{r}_J - \boldsymbol{r}_s|^3}(\boldsymbol{r}_J - \boldsymbol{r}_s) \tag{7}$$

$$\ddot{\boldsymbol{r}}_s = -\frac{GM}{|\boldsymbol{r}_s - \boldsymbol{r}_J|^3}(\boldsymbol{r}_s - \boldsymbol{r}_J) \tag{8}$$

$$\ddot{\boldsymbol{r}}_t = -\frac{GM_\odot}{|\boldsymbol{r}_t - \boldsymbol{r}_s|^3}(\boldsymbol{r}_t - \boldsymbol{r}_s) - \frac{GM}{|\boldsymbol{r}_t - \boldsymbol{r}_J|^3}(\boldsymbol{r}_t - \boldsymbol{r}_J) \tag{9}$$

where $\boldsymbol{r}_J$, $\boldsymbol{r}_s$ and $\boldsymbol{r}_t$ are the positions of Jupiter, the sun and test mass respectively.

Equations (7)-(9) can be re-written as coupled first-order ODEs and solved using routines such as LSODA.

The displacement of L4 from the origin is now non-trivial, and calls for two constants, $a$ and $\theta$ to be defined, as shown in fig. 3.

$$a = r\sqrt{k^2 - k + 1} \tag{10}$$

$$\cos\theta = \frac{2k-1}{2\sqrt{k^2-k+1}} \tag{11}$$

where

$$k = \frac{M}{M + M_\odot} \tag{12}$$

Finally, to minimise the risk of overflow, it is best to work in solar system units where unit mass is the mass of the sun ($M_\odot$), unit distance is the average earth-sun distance (1AU) and unit time is one year. The mass of Jupiter will be taken as $0.001M_\odot$ and the radius of its orbit as 5.2AU. In these units, $G = 4\pi^2$.

## 3 Computer Implementation

The program was written in *Python* and used the *NumPy*, *SciPy* and *Matplotlib* libraries. The basic approach was as follows. Objects were created for the sun, Jupiter, a test mass, and the point L4 and their initial conditions were attached to these objects in the function *run* (run.py). This function could take displacements of the test mass from L4 in the six dimensions of position and velocity space: $\Delta r$, $\Delta\theta$, $\Delta z$, $\Delta v_r$, $\Delta\omega$ and $\Delta v_z$. The objects were passed to the function *run_sim* (simulate.py). This function was the workhorse of the program; it used the ODEPACK library routine LSODA to propagate the system from the initial conditions and geometrically calculated the position of L4. This routine automatically selects between stiff and non-stiff methods, although in this program, only the non-stiff method was actually used, which is a multistep Adams method [4]. *run_sim* returned the objects, now with their propagated positions and velocities attached, back to *run*, which returned the same objects to the user. The data contained within the objects could then be passed to various functions either for visualisation or for analysis.

Two functions (in visualise.py) could be used to visualise the simulation; *plot* could plot the trajectory of the test mass in a frame that rotated with the orbit of Jupiter and *animate* created an animation showing the time-evolution of the system.

The files wander_1D.py, wander_2D.py and planet_masses.py were used to make measurements from the data. Each contained a *calc_wander* function which ran a number of simulations for different initial conditions and measured the wander and stability of the orbits which was saved .txt file. Other functions could then be ran to visualise or analyse the data. For example, *plot_wander_2D* plotted the maps presented in section 4.3 and *fit_log_wander_m* used linear regression to find a relationship between the range of wander and the planetary mass, as presented in section 4.4.

The program took an average of 0.6s to run a single simulation for 100 orbits. The full program is listed in the Appendix.

### 3.1 Testing

Tests were ran to ensure that the simulation produced accurate results. The first test was to ensure that Jupiter orbited the sun at a constant radius over short and long time scales. Fig. 4 shows the result of this test.

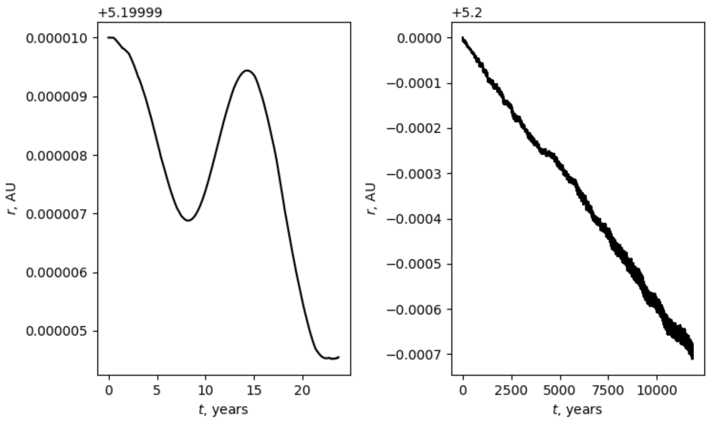

Figure 4: The radius of Jupiter's orbit plotted over 2 and 1000 orbits.

The radius of orbit varied by around $\pm 0.00001\%$ over each orbit and the average radius decreased by 0.014% over 1000 orbits. The decrease in average radius is not a problem because in most cases I used no more than 100 orbits, and the variation per orbit is so small that it can safely be considered negligible for my purposes.

I also tested that the centre-of-mass of the two-body system remained stationary. As shown in fig. 5, the movement of the centre-of-mass was of order $10^{-16}$AU per orbit.

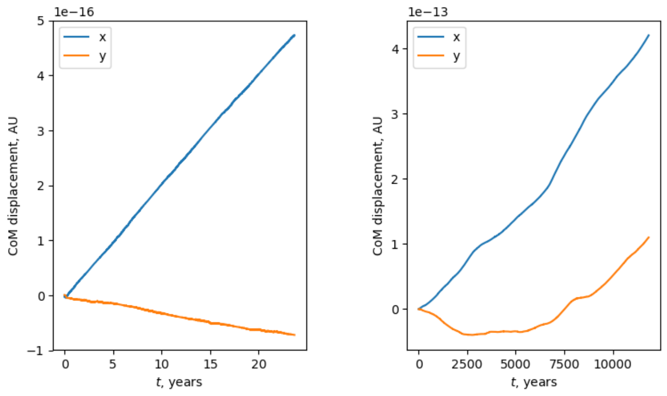

Figure 5: The displacement of the centre of mass plotted over 2 and 1000 orbits.

The stability of the time period of Jupiter's orbit was tested, and I found that any discrepancy in time period was smaller than the minimum time increment that the memory of my PC would allow, which was $2.5 \times 10^{-5}$ years. The variation in time period was therefore no more than 0.0002%.

The final test was whether a test mass that started at L4 would stay there. As shown in fig. 6, this test was successful. The oscillations shown are orbits about L4 with a maximum radius that increases to 0.0015% of the radius of Jupiter's orbit after 100 orbits. These orbits are expected, but it is unexpected that their radius should increase. Given that the magnitude of increase is nearly identical to the decrease in the radius of Jupiter's orbit, it is likely that this effect is another error associated with the integration, rather than a feature of of the Lagrange points. This and the decrease in Jupiter's radius are the most significant errors, so they can be taken as a benchmark for the error in my analysis.

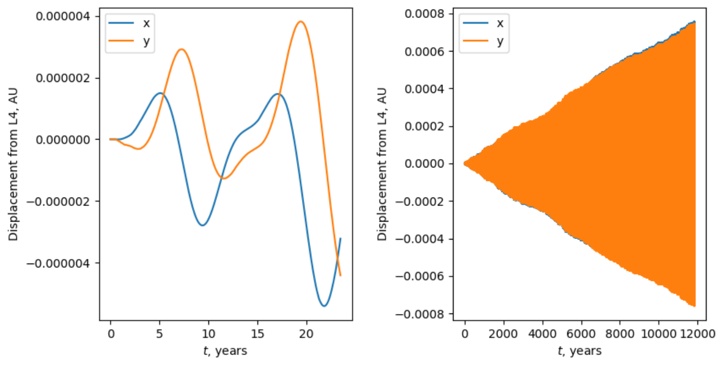

Figure 6: The displacement from L4 of a test mass starting at L4 plotted over 2 and 1000 orbits. The sun is drawn in yellow and Jupiter in blue.

## 4 Results and Discussion

### 4.1 The Stability of L4

The initial position of a test mass was slightly displaced from L4 in both position and velocity space. I found that, provided the displacement was small enough, the test mass would orbit L4 as expected. With too large a displacement, a test mass would escape from this orbit and follow an unpredictable trajectory. Typical stable and non-stable orbits plotted in a co-rotating frame are shown in fig. 7 and 8.

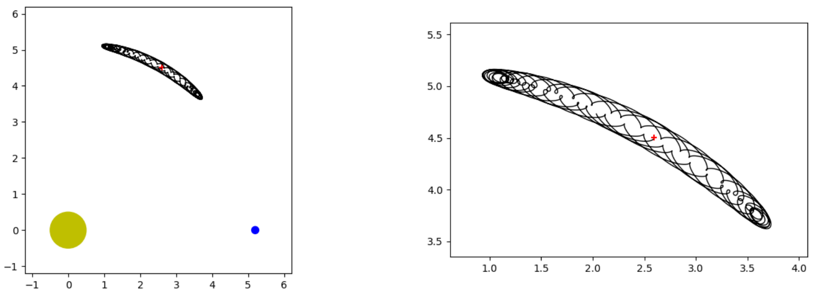

Figure 7: A typical stable orbit about L4, plotted in a co-rotating frame for 50 orbits of Jupiter ($\Delta r = -0.02$).

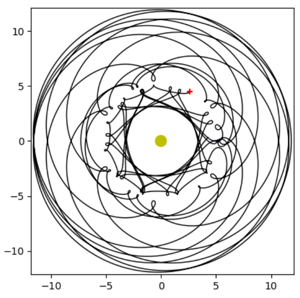

Figure 8: A typical unstable orbit of L4 ($\Delta r = -0.1$).

### 4.2 The Hilda asteroids

I was also able to simulate the "Hilda asteroids" [5] which orbit the sun. There are around 4,000 such asteroids and their key feature is that they orbit with a 3:2 resonance with Jupiter, meaning that their orbital period is 2/3 that of Jupiter. Their orbit is roughly triangular, as shown in fig. 9, with the triangle revolving around the sun with the same period as Jupiter. The asteroids essentially travel between L3, L4 and L5, allowing them to avoid collision with Jupiter and remain in a stable orbit. From the initial conditions: ($\Delta r = -1.7$, $\Delta\theta = -\pi/3$, $\Delta\omega = 0.485$), the orbit shown in fig. 9. was produced.

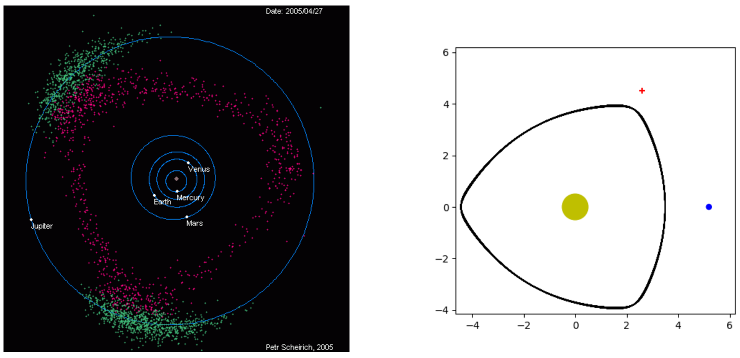

Figure 9: The locations of the Hilda (pink) and Trojan (green) asteroids [6], alongside a simulation of a Hilda asteroid, plotted over 100 orbits.

## 4.3 Range of wander from L4

The stability of L4 was further investigated by considering how far asteroids wandered from the equilibrium position given different initial conditions in displacement and velocity. The initial conditions of the test mass could be perturbed from L4 in six dimensions: $\Delta r$, $\Delta\theta$, $\Delta z$, $\Delta v_r$, $\Delta\omega$ and $\Delta v_z$. The wander was defined as the maximum distance between the test mass and L4 over 100 Jupiter orbits and an orbit was defined as stable if the mass deviated by no more than an angle of $2\pi$ either side of L4 over 100 orbits. The effect of perturbing two initial conditions simultaneously was investigated, and the maps show in fig. 10-12 were produced. The maps encode both the wander and the stability, with the coloured regions indicating the stable regions, and the white regions indicating unstable regions.

All of the maps display minimum wander when the test mass has the same initial conditions as L4, as expected. In general, the wander increases as the initial conditions are perturbed, until the orbits become unstable.

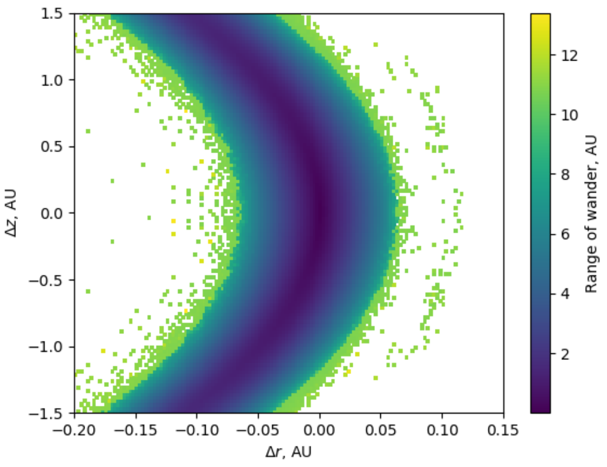

Figure 10: The effect of varying $\Delta r$ and $\Delta z$ on the range of wander.

The $\Delta r$-$\Delta z$ map (fig. 10) shows stability along a curve. The curve does not conserve the radius at 5.2AU as one might expect, but the radius increases to 5.31AU at $\Delta z = \pm 1.5$. These orbits dip back into the same plane a Jupiter after a short time.

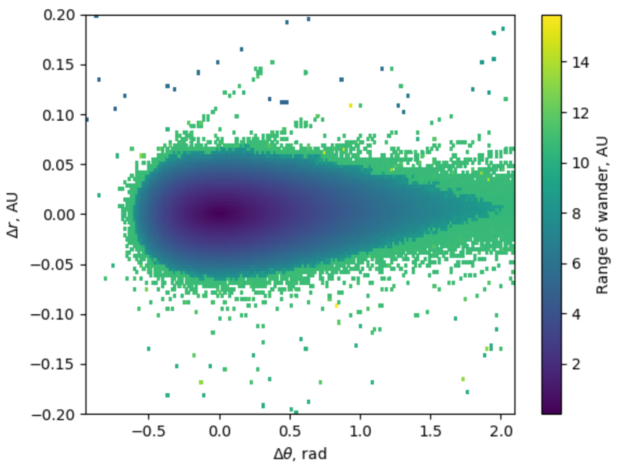

Figure 11: The effect of varying $\Delta\theta$ and $\Delta r$ on the range of wander.

The $\Delta\theta$-$\Delta r$ map (fig. 11) has a "teardrop" shape. Stability is quickly lost as $\Delta\theta$ is decreased because this puts the test mass near to Jupiter, which accelerates it away from L4. On the other hand, if $\Delta\theta$ is increased, the test mass will just orbit L5 instead. The stability is much more sensitive to changes in $\Delta r$, as these put the test mass in a non-circular orbit.

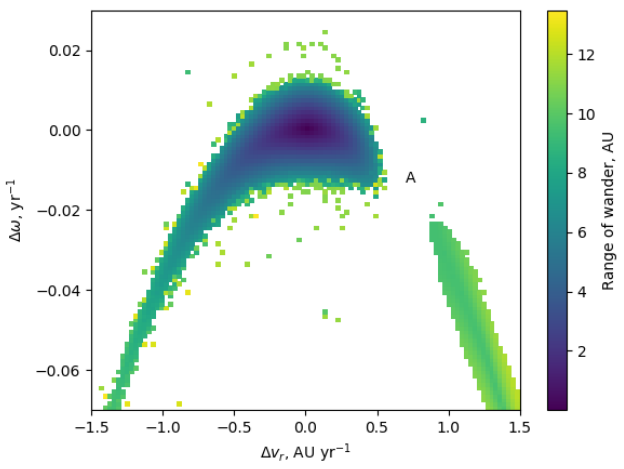

Figure 12: The effect of varying $\Delta r_v$ and $\Delta\omega$ on the range of wander.

The $\Delta v_r$-$\Delta\omega$ map (fig. 12) shows stability along a curve because this curve allows the test mass to orbit the sun in an elliptical orbit with the same time period as Jupiter. The region of instability at A is where the test mass wanders too close to Jupiter. The stability is much more sensitive to changes in $\Delta\omega$ than to changes in $\Delta v_r$

The effect of changing a single variable in phase space was investigated in more detail, and plots were produced as shown in fig. 13, 14 and 16.

When only the initial radius is varied (fig. 13), the range of wander $= 80|\Delta r|$ for small $\Delta r$. The wander increases either side of $\Delta r = 0$ until around $\Delta r = \pm 0.07$AU, beyond which the orbits become unstable. In the unstable regions, the test mass typically gets very close to Jupiter, causing it to be rapidly accelerated in some direction. This is very sensitive to the initial conditions; hence the range of wander is unpredictable in the unstable regions. A change as small as $10^{-18}$AU has unpredictable results.

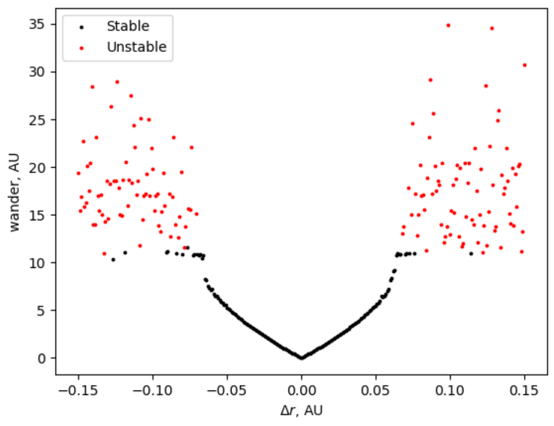

Figure 13: The effect of varying $\Delta r$ on the range of wander.

If only $\Delta\theta$ is varied, the wander varies as shown in fig. 14. Typical orbits in each regime are shown in fig. 15. At A, typical L4 orbits are observed where the maximum wander is just the distance of the starting position from L4. Therefore, for small displacements, wander $\propto 5.2|\Delta\theta|$. At C, the test mass starts starts orbiting both L4 and L5, resulting in maximum wander. At D, the test mass only orbits L5. E is a region of instability caused by the test mass getting too close to Jupiter, and F, although it appears stable is actually a region where the test mass gets so close to Jupiter that the integration breaks down and the test mass falls into the sun.

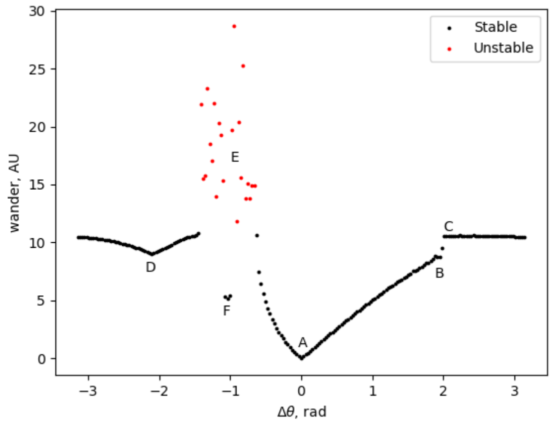

Figure 14: The effect of varying $\Delta\theta$ on the range of wander.

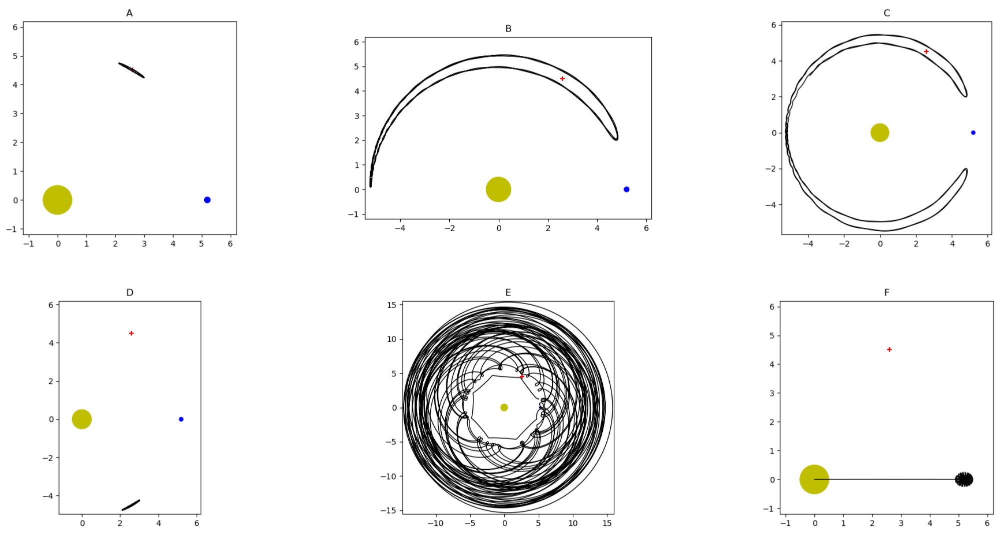

Figure 15: Typical orbits as $\Delta\theta$ is varied.

Similar plots were made for the variation of wander with $\Delta z$, $\Delta v_r$, $\Delta\omega$ and $\Delta v_z$, as shown in fig. 16. Varying these variables has a similar effect to varying $\Delta r$, with varying degrees of sensitivity.

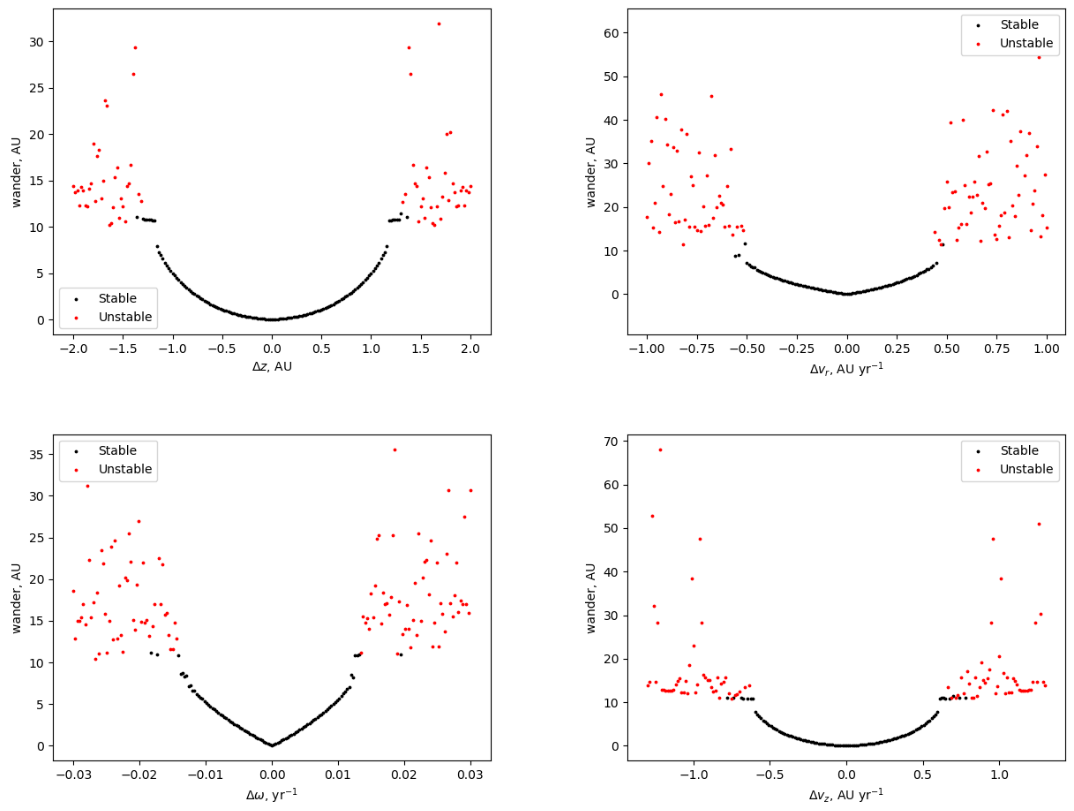

Figure 16: The effect of independently varying $\Delta z$, $\Delta v_r$, $\Delta\omega$ and $\Delta v_z$ on the range of wander.

### 4.4 The effect of planetary mass on the range of wander

Trojan objects are also present at the Lagrange points L4 and L5 of Mars (9 Trojans), Uranus (1) and Neptune (22) [2]. Furthermore, the Lagrange points of Earth's orbit are of interest to astrophysicists who wish to park satellites there. Hence it is useful to understand the effect of the planetary mass on the stability of the Lagrange points.

The wander for a range of planetary masses was plotted in fig 17 using $\Delta r = 0.001$AU. For masses around $10^{-6}M_\odot$ and an initial displacement of $10^{-5}AU$, I found that the range of wander was proportional to $M^{-\alpha}$ where $\alpha = 0.4969$. For any smaller planetary mass or initial displacement, enough orbits could not be simulated to measure the wander, or simulation inaccuracies became too significant. It seems likely, however, that the range of wander is proportional to $M^{-1/2}$, but this would have to be confirmed my more sophisticated simulations. The most stable orbit occurred at $M = 0.017M_\odot$.

It can be shown analytically [7] that L4 becomes unstable when

$$\frac{M}{M + M_\odot} > \frac{1}{2}(1 - \sqrt{\frac{23}{27}}) \tag{13}$$

or equivalently,

$$\frac{M}{M_\odot} > 0.04006 \tag{14}$$

By simulating 1000 orbits over a range of planetary masses, I found that L4 was unstable for $M/M_\odot > 0.0415$, as shown in fig. 17. The error of 3% is due to the fact that only a finite number of orbits can be simulated, so some orbits that would become unstable do not get the chance to. Memory limitations prevented longer simulations being ran so could be a point of improvement for future studies.

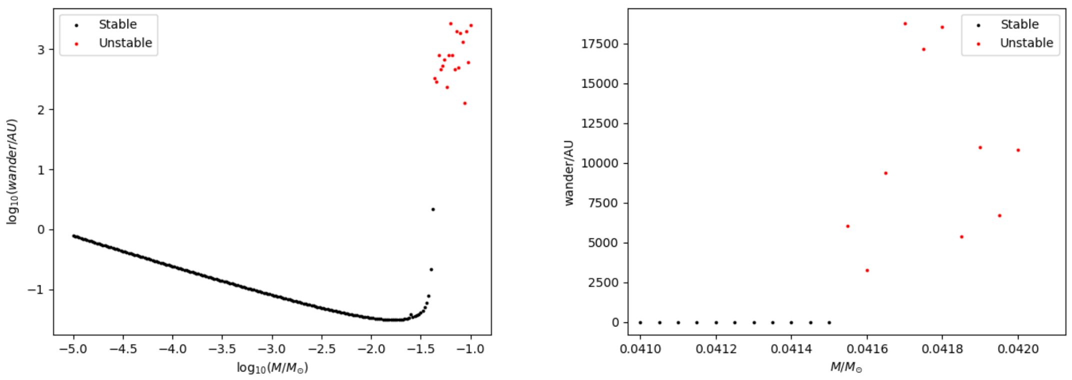

Figure 17: The effect of the planetary mass on the range of wander for $\Delta r = 0.001$AU initially over 1000 orbits. A large range of $M$ is plotted on the right, and a much narrower range is plotted on the left to find the point at which stability is lost.

## 5 Conclusions

Oscillations about Jupiter's L4 Lagrange point were simulated in *Python* and it was found that the region of stability is along an arc which is narrow in the $r$ direction and roughly traces the circumference of Jupiter's orbit. This arc extends at least 1.5 AU above and below the plane of Jupiter's orbit. In the unstable regions, it was found that the range of wander was very unpredictable, with changes as small as $10^{-18}$ having a significant effect, indicating that such a system is chaotic. For small planetary masses, $M$, and small initial perturbations, it was found that the range of wander is proportional to $M^{-0.4969}$, with no stability found for $M/M_{odot} > 0.0415$, close to the analytic critical value of 0.04006. Further conclusions could be made with more computing time and memory limitations prevented longer simulations being ran, which was an issue when estimating the effect of planetary mass on the stability.

## References

[1] NASA. *James Webb Space Telescope, Goddard Space Flight Centre.* `https://jwst.nasa.gov/index.html`. Accessed: 2019-04-09.

[2] *The International Astronomical Union Minor Planet Centre.* `https://minorplanetcenter.net//iau/lists/Trojans.html`. Accessed: 2019-04-09.

[3] *NASA Science Solar System Exploration, What is a Lagrange Point?* `https://solarsystem.nasa.gov/resources/754/what-is-a-lagrange-point/`. Accessed: 2019-04-09. Mar. 27, 2018.

[4] *R Documentation, lsoda.* `https://www.rdocumentation.org/packages/deSolve/versions/1.21/topics/lsoda`. Accessed: 2019-04-09.

[5] Tsuyoshi Terai and Fumi Yoshida. "Size Distribution of Small Hilda Asteroids". In: *The Astronomical Journal* 156.1 (June 2018), p. 30. DOI: `10.3847/1538-3881/aac81b`. URL: `https://doi.org/10.3847%2F1538-3881%2Faac81b`.

[6] M. Wade Holler. *Trojan Asteroids around Jupiter explained.* `https://www.exploremars.org/trojan-asteroids-around-jupiter-explained`. Accessed: 2019-04-09. June 29, 2013.

[7] Richard Fitzpatrick. *Stability of Lagrange Points.* `http://farside.ph.utexas.edu/teaching/336k/Newtonhtml/node126.html`. Accessed: 2019-04-09. Mar. 31, 2011.

# Appendices

## A Full program listing

### A.1 main.py

A space to run the whole simulation from, and plot graphs.

```
#this file is a useful space to run the simulation from

from run import run
from visualise import animate
from visualise import plot

times, sun, jup, test, L4 = run(rad_disp=0.05)
plot(sun, jup, test, L4)
animate(sun, jup, test, L4)
```

### A.2 run.py

Creates objects for the sun, Jupiter, test mass and L4 and attaches their initial conditions.

```
import numpy as np
from numpy import pi
from simulate import run_sim
from other_functions import polar_to_cartesian


def run(rad_disp=0, ang_disp=0, z_disp=0, rad_vel=0, ang_vel=0, z_vel=0,
  planet_mass=0.001, tot_t=100*11.85):
  '''run the simulation given the displacement of the test mass from L4.
  Returns times and sun, jup, test, L4 bodies with propagated positions
                                    attached'''

  class Sun():
    def __init__(self):
      self.loc = [0,0,0]
      self.vel = [0,0,0]
      self.m = 1

  class Planet():
    def __init__(self):
      self.loc = [0,0,0]
      self.vel = [0,0,0]
      self.m = planet_mass

  class TestMass():
    def __init__(self):
      self.loc = [0,0,0]
      self.vel = [0,0,0]

  #initialise the bodies
  sun = Sun()
  jup = Planet()
  test = TestMass()
  L4 = TestMass()
  #ang vel of orbit of jupiter in the two-body problem
  w = np.sqrt(4*pi**2*(sun.m+jup.m)/5.2**3)

  #set up initial conditions of bodies
  sun_r = 5.2*jup.m/(sun.m+jup.m)
  sun.loc, sun.vel =  polar_to_cartesian(
    np.array([[5.2*jup.m/(sun.m+jup.m), pi, 0]]),
    np.array([[0, w, 0]]))
  jup.loc, jup.vel = polar_to_cartesian(
    np.array([[5.2*sun.m/(sun.m+jup.m), 0, 0]]),
    np.array([[0, w, 0]]))

  k = sun.m/(sun.m+jup.m)     #k, a and theta as defined in equations 10-12
  a = 5.2*np.sqrt(k**2-k+1)
  theta = np.arccos((2*k-1)/(2*np.sqrt(k**2-k+1)))
  test.loc, test.vel = polar_to_cartesian(
    np.array([[a+rad_disp, theta+ang_disp, z_disp]]),
    np.array([[rad_vel, w+ang_vel, z_vel]]))


  #run the simulation.
  #We replace the initial conditions stored in the objects with a 2D
  #array of their propagated positions and velocities
  times, sun, jup, test, L4 = run_sim(sun, jup, test, L4, tot_t)
  return times, sun, jup, test, L4
```

## A.3 simulate.py

Propagates from the initial conditions and attaches the data to the objects.

```
import matplotlib.pyplot as plt
import scipy.integrate as integ
import time
import numpy as np
from numpy import pi
from other_functions import polar_to_cartesian


def run_sim(sun, jup, test, L4, tot_t):
  '''Uses scipy.odeint to propagate from the initial conditions.
  Automatically calculates position of L4.
  Replaces initial conditions in objects with propagated conditions'''
  start_time = time.time()

  #defined quantities
  dt = 1e-2
  G = 4*pi**2
  iterations = int(tot_t/dt)

  def derivs(y, t, sun, jup):
    '''Equations of motion to solve. The input y must be a 1D array,
    and is organised as [x,y,z,x',y',z'], repeated for the number of bodies.
    Sun and jup must be passed to access their masses.'''

    #converting back to an intuitive form
    sun_loc = y[0:3]
    sun_vel = y[3:6]
    jup_loc = y[6:9]
    jup_vel = y[9:12]
    test_loc = y[12:15]
    test_vel = y[15:18]

    jup_acc = -G*sun.m*(jup_loc-sun_loc)/np.linalg.norm(jup_loc-sun_loc)**3
    sun_acc = -G*jup.m*(sun_loc-jup_loc)/np.linalg.norm(sun_loc-jup_loc)**3
    test_acc = (-G*sun.m*(test_loc-sun_loc)/np.linalg.norm(test_loc-sun_loc)
                                    **3
          -G*jup.m*(test_loc-jup_loc)/np.linalg.norm(test_loc-jup_loc)**3)

    ydot = np.hstack(
      (sun_vel, sun_acc, jup_vel, jup_acc, test_vel, test_acc))
    return ydot


  #locs and vels must be in a 1D array
  y0 = np.ndarray(18)
  y0[0:3] = sun.loc[0]
  y0[3:6] = sun.vel[0]
  y0[6:9] = jup.loc[0]
  y0[9:12] = jup.vel[0]
  y0[12:15] = test.loc[0]
  y0[15:18] = test.vel[0]

  solution = integ.odeint(derivs, y0, np.linspace(0., tot_t, iterations),
                                    args=(sun, jup))

  #replace initial conditions with propagated locs and vels
  sun.loc = solution[:, 0:3]
  sun.vel = solution[:, 3:6]
  jup.loc = solution[:, 6:9]
  jup.vel = solution[:, 9:12]
  test.loc = solution[:, 12:15]
  test.vel = solution[:, 15:18]

  #Calculate position of L4
  phi = np.arctan2(jup.loc[:,1], jup.loc[:,0])
  k = sun.m/(sun.m+jup.m)
  a = np.ndarray((iterations))
  a[:] = 5.2*np.sqrt(k**2-k+1)
  theta = np.arccos((2*k-1)/(2*np.sqrt(k**2-k+1)))
  L4.loc = polar_to_cartesian(
    np.transpose([a, theta+phi, np.zeros((iterations))]),
    np.array([[0, 0, 0]]))[0]    #ang_vel doesnt matter - we only want
                                    location of L4

  times = np.arange(iterations)*dt

  print('Time taken to run sim: '+str(time.time()-start_time)+' seconds')
  return times, sun, jup, test, L4
```

## A.4 visualise.py

Contains functions for plotting and animating the evolution of the system.

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.animation import FuncAnimation
from other_functions import polar_to_cartesian

def animate(sun, jup, test, L4):
  '''takes the objects and animates them using funcanimation'''

  #defined quanitites
  num_of_frames = len(sun.loc[:,0])
  display_rate = 10   #display every n frames
  frame_interval = 1000/60

  fig = plt.figure()
  ax = plt.axes()
  ax.axis('scaled')
  ax.set_xlim(-8, 8)
  ax.set_ylim(-8, 8)
  lines = [plt.plot([], [])[0] for _ in range(4)]   #points to animate


  def init():
    '''initialisation function for FuncAnimation'''
    for i in range(len(lines)):
      lines[i].set_data([],[])
      lines[i].set_marker('.')
    lines[0].set_color('r')   #sun
    lines[1].set_color('b')   #Jupiter
    lines[2].set_color('k')   #test
    lines[3].set_color('0.5') #L4
    lines[3].set_marker('+')  #L4
    return lines


  def update_screen(i):
    '''update function for FuncAnimation'''
    frame = i*display_rate
    lines[0].set_data([sun.loc[frame,0]], [sun.loc[frame,1]]) #sun
    lines[1].set_data([jup.loc[frame,0]], [jup.loc[frame,1]]) #Jupiter
    lines[2].set_data([test.loc[frame,0]], [test.loc[frame,1]]) #test
    lines[3].set_data([L4.loc[frame,0]], [L4.loc[frame,1]])   #L4
    return lines


  ani = FuncAnimation(
    fig, update_screen, frames=np.arange(0, num_of_frames),
    init_func=init, blit=True, interval=frame_interval)

  plt.show()


def plot(sun, jup, test, L4):
  '''plots the motion of the test mass in a co-rotating frame'''
  fig, ax = plt.subplots()
  ax.axis('scaled')
  #find the coordinates in the co-rotating frame:
  test_x = test.loc[:,0]
  test_y = test.loc[:,1]
  jup_x = jup.loc[:,0]
  jup_y = jup.loc[:,1]
  jup_phi = np.arctan2(jup_y, jup_x)
  test_x_rot_frame = test_x*np.cos(jup_phi) + test_y*np.sin(jup_phi)
  test_y_rot_frame = -test_x*np.sin(jup_phi) + test_y*np.cos(jup_phi)

  #plot everything, making sure whole trajectory is shown
  ax.set_xlim(
    min(-1, min(test_x_rot_frame))-0.2,
    max(6, max(test_x_rot_frame))+0.2)
  ax.set_ylim(
    min(-1, min(test_y_rot_frame))-0.2,
    max(6, max(test_y_rot_frame))+0.2)
  ax.plot(test_x_rot_frame, test_y_rot_frame, 'k', linewidth=1) #test
  ax.add_artist(plt.Circle(sun.loc[0,0:2], 0.5, color='y'))   #sun
  ax.add_artist(plt.Circle(jup.loc[0,0:2], 0.1, color='b'))   #jupiter
  ax.scatter(L4.loc[0,0], L4.loc[0,1], c='r', marker='+')     #L4
  plt.show()
```

## A.5 other_functions.py

Contains algorithms that were frequently used.

```
import numpy as np

def polar_to_cartesian(pol_loc, pol_vel):
  '''converts an arrays of [r, phi, z] and [rad_vel, ang_vel, z_vel]
  into [x,y, z] and [x', y', z']. Takes and returns shape(N, 3)'''

  #put input into intuitive form
  r = pol_loc[:,0]
  phi = pol_loc[:,1]
  z = pol_loc[:,2]
  rad_vel = pol_vel[:,0]
  ang_vel = pol_vel[:,1]
  z_vel = pol_vel[:,2]

  #make transformation
  x = r*np.cos(phi)
  y = r*np.sin(phi)
  x_vel = rad_vel*np.cos(phi) - r*ang_vel*np.sin(phi)
  y_vel = rad_vel*np.sin(phi) + r*ang_vel*np.cos(phi)

  return np.transpose([x, y, z]), np.transpose([x_vel, y_vel, z_vel])


def get_real_phi(loc):
  '''returns the angle of an object from the origin, given its cartesian
                                                    location.
  Real_phi monotonically increases. ie not restricted to [0,2pi)'''

  phi = np.arctan2(loc[:,1], loc[:,0])      #restricted to [0,2pi)

  #every time phi decreases significantly, this means that it has gone
  #over the range [0,2pi), so need to add 2pi every time this happens
  increment = np.diff(phi)
  increment = np.insert(increment, 0, 0)    #make same length as phi
  to_add = np.cumsum(np.where(increment<-1, 2*np.pi, 0))
  real_phi = phi+to_add
  return real_phi
```

## A.6 wander_1D.py

Plots the graphs displayed in figs. 13, 14 and 16, and fits a line to the data.

```
import numpy as np
from numpy import pi
import matplotlib.pyplot as plt
from other_functions import get_real_phi
from run import run
import scipy.stats as stats
from visualise import plot


def calc_wander_1D(r):
  '''
  Calculates the wander and tests stability of the array of rs inputted.
  Saves wander.txt, stable.txt and r.txt
  Can easily be modified for a different independent variable
  '''

  def wander_func_1D(r):
    '''returns the wander and stability, given a delta r'''

    times, sun, jup, test, L4 = run(rad_disp=r)
    dist_from_L4 = np.linalg.norm(test.loc-L4.loc, axis=1)
    wander = np.max(dist_from_L4)

    #An orbit is unstable if it deviates more than 2pi from L4
    rel_phi = get_real_phi(test.loc)-get_real_phi(L4.loc)
    if np.max(rel_phi) > 2*pi or np.min(rel_phi) < -2*pi:
      stable = False
    else:
      stable = True

    return wander, stable


  vwander = np.vectorize(wander_func_1D)
  wander, stable = vwander(r)

  np.savetxt('wander.txt', wander)
  np.savetxt('stable.txt', stable)
  np.savetxt('r.txt', r)


def plot_wander_1D():
  '''Plots the wander against r. Gives different colours for stable and
                                          unstable'''
  wander = np.loadtxt('wander.txt')
  stable = np.loadtxt('stable.txt')
  r = np.loadtxt('r.txt')
  #sort the stable and unstable orbits into separate data sets
  wander_stable = np.extract(stable==True, wander)
  r_stable = np.extract(stable==True, r)
  wander_unstable = np.extract(stable==False, wander)
  r_unstable = np.extract(stable==False, r)

  plt.scatter(r_stable, wander_stable, color='k', s=3, label='Stable')
  plt.scatter(r_unstable, wander_unstable, color='r', s=3, label='Unstable')
  plt.xlabel('$\Delta r$, AU')
  plt.ylabel('wander, AU')
  plt.legend()
  plt.show()


def fit_log_wander_1D():
  '''plots log(wander) against log(r) and fits a straight line to it.
  Prints the parameters for the line'''
  r = np.loadtxt('r.txt')
  wander = np.loadtxt('wander.txt')

  reg = stats.linregress(np.log(r), np.log(wander))
  print(reg)
  def bestfit(x):
    return x*reg.slope + reg.intercept

  plt.plot(np.log(r), np.log(wander))
  plt.plot(
    [min(np.log(r)), max(np.log(r))],
    [bestfit(min(np.log(r))), bestfit(max(np.log(r)))])
  plt.xlabel('log(r)')
  plt.ylabel('log(wander)')
  plt.show()


r = np.linspace(0.01, 0.15, 11)
calc_wander_1D(r=r)
plot_wander_1D()
fit_log_wander_1D()
```

## A.7 wander_2D.py

Plots the maps displayed in figs. 10, 11 and 12.

```
import numpy as np
from numpy import pi
import matplotlib.pyplot as plt
from other_functions import get_real_phi
from other_functions import polar_to_cartesian
from run import run


def calc_wander_2D(rr, tt):
  '''
  Calculates wander and tests stability of the meshgrid of rs and ts
                                    inputted.
  t = theta = angular displacement
  Saves wander.txt, stable.txt and r.txt
```

```
    Can easily be modified for a different independent variable
    '''

    def wander_func_2D(r, t):
      '''returns the wander and stability, given a delta r and delta t'''

      times, sun, jup, test, L4 = run(rad_disp=r, ang_disp=t)
      dist_from_L4 = np.linalg.norm(test.loc-L4.loc, axis=1)
      wander = np.max(dist_from_L4)

      rel_phi = get_real_phi(test.loc)-get_real_phi(L4.loc)
      if np.max(rel_phi) > 2*pi or np.min(rel_phi) < -2*pi:
        stable = False
      else:
        stable = True

      return wander, stable

    vwander = np.vectorize(wander_func_2D)
    wander, stable = vwander(rr, tt)

    np.savetxt('wander.txt', wander)
    np.savetxt('stable.txt', stable)
    np.savetxt('rr.txt', rr)
    np.savetxt('tt.txt', tt)


def plot_wander_2D():
    '''plots a 2D map of the wander. White = unstable point'''
    wander = np.loadtxt('wander.txt')
    stable = np.loadtxt('stable.txt')
    rr = np.loadtxt('rr.txt')
    tt = np.loadtxt('tt.txt')

    #allow plt to distinguish between stable and unstable points
    wander = np.ma.masked_where(stable==False, wander)
    my_cmap = plt.get_cmap('viridis')
    my_cmap.set_bad(color='w')

    plt.pcolor(tt, rr, wander, cmap=my_cmap)
    plt.xlabel(r'$\Delta \theta$, rad')
    plt.ylabel('$\Delta r$, AU')
    cbar = plt.colorbar()
    cbar.ax.set_ylabel('Range of wander, AU')
    plt.show()


r = np.linspace(-0.15, 0.15, num=20)
t = np.linspace(-0.8, 2, num=20)
rr , tt = np.meshgrid(r, t)

calc_wander_2D(rr, tt)
plot_wander_2D()
```

## A.8 planet_masses.py

Plots the graphs displayed in fig. 17, and fits a line to the data.

```
import numpy as np
from numpy import pi
import matplotlib.pyplot as plt
from other_functions import get_real_phi
from other_functions import polar_to_cartesian
from run import run
from visualise import plot
import scipy.stats as stats


def calc_wander_m(r, m):
  '''
  Calculates the wander and tests stability of the array of ms inputted.
  Requires an initial radial displacement.
  Saves masses.txt, containing arrays for ms, stable and wander
  '''

  def wander_func_m(r, m):
    '''returns the wander and stability, given a delta r and m.
    Runs the sim for 1000 orbits, rather than the default of 100'''

    times, sun, jup, test, L4 = run(rad_disp=r, planet_mass=m, tot_t=11.85*
                                    1000)
    dist_from_L4 = np.linalg.norm(test.loc-L4.loc, axis=1)
    wander = np.max(dist_from_L4)

    rel_phi = get_real_phi(test.loc)-get_real_phi(L4.loc)
    if np.max(rel_phi) > 2*pi or np.min(rel_phi) < -2*pi:
      stable = False
    else:
      stable = True

    return wander, stable

  vwander = np.vectorize(wander_func_m)
  wander, stable = vwander(0.00001, m)
  np.savetxt('masses.txt', [m, stable, wander])


def plot_wander_m():
  ''' plots log(m) against log(wander), with different colours for stable
                                    and unstable'''
  m, stable, wander = np.loadtxt('masses.txt')

  #sort the stable and unstable orbits into separate data sets
  wander_stable = np.extract(stable==True, wander)
  m_stable = np.extract(stable==True, m)
  wander_unstable = np.extract(stable==False, wander)
  m_unstable = np.extract(stable==False, m)

  plt.scatter(
    np.log10(m_stable), np.log10(wander_stable),
    color='k', s=3, label='Stable')
  plt.scatter(
    np.log10(m_unstable), np.log10(wander_unstable),
    color='r', s=3, label='Unstable')
  plt.xlabel(r'$\log_{10}(M/M_{\odot})$')
  plt.ylabel(r'$\log_{10}(wander/AU)$')
  plt.legend()
  plt.show()


def fit_log_wander_m():
  '''fits a straight line to the log-log graph and prints the parameters'''
  m, stable, wander = np.loadtxt('masses.txt')

  reg = stats.linregress(np.log(m), np.log(wander))
  print(reg)

  def bestfit(logm):
    return logm*reg.slope + reg.intercept

  plt.plot(np.log(m), np.log(wander))
  plt.plot(
    [min(np.log(m)), max(np.log(m))],
    [bestfit(min(np.log(m))), bestfit(max(np.log(m)))])
  plt.xlabel('log(mass)')
  plt.ylabel('log(wander)')
  plt.show()


m = np.geomspace(0.0001, 0.1, num=20)
calc_wander_m(0.001, m)
plot_wander_m()
fit_log_wander_m()
```