UNIVERSITY of CALIFORNIA
SANTA CRUZ

A STUDY OF ORBITAL RESONANCES IN AN IDEALIZED MODEL OF THE SUN-JUPITER SYSTEM

A thesis submitted in partial satisfaction of the
requirements for the degree of

BACHELOR OF SCIENCE

in

PHYSICS

by

Alexander Rowe

August 2014

The thesis of Alexander Rowe is approved by:

Professor Greg Laughlin
Advisor

Professor Greg Laughlin
Theses Coordinator

Professor David P. Belanger
Chair, Department of Physics

Abstract

A Study of Orbital Resonances in an Idealized Model of the Sun-Jupiter System

by

Alexander Rowe

Jupiter's mass is large enough to have a substantial dynamical effect on test particles throughout the solar system, and in particular on bodies lying within the region occupied by the asteroid belt. We have carried out a large number of test particle integrations within the framework of the circular restricted three-body problem and have studied a variety of resonant and chaotic orbits between the Sun and Jupiter. We extensively discuss the software that was used to identify these orbits and also discuss possible applications, where the phenomenon of orbital chaos is leveraged to provide low-fuel cost trajectories between widely separated solar system destinations.

Contents

List of Figures

To Mom and Dad

Acknowledgements

I would like to thank Greg Laughlin for leading the way through the project.

1 Introduction

The circular restricted three-body problem presents a simple and elegant example of a chaotic system. No analytic closed-form solution exists, the three-body problem must be numerically integrated to determine the trajectories. Herein, we focus on a particular example of the circular restricted problem. We study the motion of a small-mass object, such as an asteroid, in the presence of the Sun and Jupiter. Because the mass is so small, we can simplify the problem without affecting the dynamics by considering it to be massless. In particular, all of the nonlinearities and resonant dynamics are retained. We give special attention to the construction of Poincare surfaces-of-section, which we generate by plotting the asteroid's x-position versus it's x-velocity every time the asteroid crosses the x-axis. By doing this for a specific energy, we are receiving a cross-section of its possible trajectories and their relative starting vectors. This cross-section reveals a highly intricate fractal structure, and delineated the intricate resonant structure of the possible orbital motions. The presence of chaotic regions allows for the interesting possibility of low-cost transport that effectively trades fuel costs for increased travel time. This method of travel is separate from the well-known method of using Lagrangian points to navigate around the solar system and the two methods could be used in junction with each other.

1.1 Sun-Jupiter System

For ease of calculation, the we will let the mass of the Sun and Jupiter respectively be $M_s = 0.9990001$ and $M_j = 0.000999$, keeping $M_s + M_j = 1$ and $M_j = M_s/1000$. The center of

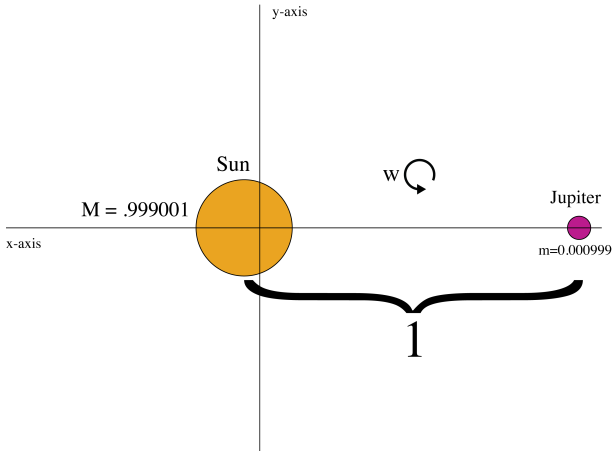


Figure 1.1: Diagram of the simulation. The test particle can start anywhere in between the Sun and Jupiter

mass is fixed at the origin, which puts the Sun at $x_s = -0.000999$ and Jupiter at $x_j = 0.9990001$. A circular orbit corresponds to angular speed $\omega = 1$. To simplify things further, we will be looking at the system in the co-rotating frame with Jupiter's orbit so that the Sun and Jupiter appear motionless. The total acceleration a on the particle is then $a = a_s + a_j + a_{centrifugal} + a_{coriolis}$, and the full equations of motion become:

$$\dot{x} = v_x,$$

$$\dot{y} = v_y,$$

$$\ddot{x} = M_s(x_s - x)/d_s^3 + M_j(x_j - x)/r_j^3 + 2v_y\omega + x\omega^2,$$

$$\ddot{y} = M_s(-y)/d_s^3 + M_j(-y)/r_j^3 + 2v_x\omega + y\omega^2.$$

Note that, conveniently, these are time-independent equations. By fixing the Sun and Jupiter's positions, we effectively reduced the number of independent variables from twelve to four.

1.2 Energy Conservation in the Sun-Jupiter System

When using a massless particle, we must use an alternative formula for energy conversation. The constant that is conserved instead, is called the Jacobian Constant, C_j. In the rotating frame,

$$C_j = n^2(x^2 + y^2) + 2\left(\frac{m_{jup}}{r_{jup}} + \frac{m_s}{r_s}\right) - \dot{x}^2 - \dot{y}^2,$$

where $n = T_{jup}/2\pi = 1$.

If we start our calculation on the x-axis with $\dot{x} = 0$, at a specific energy Cj, we can solve for $\dot{y}$ for a given x_0:

$$\dot{y}^2 = n^2x^2 + 2\left(\frac{m_{jup}}{r_{jup}} + \frac{m_s}{r_s}\right) - C_j.$$

With this equation we can sample different trajectories with the same energy C_j.

1.3 Preliminary Exercises

In order to gain an understanding of numerical integration of a three-body system, an integrator was coded from scratch. The integrator created was based on the standard forth-order Runge-Kutta equations:

$$y_{n+1} = y_n + h/6(k_1 + 2k_2 + 2k_3 + k_4),$$

$$k_1 = f(t_n, y_n),\ k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1),$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2),\ k_4 = f(t_n + h, y_n + hk_3).$$

Where h is the timestep size, and y_{n+1} is the next step in the integration.

1.3.1 Dynamic Step Size Method

Due to the nature of chaotic trajectories, integration must adhere to a high order of accuracy, resulting in a ultra small step size, and a grudgingly slow integrator. To speed up the

integration, we can take larger timesteps when we can, and take small steps when we need are navigating a close encounter. The dynamic step size algorithm takes two timestep sizes and compares them to a desired level of tolerance T. Let $f(h)$ be the calculated location of the particle for a timestep h. Then we can evaluate the accuracy of our timestep by evaluating $\Delta = T/|f(h) - f(h')|$. If this ratio is less than a small tolerance, we accept the step, and increase h for the next step by a small amount. if the ratio is greater than our tolerance, we reject the step and decrease the size until it is accepted. The following formula implements this methodology:

$$h' = h * \Delta^{0.20} \text{ if } \Delta < 1,$$

$$h' = h * \Delta^{0.25} \text{ if } \Delta > 1.$$

1.3.2 The Bulirsch-Stoer Method

The dynamic step size increases speed and accuracy, but the speed can be further increased using the Bulirsch-Stoer Method (BS). The BS-method integrates each step with numerous decreasing step sizes, i.e. $h = h_0, h_0/2, h_0/4, h_0/6, ...$, and calculates them all to the same time t. At t, the resulting vector of the integration is then considered a function of the step size, $f(h) = x_{calc}$ such that $f(0) = x_{ideal}$. By sampling different values of h we can fit a polynomial to the points, and evaluate $f(0)$. This method proves to be more accurate than the adaptive step size RK-method and is faster due to the giant steps that it can take.

Although the BS-method created is functional, the speed and accuracy is no match for the robust SciPy integrator, which will be used for further calculation.

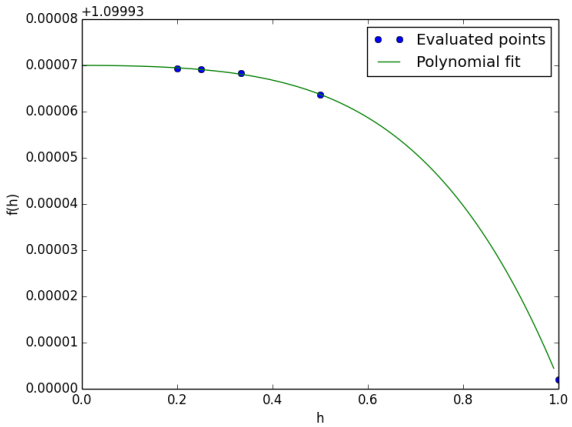


Figure 1.2: Visualization of one Bulirsch-Stoer step. The method integrates with varying step sizes of h, and computes a polynomial fit to the points. The final vector taken from this step is the extrapolated value at $h = 0$.

2 Code

All code for this project was coded in the Python language. The methods listed here are the important functions that the program uses.

2.0.3 odeint

Supplied by the Scipy Library, this is the function that integrates the equations of motion. Most of the program running time is spent inside this function. It generates a trajectory matrix of the vectors at each timestep given to it. The maximum error tolerance can be given to the function, which for all simulations is set to 10^{-12}.

2.0.4 locateCrossing

This crucial function analyzes a trajectory matrix, finding when the test particle crosses the $y = 0$ line, and interpolates the trajectory to find the position and velocity. The first part of the function scans through the entire matrix looking for when $y_i < 0$ and $y_{i+1} > 0$, which is where the particle passes the x-axis on the positive side. Once this is located, it performs an integration from t to $t + h$, consisting of 128 timesteps. Next, it locates the crossing again using a number of comparisons on the 128 timesteps, assuming that there is only one crossing in the calculated trajectory. First, $y_{64} > 0$ is checked. If true, then the crossing must be located in $i < 64$ and the successive comparison checks if $i_{32} > 0$. If not, then the following comparison is $y_{96} > 0$. By repeating this algorithm, the step we can quickly interpolate where the body crosses the x-axis.

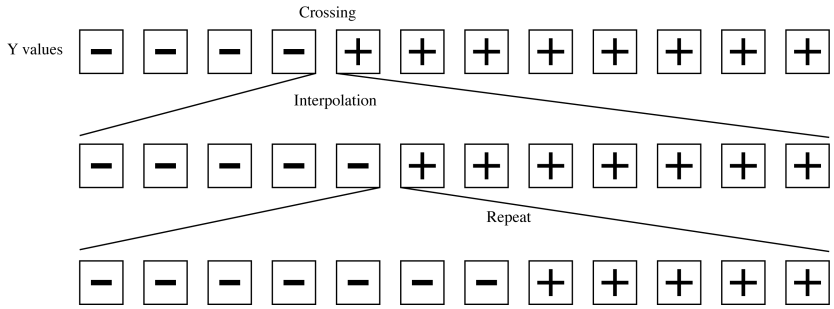


Figure 2.1: A diagram of how the crossing locator works. Each block represents a y-value in the trajectory. At the point where it turns negative to positive, the method interpolates between the two values and repeats.

Finally, the function returns a matrix containing the vectors of crossings.

2.0.5 orbitTimer

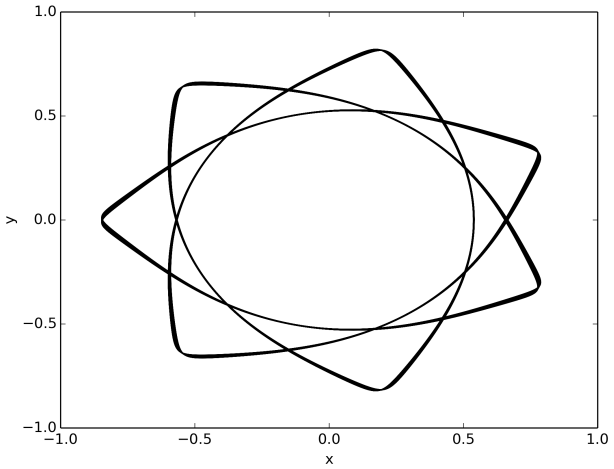
This function runs a quick integration of time T for an initial condition, and counts the number of crossings n using locateCrossing. It then returns the average orbit time n/T. By doing this we can attempt to take as large step as we can for efficiency, but small enough to have the orbit counter to register the orbit.

3 Poincare Maps

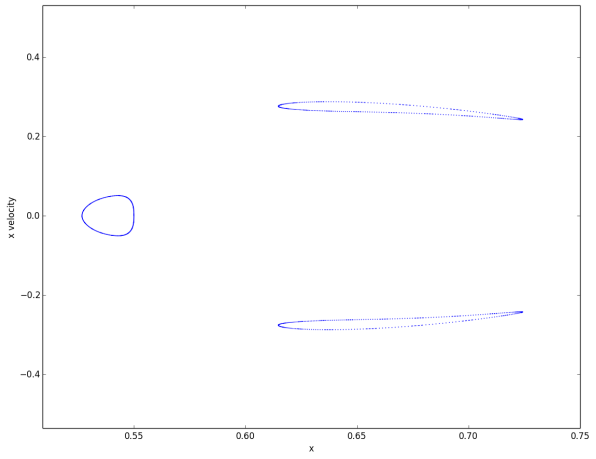
When the orbit is viewed in the rotating frame, we see that a pattern is present (Fig. 3.1a). This is a resonant orbit; the path of the particle repeats itself after a finite number of orbits. To view this more informatively, we plot a point $(x, \dot{x})$ in phase space every time the particle passes the $y = 0$ line (Fig. 3.1b). In the non-rotating frame, this is equivalent to plotting a point every time the particle passes Jupiter.

With this view, we can see the orbit creates three islands in the space. Note that one trajectory visits all islands sequentially, plotting one point in one island, then jumping to the next, rather than tracing each island one at a time. Different resonant orbits correspond to different sets of islands, which correspond to different integer multiples of the ratio of Jupiter's period with the particles. The orbital periods of Jupiter and the test particle, T_j and T_t, are related to their orbital distances r by $T^2 \propto r^3$, so $\frac{T_j^2}{T_t^2} = \frac{r_j^3}{r_t^3}$, $\frac{T_j}{T_t} = \frac{1}{0.54^{3/2}} = 5/2$. Murray et al. describes that the number of islands can be predicted using the ratio of periods $p + q : p$, where q is the number of islands In our case it is $5/2 = 2 + 3 : 2$, so there are 3 islands.

Let us now look at a chaotic orbit. Slightly modifying our starting condition to $(x_0, C_j) =$ (0.50, 3.07), our trajectory in the plane has lost its predictability and covers a large area of the possible space (Fig. 3.2a). Viewing the same trajectory in phase space, we see an intricate structure revealed (Fig. 3.2b). While resonant orbits create paths in the map, uncrossed by other starting conditions, chaotic orbits occupy a volume shared by other chaotic orbits. The chaotic orbits do not penetrate the resonant zones and as a result the resonant zones are outlined by chaotic zones.

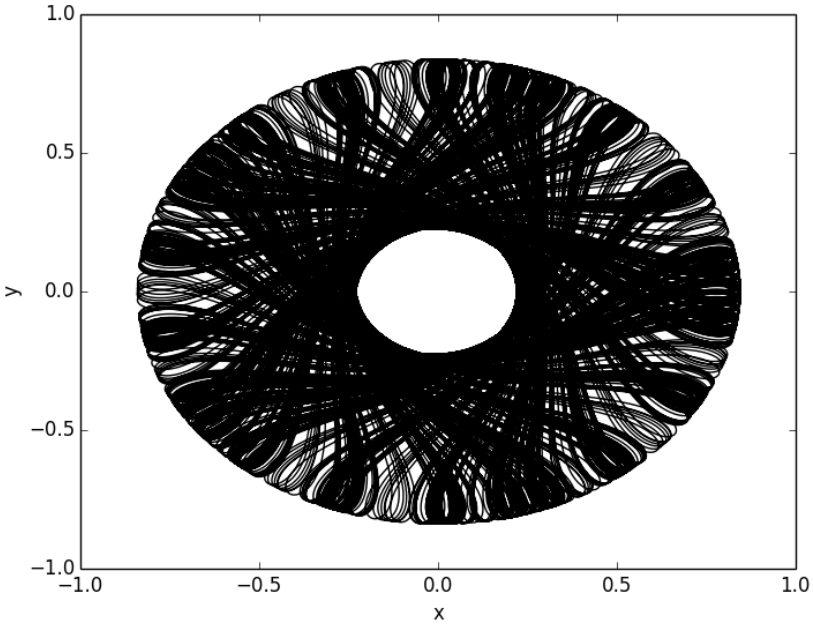


(a) Trajectory of the test particle in the rotating reference frame. Note the three different crossings that happen on the positive part of the x-axis (two happen at the same x-position, but differing velocities). These correspond to the 3 islands in the Poincare mapping.

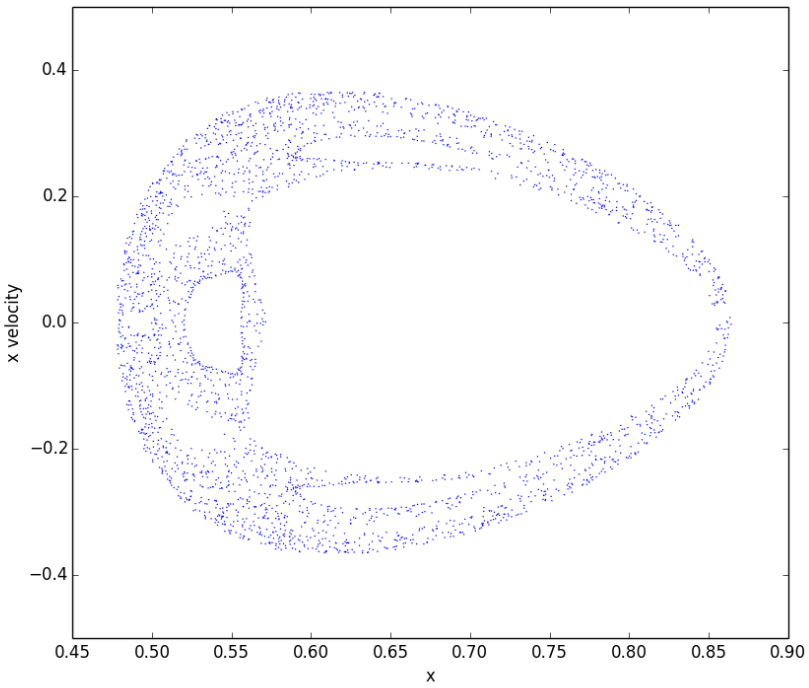


(b) Poincare map of the particles x-axis crossings.

Figure 3.1: The trajectory and corresponding map for $x_0 = 0.54$, $C_j = 3.07$.



(a) Trajectory of a chaotic orbit in the rotating frame. Given more calculation time, the donut-shape would appear completely solid.



(b) Phase space plot of the chaotic orbit.

Figure 3.2: The trajectory and corresponding phase space diagram for $x_0 = 0.50$, $C_j = 3.07$.

Each rational fraction $\frac{T_j}{T_t} = \frac{p}{q}$, $p, q \in \mathbb{N}$ creates a resonant zone in the map. The chaotic zone then takes up all of the irrational values, generating an infinitely detailed fractal structure.

3.1 Connectivity of Chaotic regions

Let us look at a highly detailed Poincare map of the chaotic zone (Fig. 3.3). This map contains 12 different starting conditions, clustered into 3 regions. With high resolution, we notice that the resonant zones are surrounded by a honeycomb of regions (Fig. 3.4). In this figure, the honeycomb ridges are occupied mainly by the teal starting condition, yet teal and the other conditions inhabit the same space for the majority of orbits. Similarly, looking at Figure 3.5, we can see that the region on the left-hand side is shared by all starting conditions, whereas the region on the right is occupied by only one.

On the right-hand side of Figure 3.3, the outer (red) and and middle (purple) zones appear to become sparse as x positively increases. The empty region just outside the elliptical shape is inaccessible, and the density of different zones increases in the narrowing region. Starting trajectories in this region reveals that there is no space between the large middle chaotic zone and the denser thin band (Fig. 3.6). Starting trajectories as close as we can to the ridge between the two regions shows that the chaotic trajectory begins in a seemingly resonant orbit, and after a period of time falls into chaos (Fig. 3.7). The closer we get to the resonant zone in our starting condition, the longer the orbit is resonant before becoming chaotic. It isn't hard to imagine that this pattern would continue indefinitely the closer we get to the resonant zone. Perhaps all resonant orbits then will all eventually break into chaos, given long enough. Our computational accuracy may not be accurate enough to derail most resonances.

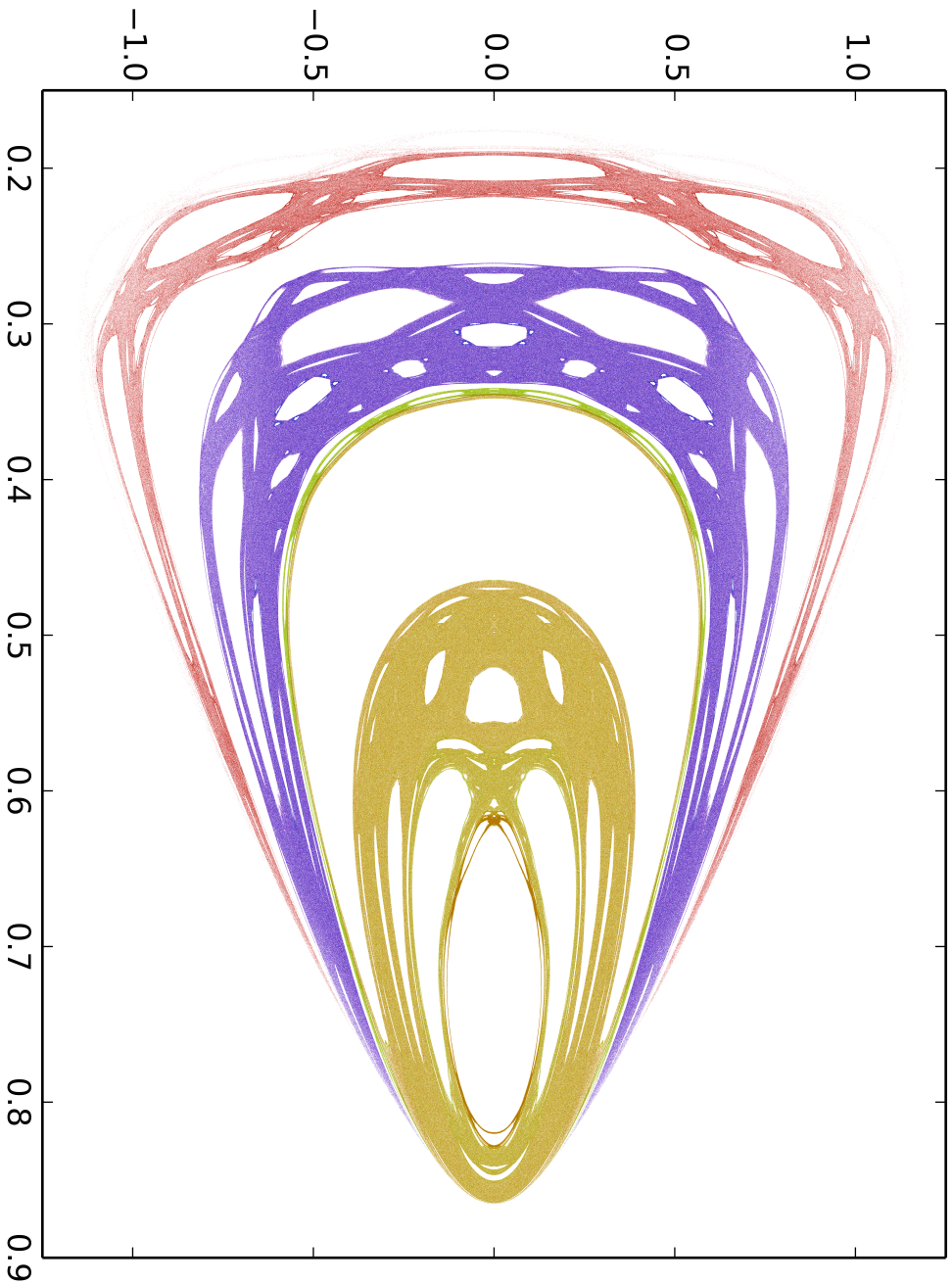


Figure 3.3: 10^7 orbits of twelve different starting conditions in $Cj = 3.07$. The twelve conditions were clustered into three regions, creating the three large chaotic regions. The plot was reflected on the x-axis to double the density of points.

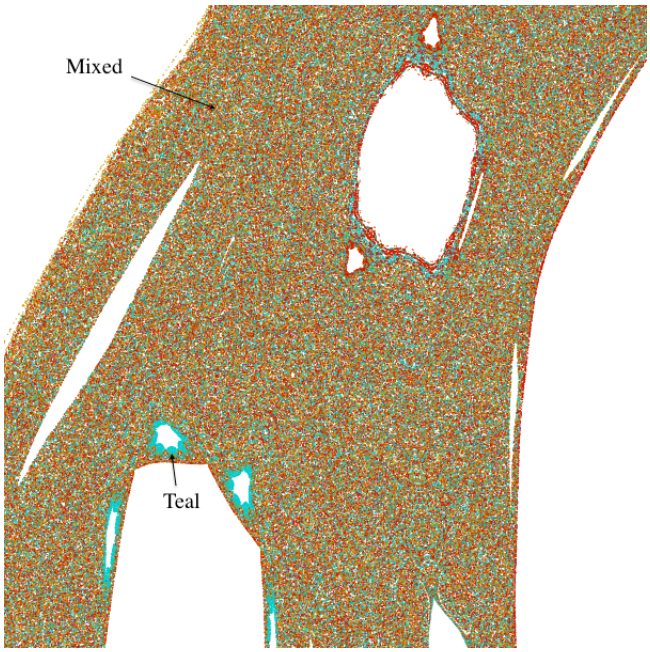


Figure 3.4: Enlarged section of Figure 3.3 (colored differently). Different colors refer to four different starting conditions. Note the honeycomb pattern that teal occupies.

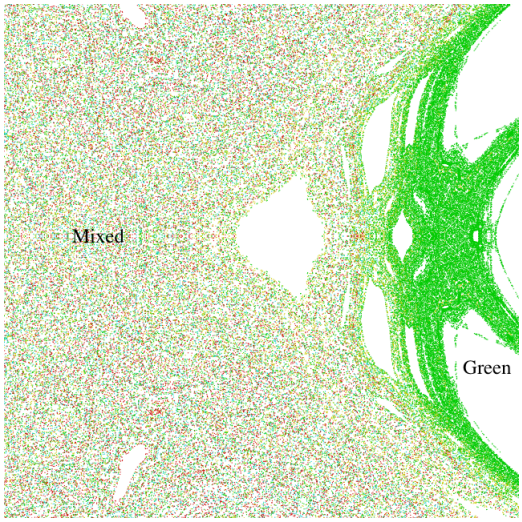


Figure 3.5: Another enlarged section of Figure 3.3. Notice the structure on the right side is occupied entirely by green, yet green also occupies the the space shared with other colors.

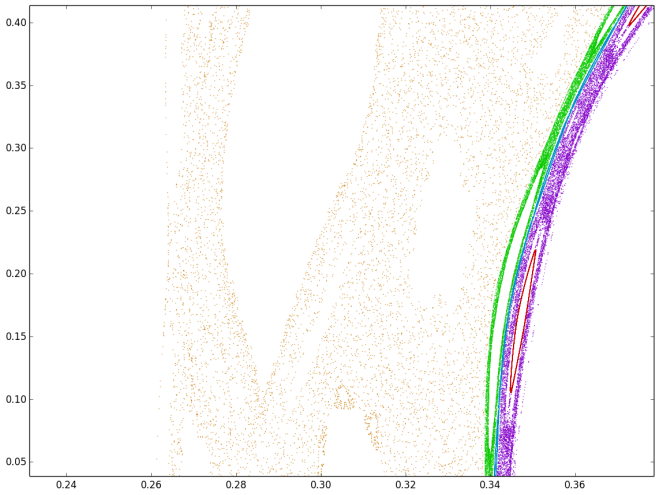


Figure 3.6: Close up of the region between a large chaotic zone and a band of various zones. The five starting conditions were equally spaced apart from each other.

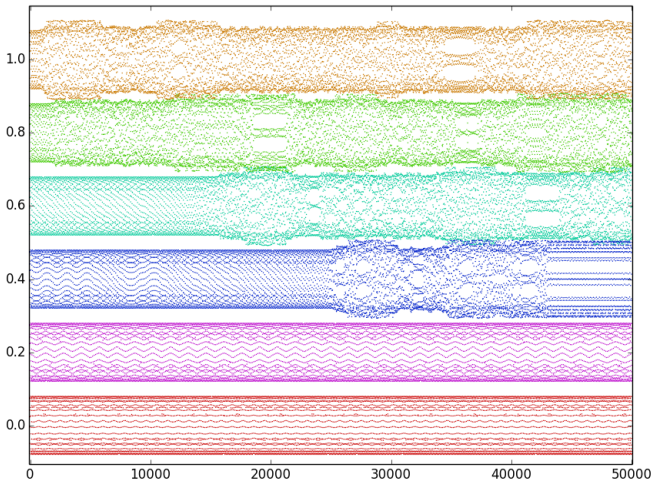


Figure 3.7: X-velocity vs time for close-resonant orbits. Plots are offset to show different starting conditions. Increasing on the y-axis is increasing distance from the resonant zone. Notice how the closest trajectory to the resonant zone takes the longest to enter into chaos.

3.2 Traveling across the zones

The topographical features of the Poincare surface brings up the possibility of low-energy interplanetary travel by hopping between zones. Referring to Figure 3.3, we see that for example, the red orbit spans almost the entire distance between the Sun and Jupiter. A traveler could reside in a chaotic zone and visit points all around the space. Once the traveler is located at the desired distance between the Sun and Jupiter, a careful shift in velocity could put them into a resonant zone. The only energy required then is move from zone to zone and the majority of the travel requires no energy. The drawback to this option is the time required to be in the desired location. The sensitivity to starting conditions means that chaotic zones are a gamble, and could take unpredictable amount of time to reach the zone. Another option would be hop just between resonant zones, but these cover significantly less space and could require more energy to get to the next zone.

3.2.1 Lagrangian Points

A separate method for low-cost interplanetary travel is using Lagrangian points. Lagrangian points are peculiar points of the constricted three-body problem that allow for slow but cost efficient travel (Fig. 3.8). The Lagrangian points are points of gravitational equilibrium; points L_4 and L_5 are fully stable whereas L_1, L_2, and L_3 are only semi-stable, sustaining carefully placed orbits for only a relatively short amount of time. By utilizing different Lagrangian points around the solar system a transport path can be mapped out. One could combine this method of travel with the previously described resonant hopping to navigate around the system.

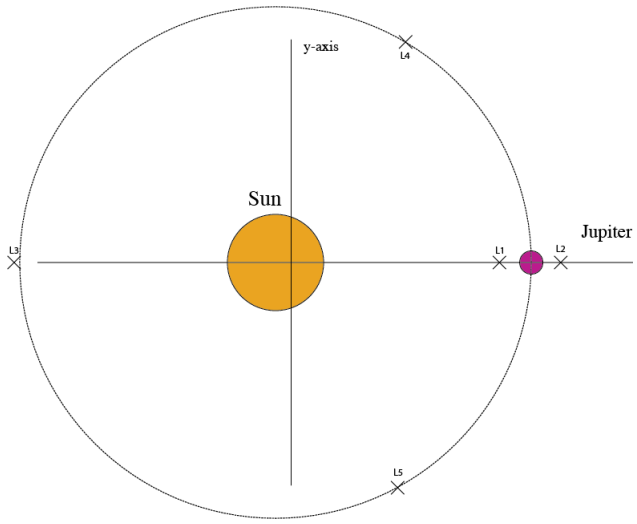


Figure 3.8: The five Lagrangian points of the Sun-Jupiter System. L_4 and L_5 are stable, where as the others are not.

4 Conclusion

The Poincare map of the restricted three-body problem reveals an interesting topography. The chaotic orbits tend to exhibit a degree of locality to them, yet share a large space with other chaotic orbits. The definition of chaotic and resonant zones is difficult when looking close at the edges of these zones. Perhaps resonant zones are in fact chaotic given enough time, and the only true resonances are single points at the center of the resonant zones. By knowing the layout of the Poincare map, one could hop zones and travel a large amount of space with very minimal energy. The trade-off is the unpredictably long time that it may take to reach the desired location. In addition to this interplanetary travel, utilizing the Lagrangian points around the solar system could maximize opportunity.

5 Appendix: Full Code

```
# Restricted three-body Poincare Mapper
# Alex Rowe
# University of California, Santa Cruz, 2014
# Python 3.4
# Repository can be found at
# https://github.com/aprowe/ThreeBodyPoincare

from numpy import *
from itertools import product
from multiprocessing import pool
from scipy import linspace

import scipy.integrate as scp
import matplotlib.pyplot as plt
import time, sys

m_s = .999001 # Mass of the Sun
m_j = .000999 # Mass of Jupiter

r_s = -m_j    # Position of Sun
r_j = m_s      # Position of Jupiter

w_j = 1       # Angular speed of Jupiter

orbits = 100  # Number of orbits to run the simulation for
tol = 1e-12      # Error Tolerance

pools = 6     # Maximum number of threads to use

filename = 'data.dat'  # File to either contnue or start

if len(sys.argv) > 1:
    filename = sys.argv[1]

if len(sys.argv) > 2:
    orbits = int(sys.argv[2])

if len(sys.argv) > 3:
    pools = int(sys.argv[3])


def main():

    x0 = [ (0.5,3.07) ] # List of starting conditions to test

    # x0 = continueCalc() # Uncomment to continue a calculation

    p = pool.Pool(pools)
    p.map(orbitCalc, x0)


# Continues a calculation alread started. Takes the filename and
# constructs a list of trajectories to continue.
def continueCalc():
    print("Continuing calculation for",filename)
    print("Adding",orbits,"orbits")

    data = loadtxt('data/'+filename)

    x0 = data[-1][5]
    cj = data[-1][6]

    init_vectors = list()
    counted = list()
    mx=0
```

```
    for row in reversed(data):
        if x0 >= mx:
            mx=x0
        x0 = row[5]
        cj = row[6]
        if not (x0,cj) in counted:
            counted.append((x0,cj))
            init_vectors.append(row)

    return init_vectors


# Calculates a particular initial condition and saves it to a file
# init - (x0, Cj) a tuple containing the initial starting x and Cj value
# init - x optionally use a starting vector with x0 and cj at end of vector
# init - (x0, Cj, x) optionally use an initial vector at end of tuple
# orbits - the rough number of orbits to calculate
# tolerance - integrator tolerance
def orbitCalc(init):
    if(len(init)==7):
        x = init[0:5]
        x0 = init[5]
        cj = init[6]

    if(len(init)==2):
        x0 = init[0]
        cj = init[1]
        x = X( x0, cj )

    if len(init)==3:
        x0 = init[0]
        cj = init[1]
        xv = init[2]
        x= X(x0,cj,xv)

    o_time = orbitTimer(x) # time the average orbit

    h = o_time/10 #cuts the orbit into 10 pieces (minimum should be 4 or 5 but 10 is safe)
    T = orbits*o_time ## final time

    print('Calculating x0=',x0,', Cj=',cj

    t = time.time() # time the process

    x_mat = ode(x,h,T) # magic happens here

    print('Counting orbits...')

    orbit_mat = locateCrossing(x_mat)
    #orbit_mat = vstack([x_mat[0],orbit_mat])

    print(len(orbit_mat)," orbits found")
    print("Took ",time.time()-t,"seconds")

    orbit_mat = col_append( orbit_mat, x0 )
    orbit_mat = col_append( orbit_mat, cj )

    print("Appending to" ,filename);

    f = open('data/'+filename, 'ab')
    savetxt(f, orbit_mat, delimiter=' ')
    f.close()

    return orbit_mat
```

```
132 # Calculates Cj for debugging purposes
133 def Cj(x):
134 
135     ds = dist([ x[0],x[1] ], [ r_s, 0 ])
136     dj = dist([ x[0],x[1] ], [ r_j, 0 ])
137 
138     C = 2 * (m_s/ds+m_j/dj)
139     C += x[0]**2 + x[1]**2
140     C += -(x[2]**2+x[3]**2)
141 
142     return C
143 
144 # Solves for a particular y-velocity and Cj
145 def ydot(x0, Cj, xd=0):
146 
147     r1 = x0-r_s # distance from sun
148     r3 = r_j-x0 # distance from jupiter
149 
150     T= 2*pi/w_j
151     n = 2*pi/T
152 
153     Yd = sqrt( - Cj
154         + n**2 * x0**2
155         + 2*(m_s/r1 + m_j/r3)
156         - xd**2)
157 
158     return Yd
159 
160 
161 # This sets up an initial vector
162 # x: inital x value
163 # Cj: Cj constant
164 # vy: use if you want to specify a specific vy
165 def X(x0, Cj=None, vx=0):
166     y=0
167     vy=ydot(x0,Cj,vx)
168 
169     return array([x0,y,vx,vy,0])
170 
171 # This is a shortcut for the ode call
172 # x - vector
173 # h - stepsize
174 # final_time - time to stop at
175 # tol - Error
176 def ode(x, h, final_time):
177     space = linspace(0,final_time,final_time/h)
178     mat = scp.odeint(f, x, space, atol=tol, rtol = tol*2)
179     return mat
180 
181 # Method called by the integrator
182 def f(x,t):
183     X  = x[0]
184     Y  = x[1]
185     VX = x[2]
186     VY = x[3]
187 
188     ds = dist([ X,Y ], [ r_s, 0 ])
189     dj = dist([ X,Y ], [ r_j, 0 ])
190 
191     grav_j = m_j / dj**3
192     grav_s = m_s / ds**3
193 
194     return(array([
195         VX, # velocity body1
196         VY,
197 
```

```
198           grav_s * (r_s-X) # x-gravity from sun
199         + grav_j * (r_j-X) # x-gravity from jupiter
200         + 2*w_j*VY         # y-coriolis force
201         + w_j**2*X         # y-centrifugal force
202         ,
203
204           grav_s * (0-Y) # y-gravity from sun
205         + grav_j * (0-Y) # y-gravity from jupiter
206         + -2*w_j*VX      # y-coriolis force
207         + w_j**2*Y ,     # y-centrifugal force
208         1                # time
209         ]))
210
211
212 # Locates all crossings of the y axis in a trajectory matrix
213 # requires a positively oriented rotation (ccw)
214 def locateCrossing(x_mat, precision = 1e-6):
215
216     def locate(tup):
217         v1 = tup[0]
218         v2 = tup[1]
219
220         if v1[1] <= 0 and v2[1] > 0 and v1[3] > 0 and v2[3] > 0:
221             return locateZero(v1, v2, precision)
222         else:
223             return None
224
225     # Copy the matrix without the first entry to shift it over
226     shifted_mat = delete(x_mat,0,0)
227
228     mat = zip(x_mat, shifted_mat) # create a list of tuples
229     mat = map(locate, mat)
230
231     return array([zero for zero in mat if zero is not None])
232
233
234 # Interpolates a zero to a tolerance, called witin locate crossing
235 # x_mat1 - Vector with a negative y
236 # x_mat2 - Vector with a positive y
237 # prec - the zero will be located to within this precision
238 def locateZero(x_mat1, x_mat2, precision):
239
240     # if x_mat1 and 2 are in the in tolerance range, return the interpolation between them
241     if( abs(x_mat1[1])+abs(x_mat2[1]) <= precision):
242         return (x_mat1+x_mat2)/2.0
243
244     # Starting and ending time
245     t = x_mat1[4]
246     t2 = x_mat2[4]
247
248
249     # Calculate the intermediate grid, cut up into 128 pieces
250     x_mat = scp.odeint(f, x_mat1, linspace(t,t2,128), atol = tol*100, rtol = tol*1000)
251
252     # get the y values
253     x = x_mat.T[1]
254
255     # Error message if somehow there is no zero
256     # Try reducing precision if this occurs
257     if not (x[0] <= 0 and x[-1] > 0):
258         print("Zero Not found Error")
259         return None
260
261     # This part quickly locates a single zero in the matrix
262     w = len(x_mat)
263     i = 0
```

```
264     while w != 1:
265          w = int(w/2.0)
266          if (x[i+w]<0):
267              i += w
268 
269     return locateZero(x_mat[i], x_mat[i+w], precision)
270 
271 
272 # Estimates the time of an orbit
273 # x - initial vector
274 # time - time to run test for
275 # Returns time per orbit
276 def orbitTimer(x,  time = 100):
277     h=0.2
278     x_mat=ode(x,h,time)
279 
280     orbit_mat = markOrbit(x_mat)
281 
282     return time/len(orbit_mat)
283 
284 # Traces the trajectory visually for debugging purposes
285 def trace(x,cj, time=1000, name=None, xd=0):
286     result = ode(X(x,cj,xd),.01,time)
287     plt.plot(result.T[0],result.T[1],color=(0,0,0))
288     plt.xlabel("x")
289     plt.ylabel("y")
290     plt.legend()
291 
292     if name is not None:
293         plt.savefig("figures/"+name,dpi=300)
294     plt.show()
295     plt.clf()
296 
297 # Simple method to add a column to a matrix
298 def col_append(x,a):
299     tmp = append(x[0],a)
300 
301     for row in x:
302         row = append(row,a)
303         tmp = vstack([tmp,row])
304 
305     tmp = delete(tmp,0,0)
306     return tmp
307 
308 # Simple method to find distance between two vectors
309 def dist(a1,b1):
310     a = array(a1)
311     b = array(b1)
312     c = linalg.norm(a-b)
313     return c
314 
315 
316 if __name__=='__main__' :
317     main()
```

Bibliography

[1] Dermott, S.F.,& Murray, C.D. *Solar System Dynamics*. Cambridge University Press, 1999.

[2] Press, W.H., Teukolsky, S.A., Vetterling, W.T., & Flannery, B.P. *Numerical Recipes*, Third Edition. Cambridge University Press, 2007.

[3] Ross, S. D. "The Interplanetary Transport Network". American Scientist 94: 230-237. doi:10.1511/2006.59.994.