Swinging Atwood's Machine

Michael Kister, Alli Behr

Introduction

In our project, we investigated the behavior of the swinging Atwood's machine in the context of chaos theory. We used the Hamiltonian from classical mechanics. Once we found the differential equations, we explored the physics of the system with the use of a novel form of a bifurcation diagram as well as Poincaré maps.

Lagrangian Function

Before beginning with our use of the Hamiltonian, we should first mention the Lagrangian functions, where the Lagrangian is defined as:

$$\mathcal{L} = \mathcal{L}(q_1, \ldots, q_n, \dot{q_1}, \ldots, \dot{q_n}, t) = T - U$$

where:

$$\frac{\partial \mathcal{L}}{\partial q_i} = \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q_{ix}}}$$

furthermore:

$$p_i = \frac{\partial \mathcal{L}}{\partial \dot{q}_i}$$

Hamiltonian Function

We now define the Hamiltonian to be:

$$\mathcal{H} = \sum_{i=1}^{n} p_i \dot{q}_i - \mathcal{L}$$

Noting that we want to keep:

$$\mathcal{H} = \mathcal{H}(q_1, \ldots, q_n, p_1, \ldots, p_n)$$

In our case, because we are dealing with a closed system and conservation of energy, we can say that:

$$\mathcal{H} = T + U$$

where the Hamiltonian equations are:

$$\dot{q}_i = \frac{\partial \mathcal{H}}{\partial p_i} \qquad \text{and} \qquad \dot{p}_i = -\frac{\partial \mathcal{H}}{\partial q_i}$$

Swinging Atwood's Machine

For our system:

$$\begin{aligned} T &= \frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_1 v_2^2 \\ &= \frac{1}{2} m_1 \left(\dot{r_1}^2 + (r_1 \dot{\theta_1})^2 \right) + \frac{1}{2} m_2 \left(\dot{r_2}^2 + (r_2 \dot{\theta_2})^2 \right) \\ U &= m_1 \, g \, \left[r_1 \left(1 - \cos\theta_1 \right) + (R - r_1) \right] + m_2 \, g \, \left[r_2 \left(1 - \cos\theta_2 \right) + (R - r_2) \right] \end{aligned}$$

However, for our system,

$$\begin{aligned} r_2 = R - r_1 \qquad &\text{and} \qquad \dot{r_2} = -\dot{r_1} \\ \dot{\theta_2} = 0 \qquad &\text{and} \qquad \theta_2 = 0 \end{aligned}$$

where R is the combined length of the two string lengths. Therefore, we say that

$$\mathcal{H} = \frac{1}{2} m_1 \left(\dot{r_1}^2 + (r_1 \dot{\theta_1})^2 \right) + \frac{1}{2} m_2 \dot{r_1}^2 - m_1 \, g \, r_1 \, \cos\theta_1 + m_2 \, g \, r_1$$

or

$$\mathcal{H} = \frac{1}{2} m_1 \left(\dot{r}^2 + (r \, \dot{\theta})^2 \right) + \frac{1}{2} m_2 \, \dot{r}^2 - m_1 \, g \, r \, \cos\theta + m_2 \, g \, r$$

Now we return to our requirement that $\mathcal{H}$ be in terms of $(q_1, \ldots, q_n, p_1, \ldots, p_n)$. To resolve this issue, we note that:

$$
\begin{aligned}
p_r = \frac{\partial \mathcal{L}}{\partial \dot{r}} &= \frac{\partial T}{\partial \dot{r}} \\
&= \frac{\partial}{\partial \dot{r}} \left(\frac{1}{2} m_1 \left(\dot{r}^2 + (r\,\dot{\theta})^2 \right) \, + \, \frac{1}{2} m_2 \, \dot{r}^2 \right) \\
&= \dot{r}(m_1 + m_2) \implies \\
\dot{r} &= \frac{p_r}{(m_1 + m_2)}
\end{aligned}
$$

and that:

$$
\begin{aligned}
p_\theta = \frac{\partial \mathcal{L}}{\partial \dot{\theta}} &= \frac{\partial T}{\partial \dot{\theta}} \\
&= \frac{\partial}{\partial \dot{\theta}} \left(\frac{1}{2} m_1 \left(\dot{r}^2 + (r\,\dot{\theta})^2 \right) \, + \, \frac{1}{2} m_2 \, \dot{r}^2 \right) \\
&= m_1 \, r^2 \, \dot{\theta} \implies \\
\dot{\theta} &= \frac{p_\theta}{m_1 \, r^2}
\end{aligned}
$$

We can thereby rewrite the Hamiltonian of the system as:

$$
\mathcal{H} = \frac{1}{2} m_1 \left[\left(\frac{p_r}{(m_1 + m_2)} \right)^2 + \left(r \, \left(\frac{p_\theta}{m_1 \, r^2} \right) \right)^2 \right] \, + \, \frac{1}{2} m_2 \, \left(\frac{p_r}{(m_1 + m_2)} \right)^2 \, - \, m_1 \, g \, r \, \cos\theta \, + \, m_2 \, g \, r
$$

which simplifies to:

$$
\mathcal{H} = \frac{p_r^2}{2(m_1 + m_2)} \, + \, \frac{p_\theta^2}{2 \, m_1 \, r^2} \, + \, m_2 \, g \, r \, - \, m_1 \, g \, r \, \cos\theta
$$

Using our Hamiltonian equations, we obtain:

$$
\begin{aligned}
\dot{\theta} = \frac{\partial \mathcal{H}}{\partial p_\theta} &= \frac{p_\theta}{m_1 \, r^2} & \qquad \dot{p_\theta} = -\frac{\partial \mathcal{H}}{\partial \theta} &= -m_1 \, g \, r \, \sin\theta \\
\dot{r} = \frac{\partial \mathcal{H}}{\partial p_r} &= \frac{p_r}{(m_1 + m_2)} & \qquad \dot{p_r} = -\frac{\partial \mathcal{H}}{\partial r} &= \frac{p_\theta^2}{m_1 \, r^3} \, - \, m_2 \, g \, + \, m_1 \, g \, \cos\theta
\end{aligned}
$$

Which are the differential equations we solved when simulating the swinging Atwood's machine.

Trajectories

Once we had the differential equations that characterized our system, we were able to solve them in ipython notebook. Given an initial set of conditions, we were able to plot the trajectories of the smaller mass for a variety of mass ratios. In Fig. 1 we compare our results with those found on Wikipedia. For this, we used our function `Solve_SAM` to get the solution of the differential equations, whereupon we simply plotted radius against angle.

From this comparison, we were able to convince ourselves that we were correctly solving the differential equations.

Bifurcation Diagram

A bifurcation diagram shows the conditions achievable for a system for a varying value. It is particularly useful for chaotic systems, where for many values of a variable, there are an infinite number of possible conditions. For our system, we came up with a novel form of bifurcation diagram where we compared possible differences in adjacent orbits.

To achieve this analysis, we first identified points where the radius reached local maximum values. We then noted that one "period" took place over the course of three of these maxima (one at the start, middle, and end). So for adjacent periods we calculated the sum in the differences of corresponding $r_i(t)$ values, where $r_i(t)$ is the temporal dependence of radius for the ith period. We calculated the sum of these differences for one hundred sets of adjacent periods for one thousand mass ratios, from which we obtained Fig. 2. For this section, we made us of our functions `differences` and `Bifurcation`.

From Fig. 2, we can clearly see mass ratios where the possible differences in orbits are more varied than for other ratios. The most notable examples of regions we could predict to be chaotic are between around 8 and 15, and past 24. On the other hand, we can see areas of very similar orbits at locations such as 7.5 and 22.5. In Fig. 3 we check the behavior of the system for mass ratios where the larger mass is 12 and 22.5 times larger than the smaller mass, and we can certainly see that the former is far more chaotic than the latter.

It should be noted that the system is not simply chaotic after $M/m > 24$, but continues to exit and reënter chaos for as long as we could see.

Phase Space

Once we (numerically) solve our set of four differential equations for $r(t)$, $\theta(t)$, $p_r(t)$, & $p_\theta(t)$, we have what could be called a four-dimensional phase space. However, since energy in our system is conserved (we are examining the case where the pulley is massless and frictionless), we can express any of the variables in terms of the other three. Therefore, a plot containing information about three of these values is sufficient for describing the state of the system. For this section, we made use of our functions `get_min_places` and `SOS_maps`.

The idea of a Poincaré is ultimately to reduce the dimensionality of the phase space so it is possible to examine. This is accomplished by examining two of the variables at points where the third variable is equal to a constant value. For our data, we examined

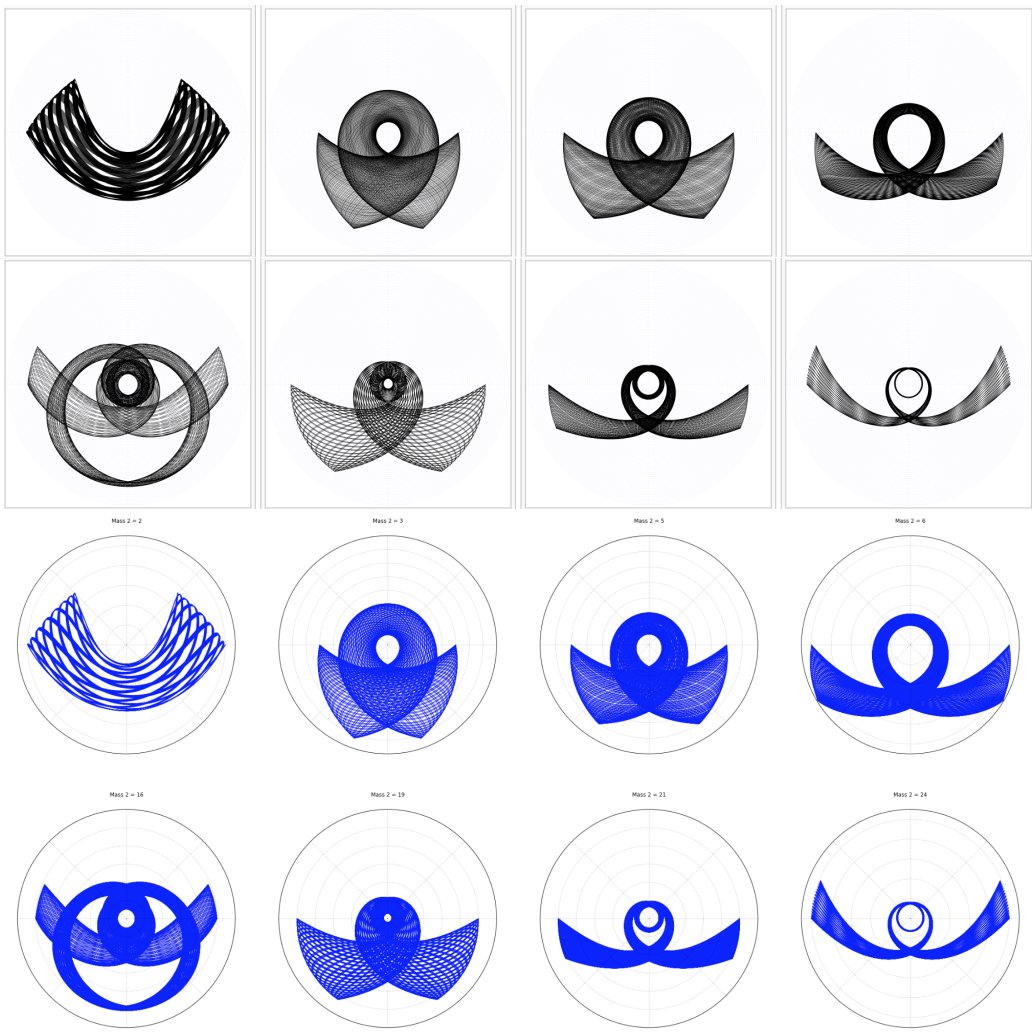

Figure 1: We compare the plots proffered by Wikipedia (gray) with our plots (blue) for corresponding mass ratios.

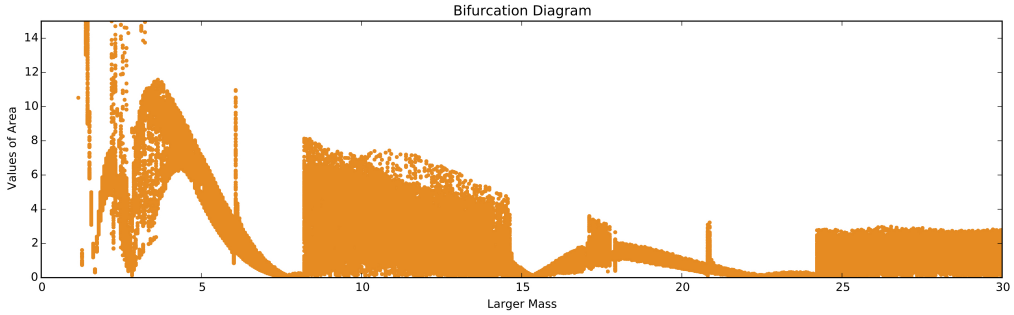

Figure 2: Our approach to a bifurcation diagram. Possible differences in orbits are plotted against a changing ratio of masses.

p_r vs r, for locations where $\theta = 0$. See Fig. 5 for some examples of the Poincaré maps we achieved in our simulation.

We note that these are different in appearance from some of the published maps, but ultimately we determined that since the sets of initial conditions were essentially infinite, and we were unable to find the precise conditions used in published works, that our differences rose from differing sets of initial conditions.

Lyapunov Exponents

Lyapunov Exponents are the exponents used to classify how chaotic a system is. These exponents specifically characterize the rate of separation of infinitesimally close trajectories in a dynamical system. It was desired to find these exponents in order to confirm the behavior of our bifurcation diagram. The equation for the change in a trajectory over time is given by:

$$|\delta Z(t)| \approx e^{\lambda t}|\delta Z_0|$$

where λ is the Lyapunov Exponent. This exponent gives an idea of the predictability of a system by giving the rate of separation of the trajectories. The maximum Lyapunov exponent for a system characterizes its overall behavior. If it is between 0 and 1, not including 0, it is considered chaotic and the greater it is, the more chaotic the system. The max Lyapunov exponent was found for our system using the equation:

$$\lambda_t \approx \frac{1}{t} \ln \frac{\delta Z_t}{\delta Z_0}$$

to find the exponents for each time and finding the maximum of all these to find the max exponent for a specific mass. This was done using the function This exponent for each mass is plotted below using the function `lyapunovExponents`. It is seen that the exponent is always positive, confirming the chaotic behavior of the machine. This plot has the same scale as our bifurcation diagram, so it can easily be compared. When done so,

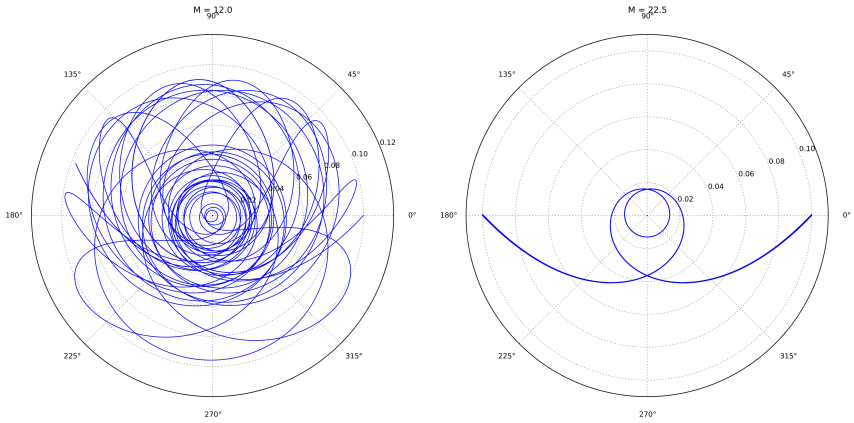

Figure 3: Two examples of trajectories. The plot on the left is for a mass ratio of 12, and the plot on the right is for a mass ratio of 22.5. These values were selected by using the bifurcation diagram to predict ratios that would be chaotic and non-chaotic.

it is seen that as more chaos is shown on the bifurcation diagram, the exponent increases, suggesting that our bifurcation accurately portrays the system's chaos.

Conclusion

Overall, we were satisfied with the information we could learn about the swinging Atwood's machine with the use of Python. Whereas solving differential equations is useful way to find an analytic solution when possible, our numerical simulations provided more insight by exploring situations in which analytic solutions are not an option.

One topic we remain curious about is an exploration of the system when it is not a closed system, for example when drag from the air, mass of the string, and friction of the string on the pulley are introduced. This would afford us a more accurate portrayal of reality, but will require much more math. We would also be interested in exploring how the concept of fractal dimensions plays a role in characterizing our phase plots.

References

[1] Antonio Politi (2013) `http://www.scholarpedia.org/article Lyapunov_exponent`

[2] N. Tufillaro. Motions of a swinging Atwood's machine. *Journal de Physique*, 1985, 46 (9), pp.1495-1500. 10.1051/jphys:019850046090149500. jpa-00210094

[3] Swinging Atwood's machine. (2014, June 15). In Wikipedia, The Free Encyclopedia. Retrieved 05:43, December 2, 2015, from `https://en.wikipedia.org/w/index.php?title=Swinging_Atwood`

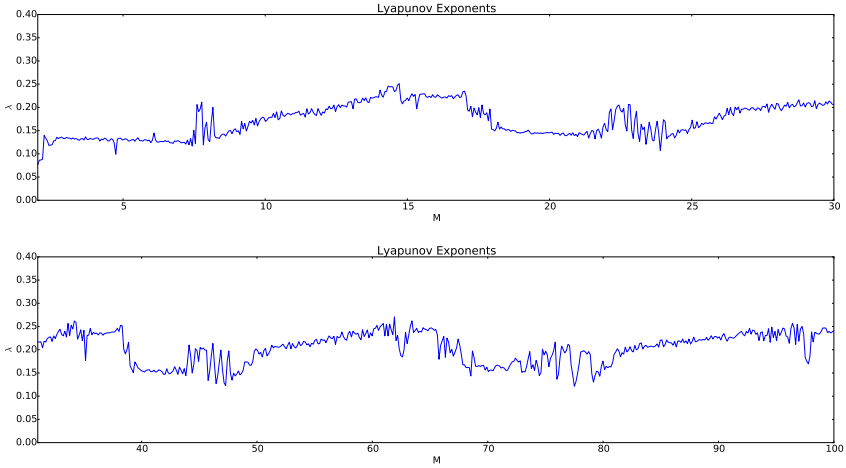

Figure 4: Here we show two excerpts of plotting the Lyapunov Exponent. The upper plot shows the Lyapunov Exponent for $M \in \{1 : 30\}$, while the lower plot shows $M \in \{31 : 100\}$

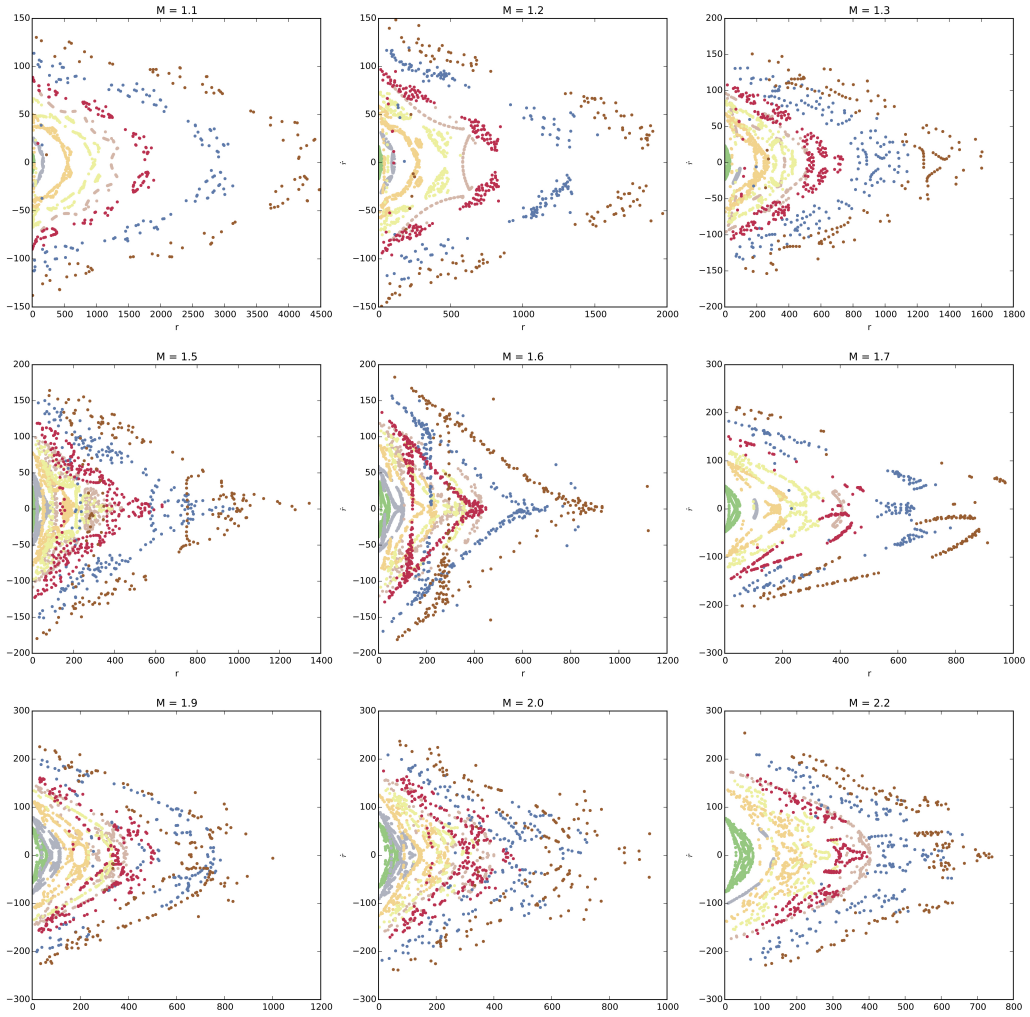

Figure 5: Nine examples of Poincaré maps for different mass ratios, each with various initial conditions.

```
# coding: utf-8

# # Solving Differential Equations for Swinging Atwood Machine (SAM)
# This is the code used for the final project. The equations we are solving are:

# $ \begin{align*} \dot{\theta} = \frac{\partial \mathscr{H}}{\partial p{\theta}}
    = \frac{p{\theta}}{m{1}\,r^{2}} \qquad&\qquad\qquad \dot{p{\theta}} = -
    \frac{\partial \mathscr{H}}{\partial \theta} = -m_{1}\,g\,r\,\sin{\theta} \
    \dot{r} = \frac{\partial \mathscr{H}}{\partial p{r}} = \frac{p{r}}
    {(m{1}+m{2})} \qquad&\qquad\qquad \dot{p{r}} = -\frac{\partial \mathscr{H}}
    {\partial r} = \frac{p{\theta}^{2}}{m{1}\,r^{3}} \,-\, m{2}\,g \,+\, m_{1}\,g
    \,\cos{\theta} \ \end{align*} $

# In[3]:

import numpy as np
import math
import scipy
from scipy.integrate import odeint
get_ipython().magic('matplotlib inline')
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import matplotlib.colors as colors
import matplotlib.cm as cmx
cmap = cmx.get_cmap('spring')
from scipy.signal import argrelextrema

cool = cm = plt.get_cmap('spring')
cNorm  = colors.Normalize(0., 31)
scalarMap = cmx.ScalarMappable(norm=cNorm, cmap=cool)

g = 9.8

M = np.array([1.0025, 1.01003, 1.02264, 1.04043, 1.06352, 1.09207,
              1.12621, 1.1661, 1.21184, 1.2635, 1.32107, 1.38444,
              1.4534, 1.52765, 1.60673, 1.69009, 1.7771, 1.86702,
              1.95908, 2.05248, 2.14642, 2.2401, 2.33278, 2.42379,
              2.51248, 2.59835, 2.68093, 2.75988, 2.83497, 2.90608,
              2.97321])
theta0 = np.linspace(.1, 3.1, 31)

#theta is the angle between the vertical and the swinging pendulum.
#r is the distance from the swinging mass to the point of rotation.
#p_theta is the angular momentum of the mass.
#p_r is the linear momentum of the swinging mass 2, pointing in the direction of
    the radius.
#m1 is the mass moving vertically
#m2 is the swinging mass.

#calculates change in theta
def dtheta(theta, r, p_theta, p_r, m1, m2):
    return p_theta / (m1 * (r**2))

#this is the change in radius caused by the vertical motion of m1
def dr(theta, r, p_theta, p_r, m1, m2):
```

```
    return p_r / (m1 + m2)

#change in rotational momentum
def dp_theta(theta, r, p_theta, p_r, m1, m2):
    return (-1 * m1 * g * r * np.sin(theta))

#change in linear momentum due to changing radius
def dp_r(theta, r, p_theta, p_r, m1, m2):
    return ((p_theta**2) / (m1 * r**3)) - (m2 * g) + (m1 * g * np.cos(theta))

#summarizes the differential equations defined above.
def dervs(state, t, m1, m2):
    theta, r, p_theta, p_r = state
    return [dtheta(theta, r, p_theta, p_r, m1, m2),
            dr(theta, r, p_theta, p_r, m1, m2),
            dp_theta(theta, r, p_theta, p_r, m1, m2),
            dp_r(theta, r, p_theta, p_r, m1, m2)]

def Solve_SAM(m1, m2, theta0, r0, p_theta0, p_r0, deriv, tf, dt):
    initial = [theta0, r0, p_theta0, p_r0]
    t = np.linspace(0,tf,round(tf/dt))
    soln_SAM = scipy.integrate.odeint(deriv, initial, t, args = (m1, m2,))
    return soln_SAM, t

#plots phase space_plots. See paper
def phase_space_plots(X):
    #X is am array of [[radius array] ,[angle array], [momentum from radius
        changing array], [momentum from angle changing]], [time array].
    r = X[0][:,1]
    theta = X[0][:,0] - np.pi/2
    p_r = X[0][:,3]
    p_theta = X[0][:,2]
    theta -= (np.ones_like(theta)*(np.pi/2))

    plt.figure(figsize=(10,10))

    ax = plt.subplot2grid((1,1), (0,0), projection='polar')
    plt.plot(theta, r)
    plt.show()

    plt.figure(figsize=(10,10))

    ax = plt.subplot2grid((3,2), (0,0))
    plt.plot(r, theta)
    plt.title('r, theta')

    ax = plt.subplot2grid((3,2), (0,1))
    plt.plot(r, p_r)
    plt.title('r, p_r')

    ax = plt.subplot2grid((3,2), (1,0))
    plt.plot(r, p_theta)
    plt.title('r, p_theta')

    ax = plt.subplot2grid((3,2), (1,1))
    plt.plot(theta, p_r)
```

```
    plt.title('theta, p_r')

    ax = plt.subplot2grid((3,2), (2,0))
    plt.plot(theta, p_theta)
    plt.title('theta, p_theta')

    ax = plt.subplot2grid((3,2), (2,1))
    plt.plot(p_r, p_theta)
    plt.title('p_r, p_theta')

    plt.show()

#take 1 dimensional slices of the z space
def z_slice(X, Y, Z, z0):
    indices = []
    Z += np.ones_like(Z)*(-min(Z) - z0)
    for i in range(len(Z)-1):
        prior = Z[i]
        follw = Z[i+1]
        if follw == 0:
            if i != 0:
                indices.append(i)
        else:
            if i != 0:
                if (prior/follw) < 0:
                    indices.append(i)
    new_X = X[np.array(indices)]
    new_Y = Y[np.array(indices)]
    return new_X, new_Y

def deep_z_slice(W, X, Y, Z, z0):
    indices = []
    Z += np.ones_like(Z)*(-min(Z) - z0)
    for i in range(len(Z)-1):
        prior = Z[i]
        follw = Z[i+1]
        if follw == 0:
            if i != 0:
                indices.append(i)
        else:
            if i != 0:
                if (prior/follw) < 0:
                    indices.append(i)
    new_W = W[np.array(indices)]
    new_X = X[np.array(indices)]
    new_Y = Y[np.array(indices)]
    return new_W, new_X, new_Y

#plotting function to create phase plotes
def plot_slices(X, Y, Z):
    slices = np.linspace(.01,.99,10)
    Z_range = max(Z) - min(Z)
    z0s = Z_range * np.array(slices)

    cool = cm = plt.get_cmap('Spectral')
    cNorm  = colors.Normalize(0., len(z0s))
```

```
    scalarMap = cmx.ScalarMappable(norm=cNorm, cmap=cool)

    plt.figure(figsize=(10,10))
    for i in range(len(z0s)):
        level = z_slice(X, Y, Z, z0s[i])
        plt.plot(level[0], level[1], '.', color=scalarMap.to_rgba(i))
    plt.show()

def threeD_plot(X, Y, Z):
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure(figsize=(20,20))
    ax = fig.gca(projection='3d')
    ax.plot(X, Y, Z)
    plt.show()

#calculates the area between two periods, caused by big M changing.
def Differences(X):
    T = np.array(argrelextrema(X, np.greater))[0]
    diffs = []
    for i in range(int(round(len(T)/2))-2):
        start1 = 0 + T[2*i]
        end1 = T[(2*i) + 1] - 1
        start2 = 0 + T[2*(i+1)]
        end2 = T[(2*(i+1)) + 1] - 1

        range1 = end1 - start1
        range2 = end2 - start2
        drange = range2 - range1

        if drange != 0:
            end2 -= drange
            range2 = end2 - start2

        area = sum(abs(X[start1:end1] - X[start2:end2]))
        diffs.append(area)
    return diffs

#creates a diagram that shows the areas between radii for differnt big Ms.
#This serves as a bifurrcation diagram, as it shows the big M's that cause chaos.
#note.

# *****calling this function requires a significant amount of time to run many
    values of M.****

def Bifurcation(M0, Mf, NM):
    plt.figure(figsize=(30,7))
    M = np.linspace(M0, Mf, NM)
    for i in range(len(M)):
        radii = Solve_SAM(1., M[i], np.pi / 2, .1, 0., 0., dervs, 100, .001)[0]
            [:,1]
        areas = Differences(radii)
        x_value = M[i] * np.ones_like(areas)
        plt.plot(x_value, areas, 'b.')
    plt.ylim([0,10])
    plt.show()
```

```
#creates bifurcation diagram sutible for ploting, in color with axes labels.
def BifurcationColor(M0, Mf, NM):
    cool = cm = plt.get_cmap('YlGnBu')
    cNorm  = colors.Normalize(0., 5.)
    scalarMap = cmx.ScalarMappable(norm=cNorm, cmap=cool)

    plt.figure(figsize=(30,7))

    M = np.linspace(M0, Mf, NM)
    for i in range(len(M)):
        radii = Solve_SAM(1., M[i], np.pi / 2, .1, 0., 0., dervs, 1000, .001)[0]
            [:,1]
        areas = np.array(Differences(radii))
        p_areas = np.round(areas*100)/100
        if len(set(p_areas)) < 2:
            p_areas[0] += .001
        freq = np.histogram(p_areas, int(300*(max(p_areas)-min(p_areas))+1))

        for j in range(len(np.array(freq[1]))-1):
            colorVal = scalarMap.to_rgba(np.array(freq[0])[j])
            plt.plot(M[i], np.array(freq[1])[j], '.', ms=2,color=colorVal)

    plt.ylim([0,15])
    plt.show()

#returns the places where the angles are a minimum in x and y plane.
def get_min_places(X, Y, Z):
    indices = argrelextrema(Z, np.less)
    new_X = X[indices]
    new_Y = Y[indices]
    return new_X, new_Y

#returns the momentum of the system at an angle, radius, energy, and m.
def get_p_r(E, M, r, theta, p_theta):
    return ( ( E - (p_theta**2/(2*r**2)) - (M*g*r) + (g*r*np.cos(theta)) )*( 2*(M
        +1) ) )

#plots maps of phase space
def SOS_maps(E, M, nTs, nDt, N):
    cool = cm = plt.get_cmap('Accent')
    cNorm  = colors.Normalize(0., nTs*nDt)
    scalarMap = cmx.ScalarMappable(norm=cNorm, cmap=cool)
    Ts = np.linspace(np.pi/4, 3*np.pi/3, 3)
    Dt = np.linspace(0, 5, 3)

    R0 = .9
    it = 0
    for i in range(nTs):
        theta = Ts[i]
        for j in range(nDt):
            p_theta = Dt[j]
            p_r = get_p_r(E, M, R0, theta, p_theta)
            X = Solve_SAM(1., M, theta, R0, p_theta, p_r, dervs, N, .001)
            ar = X[0][:,1]
            atheta = X[0][:,0]
            ap_r = X[0][:,3]
```

```
            ap_theta = X[0][:,2]

            Y = get_min_places(ar, ap_r, atheta)
            some_r = Y[0]
            some_theta = Y[1]
            plt.plot(some_r, some_theta, '.',
                     label=r'$\theta$={0}, $P_\theta$={1}'.format(str(theta)[:5],
                         p_theta),
                     color=scalarMap.to_rgba(nTs*i+j))
            it += 1
    #plt.legend()
    plt.xlabel('r')
    plt.ylabel(r'$\dot{r}$')
    plt.title('M = {0}'.format(str(M)[:3]), fontsize=24)

#plots 9 phase plots for different masses
def three_SOS_maps(E, M0, Mf, N):
    Ms = np.linspace(M0, Mf, 9)
    print(Ms)
    Ts = np.linspace(np.pi/4, 3*np.pi/3, 3)
    Dt = np.linspace(0, 5, 3)

    plt.figure(figsize=(20,20))
    for i in range(3):
        for j in range(3):
            ax = plt.subplot2grid((3, 3), (i, j))
            SOS_maps(E, Ms[(3*i)+j], int(len(Ts)), int(len(Dt)), N)


# In[5]:

#RUNNING THIS FUNCTION TAKES A SIGNIFICANT AMOUNT OF TIME FOR MANY DATA POINTS
Bifurcation(1.1, 30, 1001)


# In[6]:

Bifurcation(30, 100, 1001)


# In[7]:

MMM = 7.


# In[17]:

X = Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 10, .001)
r = X[0][:,1]
theta = X[0][:,0]
p_r = X[0][:,3]
p_theta = X[0][:,2]
SLICE = z_slice(r, p_r, theta, 0.00001)

#phase_space_plots(X)
```

```
# In[20]:

three_SOS_maps(1, 1.1, 2.2, 10000)


# In[123]:

#series of trajectorings plotted for different mass ratios.
#These are shown in polar maps so they are real life representations of the
    atwood machine.
#note that

plt.figure(figsize=(40,20))

MMM = 2
ax = plt.subplot2grid((2,4),(0,0) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=30)
ax.set_xticklabels([])
ax.set_yticklabels([])

MMM = 3
ax = plt.subplot2grid((2,4),(0,1) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=30)
ax.set_xticklabels([])
ax.set_yticklabels([])

MMM = 5
ax = plt.subplot2grid((2,4),(0,2) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=30)
ax.set_xticklabels([])
ax.set_yticklabels([])

MMM = 6
ax = plt.subplot2grid((2,4),(0,3) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=30)
```

```
ax.set_xticklabels([])
ax.set_yticklabels([])

MMM = 16
ax = plt.subplot2grid((2,4),(1,0) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=30)
ax.set_xticklabels([])
ax.set_yticklabels([])

MMM = 19
ax = plt.subplot2grid((2,4),(1,1) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=30)
ax.set_xticklabels([])
ax.set_yticklabels([])

MMM = 21
ax = plt.subplot2grid((2,4),(1,2) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=30)
ax.set_xticklabels([])
ax.set_yticklabels([])

MMM = 24
ax = plt.subplot2grid((2,4),(1,3) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 50, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=30)
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.savefig("tragectories.pdf", format = "pdf")
plt.show()


# In[33]:

# shows one larger trajectory in chaos.

plt.figure(figsize=(40,20))
```

```
MMM = 24
ax = plt.subplot2grid((2,2),(1,1) ,projection='polar')
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 1, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 1, .001)
             [0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 1, .001)[0][:,1])
plt.title('Mass 2 = %s' % MMM, fontsize=24)
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.show()


# In[23]:


#not plotted in polar to show how stagectory is diverging. used to investigate
    lyapunov exponenets.
plt.plot(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 1.2, .001)[0][:,0] -
         np.ones_like(Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 1.2, .
             001)[0][:,0])*
          (np.pi/2),
         Solve_SAM(1., MMM, np.pi / 2., .1, 0., 0., dervs, 1.2, .001)[0][:,1])


# In[96]:

#used to find the difference between trajectories
def Differences2(X):
    T = np.array(argrelextrema(X, np.greater))[0]#array of angles
    diffs = []
    for i in range(int(round(len(T)/2))-2):
        start1 = 0 + T[2*i]
        end1 = T[(2*i) + 1] - 1
        start2 = 0 + T[2*(i+1)]
        end2 = T[(2*(i+1)) + 1] - 1

        range1 = end1 - start1
        range2 = end2 - start2
        drange = range2 - range1

        if drange != 0:
            end2 -= drange
            range2 = end2 - start2

        difference = (abs(X[start1:end1] - X[start2:end2]))
        diffs.append(difference)
    return diffs

#finds lyapunov exponents
def findExponent(X):
    diff = Differences2(X)
    exp_diff = []
    N = 0
    for ar in diff:
```

```
        npar = np.array(ar)
        min_ = findmin(ar)

        for i in range(len(npar)-1):
            if ar[i] != 0:
                if ar[i+1] != 0:

                    log_2 = (np.log2(max(ar[i+1],ar[i])/min(ar[i+1], ar[i])))
                    exp_diff.append(log_2)

    np_exp = np.array(exp_diff)
    lya = (2./len(np_exp))*sum(np_exp)
    return lya

#plots exponents
def lyapunovExponents(M0, Mf, NM):
    plt.figure(figsize=(30,7))
    M = np.linspace(M0, Mf, NM)
    exponents = []
    for i in range(len(M)):
        radii = Solve_SAM(1., M[i], np.pi / 2, .1, 0., 0., dervs, 100, .001)[0]
            [:,1]
        exponent = findExponent(radii)
        exponents.append(exponent)

    exponentsnp = np.array(exponents)
    plt.plot(M, exponentsnp)
    plt.xlim(int(M0),int(Mf))
    plt.title('Lyapunov Exponents', fontsize = 24)
    plt.xlabel('M', fontsize = 20)
    plt.ylabel(r'$\lambda$', fontsize = 20)
    plt.ylim(0,.4)
    plt.tick_params(axis='y', labelsize=20)
    plt.tick_params(axis='x', labelsize=20)
    plt.savefig("Lyapunov" + "_"+ str(int(M0)) + "_" + str(int(Mf))
        +".pdf",format = "pdf" )
    plt.show()


# In[102]:

X = Solve_SAM(1., 2., np.pi / 2, .1, 0., 0., dervs, 100, .001)[0][:,1]
findExponent(X)


# In[99]:

lyapunovExponents(2., 30., 501.)


# In[100]:

lyapunovExponents(31., 100., 501.)


# In[ ]:
```