# Double Pendulum

Ronan Waroquet, Michael T. McCannon

December 2022

## Introduction

A double pendulum is simply a pendulum with another pendulum attached to it. Unfortunately this is where the simplicity ends. Double pendulums unlike single pendulums exhibit notoriously chaotic motion and thus an interesting object to study for many. A single pendulum exhibits harmonic motion with equations $x = l\sin(\theta), y = -l\cos(\theta)$ where $l$ is the length of a mass less rod or rope and $\theta$ is the angle between equilibrium and the rod. Thus not only straightforward to model a single pendulum, it is also straightforward to solve the equations. However attaching a second pendulum really complicates the equations as shown later on. Thus solving these equations is difficult if not impossible, so the next best option is to model this chaotic motion which as demonstrated in this report is not easy. First the equations of motion needed to be derived and then plugged into an iterative algorithm. Using a graphing package with Python, these values from the algorithm were able to be modeled with its trace to see the motion of the double pendulum. While this process is not easy it is very rewarding to see this notoriously chaotic motion.

## Equations of Motion

The first step is to derive the equations of motion which is necessary to simulate our double pendulum. This turns out to be the longest portion of our modeling project. Using an iterative algorithm to model this complicated motion proved to be quite difficult. To do so, it is easiest by first, defining our grid and system, as shown below.

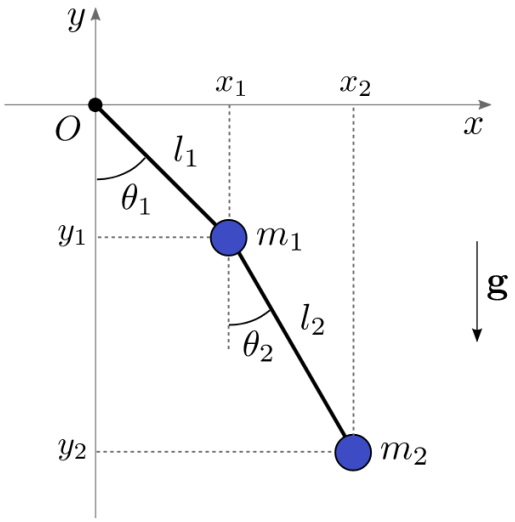


Figure 1: Double Pendulum System

Therefore, the masses at their respective positions will be given as,

$$x_1 = l_1 sin(\theta_1) \qquad y_1 = -l_1 cos(\theta_1) \tag{1}$$

$$y_2 = l_1 sin(\theta_1) + l_2 sin(\theta_2) \qquad y_2 = -l_1 cos(\theta_1) - l_2 cos(\theta_2) \tag{2}$$

We will proceed by finding the equations of motion by the Lagrangian method. The Lagrangian method utilizes the energy of the system, rather than the mechanical motion of the system. The Lagrangian says that for a system of particles or point masses, it can be written as,

$$L = T - V \tag{3}$$

where T is the kinetic energy of the system and V is the potential energy of the system. After obtaining this equation, the Lagrangian also states that

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_i}\right) - \frac{\partial L}{\partial \theta_i} = 0 \quad \text{for} \quad i = 1, 2, ... \tag{4}$$

In this case, we use $\theta$ as our variable of motion since $\theta$ changes with time. We first being by obtaining the kinetic energy of each points mass (there are 2 point masses) of the system. In the Lagrangian, $\dot{x}$ and $\dot{y}$ are the derivatives of $x$ and $y$ with respect to time which is the velocity in Cartesian coordinates.

$$\begin{aligned} T &= \frac{1}{2}m_1 v_1^2 + \frac{1}{2}m_2 v_2^2 \\ &= \frac{1}{2}m_1(\dot{x}_1^2 + \dot{y}_1^2) + \frac{1}{2}m_2(\dot{x}_2^2 + \dot{y}_2^2) \end{aligned} \tag{5}$$

We then substitute $\dot{x}$ and $\dot{y}$ for the time derivatives of equations (1) and (2). After simplifying, we then have

$$T = \frac{1}{2}m_1 l_1^2 \dot{\theta}_1^2 + \frac{1}{2}m_2\left[l_1^2\dot{\theta}_1^2 + l_2^2\dot{\theta}_2^2 + 2l_1 l_2 \dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2)\right] \tag{6}$$

The next part is then to find the potential energy of the system. This is pretty straightforward as it is the vertical displacement of the masses multiplied by $mg$. Therefore, we have

$$\begin{aligned} V &= m_1 g y_1 + m_2 g y_2 \\ &= -m_1 g l_1 \cos\theta_1 - m_2 g (l_1 \cos\theta_1 + l_2 \cos\theta_2) \\ &= -(m_1 + m_2) g l_1 \cos\theta_1 - m_2 g l_2 \cos\theta_2 \end{aligned} \tag{7}$$

We can therefore plug in T and V into equation (3), by definition of the Lagrangian. We then yield

$$\begin{aligned} L = \frac{1}{2} m_1 l_1^2 \dot{\theta}_1^2 &+ \frac{1}{2} m_2 \left[ l_1^2 \dot{\theta}_1^2 + l_2^2 \dot{\theta}_2^2 + 2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2) \right] \\ &+ (m_1 + m_2) g l_1 \cos\theta_1 + m_2 g l_2 \cos\theta_2 \end{aligned}$$

Simplifying...

$$\begin{aligned} L = \frac{1}{2} (m_1 + m_2) l_1^2 \dot{\theta}_1^2 &+ \frac{1}{2} m_2 l_2^2 \dot{\theta}_2^2 + m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2) \\ &+ (m_1 + m_2) g l_1 \cos\theta_1 + m_2 g l_2 \cos\theta_2 \end{aligned} \tag{8}$$

From the definition of the Lagrangian, we need to use equation (8) and plug it in into equation (4) for each theta. We have $\theta_1$ and $\theta_2$ therefore we will end up with two equations.

Lets start by calculating

$$\frac{\partial L}{\partial \dot{\theta}_i} \quad \text{and} \quad \frac{\partial L}{\partial \theta_i} \quad for \quad \theta_1 \quad \text{and} \quad \theta_2$$

We then have...

$$\frac{\partial L}{\partial \dot{\theta}_1} = = m_2 l_2^2 \dot{\theta}_2 + m_2 l_1 l_2 \dot{\theta}_1 \cos(\theta_1 - \theta_2) \tag{9}$$

$$\frac{\partial L}{\partial \dot{\theta}_2} = m_2 l_2^2 \dot{\theta}_2 + m_2 l_1 l_2 \dot{\theta}_1 \cos(\theta_1 - \theta_2) \tag{10}$$

$$\frac{\partial L}{\partial \theta_1} = -m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \sin(\theta_1 - \theta_2) - (m_1 + m_2) g l_1 \sin\theta_1 \tag{11}$$

$$\frac{\partial L}{\partial \theta_2} = m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \sin(\theta_1 - \theta_2) - m_2 g l_2 \sin\theta_2 \tag{12}$$

We can now proceed by taking the derivative of (9) and (10) with respect to t and equaling it to their partial derivatives of (11) and (12)

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_i}\right) = \frac{\partial L}{\partial \theta_i} \quad \text{for} \quad i = 1, 2$$

$$(m_1 + m_2)l_1\ddot{\theta_1} + m_2 l_2 \ddot{\theta_2}\cos(\theta_1 - \theta_2) - m_2 l_2 \dot{\theta_2}^2 \sin(\theta_1 - \theta_2) = -g(m_1 + m_2)\sin\theta_1 \tag{13}$$

$$m_2 l_2^2 + m_2 l_1 l_2 \ddot{\theta_1}\cos(\theta_1 - \theta_2) - m_2 l_1 l_2 \dot{\theta_1}^2 \sin(\theta_1 - \theta_2) = -g m_2 l_2 \sin\theta_2 \tag{14}$$

After simplifying, we now have complete equations of motion for $\theta_1$ and $\theta_2$.

# Algorithm

We proceeded by finding a way to find angles $\theta_1$ and $\theta_2$ at any time $t$. There are three ways we could have proceeded. The first way we could have solves for angles $\theta_1$ and $\theta_2$ is by solving the differential equations (13) and (14). We were not sure of any analytical solution to these differential equations so we solved it numerically with Python. This leaves us with two different ways. The second path we could have pursued is solving the differential equation numerically but using a built in ODE solving function. Python has many libraries for scientific computing. Sympy and numpy are the most commonly used. These libraries are able to solve complex differential equations. The last way, the one we used, is more engaging. We take the second degree differential equation, reduce the order of the equations by makign a substitution and use euler's method to compute the changes of $\theta_1$ and $\theta_2$.

We want to solve for $\theta_1$ and $\theta_2$. Let's make a substitution by setting $z_i = \dot{\theta}_i$ and $\dot{z_i} = \ddot{\theta}_i$ for $i = 1, 2$. Using these substitutions in equations (13) and (14), solving for $\dot{z_1}$ and $\dot{z_2}$, we have

$$\dot{z_1} = \frac{m_2 g \sin\theta_2 \cos(\theta_1 - \theta_2) - m_2 \sin(\theta_1 - \theta_2)(l_1 z_1^2 \cos(\theta_1 - \theta_2) + l_2 z_2^2) - (m_1 + m_2) g \sin\theta_1}{l_1(m_1 + m_2 \sin^2(\theta_1 - \theta_2))} \tag{15}$$

$$\dot{z_2} = \frac{(m_1 + m_2)(l_1 z_1^2 \sin(\theta_1 - \theta_2) - g\sin\theta_2 + g\sin\theta_1\cos(\theta_1 - \theta_2)) + m_2 l_2 z_2^2 \sin(\theta_1 - \theta_2)\cos(\theta_1 - \theta_2)}{l_2(m_1 + m_2\sin^2(\theta_1 - \theta_2))} \tag{16}$$

Notice how $z_1 = \frac{d\theta_1}{dt}$ and $z_2 = \frac{d\theta_2}{dt}$. We then have that $d\theta_1 = z_1 dt$ and $d\theta_2 = z_2 dt$. We, conveniently, have a way of solve for each theta by utilizing the change of $z_1$ and $z_2$ (acceleration) and the timestep $dt$. Since $z_1$ and $z_2$ are the change of velocities of angles $\theta_1$ and $\theta_2$, we need to find a z value at each time value we iterate the algorithm by. We do not have equations $z_1$ and $z_2$. But, we do know the

change of $z_1$ and $z_2$. These are equations (15) and (16). We can now, once again, use euler's method to find $z_1$ and $z_2$ at any time $t$. Bringing it all together, we have the following equations...

$$z_i(t) = z_i(t-dt) + \dot{z}_i(t-dt)dt \quad \text{for} \quad i = 1, 2 \tag{17}$$

$$\theta_i(t) = \theta_i(t-dt) + z_i(t-dt)dt \quad \text{for} \quad i = 1, 2 \tag{18}$$

We are now complete with finding angles $\theta_1$ and $\theta_2$ at any time t. All we need to do is provide the euler's method functions (17) and (18) with initial positions, velocities, and accelerations. When we find these angles $\theta_1$ $\theta_2$ at any time t in our algorithm. The code below this is for any number of iterations we provide the for loop. See code below.

```
t = np.arange(t_initial, t_final, dt) #define array of all time steps

def z1_deriv(m1, m2, l1, l2, theta1, theta2, z1, z2): #function to define change in
                                                      z_1
    delta = theta1 - theta2

    numer = m2 * G * np.sin(theta2) * np.cos(delta) - m2 * np.sin(delta) * (  l1 * z1
                                                        * z1 * np.cos(delta) + l2 * z2 * z2  )
                                                        - \
    (m1 + m2) * G * np.sin(theta1)

    denom = l1 * (m1 + m2 * np.sin(delta) * np.sin(delta))

    z1_dot = numer/denom


    return z1_dot

def z2_deriv(m1, m2, l1, l2, theta1, theta2, z1, z2): #function to define change in
                                                      z_2
    delta = theta1 - theta2

    numer = (m1 + m2) * (l1 * z1 * z1 * np.sin(delta) - G * np.sin(theta2) + G * np.
                                                        sin(theta1) * np.cos(delta) ) + \
    m2 * l2 * z2 * z2 * np.sin(delta) * np.cos(delta)

    denom = l2 * (m1 + m2 * np.sin(delta) * np.sin(delta))

    z2_dot = numer/denom

    return z2_dot
```

```
z1, z2 = [w1], [w1] #initial velocities
th1, th2 = [THETA1], [THETA2] #initial angles
numIterations = (t_final - t_initial) / dt

for i in range(1,int(numIterations)):
    z1_value = z1[i-1] + z1_deriv(M1, M2, L1, L2, th1[i-1], th2[i-1], z1[i-1], z2[i-1]
                                                    )*dt
    z1.append(z1_value)

    z2_value = z2[i-1] + z2_deriv(M1, M2, L1, L2, th1[i-1], th2[i-1], z1[i-1], z2[i-1]
                                                    )*dt
    z2.append(z2_value)

    th1_value = th1[i-1] + z1[i-1]*dt
    th1.append(th1_value)

    th2_value = th2[i-1] + z2[i-1]*dt
    th2.append(th2_value)
```

Rather than just modeling the angles vs time, we constructed a simulation using a Python library matplotlib.animation. What we essentially do now is at each time step dt, we convert the angles to Cartesian coordinates, and store the $x$ and $y$ variables in an array to use for simulation and graphing.

```
x1 = L1 * np.sin(th1)
y1 = -L1 * np.cos(th1)

x2 = L1 * np.sin(th1) + L2 * np.sin(th2)
y2 = -L1 * np.cos(th1) - L2 * np.cos(th2)
```

We have completed the algorithm. We now need to run the algorithm, retrieve angle values and graph them.

## Graphing and Modeling

After solving for all $\theta_1$ and $\theta_2$ values, we've graphed the angles vs time.

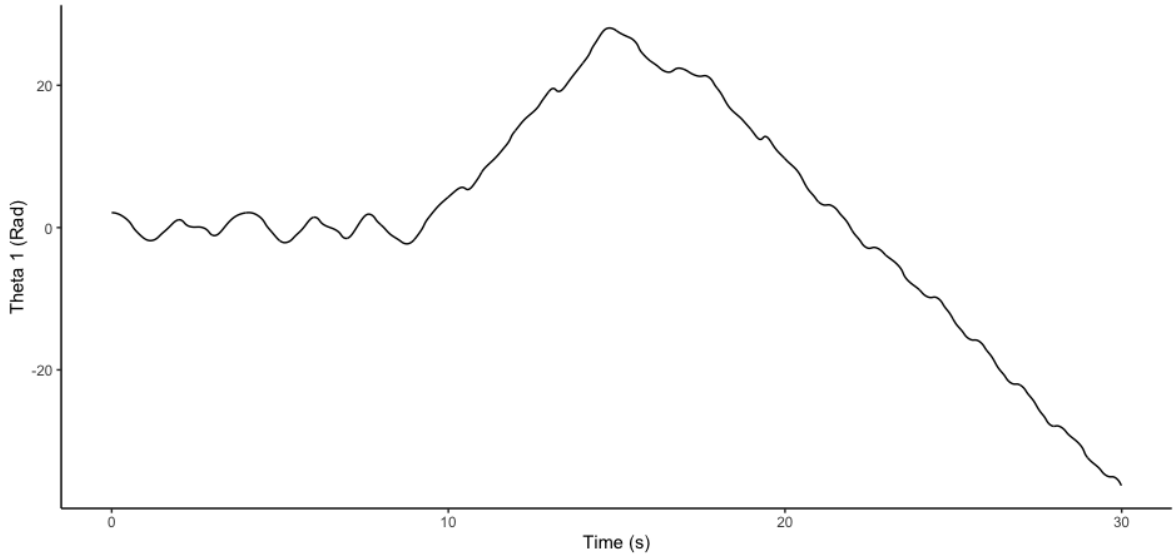


Figure 2: $\theta_1$ vs time

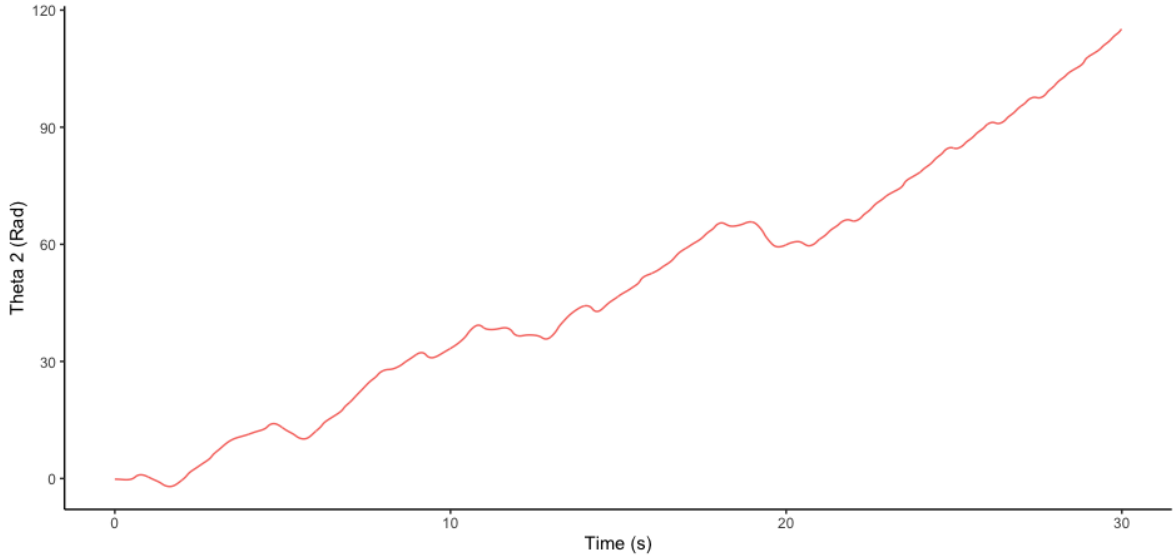


Figure 3: $\theta_2$ vs time

In our simulation, we plotted the double pendulum and traced the location of the second mass. The following graph (figure 4) produced just this. The y-axis is measured in radians and 0 implies 0 degrees with respect to our initially defined system (figure 1). Of course, we need initial values. We therefore choose slight deviations from 0 radians of both $\theta_1$ and $\theta_2$. There is not much to disect from these graphs other than we can observe constant circular rotation as the radians of both angles increase and decrease over time. Due to the nature of our code, we don't restrict radian measurements in the interval $[2, \pi]$.

Back to our previous discussion following algorithms, I mentioned that there is a way to solve for angles $\theta_1$ and $\theta_2$ using a built in differential equaiton solver using Python's scipy library. The angles calculated using this library should be even more accurate. The following code is used to calculate angle and velocity values using the scipy library in a more accurate fashion.

```
from scipy.integrate import odeint

def diff(y, t, m1, m2, l1, l2):
    theta1, z1, theta2, z2 = y

    delta = theta1 - theta2

    numer = (m1 + m2) * (l1 * z1 * z1 * np.sin(delta) - G * np.sin(theta2) + G * np.
                                                    sin(theta1) * np.cos(delta) ) + \
    m2 * l2 * z2 * z2 * np.sin(delta) * np.cos(delta)

    denom = l2 * (m1 + m2 * np.sin(delta) * np.sin(delta))

    theta1dot = z1

    z2_dot = numer/denom

    numer = m2 * G * np.sin(theta2) * np.cos(delta) - m2 * np.sin(delta) * (  l1 * z1
                                                    * z1 * np.cos(delta) + l2 * z2 * z2  )
                                                    - \
    (m1 + m2) * G * np.sin(theta1)

    denom = l1 * (m1 + m2 * np.sin(delta) * np.sin(delta))

    theta2dot = z2

    z1_dot = numer/denom

    return theta1dot, z1_dot, theta2dot, z2_dot

y0 = np.array([THETA1, w1, THETA2, w2])

y = odeint(diff, y0, t, args = (M1, M2, L1, L2))
```

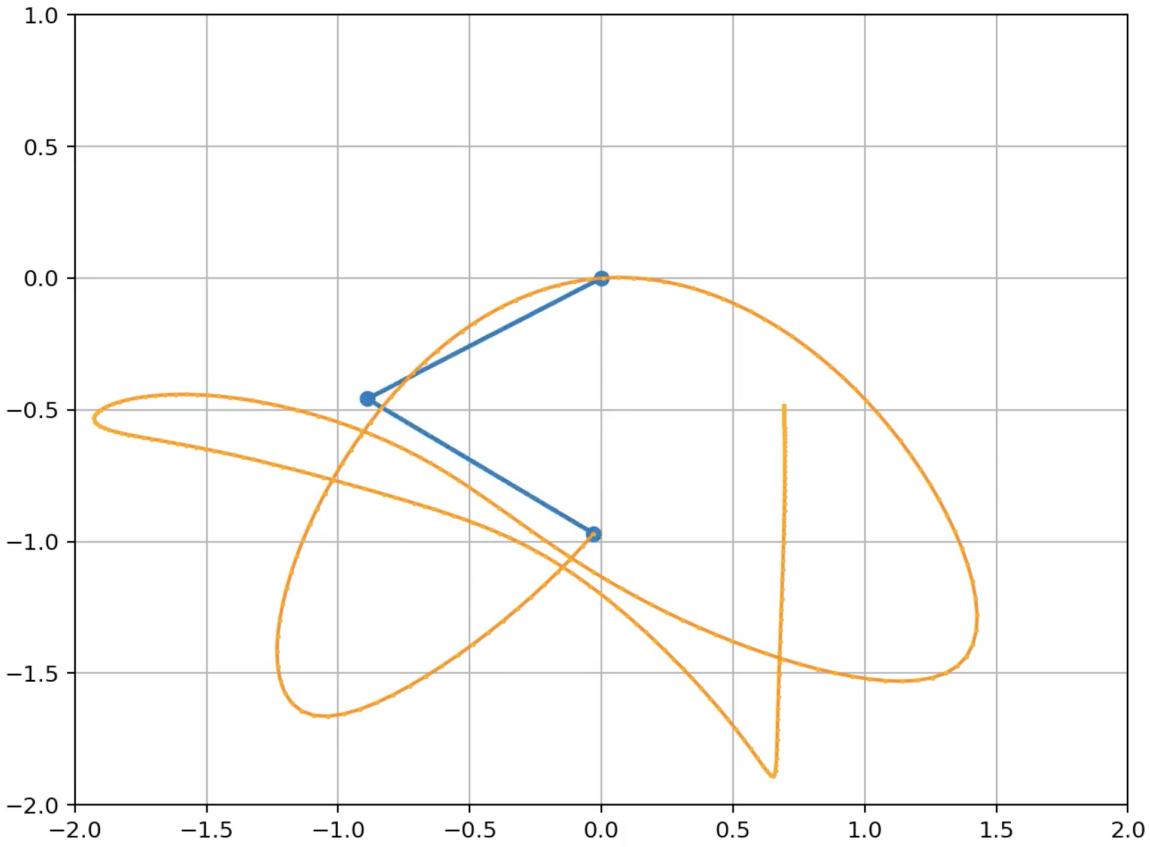


Figure 4: Double Pendulum with Trace

Now that we assume we have more accurate values, we can calculate the Relative Error of our algorithm. Observe the following two graphs which show the relative error (in %) of the measured angles.

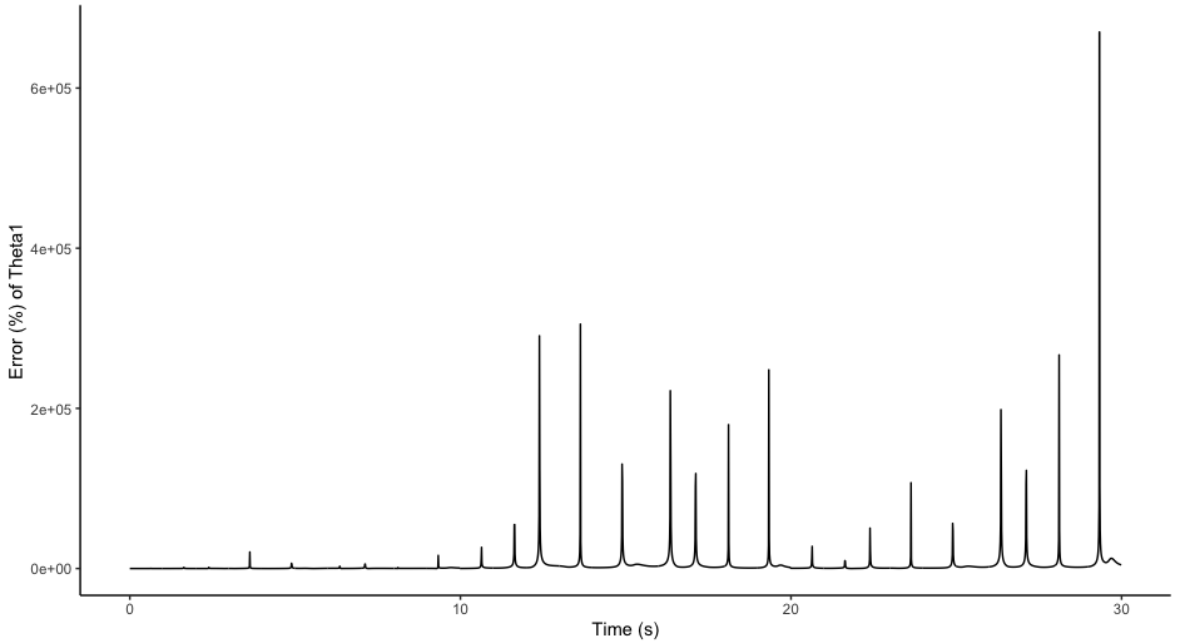


Figure 5: Relative Error of $\theta_1$

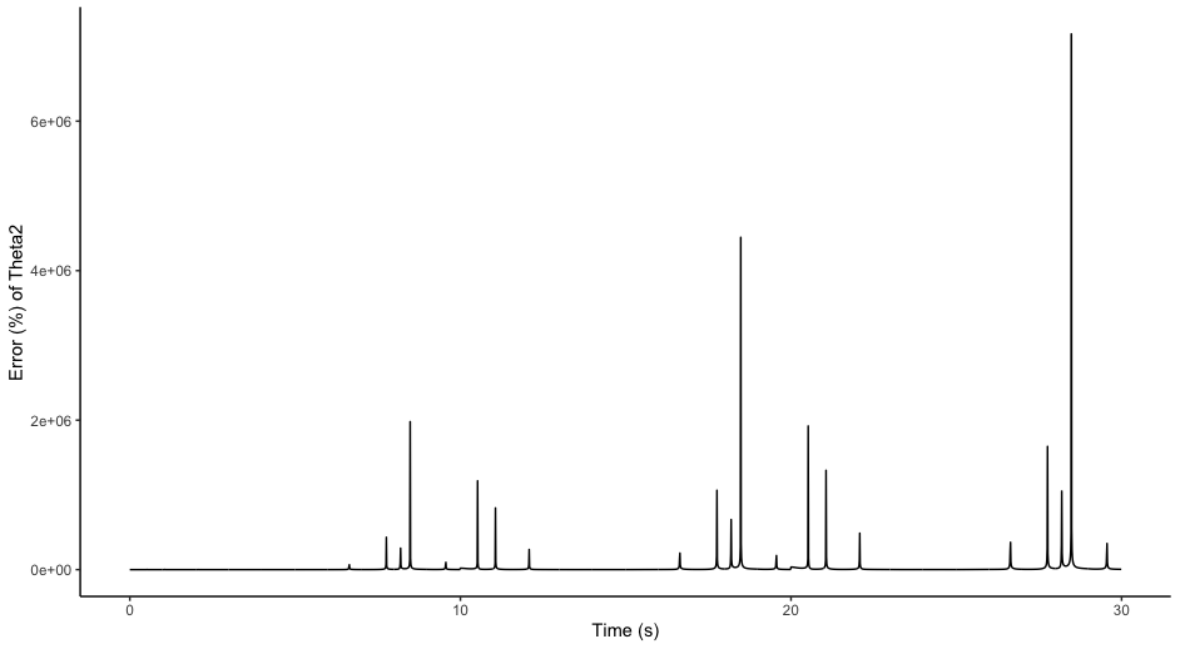


Figure 6: Relative Error of $\theta_2$

As you can see, there are extremely large jumps in relative error at specific times. The max values jump to 6 million percent Relative Error. Since we use a very small time interval $dt = 0.01$, the program has no choice but to calculate very small values. Since we are unsure as to how the scipy library program

solves differential equations, we are not sure if the library outputted accurate values. Let's magnify these error values to a more appropriate interval.

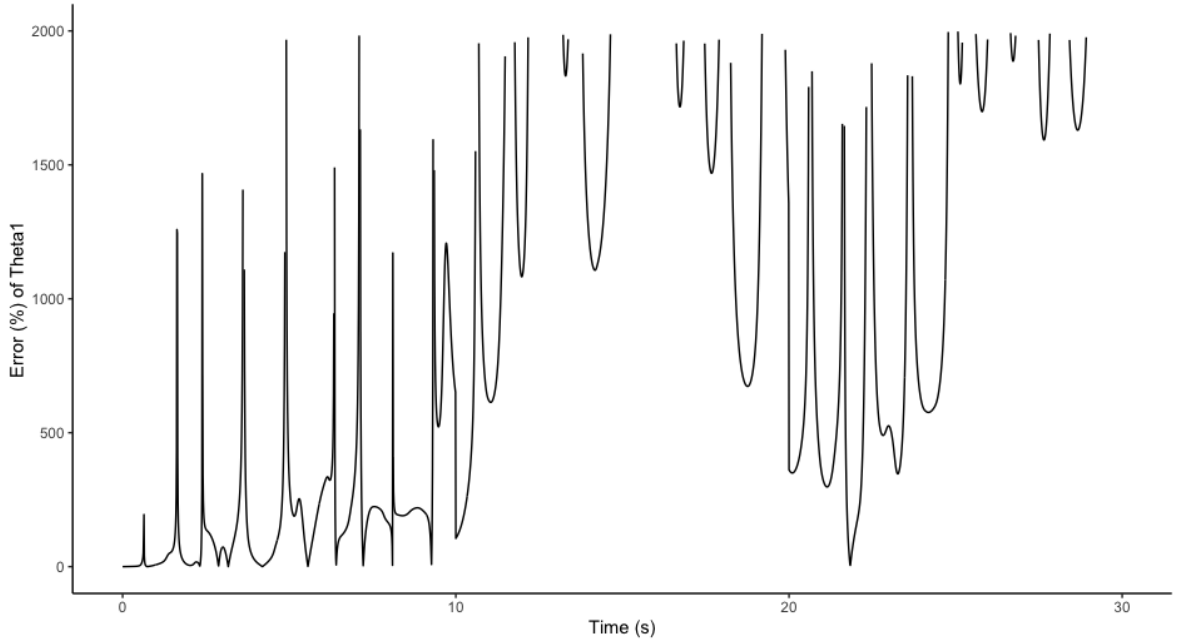


Figure 7: Relative Error of $\theta_1$

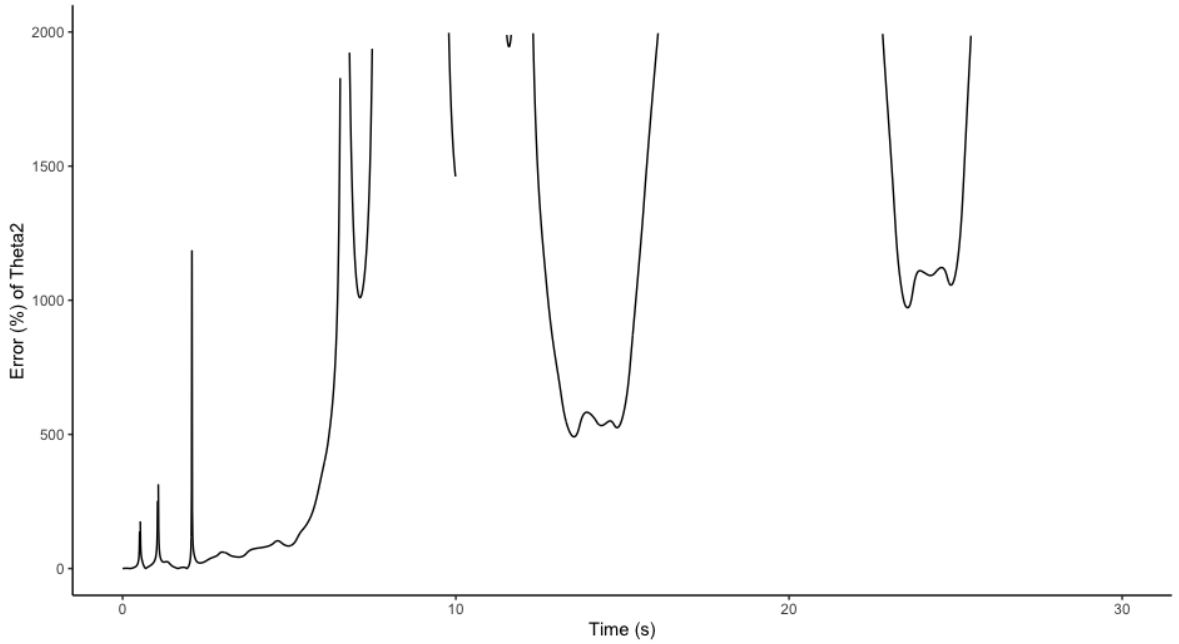


Figure 8: Relative Error of $\theta_2$

If we assume our ODE solver is correct, our algorithm accumulates errors very quickly. This leads us to limitations of our algorithm. In order to find out why, we need to re-code our algorithm to utilize as little calculations as possible or find an alternative method to euler's method.

## Limitations

There are three main limitations to this model, first there is no air resistance, second we are assuming that the connecting ropes are mass less, and that the pivot points are friction less. To implement one of these forces would be very difficult since it would make already complicated equations even more difficult, let alone trying to implement all three. The reason why these are major limitations is because with all these taken into account especially air resistance and the pivot friction, the system would lose energy and thus stop eventually. Our model goes on forever. Third, and most obvious, are the limitations of the code itself. As stated before, our algorithm, it seems, to accumulate lots of error with very large and sharp errors at specific times. We were unable to find a cause and correlation between $\theta_1$ and $\theta_2$ positions and error. The only conclusions we are able to draw from this scenario is that we utilize the euler's method in two ways. One is essentially embedded in the other. Given the nature of euler's method and it's accumulation of errors, we can make the conclusion that we need to implement an alternative algorithm to that of euler's method.

## Conclusion

A double pendulum system is very complicated. It involved different steady and chaotic states and this is why double pendulums are so heavily studied. After completion of the code, it is easy to extend this situation to that of a triple or quadruple pendulum system. The Lagrangian will be very long to compute but the implementation is the same. Before we even consider of extending this system, we will need to implement a different iterative algorithm. If we look back to our derivations of the motion of the equation and the utilization of euler's method in solving for $\theta_1$, $\theta_2$, $\frac{d\theta_1}{dt}$ $(z_1)$, and $\frac{d\theta_2}{dt}$ $(z_2)$, euler's method can be used perfectly in this situation. That's why we would like to make the conclusion that the main and primary limitation is the method of implementation. Alternative methods are the utilization of the ODE solver using our scipy library or the Runge-Kutta method. Additional method need to be researched.

## Full Code

```
import sympy as sym
import matplotlib.pyplot as plt
import matplotlib.animation
import numpy as np
from scipy.integrate import odeint

#Declarations
L1 = 1.0
L2 = 1.0
L = L1 + L2
G = 9.81
M1 = 1.0
M2 = 1.0
THETA1 = 5*np.pi/6
THETA2 = np.pi
w1 = 0
w2 = 0
t_initial = 0
t_final = 15 #seconds
dt = .01

t = np.arange(t_initial, t_final, dt) #define array of all time steps

def z1_deriv(m1, m2, l1, l2, theta1, theta2, z1, z2): #function to define change in
                                                      z_1
    delta = theta1 - theta2

    numer = m2 * G * np.sin(theta2) * np.cos(delta) - m2 * np.sin(delta) * (  l1 * z1
                                                        * z1 * np.cos(delta) + l2 * z2 * z2  )
                                                        - \
    (m1 + m2) * G * np.sin(theta1)

    denom = l1 * (m1 + m2 * np.sin(delta) * np.sin(delta))

    z1_dot = numer/denom


    return z1_dot

def z2_deriv(m1, m2, l1, l2, theta1, theta2, z1, z2): #function to define change in
                                                      z_2
    delta = theta1 - theta2

    numer = (m1 + m2) * (l1 * z1 * z1 * np.sin(delta) - G * np.sin(theta2) + G * np.
                                                        sin(theta1) * np.cos(delta) ) + \
    m2 * l2 * z2 * z2 * np.sin(delta) * np.cos(delta)
```

```
    denom = l2 * (m1 + m2 * np.sin(delta) * np.sin(delta))

    z2_dot = numer/denom

    return z2_dot

def diff(y, t, m1, m2, l1, l2): #used in solving the ODE with scipy library
    theta1, z1, theta2, z2 = y

    delta = theta1 - theta2

    numer = (m1 + m2) * (l1 * z1 * z1 * np.sin(delta) - G * np.sin(theta2) + G * np.
                                                sin(theta1) * np.cos(delta) ) + \
    m2 * l2 * z2 * z2 * np.sin(delta) * np.cos(delta)

    denom = l2 * (m1 + m2 * np.sin(delta) * np.sin(delta))

    theta1dot = z1

    z2_dot = numer/denom

    numer = m2 * G * np.sin(theta2) * np.cos(delta) - m2 * np.sin(delta) * (  l1 * z1
                                                * z1 * np.cos(delta) + l2 * z2 * z2  )
                                                - \
    (m1 + m2) * G * np.sin(theta1)

    denom = l1 * (m1 + m2 * np.sin(delta) * np.sin(delta))

    theta2dot = z2

    z1_dot = numer/denom

    return theta1dot, z1_dot, theta2dot, z2_dot

y0 = np.array([THETA1, w1, THETA2, w2])
y = odeint(diff, y0, t, args = (M1, M2, L1, L2))

z1, z2 = [w1], [w1] #initial velocities
th1, th2 = [THETA1], [THETA2] #initial angles

numIterations = (t_final - t_initial) / dt

for i in range(1,int(numIterations)):
    z1_value = z1[i-1] + z1_deriv(M1, M2, L1, L2, th1[i-1], th2[i-1], z1[i-1], z2[i-1]
                                                )*dt
    z1.append(z1_value)
```

```
    z2_value = z2[i-1] + z2_deriv(M1, M2, L1, L2, th1[i-1], th2[i-1], z1[i-1], z2[i-1]
                                                    )*dt
    z2.append(z2_value)

    th1_value = th1[i-1] + z1[i-1]*dt
    th1.append(th1_value)

    th2_value = th2[i-1] + z2[i-1]*dt
    th2.append(th2_value)

x1 = L1 * np.sin(th1)
y1 = -L1 * np.cos(th1)

x2 = L1 * np.sin(th1) + L2 * np.sin(th2)
y2 = -L1 * np.cos(th1) + L2 * np.cos(th2)


#SIMULATION
fig = plt.figure(figsize=(5,4))
ax = fig.add_subplot(autoscale_on=False, xlim=(-L, L), ylim=(-L, 1.))
ax.set_aspect('equal')
ax.grid()

line, = ax.plot([], [], 'o-', lw = 2)

point_s,  = ax.plot([0], [0], marker="o"
                      , markersize=1
                      , markeredgecolor="orange"
                      , markerfacecolor="orange")

xdata, ydata = [],[]

def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    xdata.append(x2[i])
    ydata.append(y2[i])

    point_s.set_data(xdata,ydata)

    line.set_data(thisx, thisy)
    return line, point_s

anim = animation.FuncAnimation(fig,func=animate,frames=int(numIterations),interval=dt*
                                          int(numIterations),blit=True)

plt.show()
```