Simulating Chaos in the Astrojax Pendulum with Automatic Differentiation

Gene Siriviboon

Phys2300, Prof.Brad Marston

Abstract

The Newton-Raphson method and the automatic differentiation are used to simulate the system of Astrojax. Although both methods agree to the order of 10^{-6}, the automatic differentiation is significantly computationally cheaper, allowing the extensive exploration in the phase space of the system. The Lyapunov exponents, the Poincare section, and the flip time diagram are used to classify the stability of the orbit and the sensitivity to the initial condition of the system.

Keywords: Chaos Theory, Automatic Differentiation

1. Introduction

Astrojax is a system of two masses connected with a single string. The system resembles a double pendulum except that there is an extra degree of freedom from the mass sliding along the string, which could give the system a richer possible phenomenon. However, due to its high degree of freedom, the equation of motion of the system is extremely complex. Even though the formulation of Lagrangian with constrain could simplify the system, the solution to the Lagrange multiplier still need numerical algorithms such as the Newton-Raphson method [1], making it computationally expensive. This is limiting especially for phase space analysis which needs an extensive search on the parameter space. This project aims to tackle both the methodology and the nature of the Astrojax system itself with the following questions: Is there a natural way to simulate the Astrojax system without resorting to root finding numerical method? and what is the stability of the orbit for each initial condition?

We would explore the possibility of using the automatic differentiation to compute the equation of the motion of the system from Lagrangian with the equation of constrained. The stability of the state at each configuration would be calculated by the Lyapunov exponent, the Poincare section visualized, and the flip time diagram of the system will be simulated.

2. Background and Theory

2.1. Astrojax system

The figure 1 shows the system of Astrojax pendulum. The endpoint of the pendulum would be fixed at $(0,0,0)$ while both of the mass can move in the three-dimensional space with the holonomic constrained $\Phi((r_1),(r_2) = 0$ such that

$$\Phi(\mathbf{r_1},\mathbf{r_2}) = \|\mathbf{r_1}\| + \|\mathbf{r_2} - \mathbf{r_1}\| - l \tag{1}$$

which represent a friction-less string attaching to the endpoint and the second mass. The Lagrangian of the system can be written as

$$\begin{aligned} L(\mathbf{r_1},\mathbf{r_2},\dot{\mathbf{r}}_\mathbf{1},\dot{\mathbf{r}}_\mathbf{2},t) = \frac{1}{2}m_1 \|\dot{\mathbf{r}}_\mathbf{1}\|^2 + \frac{1}{2}m_2 \|\dot{\mathbf{r}}_\mathbf{2}\|^2 + m_1 g(\mathbf{r_1}\cdot\hat{z}) \\ + m_2 g(\mathbf{r_2}\cdot\hat{z}) + \lambda(t)\Phi(\mathbf{r_1},\mathbf{r_2}) \end{aligned} \tag{2}$$

where $m_1 = m_2 = m$ is the first and second mass and g is the gravitational acceleration which is pointed in the $-\hat{z}$ direction. Since the Lagrangian is well-defined up to the additive and multiplicative constant, we can choose the normalized unit of time $\sqrt{\frac{g}{l}}$ and length l so that in this unit $m = g = l = 1$

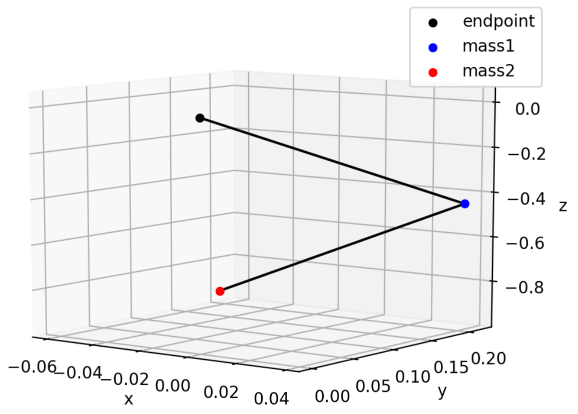


Figure 1: An example of Astrojax configuration

From the Lagrangian, we could calculate the Euler-Lagrange equation.

$$m\ddot{\mathbf{r}}_{\mathbf{1}} = -mg\hat{z} + \lambda(\hat{r}_1 - \hat{r}_2) \tag{3}$$

$$m\ddot{\mathbf{r}}_{\mathbf{2}} = -mg\hat{z} + \lambda\hat{r}_2 \tag{4}$$

where

$$\hat{r}_1 = \frac{\mathbf{r_1}}{\|\mathbf{r_1}\|} \tag{5}$$

$$\hat{r}_2 = \frac{\mathbf{r_1} - \mathbf{r_2}}{\|\mathbf{r_1} - \mathbf{r_2}\|} \tag{6}$$

Computationally-wise, we can see that the terms $\lambda(t)$ would be the hardest term to compute as we would need to solve the non-linear equation from $f(\mathbf{r_1}, \mathbf{r_2})$. Section 3.1 will discuss on the numerical method for solving these system of equations.

On the other hand, we could mitigate the problem by choosing a new set of generalized coordinate such that the constrained is implicitly defined. For this system, we can the define the configuration space as

$$\Omega = S^2 \times S^2 \times [0, l] \tag{7}$$

where $(\theta_1, \phi_1, \theta_2, \phi_2, r) \in \Omega$ can be converted back to Cartesian coordinate as

$$x_1 = r \sin\theta \cos\phi \tag{8}$$
$$y_1 = r \sin\theta \sin\phi \tag{9}$$
$$z_1 = r_1 \cos\theta \tag{10}$$
$$x_2 = x_1 + (1-r) \sin\theta \cos\phi \tag{11}$$
$$y_2 = y_1 + (1-r) \sin\theta \sin\phi \tag{12}$$
$$z_2 = z_1 + (1-r) \cos\theta \tag{13}$$

However, the Euler-Lagrange equation of such coordinate system would be extremely complex to be analyze by hand.

2.2. Automatic Differentiation

To mitigate the complexity of the analytical solution for the constrained of motion, we can use the tools of automatic differentiation from machine learning to help calculating gradient of the function without sacrificing the error from finite difference method. It has been shown that we can write the Euler-Lagrange equation in the form of gradient [2]

$$\frac{d}{dt}\nabla_{\dot{q}} L(q, \dot{q}, t) = \nabla_q L(q, \dot{q}, t) + (\nabla_q \Phi)^T \lambda \tag{14}$$

From chain rule,

$$\frac{d}{dt}\nabla_{\dot{q}} L(q, \dot{q}, t) = \dot{q}\nabla_q \nabla_{\dot{q}} L(q, \dot{q}, t) + \ddot{q}\nabla_{\dot{q}} \nabla_{\dot{q}} L(q, \dot{q}, t) \tag{15}$$

Therefore,

$$\ddot{q}\nabla_{\dot{q}} \nabla_{\dot{q}} L(q, \dot{q}, t) = \nabla_q L(q, \dot{q}, t) + (\nabla_q \Phi)^T \lambda - \nabla_q \nabla_{\dot{q}} L(q, \dot{q}, t)\dot{q} \tag{16}$$
$$\ddot{q} = M^{-1}(f + (\nabla_q \Phi)^T \lambda) \tag{17}$$

Where $M = \nabla_{\dot{q}} \nabla_{\dot{q}} L(q, \dot{q}, t)$ is the hessian of the Lagrangian over the velocity, equivalent to mass term in Cartesian coordinate.

$$f = \nabla_q L(q, \dot{q}, t) - (\nabla_q \nabla_{\dot{q}} L(q, \dot{q}, t))\dot{q} \tag{18}$$

is the external force term. Note that in the Cartesian coordinate this would be reduced to $-\nabla_q V$. To eliminate the constrained of motion, we could take use the condition

$$0 = \dot{\Phi} = \dot{q}\nabla_q \Phi \tag{19}$$

and take the time derivative

$$0 = \ddot{q}\nabla_q \Phi + \dot{q}\nabla_q \dot{\Phi} \tag{20}$$

Multiplying equation 17 with $\nabla_q \Phi$,

$$\ddot{q}\nabla_q \Phi = \nabla_q \Phi M^{-1}(f + (\nabla_q \Phi)^T \lambda) \tag{21}$$

will give

$$\begin{align} -(\nabla_q \dot{\Phi})\dot{q} &= \nabla_q \Phi M^{-1}(f + (\nabla_q \Phi)^T \lambda) \tag{22} \\ -(\nabla_q \dot{\Phi})\dot{q} &= \nabla_q \Phi M^{-1} f + \nabla_q \Phi M^{-1}(\nabla_q \Phi)^T \lambda \tag{23} \\ \nabla_q \Phi M^{-1}(\nabla_q \Phi)^T \lambda &= -\dot{q}\nabla_q \dot{\Phi} - \nabla_q \Phi M^{-1} f \tag{24} \\ \lambda &= (\nabla_q \Phi M^{-1}(\nabla_q \Phi)^T)^{-1}(-(\nabla_q \dot{\Phi})\dot{q} - \nabla_q \Phi M^{-1} f) \tag{25} \end{align}$$

Therefore

$$\begin{align} \ddot{q} &= M^{-1}(f - (\nabla_q \Phi)^T(\nabla_q \Phi M^{-1}(\nabla_q \Phi)^T)^{-1}(-(\nabla_q \dot{\Phi})\dot{q} - \nabla_q \Phi M^{-1} f)) \tag{26} \\ &= M^{-1} f - (M^{-1}(\nabla_q \Phi)^T)(\nabla_q \Phi M^{-1}(\nabla_q \Phi)^T)^{-1}((\nabla_q \dot{\Phi})\dot{q} + \nabla_q \Phi M^{-1} f)) \tag{27} \end{align}$$

Although this condition can be troublesome in general coordinate, this term can be simplify in our case with the Cartesian coordinate with $M = \mathbb{1}$

$$\ddot{q} = f - (\nabla_q \Phi)^T(\nabla_q \Phi(\nabla_q \Phi)^T)^{-1}((\nabla_q \dot{\Phi})\dot{q} + \nabla_q \Phi f)) \tag{28}$$

By expressing the acceleration in this form, we could simply use automatic differentiation to build the computational graph of $L(q, \dot{q}, t)$, $\Phi(q)$, and $\dot{\Phi}(q)$ in order to compute the gradient with respect to $q, \dot{q}$ without having to deal with the complex analytical solution.

2.3. Lyapunov Exponent

For the non-linear dynamical system, we can characterized by how a small perturbation of the system could affect the system. For the linear system, the small perturbation would only result in the small shift in the solution. However, for the non-linear system, we could define the evolution of perturbation $\delta(t)$ as

$$\delta(t) = \delta_0 e^{\lambda t} \tag{29}$$

where $\delta_0 << 1$ and λ is the Lyapunov exponent. The negative Lyapunov exponent indicated that the system is near the basin of attraction and is converging to the stable state. In contrary, he positive Lyapunov exponent indicate the initial-condition sensitivity and the non-predictability of the system as we would need an infinitely precise initial condition in order to predict the system perfectly. Else, the solution would diverge from the expected one in an exponential rate.

2.4. Poincaré Section

Since the system of interest have $5 + 5 = 10$ degrees of freedom in the phase space, we would need a tool to reduce the dimension of the trajectory to visualize the trajectory. Poincaré Section is a representation where we choose a plane and plotting only the trajectory that intersect with such plane. For this work, the section would be chosen as $\theta_1 = 0$ with $\dot{\theta_1} > 0$, following from Stachowiak and Okada work[3].

3. Methodologies

3.1. Newton-Raphson Method

Du Toit's work[1] is used as a baseline algorithm for calculating the force of constrain. From equation 1, 3, 4, we can rewrite the problem into root-finding problem of

$$0 = G(\ddot{\mathbf{r}}_\mathbf{1}, \ddot{\mathbf{r}}_\mathbf{2}, \lambda) = \begin{pmatrix} \ddot{x_1} - \lambda(\alpha x_1 - \beta(x_2 - x_1) \\ \ddot{y_1} - \lambda(\alpha y_1 - \beta(y_2 - y_1) \\ \ddot{z_1} - \lambda(\alpha z_1 - \beta(z_2 - z_1) + 1 \\ \ddot{x_2} - \lambda\beta(x_2 - x_1) \\ \ddot{y_2} - \lambda\beta(y_2 - y_1) \\ \ddot{z_2} - \lambda\beta(z_2 - z_1) + 1 \\ \frac{d}{dt^2}(\|\mathbf{r_1}\| + \|\mathbf{r_2} - \mathbf{r_1}\| - l) \end{pmatrix} \tag{30}$$

where

$$\alpha = \frac{1}{\|\mathbf{r_1}\|} \tag{31}$$

$$\beta = \frac{1}{\|\mathbf{r_2} - \mathbf{r_1}\|} \tag{32}$$

here the normalized unit is used. The first 6 rows of the vector determine the satisfiability of the equation of motion where the last row determine the satisfiability of the equation of constrained. From here we can use the Newton-Raphson method to find the solution from the iterative application

$$(\ddot{\mathbf{r}}_{\mathbf{1}}, \ddot{\mathbf{r}}_{\mathbf{2}}, \lambda)^{i+1} = (\ddot{\mathbf{r}}_{\mathbf{1}}, \ddot{\mathbf{r}}_{\mathbf{2}}, \lambda)^{i} - J((\ddot{\mathbf{r}}_{\mathbf{1}}, \ddot{\mathbf{r}}_{\mathbf{2}}, \lambda)^{i})^{-1} G(\ddot{\mathbf{r}}_{\mathbf{1}}, \ddot{\mathbf{r}}_{\mathbf{2}}, \lambda)^{i} \tag{33}$$

where J is the Jacobian of G

$$J = \nabla_{(\ddot{\mathbf{r}}_{\mathbf{1}}, \ddot{\mathbf{r}}_{\mathbf{2}}, \lambda)} G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & j_1 \\ 0 & 1 & 0 & 0 & 0 & 0 & j_2 \\ 0 & 0 & 1 & 0 & 0 & 0 & j_3 \\ 0 & 0 & 0 & 1 & 0 & 0 & j_4 \\ 0 & 0 & 0 & 0 & 1 & 0 & j_5 \\ 0 & 0 & 0 & 0 & 0 & 1 & j_6 \\ j_1 & j_1 & j_3 & j_4 & j_5 & j_6 & 0 \end{pmatrix} \tag{34}$$

where

$$j = \begin{pmatrix} \alpha x_1 - \beta(x_2 - x_1) \\ \alpha y_1 - \beta(y_2 - y_1) \\ \alpha z_1 - \beta(z_2 - z_1) \\ \beta(x_2 - x_1) \\ \beta(y_2 - y_1) \\ \beta(z_2 - z_1) \end{pmatrix} \tag{35}$$

3.2. JAX Autograd

As we have shown before that we could rewrite the equation of motion with only with the gradient and hessian term. We could see that equation 28 fits into the JAX, which is a python package for automatic differentiation, nicely.

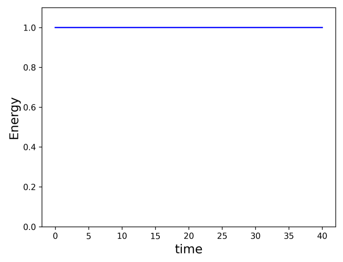
```python
def equation_of_motion(f, dPhi, dPhi_dot, state, t=None):
    q, q_t = jnp.split(state, 2)
    dphi = dPhi(q,0).reshape([1, -1])
    dphi_dot = dPhi_dot(q, q_t, 0).reshape([1, -1])
    q_tt = f - dphi.T @ jnp.linalg.pinv(dphi @ dphi.T) \
           @ (dphi @ f + dphi_dot @ q_t)
    dstate = jnp.concatenate([q_t, q_tt])
    return dstate
```

Listing 1: The snippet of python code for calculating the equation of motion

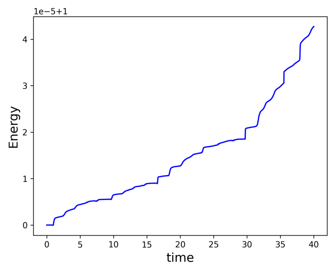
4. Results and Discussion

4.1. The Conserved Quantity and the Comparison with Baseline Method

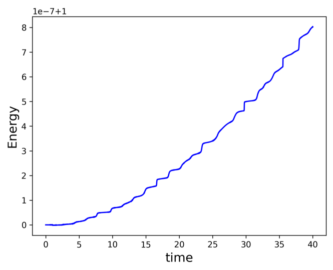
In order to justified that the trajectory we simulated, we will be focused on the conservation of total energy and the total length of the string. Figure 2 and 3 shows the evolution of the energy and length of the system over time. We can see that both of the simulation preserved these quantity fairly well, with the automatic differentiation preserving energy with the higher precision of 10^{-7} unit comparing to 10^{-5} unit in the Newton-Raphson method. This suggest the nature of the automatic differentiation which can calculate the gradient with comparable precision to the analytical solution.



(a) The total energy of the system over time



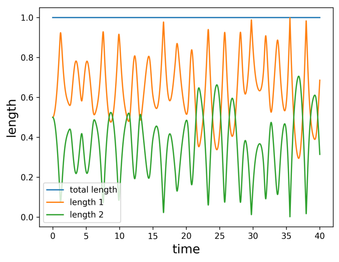
(b) The total energy of the system over time using the Newton-Raphson method



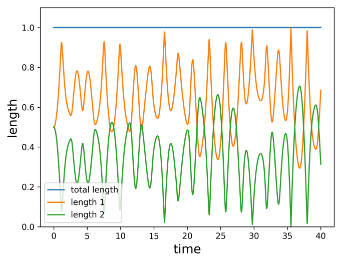
(c) The total energy of the system over time using the automatic differentiation method

Figure 2

Figure 4 shows the comparison between the trajectory upon perturbation and different simulation algorithm. We can see that the difference in the trajectory of the two algorithm is equivalent of the pertubation of the order $10^{-7} - 10^{-6}$.

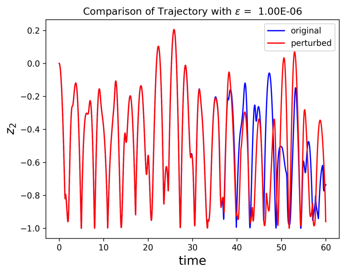


(a) The total length of the string over time using the Newton-Raphson method

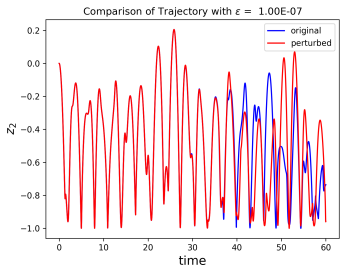


(b) The total length of the string over time using the automatic differentiation method

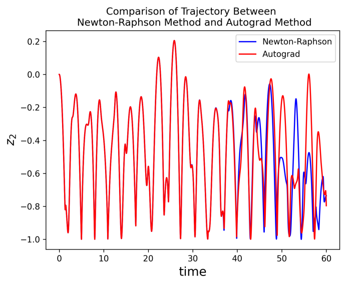
Figure 3



(a) The comparison of the trajectory of the Astrojax upon the perturbation of order 10^{-6}



(b) The comparison of the trajectory of the Astrojax upon the perturbation of order 10^{-7}



(c) The comparison of the trajectory of the Astrojax calculated from Newton-Raphson method and Automatic differentiation method

Figure 4

From this point forward, all the calculation will be from the automatic differentiation as we observed that it conserve the energy better and is far less computationally expensive.

4.2. Lyapunov Exponent as a Function of Initial Condition

Figure 5 shows the Lyapunov exponent of the system at different initial angles. We can see that the Lyapunov exponent is minimal around $\theta_1 = \theta_2 = \pi$ which is reasonable as such that is the state with lowest total energy. Still, it is worth noting that the negative Lyapunov is not found in the state space we explored, indicating that the chaotic regime for this system is not limited to the high energy state.

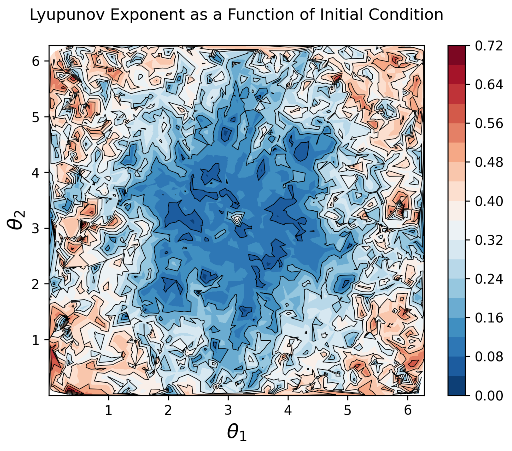


Figure 5: Lyapunov Exponent as a function of initial condition with $r = 0.5$ and $\dot{\theta}_i = \phi_i = \dot{\phi}_i = \dot{r} = 0$.

4.3. Poincare Section

Below shows the Poincare section at low energy limit (Fig. 6) and high energy limit (Fig. 7). We can see that although the system doe not exhibit a stable oscillation, the Poincare section still have some pattern emerge at low energy while the pattern diffuses into randomness as the energy increase.

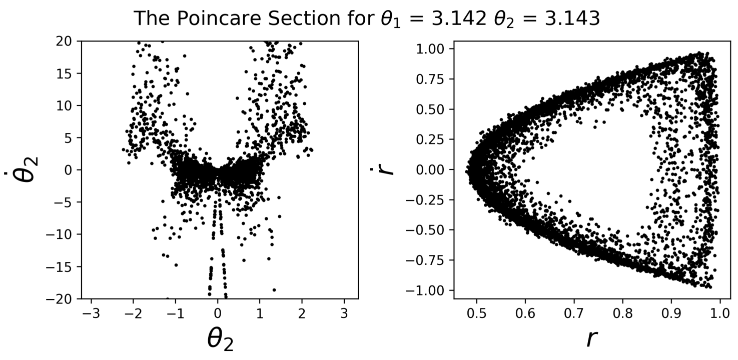


Figure 6: The Poincare section of the Astrojax system on the initial condition of $\theta_1 = 3.142, \theta_2 = 3.143, r = 0.5$

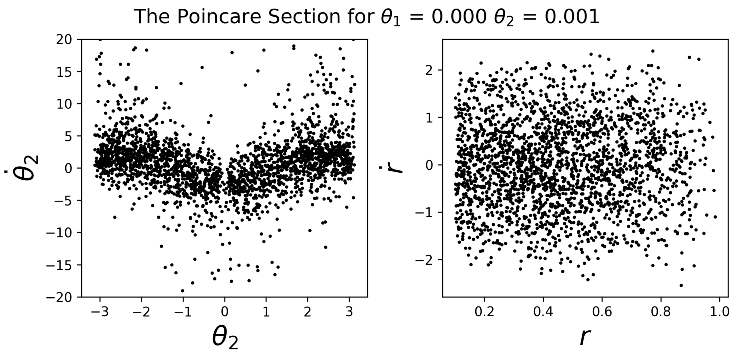


Figure 7: The Poincare section of the Astrojax system on the initial condition of $\theta_1 = 0, \theta_2 = 0.001, r = 0.5$

4.4. Flip time

To study the long-term behaviors of the system, we examine the time the Astrojax system use to rotate for full 2π radian around the first mass. We can see that from the energy perspective, we can predict the region enclosed by the black line, which is energy of the system is less than the amount of energy need to flip, by

$$H(\theta_1, \theta_2, r = 0.5) < H_{flip} \tag{36}$$

$$0.5(2\cos\theta_1 + \cos\theta_2) < 0.0(2\cos 0) + 1.0(\cos(\pi)) \tag{37}$$

$$2\cos\theta_1 + \cos\theta_2 < -2 \tag{38}$$

Where the flip energy is defined as the configuration which the second mass is at its lowest while the first mass is still at position $\theta_1 = 0$.

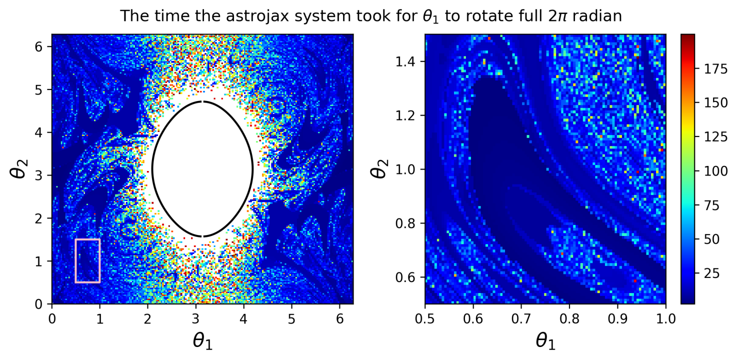


Figure 8: The flip time of the Astrojax as a function of initial condition with the close up figure on the right. The white pixel are the system which does not flip within $200\sqrt{l/g}$. The black line is defined as the region where the energy of the system is just enough to flip.

We can see the similar structure between this figure and the flip time for the double pendulum [4] as it exhibits a well-defined region for the question of whether the Astrojax will flip. However, the sensitivity to the initial condition of the system emerges in a form of the fractal structure as one can see that some nearby initial condition could leads to a huge difference in flipping time. With more computing power, we could calculate a more quantitative description of the flip time with the fractal dimension.

5. Conclusion

We have shown that the automatic differentiation technique could prove to be useful as a simulation technique of the system with a non-trivial constrained. The numerical simulation of the Astrojax pendulum is made and the chaos nature of the system is studied. The Lyapunov exponent and Poincare section shows that the system, in our region of interest, does not exhibits the stable trajectory. The flip time diagram exhibits the fracture structure, indicating the high sensitivity to the initial condition of the system.

References

[1] P. D. Toit, The Astrojax Pendulum and the N -Body Problem on the Sphere : A study in reduction, variational integration, and pattern evocation, Retrieved October 29, 2020, from https://iypt.ru/wp-content/uploads/2019/07/The-Astrojax-Pendulum-and-the-N-Body.pdf, 2005.

[2] M. Finzi, K. A. Wang, A. G. Wilson, Simplifying hamiltonian and lagrangian neural networks via explicit constraints, Part of Advances in Neural Information Processing Systems 33 pre-proceedings (NeurIPS 2020). Retrieved November 28, 2020, from https://papers.nips.cc/paper/2020/file/9f655cc8884fda7ad6d8a6fb15cc001e-Paper, 2020.

[3] T. Stachowiak, T. Okada, A numerical analysis of chaos in the double pendulum, Chaos, Solitons & Fractals 29 (2006) 417–422.

[4] J. S. Heyl, The Double Pendulum Fractal, Retrieved November 28, 2020, from https://www.famaf.unc.edu.ar/~vmarconi/fiscomp/Double, 2008.

[5] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, S. Ho, Lagrangian neural networks, arXiv preprint arXiv:2003.04630 (2020).

6. Appendix

The source code can be found here https://github.com/GeneSiriviboon/AstrojaxSolver.git

6.1. Solving Astrojax with Newton-Raphson Method

```
import numpy as np
from scipy.optimize import root, approx_fprime
from scipy.integrate import odeint

from util import *
from visualize import *
```

```
class Astrojax():

    def __init__(self):
        self.guess = np.array([0,0,-1,0,0,-1, 1])

    """
    state - representation

    state = [x1, y1, z1, x2, y2, z2, vx1, vy1, vz1, vx2
        , vy2, vz2]
    acceleration_tension = [a1x, a1y, a1z, a2x, a2y,
        a2z, T]
    m1 - mass of the upper ball
    m2 - mass of the lower ball
    A - function that takes in time t and derivative n
        and give out the nth derivative of endpoint over
         time
    """


    """
    creating constrain condition as a function of
        acceleration_tension

    [a - F/m, ldotdot]

    """
    def g_val(self, state, t):
        def g_func(acceleration_tension):


            rel1 = state[:3]
            rel2 = state[3:6] - state[:3]

            l1 = np.linalg.norm(rel1)
            l2 = np.linalg.norm(rel2)
            dL = l_dotdot(state[:6], state[6:12],
                acceleration_tension[:6],
                acceleration_tension[-1])


            g1 = acceleration_tension[:3] +
                acceleration_tension[6] * (rel1/l1 -
                rel2/l2) - np.array([0, 0, -1])
            g2 = acceleration_tension[3:6] +
                acceleration_tension[6] * (rel2/l2) -
                np.array([0, 0, -1])
            return np.hstack([g1, g2, [dL]])

        return g_func

    """
    calculate the jacobian of the constrain
    """
    def jacobian(self, state, t):
        def jac(acceleration_tension):

            rel1 = state[:3]
            rel2 = state[3:6] - state[:3]

            l1 = np.linalg.norm(rel1)
            l2 = np.linalg.norm(rel2)

            r1 = np.hstack([rel1/l1 - rel2/l2, rel2/l2
                ])
            r2 = np.hstack([rel1/l1 - rel2/l2, rel2/l2
                ])

            j = np.eye(7)
            j[:-1, 6] = r2
            j[6, :-1] = r1
            j[6, 6] = 0

            return j
        return jac

    def solveAcc(self, state, t):
        Root = root(self.g_val(state = state, t = t), \
                    self.guess, tol = 1e-10, jac = self.
                       jacobian(state = state, t = t))

        return Root.x



    """
    solve newton equation to find acceleration
    """
    def dstate(self, t_end, p = True):
        def helper(state, t):
            acceleration_tension  = self.solveAcc(state
               , t)

            self.guess = acceleration_tension

            progress = t/t_end

            if p:
                print_progress(progress,length = 30)

            return np.hstack([state[6:],
               acceleration_tension[:-1]])

        return helper

    def trajectory(self, initial, t):
        Dstate = self.dstate(t_end = t[-1])
        print('checking initial condition...')
        assert(abs(ldot(initial[:6], initial[6:])) < 1e
           -10)
        print('solving trajectory...')
        trajectory = odeint(Dstate, initial, t)
        print()
        print('trajectory obtained')
        return trajectory



if __name__ == '__main__':



    astrojax = Astrojax()
    t = np.linspace(0, 40, num = 4000)

    initial_state = np.array([0.0, 0.0, -0.5, 0.5, 0.0,
        -0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

    trajectory = astrojax.trajectory(initial_state, t)

    l = []
    l1 = []
    E = []
    for i in range(trajectory.shape[0]):
        l.append(length(trajectory[i]))
        E.append(Energy(trajectory[i]))

        endpoint = 0 #A * np.sin(w * t[i])

        rel1 = trajectory[i, :3] - endpoint
        rel2 = trajectory[i, 3:6] - trajectory[i, :3]

        l1.append(np.linalg.norm(rel1))

    l = np.array(l)
    l1 = np.array(l1)
    E = np.array(E)
    plt.figure()
    plt.xlabel('time', fontsize=15)
    plt.ylabel('Energy', fontsize=15)
    plt.plot(t, E, 'b')
    plt.ylim([0, np.max(E) * 1.1])
    plt.savefig('BaselineEnergy.png', dpi = 300)
    plt.show()


    print(trajectory[:, 6:12])
```

6.2. Solving Lagrangian with Holonomic Constraint

```
import jax
import jax.numpy as jnp
import numpy as np
from jax.experimental.ode import odeint
# from scipy.integrate import odeint
import matplotlib.pyplot as plt
from functools import partial # reduces arguments to
   function by making some subset implicit

from scipy.optimize.nonlin import Jacobian

from jax.config import config;
config.update("jax_enable_x64", True)

def equation_of_motion(f, dPhi, dPhi_dot, state, t=None
   ):
    q, q_t = jnp.split(state, 2)
    dphi = dPhi(q,0).reshape([1, -1])
    dphi_dot = dPhi_dot(q, q_t, 0).reshape([1, -1])
    q_tt = f - dphi.T @ jnp.linalg.pinv(dphi @ dphi.T)
       @(dphi @ f + dphi_dot @ q_t)
    dstate = jnp.concatenate([q_t, q_tt])
    return dstate
```

```
def solve_lagrangian(force, dPhi, dPhi_dot,
   initial_state, **kwargs):
    # We currently run odeint on CPUs only, because its
        cost is dominated by
    # control flow, which is slow on GPUs.
    @partial(jax.jit, backend='cpu')
    def f(initial_state):
        return odeint(partial(equation_of_motion, force
           , dPhi, dPhi_dot),
                    initial_state, **kwargs)
    return f(initial_state)

def solve_autograd(f, dPhi, dPhi_t):
    @partial(jax.jit, backend='cpu')
    def helper(initial_state, times):
        return solve_lagrangian(f, dPhi, dPhi_t,
           initial_state, t=times, rtol=1e-10, atol=1e
           -10)
    return helper



if __name__ == '__main__':

    state = np.array([1.0, 0.0, 0.0, 0.0])
    states = np.random.random([100, 4])
    f = np.array([0,-1])
    times = np.linspace(0, 10, num = 100)
    dPhi = lambda q, t: 2 * q
    dPhi_t = lambda q, q_t, t: q_t @ jax.jacobian(dPhi)
       (q, t).T
    print(equation_of_motion(f, dPhi, dPhi_t, state))
    ans = [equation_of_motion(f, dPhi, dPhi_t, state)
       for state in states ]
    maps_ans = jax.vmap(partial(equation_of_motion, f,
       dPhi, dPhi_t), 0)(states)

    print(np.allclose(ans, maps_ans))
```

6.3. Solving Astrojax with Automatic Differentiation

```
from CLNNs import *
import jax.numpy as jnp
import numpy as np
from util import *

from jax.config import config;
config.update("jax_enable_x64", True)

def DPhi(q, t):
    q1, q2 = jnp.split(q, 2)

    rel1 = q1
    rel2 = q2 - q1

    l1 = jnp.linalg.norm(rel1)
    l2 = jnp.linalg.norm(rel2)

    r1 = jnp.hstack([rel1/l1 - rel2/l2, rel2/l2])
    return r1

def DPhi_dot(q, q_dot,  t):
    return q_dot @ jax.jacobian(DPhi)(q, t).T

def solveAstrojax(initial, t):
    f = np.array([0, 0, -1, 0, 0, -1])
    trajectory = jax.device_get(solve_autograd(f, DPhi,
        DPhi_dot)(initial, t))
    return trajectory


if __name__ == '__main__':
    state1 = np.array([0.5, 0.0, 0.0, 1.0, 0.0, 0.0,
                       0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

    t = np.linspace(0, 40, num = 4000)
    trajectory1 = solveAstrojax(state1, t)
    E = [Energy(state) for state in trajectory1]
    l = [length(state) for state in trajectory1]

    plt.plot(t, E, 'g')
    plt.plot(t, l, 'r')
    plt.show()
```

6.4. Compare the two simulation together

```
from astrojax import *
from astrojax_Newton import *


def compare(initial_state, t):
    astrojax = Astrojax()
    trajectory1 = astrojax.trajectory(initial_state, t)
    trajectory2 = solveAstrojax(initial_state, t)

    plt.xlabel('time', fontsize=15)
    plt.ylabel(r'$z_2$', fontsize=15)
    plt.title('Comparison of Trajectory Between \n 
       Newton-Raphson Method and Autograd Method')
    plt.plot(t, trajectory1[:, 5], 'b', label = 'Newton
       -Raphson')
    plt.plot(t, trajectory2[:, 5], 'r', label = '
       Autograd')

    plt.legend()

def compare_err(initial_state1, initial_state2, t, err)
   :
    astrojax = Astrojax()
    trajectory1 = astrojax.trajectory(initial_state1, t
       )
    trajectory2 = astrojax.trajectory(initial_state2, t
       )

    plt.xlabel('time', fontsize=15)
    plt.ylabel(r'$z_2$', fontsize=15)
    plt.title(r'Comparison of Trajectory with $\
       epsilon$ =  {:.2E}'.format(err))
    plt.plot(t, trajectory1[:, 5], 'b', label = '
       original')
    plt.plot(t, trajectory2[:, 5], 'r', label = '
       perturbed')

    plt.legend()


if __name__ == '__main__':

    err = 1e-6

    state = np.array([0.5, 0.0, 0.0, 1.0, 0.0, 0.0,
                      0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

    state2 = np.array([0.5 - err, 0.0, 0.0, 1.0, 0.0,
       0.0,
                     0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

    t = np.linspace(0, 60, num = 6000)
    compare_err(state, state2, t, 1e-7)
    plt.savefig('./compare_err7.png', dpi = 300)
    plt.show()
```

6.5. Calculating Lyupunov Exponent

```
import jax.numpy as jnp
from matplotlib.pyplot import plot
import numpy as np
from astrojax import *
from functools import partial
from util import *
import matplotlib.pyplot as plt
from tqdm import tqdm
from visualize import *
```

```
def randomPolar(num):
    theta1s = np.random.random(num) * np.pi
    theta2s = np.random.random(num) * np.pi
    phi1s = np.random.random(num) * np.pi * 2
    phi2s = np.random.random(num) * np.pi * 2
    theta1sdot = np.random.random(num) * np.pi
    theta2sdot = np.random.random(num) * np.pi
    phi1sdot = np.random.random(num) * np.pi * 2
    phi2sdot = np.random.random(num) * np.pi * 2
    r = np.random.random(num)
    r_dot = np.random.random(num)

    return (theta1s,phi1s,theta2s,phi2s, r), (
       theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)


def randomStateInitializer(num):

    x, v = randomPolar(num)

    return (x, v)

def randomPair(num, epsilon):
    x, v = randomPolar(num)
    state1 =  polar2Cartesian(x, v)

    theta1, phi1, theta2, phi2, r = x
    x2 = (theta1, phi1, theta2 + epsilon, phi2, r)

    # state2 = polar2Cartesian((theta1, phi1, theta2 +
       epsilon, phi2, r), v)

    return (x, v), (x2, v)

def solveAstrojaxs(states, t):
    return jax.vmap(partial(solveAstrojax, t = t)) (
       states)
```

```
def lyupunov(PolarStates1, PolarStates2, t = 3, num =
    400):
    time = np.linspace(0, t, num = num)

    states1 = polar2Cartesian(PolarStates1[0],
        PolarStates1[1])
    states2 = polar2Cartesian(PolarStates2[0],
        PolarStates2[1])

    traj1 = np.array([solveAstrojax(state1, time) for
        state1 in tqdm(states1)])
    traj2 = np.array([solveAstrojax(state2, time) for
        state2 in tqdm(states2)])

    diffs = np.sum((traj1 - traj2)**2, axis = -1)**0.5
    slope = [np.polyfit(time, np.log(diff), 1)[0] for
        diff in diffs]

    return slope, traj1, traj2

def Cartesian2Polar(traj, t):
    r = np.sum(traj[:, :3]**2, axis = 1)**0.5

    theta1 = np.unwrap(np.arccos(traj[:, 2]/r) * np.
        sign(traj[:, 0]))
    theta2 = np.unwrap(np.arccos((traj[:, 5] - traj[:,
        2])/(1-r)) * np.sign(traj[:, 3] - traj[:, 0]))
    # theta1dot = - traj[:, 8] /(r * np.sin(theta1))
    # theta2dot = - (traj[:, 11] - traj[:, 8]) /((1-r)
        * np.sin(theta2))
    theta1dot = (theta1[2:] - theta1[:-2])/(t[1] - t
        [0])/2
    theta2dot = (theta2[2:] - theta2[:-2])/(t[1] - t
        [0])/2
    r_dot = (r[2:] - r[:-2])/(t[1] - t[0])/2
    return theta1[1:-1], theta2[1:-1], theta1dot,
        theta2dot, r[1:-1], r_dot, t[1:-1]
```

```
if __name__ == '__main__':
    theta1s = np.array([np.pi]); theta2s = np.array([np
       .pi - 1e-3])
    phi1s = np.array([0]); phi2s = np.array([0])
    theta1sdot = np.array([0])  ; theta2sdot = np.array
       ([0]); phi1sdot = np.array([0]); phi2sdot = np.
       array([0])

    r = np.array([0.5])
    r_dot = np.array([0])

    x, v = (theta1s,phi1s,theta2s,phi2s, r), (
       theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)

    t = np.linspace(0, 20000, num = 2000000)

    state1 = polar2Cartesian(x, v)[0]

    traj = solveAstrojax(state1, t)

    theta1, theta2, theta1dot, theta2dot, r, r_dot, t1
       = Cartesian2Polar(traj, t)
    theta1 = theta1 % (2 * np.pi) - np.pi
    theta2 = theta2 % (2 * np.pi) - np.pi
    zero_crossings = np.where(np.diff(np.sign(theta1)))
       [0]
    zero_crossings = zero_crossings[np.abs(theta1[
       zero_crossings]) < 1]
    zero_crossings = zero_crossings[theta1dot[
       zero_crossings] > 0]
    zero_crossings = zero_crossings[r[zero_crossings] >
        1e-1]

    fig, axs = plt.subplots(1, 2, figsize = [8,4])
    for i in range(zero_crossings.shape[0]):
        axs[0].plot(theta2[zero_crossings[i]],
            theta2dot[zero_crossings[i]], 'k.',
            markersize = 3)
        axs[0].set_xlabel(r'$\theta_2$', fontsize=20)
        axs[0].set_ylabel(r'$\dot{\theta}_2$', fontsize
            =20)
        axs[0].set_ylim([-20, 20])
        axs[1].plot(r[zero_crossings[i]], r_dot[
            zero_crossings[i]], 'k.', markersize = 3)
        axs[1].set_xlabel(r'$r$', fontsize=20)
        axs[1].set_ylabel(r'$\dot{r}$', fontsize=20)
        fig.tight_layout(rect=[0, 0.03, 1, 0.95])
        fig.suptitle(r'The Poincare Section for $\
            theta_1$ = {:.3f} $\theta_2$ = {:.3f}'.
            format(theta1s[0], theta2s[0]), fontsize =
            15)
        plt.pause(0.01)
    plt.show()
```

6.6. Utility Function

```
import tensorflow as tf
import numpy as np


tf.keras.backend.set_floatx('float64')

"""
print the progress of solving trajectory
 - input
    progress - float from 0(start) - 1(finish)'
    length - langth of the progressbar
"""
def print_progress(progress,length = 30):
    n = int(progress * length)
    print('|',*['#']*n, *[' ']*(length - n), '|', ' {:d
       }'.format(int(progress * 100)), '%', end = '\r',
        sep = '')
```

```
def dot(u, v):
    return tf.einsum('ij,ij->i',u, v)

"""
calculate the length of the rope (use to check the
   constrain)
"""
def length(state):
    q1, q2, q1_t, q2_t = np.split(state, 4)

    rel1 = q1
    rel2 = q2 - q1

    l1 = np.linalg.norm(rel1)
    l2 = np.linalg.norm(rel2)
    return l1+l2

def Energy(state):
    q1, q2, q1_t, q2_t = np.split(state, 4)
    return 0.5*np.sum(q1_t**2 + q2_t **2) + q1[-1] + q2
       [-1] + 2

"""
calculate rate of rope length changes (use to check the
    constrain)
"""
def ldot(pos, vel):


    rel1 = pos[:3]
    rel2 = pos[3:] - pos[:3]

    l1 = np.linalg.norm(rel1)
    l2 = np.linalg.norm(rel2)

    v_rel_1 = vel[:3]
    v_rel_2 = vel[3:] - vel[:3]
    return np.dot(v_rel_1, rel1)/l1 + np.dot(v_rel_2,
        rel2)/l2

# numpy version
def l_dotdot(pos, vel, acc, tension):

    rel1 = pos[:3]
    rel2 = pos[3:6] - pos[:3]

    l1 = np.linalg.norm(rel1)
    l2 = np.linalg.norm(rel2)

    v_rel_1 = vel[:3]
    v_rel_2 = vel[3:6] - vel[:3]

    dL = np.dot(acc[:3], rel1)/l1 \
         + np.linalg.norm(v_rel_1)**2/l1 \
         - np.dot(v_rel_1, rel1)**2 / l1**3\
         + np.dot(acc[3:6] - acc[:3], rel2)/l2 \
         + np.linalg.norm(v_rel_2)**2/l2 \
         - np.dot(v_rel_2, rel2)**2 / l2**3

    return dL

def polar2Cartesian(polarPos, polarVel):
    (theta1, phi1, theta2, phi2, r) = polarPos
    (theta1dot, phi1dot, theta2dot, phi2dot, rdot) =
        polarVel

    sinTheta1 = np.sin(theta1)
    cosTheta1 = np.cos(theta1)
    sinTheta2 = np.sin(theta2)
    cosTheta2 = np.cos(theta2)
    sinPhi1 = np.sin(phi1)
    cosPhi1 = np.cos(phi1)
    sinPhi2 = np.sin(phi2)
    cosPhi2 = np.cos(phi2)
    l1 = r[:, np.newaxis]
    l1_dot = rdot[:, np.newaxis]

    Omega1 = 2 * np.pi * np.stack([-theta1dot * sinPhi1
        , theta1dot * cosPhi1, phi1dot], axis = 1)
    Omega2 = 2 * np.pi * np.stack([-theta2dot * sinPhi2
        , theta2dot * cosPhi2, phi2dot], axis = 1)

    s1 = np.stack([sinTheta1 * cosPhi1, sinTheta1 *
        sinPhi1, cosTheta1], axis = 1)
    s2 = np.stack([sinTheta2 * cosPhi2, sinTheta2 *
        sinPhi2, cosTheta2], axis = 1)

    pos = np.hstack([s1 * l1, s1 * l1 + s2 * (1 - l1)])

    vel1 = np.hstack([s1 * l1_dot, s1 * l1_dot - s2 *
        l1_dot])
    vel2 = np.hstack([np.cross(Omega1, s1 * l1),
                        np.cross(Omega1, s1 * l1) +
                            np.cross(Omega2, s2 * (1-
                            l1))])

    vel = vel1 + vel2

    return np.hstack([pos, vel])
```
6.7. Lyapunov Exponent mapping and Flipping Time Diagram

Lyupunov

December 3, 2020

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
from mpl_toolkits.mplot3d import Axes3D
from ensemble import *
from IPython.display import HTML
from matplotlib import animation, rc
%matplotlib notebook
```

[3]:
```
num = 3000
epsilon= 1e-8

theta1s = np.random.random(num) * np.pi *2
theta2s = np.random.random(num) * np.pi * 2

phi1s = np.random.random(num) * np.pi * 2 * 0
phi2s = phi1s
theta1sdot = np.random.random(num) * np.pi * 0
theta2sdot = np.random.random(num) * np.pi * 0
phi1sdot = np.random.random(num) * np.pi * 2 * 0
phi2sdot = np.random.random(num) * np.pi * 2 * 0
r = np.ones(num) * 0.5
r_dot = np.random.random(num) * 0

x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
 ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)
x2 = (theta1s + epsilon,phi1s,theta2s,phi2s, r)
v2 = (theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)

l, traj1, traj2 = lyupunov((x, v), (x2, v2), t = 20, num = 1000)

np.savez('2Dsample2.npz', theta1 = theta1s, thetha2 = theta2s, lyupunov = l)

plt.scatter(theta1s, theta2s, c=l, s=2, cmap='seismic')
plt.colorbar()

plt.xlabel(r'$\theta_1$')
plt.ylabel(r'$\theta_2$')
plt.title('Lyupunov Exponent')

plt.savefig('lyupunov3.png', dpi = 300)
plt.show()
```

```
  0%|          | 0/3000 [00:00<?, ?it/s]WARNING:absl:No GPU/TPU found, falling
back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)
100%|      | 3000/3000 [55:21<00:00,  1.11s/it]
100%|      | 3000/3000 [56:15<00:00,  1.13s/it]

<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

[6]:
```
num = 3000
epsilon= 1e-3

theta1s = np.ones(num)* np.pi  # np.random.random(num) * np.pi
theta2s = np.random.random(num) * np.pi

phi1s = np.random.random(num) * np.pi * 2 * 0
phi2s = phi1s
theta1sdot = np.random.random(num) * np.pi * 0
theta2sdot = np.random.random(num) * np.pi * 0
phi1sdot = np.random.random(num) * np.pi * 2 * 0
phi2sdot = np.random.random(num) * np.pi * 2 * 0
r = np.ones(num) * 0.5
r_dot = np.random.random(num) * 0

x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
 ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)
x2 = (theta1s + epsilon,phi1s,theta2s,phi2s, r)
v2 = (theta1sdot,phi1sdot + epsilon,theta2sdot,phi2sdot, r_dot)

l, traj1, traj2 = lyupunov((x, v), (x2, v2), t = 10, num = 500)

np.savez('2Dsample1D.npz', theta1 = theta1s, thetha2 = theta2s, lyupunov = l)

plt.plot(theta2s, l, '.')

plt.xlabel(r'$\theta_2$')
plt.ylabel("Lyupunov Exponent")
plt.title(r'Lyupunov Exponent as a Function of Initial Condition ($\theta_1 =␣
 ↪\pi$)')
```

```
plt.savefig('lyupunov1D.png', dpi = 300)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[4]:
```
data = np.load('2Dsample2.npz')
theta1 = data['theta1']
theta2 = data['thetha2']
lyupunov_dat = data['lyupunov']
list(data.keys())
```

[4]:
```
['theta1', 'thetha2', 'lyupunov']
```

[5]:
```
points = 500
data = np.zeros([points,3])
x = theta1
y = theta2
z = lyupunov_dat

triang = mtri.Triangulation(x, y)
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

ax.triplot(triang, c="#D3D3D3", marker='.', markerfacecolor="#DC143C",
    markeredgecolor="black", markersize=10)

ax.set_xlabel('X')
ax.set_ylabel('Y')
plt.show()

q1 = np.quantile(z, 0.25)
q3 = np.quantile(z, 0.75)
# isBad = z > q3 + 10 * (q3 - q1)

# mask = np.any(isBad[triang.triangles],axis=1)
# triang.set_mask(mask)

idx = np.argsort(z)[::-1]
print(x[idx], y[idx])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
[0.33796975 0.57253308 0.41471229 … 3.2502429  2.95280541 2.59087961]
[0.10160304 3.07044123 5.69341291 … 1.54475049 4.72989789 2.12792928]
```

[11]:
```python
fig = plt.figure()
ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_trisurf(triang, z, cmap='jet')
# ax.scatter(x,y,z, marker='.', s=10, c="black", alpha=0.)
ax.view_init(elev=60, azim=-45)

ax.set_xlabel(r'$\theta_1$')
ax.set_ylabel(r'$\theta_2$')
ax.set_zlabel(r'$\lambda$')
ax.set_title('Lyupunov Exponent as a Function of Initial Condition\n\n')
# plt.savefig('lyupunov_surface.png', dpi = 300)
plt.show()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

[10]:
```python
fig, ax2 = plt.subplots(nrows=1)
ax2.tricontour(x, y, z, levels=10, linewidths=0.5, colors='k')
cntr2 = ax2.tricontourf(x, y, z, levels=20, cmap="RdBu_r")

fig.colorbar(cntr2, ax=ax2)
# ax2.plot(x, y, 'ko', ms=3)
# ax2.set(xlim=(-2, 2), ylim=(-2, 2))
# ax2.set_title('tricontour (%d points)' % npts)

plt.subplots_adjust(hspace=0.5)
plt.xlabel(r'$\theta_1$')
plt.ylabel(r'$\theta_2$')
# ax.set_zlabel(r'$\lambda$')
plt.title('Lyupunov Exponent as a Function of Initial Condition\n')
plt.savefig('lyupunov_surface.png', dpi = 300)

plt.show()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

[2]:
```python
def fliptime(theta1, theta2):
    theta1s = np.array([theta1]); theta2s = np.array([theta2])
    phi1s = np.array([0]); phi2s = np.array([0]);
    r = np.array([0.5]); r_dot = np.array([0.0])
    theta1sdot = np.array([0]) ; theta2sdot = np.array([0]); phi1sdot = np.
 ↪array([0]); phi2sdot = np.array([0])

    x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
 ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)
    state1 = polar2Cartesian(x, v)[0]

    t_max  = 200

    t = np.linspace(0, t_max, num = t_max * 100)
    traj = solveAstrojax(state1, t)

    theta1, theta2, theta1dot, theta2dot,r, r_dot, t1 = Cartesian2Polar(traj, t)
    theta1_motion = np.abs(theta1 - theta1[0])
    theta2 = theta2 % (2 * np.pi) - np.pi
    zero_crossings = np.where(np.diff(np.sign(theta1_motion - 2 * np.pi)))[0]
    if len(zero_crossings) > 0:
        return t1[zero_crossings[0]]
    else:
        return np.nan
```

[3]:
```
# num = 5000
theta1 = 2 * np.pi * np.linspace(0, 1, num = 200)
theta2 = 2 * np.pi * np.linspace(0, 1, num = 200)
t1, t2 = np.meshgrid(theta1, theta2)
t1 = t1.flatten()
t2 = t2.flatten()
print(t1.shape)
isPossible = 2 * np.cos(t1) + np.cos(t2) > - 2
flips = []
for i in tqdm(range(t1.shape[0])):
    if isPossible[i]:
        flips.append(fliptime(t1[i], t2[i]))
    else:
        flips.append(np.nan)

np.savez('flips0.npz', theta1 = t1, theta2 = t2, flip = flips)
```

```
  0%|          | 0/40000 [00:00<?, ?it/s]WARNING:absl:No GPU/TPU found, falling
back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)

(40000,)

/Users/gene/Brown-Class/Juniour-Fall/phys2030/Final Project/code/ensemble.py:65:
RuntimeWarning: invalid value encountered in arccos
  theta2 = np.unwrap(np.arccos((traj[:, 5] - traj[:, 2])/(1-r)) *
np.sign(traj[:, 3] - traj[:, 0]))
/usr/local/opt/pyenv/versions/cv/lib/python3.7/site-
packages/numpy/lib/function_base.py:1520: RuntimeWarning: invalid value
encountered in greater
  _nx.copyto(ddmod, pi, where=(ddmod == -pi) & (dd > 0))
/usr/local/opt/pyenv/versions/cv/lib/python3.7/site-
packages/numpy/lib/function_base.py:1522: RuntimeWarning: invalid value
encountered in less
  _nx.copyto(ph_correct, 0, where=abs(dd) < discont)
100%|      | 40000/40000 [12:26:53<00:00,  1.12s/it]
```

[5]:

```

```

[68]:

```python
dat = np.load('fkips.npz')
theta1 = dat['theta1']
theta2 = dat['thetha2']
flips = dat['flip']
plt.scatter(theta1, theta2, c=flips, s=2, cmap='seismic')
plt.colorbar()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

[68]:

```
<matplotlib.colorbar.Colorbar at 0x194ac20d0>
```

[69]:

```python
theta1 = 2 * np.pi * np.linspace(0, 1, num = 100)
theta2 = 2 * np.pi * np.linspace(0, 1, num = 100)

t1, t2 = np.meshgrid(theta1, theta2)
z = 2 * np.cos(t1) + np.cos(t2) > - 2
print(z)
plt.pcolormesh(t1, t2, z)
```

```
[[ True  True  True … True  True  True]
 [ True  True  True … True  True  True]
 [ True  True  True … True  True  True]
 …
 [ True  True  True … True  True  True]
 [ True  True  True … True  True  True]
 [ True  True  True … True  True  True]]

<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

```
[69]: <matplotlib.collections.QuadMesh at 0x1532d8910>
```

```
[12]: ~np.isnan(flips)
```

```
[12]: array([ True,  True,  True, …, False,  True,  True])
```

```
[35]: dat = np.load('flips0.npz')
      theta1 = dat['theta1']
      theta2 = dat['theta2']
      flips = dat['flip']

      fig, axs = plt.subplots(1, 2, figsize = [8, 4])
      c = axs[0].pcolormesh(theta1.reshape([200,200]), theta2.reshape([200,200]),␣
       ↪flips.reshape([200,200]), cmap = 'jet')
      axs[0].plot([0.5, 1, 1, 0.5, 0.5], [0.5, 0.5, 1.5, 1.5, 0.5], 'pink')
      axs[0].set_xlabel(r'$\theta_1$', fontsize = 15)
      axs[0].set_ylabel(r'$\theta_2$', fontsize = 15)
      plt.colorbar(c)

      dat = np.load('flips4.npz')
      theta1 = dat['theta1']
      theta2 = dat['theta2']
      flips = dat['flip']
      axs[1].pcolormesh(theta1.reshape([100,100]), theta2.reshape([100,100]), flips.
       ↪reshape([100,100]), cmap = 'jet')
      axs[1].set_xlabel(r'$\theta_1$', fontsize = 15)
      axs[1].set_ylabel(r'$\theta_2$', fontsize = 15)
      fig.tight_layout(rect=[0, 0.03, 1, 0.95])
      fig.suptitle(r'The time the astrojax system took for $\theta_1$ to rotate full␣
       ↪$2\pi$ radian')
      # plt.savefig('flip.png', dpi = 300)
```

```
/usr/local/opt/pyenv/versions/cv/lib/python3.7/site-
packages/ipykernel_launcher.py:6: RuntimeWarning: invalid value encountered in
greater



        ␣
 ↪---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call␣
 ↪last)

        <ipython-input-35-41cbb31c62dc> in <module>
         10 z = 2 * np.cos(t1) + np.cos(t2) > - 2
         11 plt.pcolormesh(t1, t2, z)
```

```
   ---> 12 plt.pcolormesh(theta1.reshape([200,200]), theta2.reshape([200,200]),␣
↪flips.reshape([200,200]), cmap = 'jet')
        13 plt.colorbar()


       ValueError: cannot reshape array of size 10000 into shape (200,200)
```

[60]:
```
dat = np.load('flips0.npz')
theta1 = dat['theta1']
theta2 = dat['theta2']
flips = dat['flip']
t1 = np.linspace(0, 2 * np.pi, num = 1000)
t2 = np.linspace(0, 2 * np.pi, num = 1000)

fig, axs = plt.subplots(1, 2, figsize = [8, 4])
c = axs[0].pcolormesh(theta1.reshape([200,200]), theta2.reshape([200,200]),␣
 ↪flips.reshape([200,200]), cmap = 'jet')
axs[0].plot([0.5, 1, 1, 0.5, 0.5], [0.5, 0.5, 1.5, 1.5, 0.5], 'pink')

axs[0].set_xlabel(r'$\theta_1$', fontsize = 15)
axs[0].set_ylabel(r'$\theta_2$', fontsize = 15)
# z = 2 * np.cos(t1) + np.cos(t2) > - 2
axs[0].plot(np.arccos((-2 - np.cos(t2))/2), t2 ,'k')
axs[0].plot(2*np.pi - np.arccos((-2 - np.cos(t2))/2), t2 ,'k')
# axs[0].plot(t1, t2, '.')
plt.colorbar(c)

dat = np.load('flips4.npz')
theta1 = dat['theta1']
theta2 = dat['theta2']
flips = dat['flip']

axs[1].pcolormesh(theta1.reshape([100,100]), theta2.reshape([100,100]), flips.
 ↪reshape([100,100]), cmap = 'jet')
axs[1].set_xlabel(r'$\theta_1$', fontsize = 15)
axs[1].set_ylabel(r'$\theta_2$', fontsize = 15)
fig.tight_layout(rect=[0, 0.03, 1, 0.95])
fig.suptitle(r'The time the astrojax system took for $\theta_1$ to rotate full␣
 ↪$2\pi$ radian')
plt.savefig('flip.png', dpi = 300)
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

```
/usr/local/opt/pyenv/versions/cv/lib/python3.7/site-
packages/ipykernel_launcher.py:15: RuntimeWarning: invalid value encountered in
arccos
  from ipykernel import kernelapp as app
/usr/local/opt/pyenv/versions/cv/lib/python3.7/site-
packages/ipykernel_launcher.py:16: RuntimeWarning: invalid value encountered in
arccos
  app.launch_new_instance()
```

```
[9]: # num = 5000
theta1 = np.linspace(0.5, 1, num = 100)
theta2 = np.linspace(0.5, 1.5, num = 100)
t1, t2 = np.meshgrid(theta1, theta2)
t1 = t1.flatten()
t2 = t2.flatten()
print(t1.shape)
isPossible = 2 * np.cos(t1) + np.cos(t2) > - 2
flips = []
for i in tqdm(range(t1.shape[0])):
    if isPossible[i]:
        flips.append(fliptime(t1[i], t2[i]))
    else:
        flips.append(np.nan)

np.savez('flips4.npz', theta1 = t1, theta2 = t2, flip = flips)
```

```
  0%|          | 0/10000 [00:00<?, ?it/s]

(10000,)

100%|     | 10000/10000 [3:30:06<00:00,  1.26s/it]
```

```
[14]: dat = np.load('flips4.npz')
theta1 = dat['theta1']
theta2 = dat['theta2']
flips = dat['flip']
plt.pcolormesh(theta1.reshape([100,100]), theta2.reshape([100,100]), flips.
 ↪reshape([100,100]), cmap = 'jet')
plt.colorbar()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

```
[14]: <matplotlib.colorbar.Colorbar at 0x160bea2d0>
```

```
[ ]:
```

6.8. Poincare Section

Trajectory

December 3, 2020

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
from mpl_toolkits.mplot3d import Axes3D
from ensemble import *
from IPython.display import HTML
from matplotlib import animation, rc
%matplotlib notebook
```

[2]:
```
num = 1
epsilon= 1e-3
theta1s = np.array([3.0])
theta2s = np.array([1.0])
phi1s = np.array([0])
phi2s = np.array([0])
theta1sdot = np.array([0])
theta2sdot = np.array([0])
phi1sdot = np.array([0])
phi2sdot = np.array([0])
r = np.array([0.5])
r_dot = np.array([0])

x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
 ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)
x2 = (theta1s + epsilon,phi1s,theta2s,phi2s, r)
v2 = (theta1sdot,phi1sdot + epsilon,theta2sdot,phi2sdot, r_dot)

t = np.linspace(0, 10, num = 500)

state1 = polar2Cartesian(x, v)[0]

traj = solveAstrojax(state1, t)

anim = animate2D(traj, t)

anim
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
and rerun for more info.)

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>
```

```
[2]: <matplotlib.animation.FuncAnimation at 0x156ef4490>
```

```
[4]: t = np.linspace(0, 100, num = 10000)

     state1 = polar2Cartesian(x, v)[0]

     traj = solveAstrojax(state1, t)

     theta1, theta2, theta1dot, theta2dot,r, r_dot, t1 = Cartesian2Polar(traj, t)
     theta1 = theta1 % (2 * np.pi) - np.pi
     theta2 = theta2 % (2 * np.pi) - np.pi
     # plt.plot(t1, theta1dot, 'g')
     plt.plot(t1, theta1, 'b.', markersize = 1)
     zero_crossings = np.where(np.diff(np.sign(theta1%(2 * np.pi) - np.pi)))[0]
     zero_crossings = zero_crossings[np.abs(theta1[zero_crossings]) < 1]
     zero_crossings = zero_crossings[theta1dot[zero_crossings] > 0]
     zero_crossings = zero_crossings[r[zero_crossings] > 1e-1]
     plt.plot(t1[zero_crossings], theta1[zero_crossings], 'rx')
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>
```

```
[4]: [<matplotlib.lines.Line2D at 0x1570f9350>]
```

```
[ ]: theta1s = np.array([3.0]); theta2s = np.array([1.0])
     phi1s = np.array([0]); phi2s = np.array([0])
     theta1sdot = np.array([0]) ; theta2sdot = np.array([0]); phi1sdot = np.
      ↪array([0]); phi2sdot = np.array([0])

     r = np.array([0.5])
     r_dot = np.array([0])

     x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
      ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)

     t = np.linspace(0, 20000, num = 2000000)

     state1 = polar2Cartesian(x, v)[0]
```

```
traj = solveAstrojax(state1, t)

theta1, theta2, theta1dot, theta2dot, r, r_dot, t1 = Cartesian2Polar(traj, t)
theta1 = theta1 % (2 * np.pi) - np.pi
theta2 = theta2 % (2 * np.pi) - np.pi
# plt.plot(t1, theta1dot, 'r.')
# plt.plot(t1, theta1, 'b.')
zero_crossings = np.where(np.diff(np.sign(theta1)))[0]
zero_crossings = zero_crossings[np.abs(theta1[zero_crossings]) < 1]
zero_crossings = zero_crossings[theta1dot[zero_crossings] > 0]
zero_crossings = zero_crossings[r[zero_crossings] > 1e-1]

fig, axs = plt.subplots(1, 2, figsize = [4,8])
axs[0].plot(theta2[zero_crossings], theta2dot[zero_crossings], 'k.', markersize␣
 ↪= 3)
axs[0].set_xlabel(r'$\theta_2$', fontsize=20)
axs[0].set_ylabel(r'$\dot{\theta}_2$', fontsize=20)
axs[0].set_ylim([-20, 20])
axs[1].plot(r[zero_crossings], r_dot[zero_crossings], 'k.', markersize = 3)
```

[6]:
```
theta1s = np.array([0.2]); theta2s = np.array([3.0])
phi1s = np.array([0]); phi2s = np.array([0])
theta1sdot = np.array([0]) ; theta2sdot = np.array([0]); phi1sdot = np.
 ↪array([0]); phi2sdot = np.array([0])

r = np.array([0.5])
r_dot = np.array([0])

x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
 ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)

t = np.linspace(0, 10000, num = 1000000)

state1 = polar2Cartesian(x, v)[0]

traj = solveAstrojax(state1, t)

theta1, theta2, theta1dot, theta2dot, r, r_dot, t1 = Cartesian2Polar(traj, t)
theta1 = theta1 % (2 * np.pi) - np.pi
theta2 = theta2 % (2 * np.pi) - np.pi
# plt.plot(t1, theta1dot, 'r.')
# plt.plot(t1, theta1, 'b.')
zero_crossings = np.where(np.diff(np.sign(theta1%(2 * np.pi) - np.pi)))[0]
zero_crossings = zero_crossings[np.abs(theta1[zero_crossings]) < 1]
zero_crossings = zero_crossings[theta1dot[zero_crossings] > 0]
zero_crossings = zero_crossings[r[zero_crossings] > 1e-1]
plt.figure()
plt.plot(theta2[zero_crossings], theta2dot[zero_crossings], 'k.', markersize =␣
 ↪3)
plt.ylim([-20, 20])
plt.figure()
plt.plot(r[zero_crossings], r_dot[zero_crossings], 'k.', markersize = 3)
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>
```

[6]:
```
[<matplotlib.lines.Line2D at 0x15bf95450>]
```

[9]:
```
data = np.load('2Dsample2.npz')
theta1 = data['theta1']
theta2 = data['thetha2']
lyupunov_dat = data['lyupunov']
list(data.keys())
idx = np.argmin(lyupunov_dat)
print(theta1[idx], theta2[idx], lyupunov_dat[idx])
```

```
1.4931048653369354 5.89676180994061 -0.07933807120342448
```

[31]:
```
num = 1
epsilon= 1e-8
theta1s = np.array([0.1])
theta2s = np.array([0.5])
phi1s = np.array([0])
phi2s = np.array([0])
theta1sdot = np.array([0])
theta2sdot = np.array([0])
phi1sdot = np.array([0])
phi2sdot = np.array([0])
r = np.array([0.5])
r_dot = np.array([0])

x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
 ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)
x2 = (theta1s + epsilon,phi1s,theta2s,phi2s, r)
v2 = (theta1sdot, phi1sdot,theta2sdot,phi2sdot, r_dot)

t = np.linspace(0, 20, num = 1000)
```

```
state1 = polar2Cartesian(x, v)[0]
state2 = polar2Cartesian(x2, v2)[0]

traj1 = solveAstrojax(state1, t)
traj2 = solveAstrojax(state2, t)

diffs = np.sum((traj1[:, :] - traj2[:, :])**2, axis = -1)**0.5

plt.plot(t, diffs)
plt.yscale('log')
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

[32]:
```
animate2D(traj1, t)
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

[32]: `<matplotlib.animation.FuncAnimation at 0x1688b57d0>`

[21]:
```
animate2D(traj2, t)
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

[21]: `<matplotlib.animation.FuncAnimation at 0x161ba6c50>`

[57]:
```
theta1s = np.array([np.pi]); theta2s = np.array([np.pi + 0.1])
phi1s = np.array([0]); phi2s = np.array([0])
theta1sdot = np.array([0]) ; theta2sdot = np.array([0]); phi1sdot = np.
 ↪array([0]); phi2sdot = np.array([0])
t = np.linspace(0, 100, num = 10000)
x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
 ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)
state1 = polar2Cartesian(x, v)[0]

traj = solveAstrojax(state1, t)

theta1, theta2, theta1dot, theta2dot,r, r_dot, t1 = Cartesian2Polar(traj, t)
```

```
theta1_motion = np.abs(theta1 - theta1[0])
theta2 = theta2 % (2 * np.pi) - np.pi
# plt.plot(t1, theta1dot, 'g')
plt.plot(t1, theta1_motion, 'b.', markersize = 1)
zero_crossings = np.where(np.diff(np.sign(theta1_motion - 2 * np.pi)))[0]
plt.plot(t1[zero_crossings], theta1_motion[zero_crossings], 'rx')
print(t1[zero_crossings])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
[]
```

[67]:

```

```

[76]:

```

```

```
7it [00:07,  1.18s/it]/Users/gene/Brown-Class/Juniour-Fall/phys2030/Final
Project/code/ensemble.py:65: RuntimeWarning: invalid value encountered in arccos
  theta2 = np.unwrap(np.arccos((traj[:, 5] - traj[:, 2])/(1-r)) *
np.sign(traj[:, 3] - traj[:, 0]))
/usr/local/opt/pyenv/versions/cv/lib/python3.7/site-
packages/numpy/lib/function_base.py:1520: RuntimeWarning: invalid value
encountered in greater
  _nx.copyto(ddmod, pi, where=(ddmod == -pi) & (dd > 0))
/usr/local/opt/pyenv/versions/cv/lib/python3.7/site-
packages/numpy/lib/function_base.py:1522: RuntimeWarning: invalid value
encountered in less
  _nx.copyto(ph_correct, 0, where=abs(dd) < discont)
100it [01:48,  1.09s/it]
```

[77]:

```
plt.scatter(theta1, theta2, c=flips, s=2, cmap='seismic')
plt.colorbar()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[77]:

```
<matplotlib.colorbar.Colorbar at 0x173603050>
```

[78]:

```
num = 1
epsilon= 1e-3
theta1s = np.array([3.0])
theta2s = np.array([3.0])
```

```
phi1s = np.array([0])
phi2s = np.array([0])
theta1sdot = np.array([0])
theta2sdot = np.array([0])
phi1sdot = np.array([0])
phi2sdot = np.array([0])
r = np.array([0.5])
r_dot = np.array([0])

x, v = (theta1s,phi1s,theta2s,phi2s, r),␣
 ↪(theta1sdot,phi1sdot,theta2sdot,phi2sdot, r_dot)
x2 = (theta1s + epsilon,phi1s,theta2s,phi2s, r)
v2 = (theta1sdot,phi1sdot + epsilon,theta2sdot,phi2sdot, r_dot)

t = np.linspace(0, 10, num = 500)

state1 = polar2Cartesian(x, v)[0]

traj = solveAstrojax(state1, t)

anim = animate2D(traj, t)

anim
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>
```

```
[78]: <matplotlib.animation.FuncAnimation at 0x177543e50>
```

```
[ ]:
```