Christopher Martinez
MAE 271
Prof. Margolis

Lab 2 Report

Problem Overview

An inverted pendulum attached to a vertically oscillating cart can counterintuitively stabilize in the upright position despite initial angular displacements. This lab investigates the stabilization by mapping regions in the frequency-amplitude parameter space that result in convergence to the upright position for initial angles of 10°, 20°, and 30°, and explores whether this behavior extends to larger angles up to 90°.

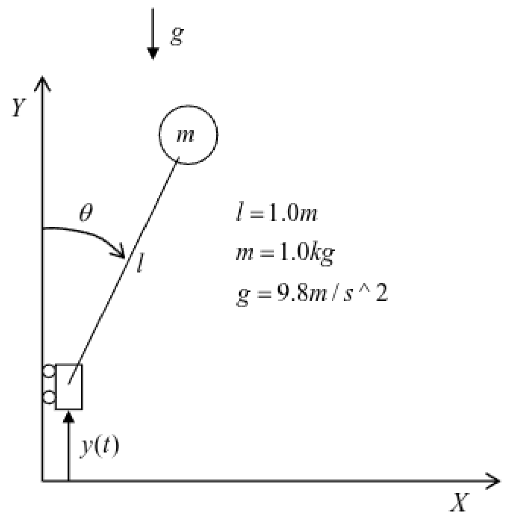

Figure 1: Inverted Pendulum System

Simulation Purpose

By creating a simulation of the inverted pendulum system, the stability of any given system can be observed and measured, allowing the space of all variable parameters to be mapped for stability by sampling a range of amplitudes and frequencies for the cart's motion at a given starting angle θ. Because stable systems will bias towards an upright position, each simulation can measure the time spent above a critical angle to assign as a score. Then, by iteratively simulating new points across the parameter space, this score transforms the amplitude-frequency parameter space into a heat map for a single starting angle, distinguishing regions of stability from instability.

Simulation Setup

Using Python, a time-step environment is created to run with a maximum time span of 10 seconds and a time step of 1 ms. The equations of motion necessary to perform Euler integration are the pendulum's angular velocity and the horizontal force acting on the pendulum mass, both derived from the system bond graph, depicted below. All simulations used the following fixed parameters: pendulum mass (m) = 1.0 kg, pendulum length (l) = 1 m, and gravity = 9.8 m/s^2. A critical angle was set at ±95°, and simulations that exceed this angle would end early. Simulations will be scored according to time elapsed above this critical angle to select for stable parameters. The parameter space encompassed amplitudes from -0.5 m to 0.5 m, where the negative amplitude indicates that the cart begins moving downwards rather than upwards, and frequencies from 0 to 100 Hz. Points within the parameter space were selected using an evenly spaced 100 by 100 grid. For a given starting angle, 10,000 simulations are performed in order to create a stability map within the parameter space. Given three starting angles of 10°, 20°, and 30°, and three additional angles to explore the potential for stable systems at higher starting displacements: 70°, 80°, and 90°, a total of 60,000 simulations will need to be performed.

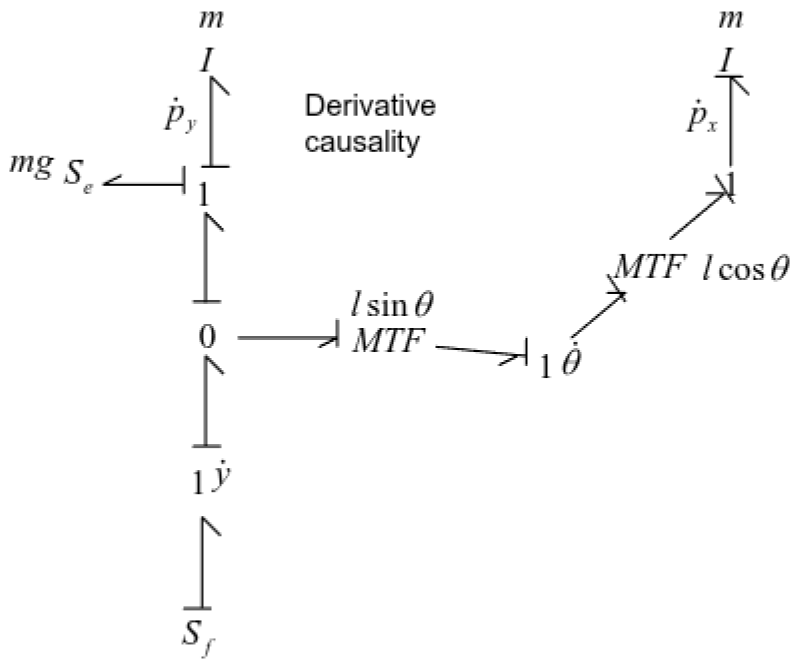

Figure 2: System Bond Graph

$$\dot{\theta} = \frac{1}{l\cos\theta}\frac{p_x}{m}$$

Figure 3: Angular Velocity of the Pendulum

$$\dot{p}_x = mg\sin\theta\cos\theta + m\ddot{y}\sin\theta\cos\theta - \frac{\sin\theta}{\cos\theta}\dot{\theta}p_x$$

Figure 4: Horizontal Force on the Pendulum Mass

<u>Results</u>

When performing the simulations, stable systems ran to completion (10 seconds) while systems exceeding the maximum displacement angle were terminated early and scored by their elapsed time. The resulting heatmaps show stable systems in white and unstable systems with a color gradient indicating time-to-failure. While analyzing the systems at the boundary of stable and unstable systems, an unexpected behavior was observed; the pendulum neither converged to the upright position nor diverged into a chaotic state, but rather oscillated in a tight horizontal position just above 90°. Because these stable systems do not “converge to the upright position” this new mode of stability will be referred to as “pseudo-stability”. As a creative liberty, pseudo-states were reevaluated to have black pixels to distinguish it from both the stable and unstable regions.

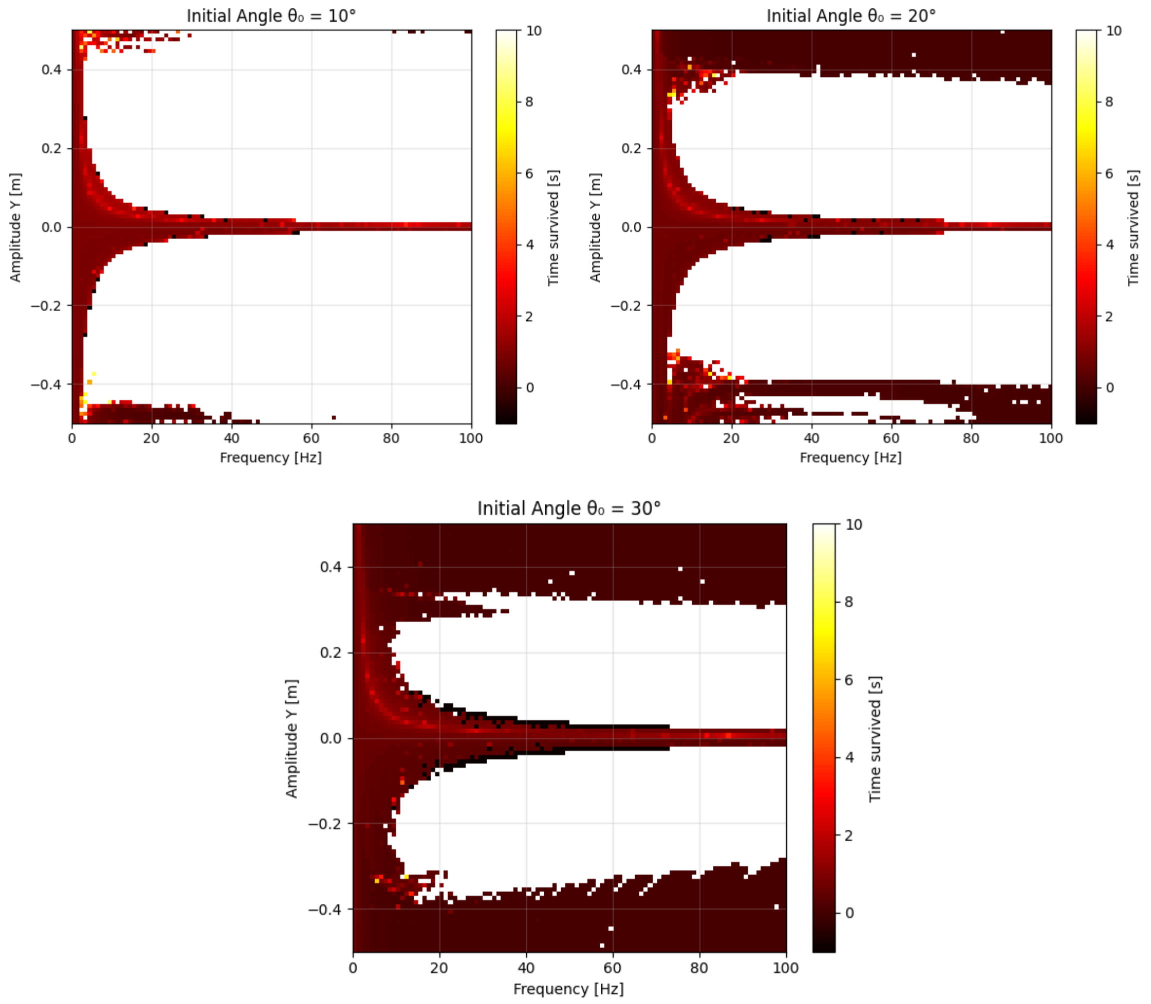

Figure 5: Parameter Space Heat Maps for Small Starting Angles (10°, 20°, and 30°)

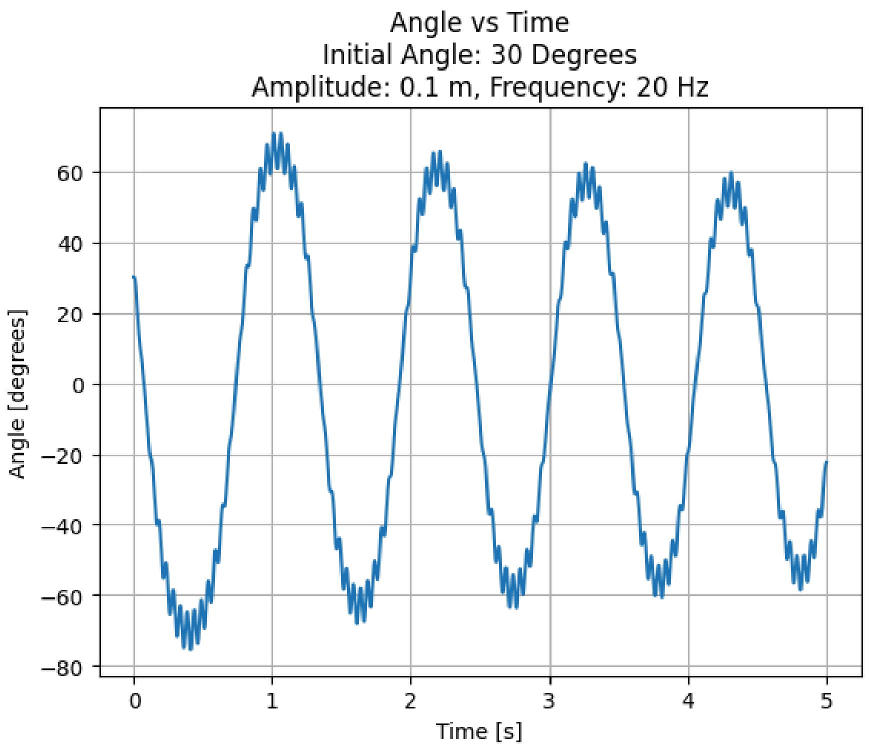

Figure 6: Stable System Time Evolution

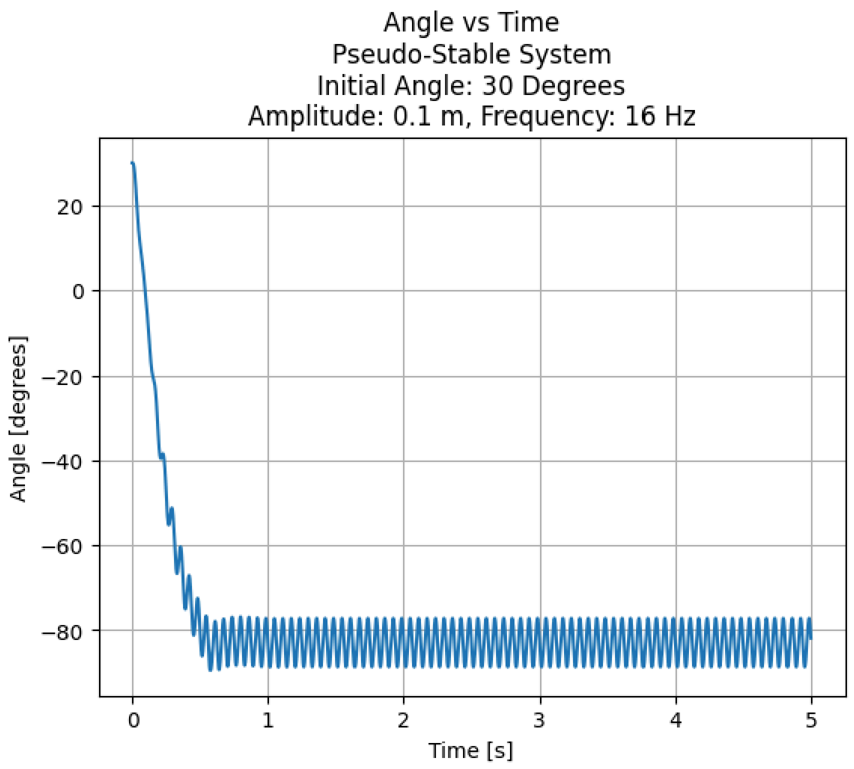

Figure 7: Pseudo-Stable System Time Evolution

Additional parameter spaces were simulated at initial angles of 70°, 80°, and 90° to investigate if higher starting angles would be capable of achieving stable systems. The results reveal significant regions for both stable and pseudo-stable systems, confirming that even at 90° initial displacement, the system can achieve upright stability. Notably, when the starting angle is 90°, there is a significantly larger region of pseudo-stable systems present than in any other parameter space.

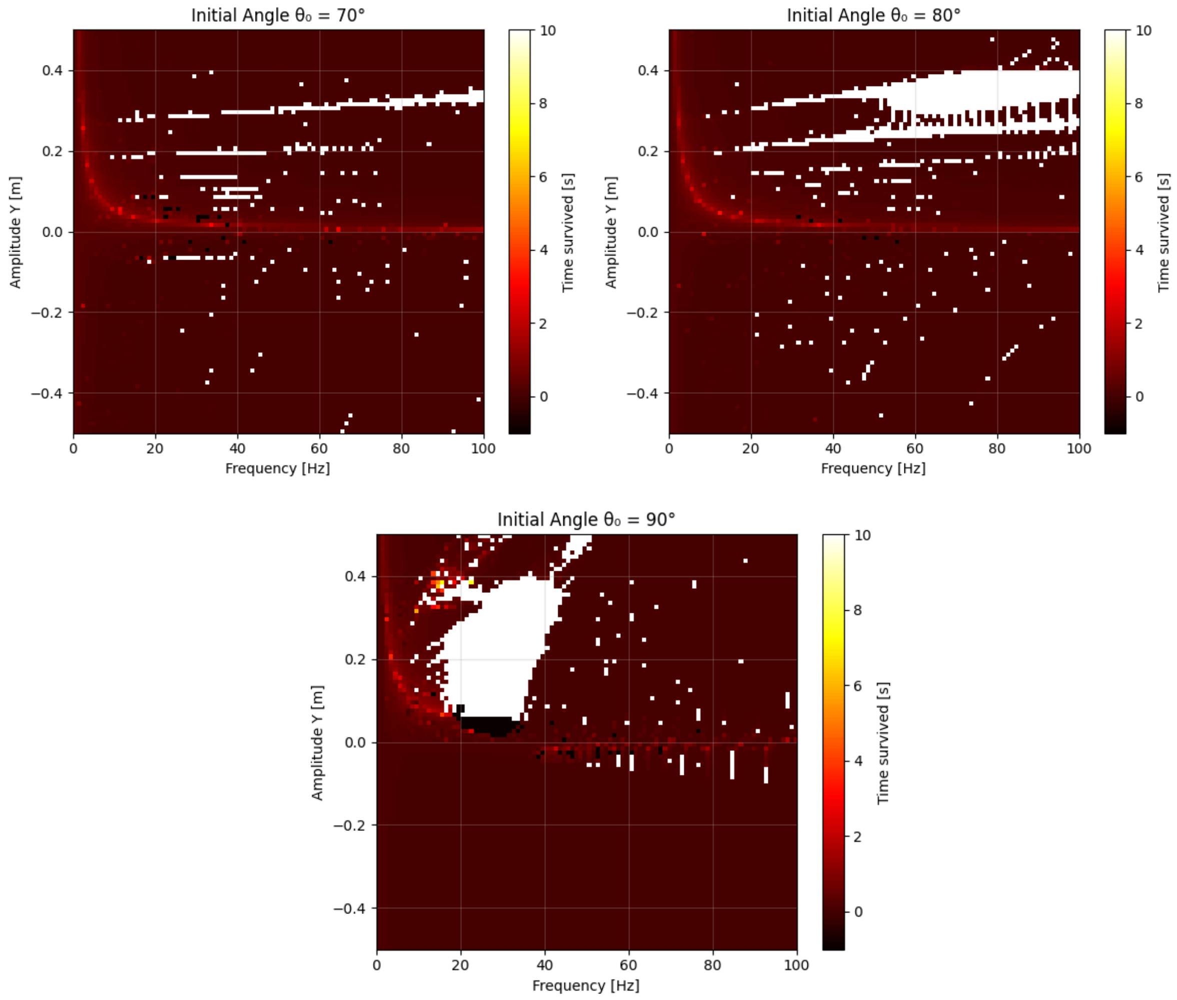

Figure 8: Parameter Space Heat Maps for Large Starting Angles (70°, 80°, and 90°)

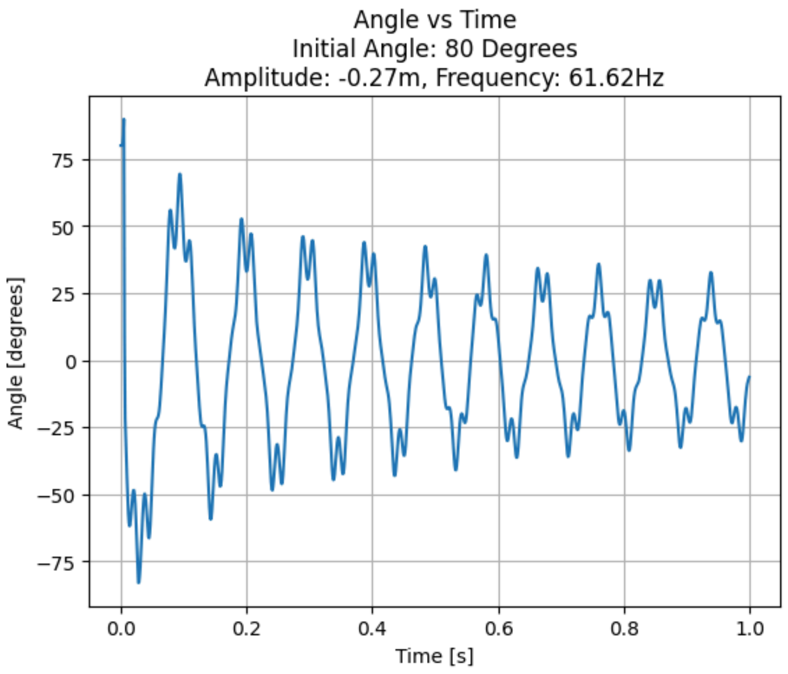

Figure 9: Stable System at a High Starting Angle

Discussion

While the decision to enforce a maximum angle may have resulted in potentially stable parameters to be incorrectly marked as unstable, the heatmaps still reveal a subset of the true regions of stability within the parameter space. Likewise, the angle over time charts clearly demonstrate convergence towards the vertical position for stable systems and a horizontal position in pseudo-stable systems. The stabilizing behavior of this inverted pendulum system has therefore been verified for systems with an initial displacement angle of up to 90 degrees.

Conclusion

By programming an inverted pendulum system attached to a vertically oscillating cart in Python, simulations were performed across a limited parameter space to determine the amplitude and frequencies of the cart's motion that stabilizes the pendulum into an upright position. Stable regions in parameter space were found and verified for systems with starting angles up to 90 degrees, but does not exclude the possibility of greater starting angles also being capable of achieving stability. Finally, by imposing a simulation constraint that enforced a minimum number of oscillations about the vertical position, some stable systems distinguished themselves with a new behavior aligning to a horizontal position, oscillating just above 90 degrees from the upright position and remaining nearly parallel to the ground.

Appendix

Python Code – Jupyter Notebook Implementation

```python
'''
JUPYTER NOTEBOOK: CELL 1
'''
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

class Cart_and_mass:
    def __init__(self, Y, w, theta, length=1) -> None:
        # Parameters
        self.mass = 1            # kg
        self.length = length     # bar length
        self.gravity = 9.8       # m/s^2
        self.amplitude = Y   # y(t) = Y*sin(w*t)
        self.omega = w       #         ^     ^
        # Initial state
        self.time = 0
        self.cart_y = 0      # cart position (initial: y=0)
        self.theta = theta   # bar angle (y-axis --> bar)
        self.theta_dot = 0
        self.p_x = 0     # mass momentum in the x-direction
        self.p_y = 0     # mass momentum in the y-direction (unused)
        self.mass_x = self.length*np.cos(self.theta) # UNUSED position data
        self.mass_y = self.length*np.sin(self.theta) # UNUSED position data
        # States
        self.history = [self.get_state()]    # use to store simulation data
(states and observables)
        self.history_desc = "[cart_y, theta, theta_dot, p_x, time]"
        self.observable = [self.get_observables]

    def get_state(self)->list:
        return [
            self.cart_y,
            self.theta,
            self.theta_dot,
            self.p_x,
            self.time
        ]
    def get_observables(self)->list:
        return [
            self.cart_y,
            self.theta,
            self.theta_dot,
            self.p_x,
            self.p_y,
            self.mass_x,
            self.mass_y,
            self.time,
        ]

    def get_cart_position(self, time=None):
        if time==None:
            time = self.time
        return self.amplitude*np.sin(self.omega*time)
    def get_cart_velocity(self, time=None):
        if time==None:
            time = self.time
        return self.amplitude*self.omega*np.cos(self.omega*time)
    def get_cart_acceleration(self, time=None):
        if time==None:
            time = self.time
        return -(self.omega**2)*self.amplitude*np.sin(self.omega*time)  #
-(w^2)Ysin(wt)

    def calc_Px_dot(self, cart_acceleration, theta, theta_dot, P_x):
        '''
        Formula (4) from the lab pdf. Must provide the function parameters
        '''
        term1 = self.mass*self.gravity*np.sin(theta)*np.cos(theta)
        term2 = self.mass*cart_acceleration*np.sin(theta)*np.cos(theta)
        term3 = (np.sin(theta)/np.cos(theta))*theta_dot*P_x
        return term1 + term2 - term3    # subtracting term3, see eq (4) in
the pdf
    def calc_Py_dot(self, cart_acceleration, theta, theta_dot, P_x):
        '''
        Formula (2)
        '''
        P_x_dot = self.calc_Px_dot(
            cart_acceleration=cart_acceleration,
            theta=theta,
            theta_dot=theta_dot,
            P_x=P_x
            )
        #terms inside parentheses
        term1 = cart_acceleration
        term2 = (np.sin(theta)/np.cos(theta))*(P_x_dot/self.mass)
        term3 = (P_x/self.mass)*(theta_dot/(np.cos(theta)**2))
        return self.mass*(term1 - term2 - term3)
    def calc_theta_dot(self, theta, P_x):
        '''
        Equation (5)
        '''
        return (1/(self.length*np.cos(theta)))*(P_x/self.mass)

    def update_observables(self):
        #update the unimportant stuff (Py, mass x&y, )
        pass

    def time_step(self, dt):
        # increment time (current state still holds "previous" values)
        cart_accel = self.get_cart_acceleration(self.time) #instantaneous
acceleration
        self.time += dt
        self.cart_y = self.get_cart_position(self.time)
        # calculating forces acting on the mass
        Fx = self.calc_Px_dot(
            cart_acceleration=cart_accel,
            theta=self.theta,
            theta_dot=self.theta_dot,
            P_x=self.p_x
        )
        # inserting new values: state = [cart_y, theta, theta_dot, p_x,
time]
        self.theta_dot = self.calc_theta_dot(
            theta=self.theta,
            P_x=self.p_x
            )
        self.p_x += Fx*dt  # new momentum
        self.theta += self.theta_dot*dt # updated angle
        if abs(self.theta) > np.pi*2:
            self.theta %= 2*np.pi           # constrain from 0 to 2pi
        # Updates
        self.history.append(self.get_state())
        #TODO: track observables
        return
    def reset(self):
        #TODO:return to initial state
        pass

    def search_time_step(self, dt):
        self.time_step(dt)
        if self.theta > np.deg2rad(90) or self.theta < np.deg2rad(-90):
            raise Exception(f"Angle out of bounds:
{self.get_state()=}\n{self.history_desc}")
```

```
'''
CELL 2
'''
def deg_to_rad(deg):
    return deg*np.pi/180

# Simulation with provided parameters
y = 0.1 # m
f = 20  # Hz
sim = Cart_and_mass(
    Y=y,
    w=2*np.pi*f,
    theta=deg_to_rad(30),
    length=1
    )
dt = 0.001
time_span = np.arange(0,100,dt)
for moment in time_span:
    sim.time_step(dt)
states = sim.history[:len(sim.history)]
x = [t[-1] for t in states]
y = [np.rad2deg(t[1]) for t in states]
```

```
plt.plot(x,y)
'''
CELL 3
'''
# Creating a phase space for simulation parameters and to score simulation
stability
class Phase_space:
   def __init__(self, angles = [deg_to_rad(num) for num in [10,20,30]]):
       self.angles = angles
       self.Y_space = np.linspace(-.5, .5, 100)
       self.w_space = 2*np.pi*np.linspace(0, 100, 100)
       self.PS_times = np.array(

[[np.zeros(len(self.w_space))]*len(self.Y_space)]*len(self.angles)
           )
# PS_times array index as follows: [i_angle, i_Y, i_w]
   def query_phase_space(self, Y_val, w_val):
       closest_indecies = []
       for i_y, val in enumerate(self.Y_space):
           if val > Y_val:
               closest_indecies.append(i_y)
               break
       for i_w, val in enumerate(self.w_space):
           if val > w_val:
               closest_indecies.append(i_w)
               break
       return closest_indecies # not accurate but good enough

def simulation(angle, amplitude, omega, time_span, dt, test=False):
   temp_sim = Cart_and_mass(
       Y=amplitude,
       w=omega,
       theta=angle
   )
   threshold = 95
   positive = (temp_sim.theta > 0)
   oscillation_count = 0
   for moment in time_span:
       temp_sim.time_step(dt)
       if positive and (temp_sim.theta < 0):
           print(moment, "oscillated to negative") if test else ""
           oscillation_count+=1
           positive = False
       elif (not positive) and (temp_sim.theta > 0):
               print(moment, "oscillated to positive") if test else ""
               positive = True
               oscillation_count+=1
       if abs(np.rad2deg(temp_sim.theta)) > threshold:
           print(moment, "Over 90 degrees") if test else ""
           return moment
   if oscillation_count < 5:
       return -1 # stable, but not around the vertical position
   return time_span[-1]
'''
CELL 4
'''
from joblib import Parallel, delayed
# Separated param setup to avoid overwriting data if the next cell is
accidentally reran
data = Phase_space()
dt = 0.001
max_time = 10
time_span = np.arange(0, max_time, dt)
# Create list of all parameter combinations
params = []
for i_angle, angle in enumerate(data.angles):
   for i_Y, amplitude in enumerate(data.Y_space):
       for i_w, omega in enumerate(data.w_space):
           params.append((i_angle, i_Y, i_w, angle, amplitude, omega))
'''
CELL 5
'''
if params != []:
   print(f"Running {len(params)} simulations in parallel...")
   # Run simulations in parallel
   results = Parallel(n_jobs=-2, verbose=5)(
       delayed(simulation)(angle, amplitude, omega, time_span, dt) #
creates function...
       for i_angle, i_Y, i_w, angle, amplitude, omega in params    #
...for each set of parameters
```

```
    )
    # Fill in the results matrix
    for (i_angle, i_Y, i_w, *_), time in zip(params, results):
        data.PS_times[i_angle, i_Y, i_w] = time
params = [] # prevents me from overwriting the data if I rerun this cell
print("Grid search complete!")
'''
CELL 6
'''
# Same as previous cells, but mapping the phase space for 70, 80, and 90
degrees
large_angle_data = Phase_space([deg_to_rad(num) for num in [70,80,90]])
# Searching phase space for large angles
large_angle_params = []
for i_angle, angle in enumerate(large_angle_data.angles):
    for i_Y, amplitude in enumerate(large_angle_data.Y_space):
        for i_w, omega in enumerate(large_angle_data.w_space):
            large_angle_params.append((i_angle, i_Y, i_w, angle, amplitude,
omega))
'''
CELL 7
'''
if large_angle_params != []:
    print(f"Running {len(large_angle_params)} simulations in parallel...")
    # Run simulations in parallel
    large_angle_results = Parallel(n_jobs=-2, verbose=5)(
        delayed(simulation)(angle, amplitude, omega, time_span, dt)
        for i_angle, i_Y, i_w, angle, amplitude, omega in
large_angle_params
    )
    # Fill in the results matrix
    for (i_angle, i_Y, i_w, *_), time in zip(large_angle_params,
large_angle_results):
        large_angle_data.PS_times[i_angle, i_Y, i_w] = time

large_angle_params = []
print("Grid search complete!")
'''
CELL 8
'''
```

```
import matplotlib.pyplot as plt
import numpy as np

def heatmap(angle_labels = ['10°', '20°', '30°'], PS=data,
title='stability_heatmaps.png'):
   fig, axes = plt.subplots(1, 3, figsize=(18, 5))
   for i, (ax, angle_label) in enumerate(zip(axes, angle_labels)):
       # Extract the 2D slice for this angle
       time_grid = PS.PS_times[i, :, :]
       # Create heatmap
       im = ax.imshow(time_grid,
                   extent=[0, 100, -0.5, 0.5],  # [freq_min, freq_max,
Y_min, Y_max]
                   aspect='auto',
                   origin='lower',
                   cmap='hot',
                   vmin=-1,
                   vmax=10)

       ax.set_xlabel('Frequency [Hz]')
       ax.set_ylabel('Amplitude Y [m]')
       ax.set_title(f'Initial Angle θ₀ = {angle_label}')
       ax.grid(True, alpha=0.3)
       cbar = plt.colorbar(im, ax=ax)
       cbar.set_label('Time survived [s]')
   plt.tight_layout()
   plt.savefig(title, dpi=150, bbox_inches='tight')
   plt.show()
   print("Heatmap saved as 'stability_heatmaps.png'")
'''
CELL 9
'''
heatmap(PS=data)
heatmap(angle_labels=['70°', '80°', '90°'], PS=large_angle_data,
title="large angle heatmap.png")
'''
CELL 10
'''
def angle_time_plot(times, angles, radians=True, fig=0, title="Angle vs
Time"):
   if radians:
       angles = np.rad2deg(angles)
   plt.figure(fig)
   plt.plot(times, angles)
   plt.grid(True)
   plt.title(title)
   plt.xlabel("Time [s]")
   plt.ylabel("Angle [degrees]")
   plt.show()

def plot_sim(Y, w, theta, time_span, dt, title="Angle vs Time",
subset_ratio=1.0):
   temp_sim = Cart_and_mass(
       Y=Y,
       w=w,
       theta=theta
   )
   for moment in time_span:
       temp_sim.time_step(dt)
   states = temp_sim.history
   subset = states[:int(len(states)//(1/subset_ratio))]
   times = [state[-1] for state in subset]
   angles = [np.rad2deg(state[1]) for state in subset]

   angle_time_plot(times=times, angles=angles, radians=False, title=title)
'''
CELL 11
'''
# Lone point on the 80 degree heat map
plot_sim(
   Y = large_angle_data.Y_space[23],
   w = large_angle_data.w_space[61],
   theta = deg_to_rad(80),
   time_span=time_span,
   dt=dt,
   title=f"Angle vs Time\nInitial Angle: 80 Degrees\nAmplitude:
{large_angle_data.Y_space[23]:.2f}m, Frequency:
{large_angle_data.w_space[61]/(2*np.pi):.2f}Hz",
   subset_ratio=.1,
)
```

```
'''
CELL 12
'''
# Point on the 90 degree heat map
plot_sim(
   Y = large_angle_data.Y_space[75],
   w = large_angle_data.w_space[25],
   theta = deg_to_rad(90),
   time_span=time_span,
   dt=dt,
   title=f"Angle vs Time\nInitial Angle: 90 Degrees\nAmplitude:
{large_angle_data.Y_space[75]:.2f}m, Frequency:
{large_angle_data.w_space[25]/(2*np.pi):.2f}Hz",
   subset_ratio=.5,
)
'''
CELL 13
'''
# Provided Parameters (Y=0.1, f=20)
plot_sim(
   Y = 0.1,
   w = 20*2*np.pi,
   theta = deg_to_rad(30),
   time_span=time_span,
   dt=dt,
   title="Angle vs Time\nInitial Angle: 30 Degrees\nAmplitude: 0.1 m,
Frequency: 20 Hz",
   subset_ratio=.5,
)
'''
CELL 14
'''
# Pseudo Stable System
plot_sim(
   Y = 0.1,
   w = 16*2*np.pi,
   theta = deg_to_rad(30),
   time_span=time_span,
   dt=dt,
   title="Angle vs Time\nPseudo-Stable System\nInitial Angle: 30 Degrees\nAmplitude: 0.1 m, Frequency: 16 Hz",
   subset_ratio=.5,
)
```