Chaos

- **Author:** Aidan Andrews
- **Date:** 24 February 2025
- **Time spent on this assignment:** 4 hours

Note: You must answer things inside the answer tags as well as questions which have an **A:**.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
#from jax.config import config
#config.update("jax_enable_x64", True)
#from jax import jit, grad
#import jax.numpy as jnp
#import jax

import matplotlib.animation as animation
from IPython.display import HTML
def resetMe(keepList=[]):
    ll=%who_ls
    keepList=keepList+['resetMe','np','plt','math','jax','jnp','jit','grad',
    for iiii in keepList:
        if iiii in ll:
            ll.remove(iiii)
    for iiii in ll:
        jjjj="^"+iiii+"$"
        %reset_selective -f {jjjj}
    ll=%who_ls
    plt.rcParams.update({"font.size": 14})
    return
resetMe()
import datetime;datetime.datetime.now()
```

Out[1]:
```
datetime.datetime(2025, 2, 28, 1, 35, 8, 673135)
```

In this project we are going to learn about pendula and chaos. We will start with a single pendulum, examining phase space diagrams and animating trajectories, and moving toward a double pendulum. Then we'll wrap up by using automatic differentiation to look at the same problem, recast in a different form, to write a more general solution to the double pendulum problem.

Exercise 1. Single Pendulum

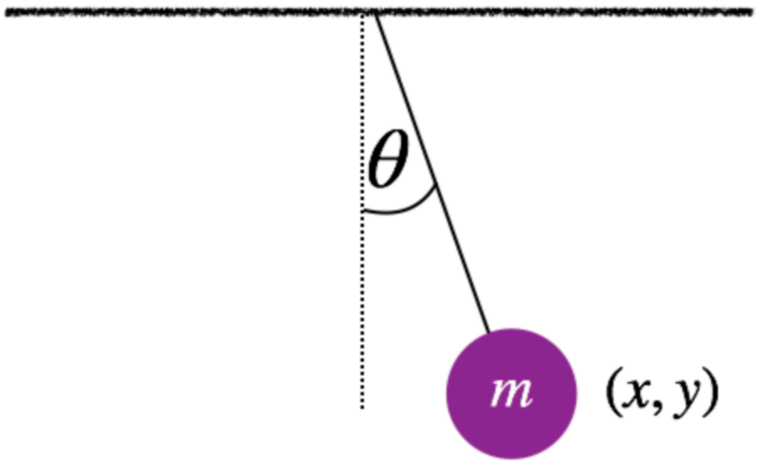
- **List of collaborators:**
- **References you used in developing your code:**

You will use the following animation code for exercise 1

In [19]:
```
def animateMe_singlePendula(positions):
    positionArray = []
    for position in positions:
        theta1 = position[:, 0]
        x1 = params['l1'] * np.sin(theta1)
        y1 = -params['l1'] * np.cos(theta1)
        l = len(x1)
        pos = np.zeros((l, 2))
        pos[:, 0] = x1
        pos[:, 1] = y1
        positionArray.append(pos)

    fig, ax = plt.subplots()
    x_min = np.min([np.min(pos[:, 0]) for pos in positionArray]) * 1.1
    x_max = np.max([np.max(pos[:, 0]) for pos in positionArray]) * 1.1
    y_min = np.min([np.min(pos[:, 1]) for pos in positionArray]) * 1.1
    y_max = np.max([np.max(pos[:, 1]) for pos in positionArray]) * 1.1
    y_max = np.max([y_max, 0.1])
    y_max = np.max([y_max, x_max])
    x_max = y_max
    y_min = np.min([y_min, x_min])
    x_min = y_min
    ax = plt.axes(xlim=(x_min, x_max), ylim=(y_min, y_max))
    ax.plot([0], [0], 'o')
    lines = []
    for pos in positionArray:
        lines.append(ax.plot([], [], 'o', color="g")[0])
        lines.append(ax.plot([], [], '-', color="g")[0])

    def update(i, positionArray, lines):
        for idx, pos in enumerate(positionArray):
            lines[2*idx].set_data([pos[i, 0]], [pos[i, 1]])
            lines[2*idx+1].set_data([0, pos[i, 0]], [0, pos[i, 1]])
        return lines

    ani = animation.FuncAnimation(fig, update, len(positionArray[0]), fargs=
                                  interval=20, blit=True, repeat=False)
    plt.close()
    return ani
```

a. A Single Pendula

When working with pendula, instead of keeping track of the position (x, y), we instead are going to keep track of the angle θ.



In this exercise, we are going to simulate a pendulum with a rigid rod L and a fixed mass m at the end of it. We can define the position of the pendulum by the angle it makes with respect to hanging down (at $\theta = 0$).

You can force your code to use theta simply by giving initial conditions and velocities for a single dimension (i.e. `params['initPos']=np.array([0.1])` would start you pendulum at 0.1 radians.

The "force" (angular acceleration) of a single pendulum is $\alpha(\theta) = -g\sin(\theta)/L$. Modify the force function to use this new "force".

🦉 Run a simulation with no initial velocity and an initial angle of $\theta = 0.6$ and a $dt = 0.01$ for a time $T = 10$. Set $L = 1$ (`params['l1']=1`).

Plot

- θ vs. t
- y vs. x (include the origin on this plot, and equalize the axes scale)
- a *phase space diagram* (ω vs θ)
- and animate the pendula.

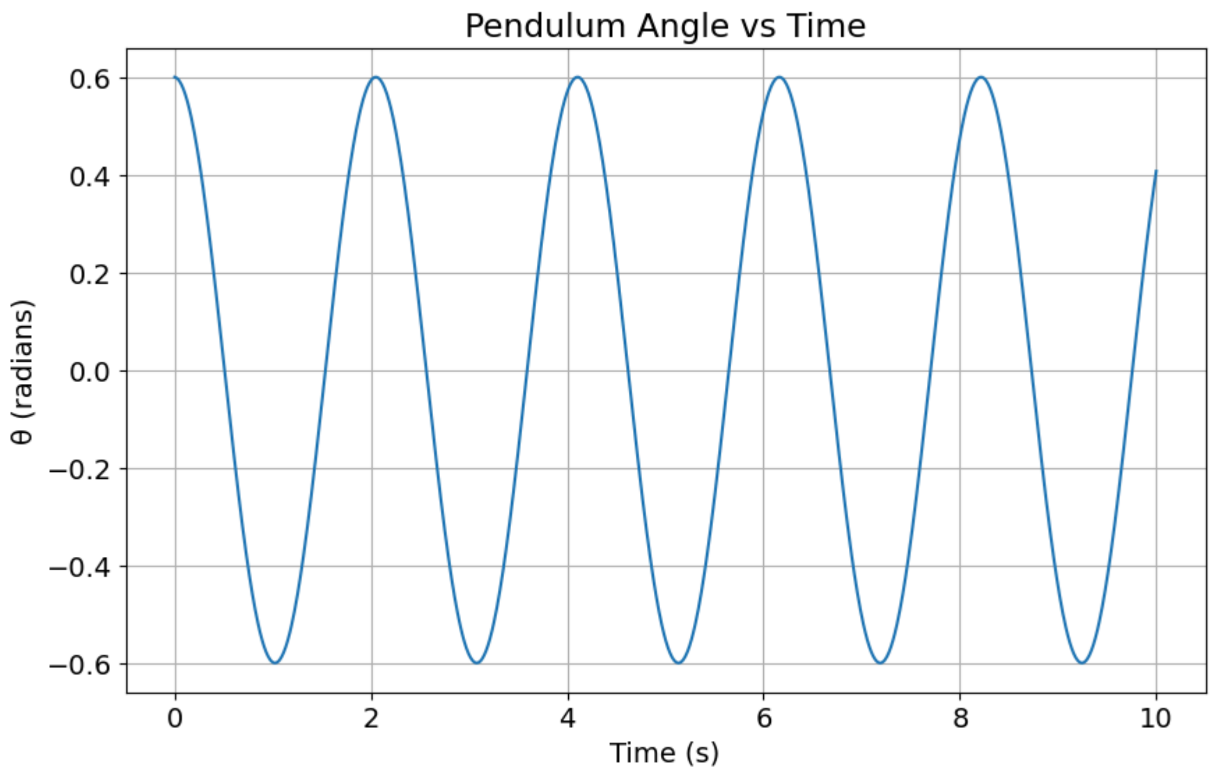
A phase space diagram is a graph displaying the position and velocity of an object on the abscissa (x axis) and ordinate (y axis). Since an undamped, undriven harmonic oscillator moves with

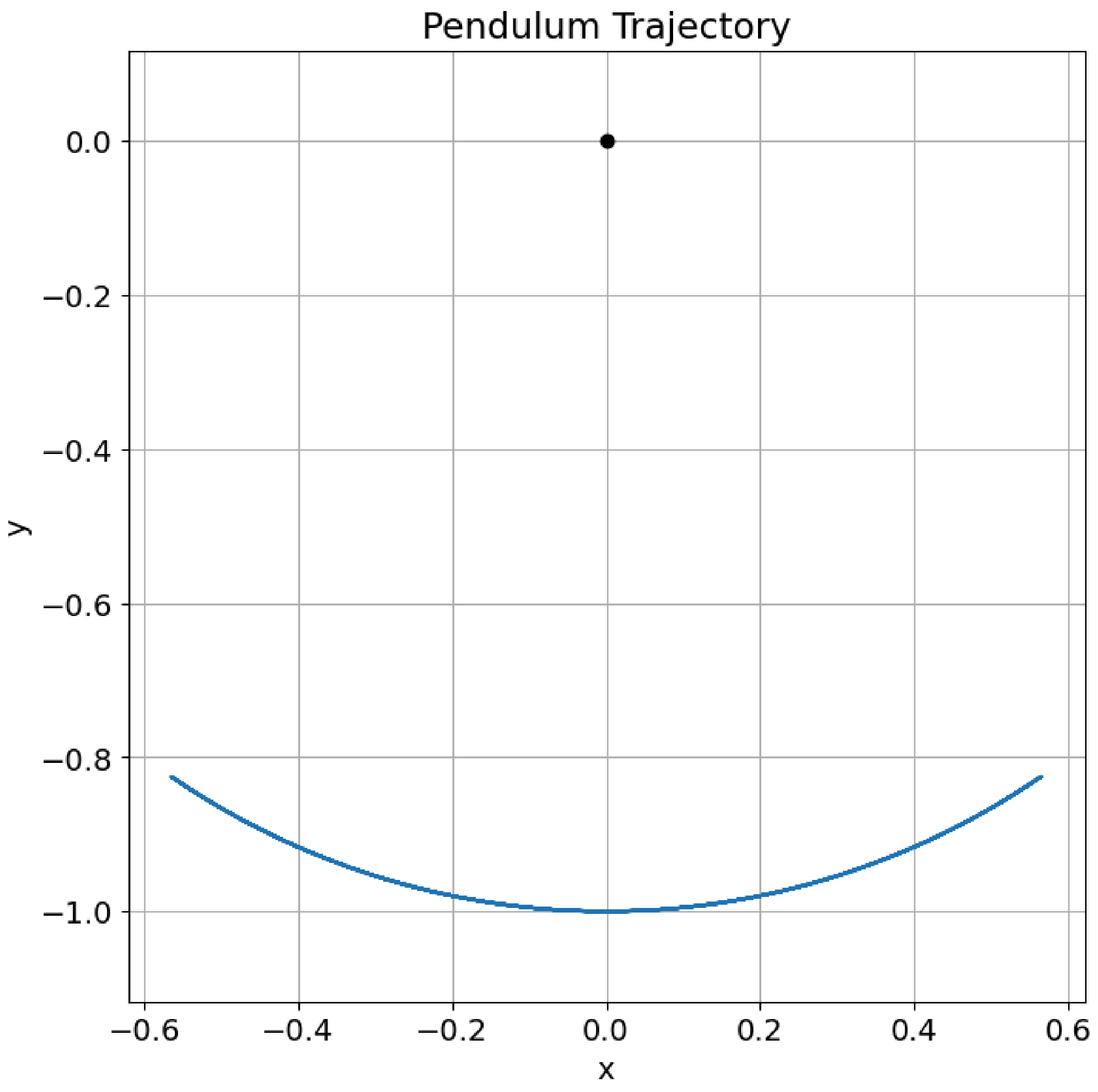
$$\theta(t) = A\cos(\Omega t + \phi) \qquad\qquad \omega(t) = -A\Omega\sin(\Omega t + \phi)$$

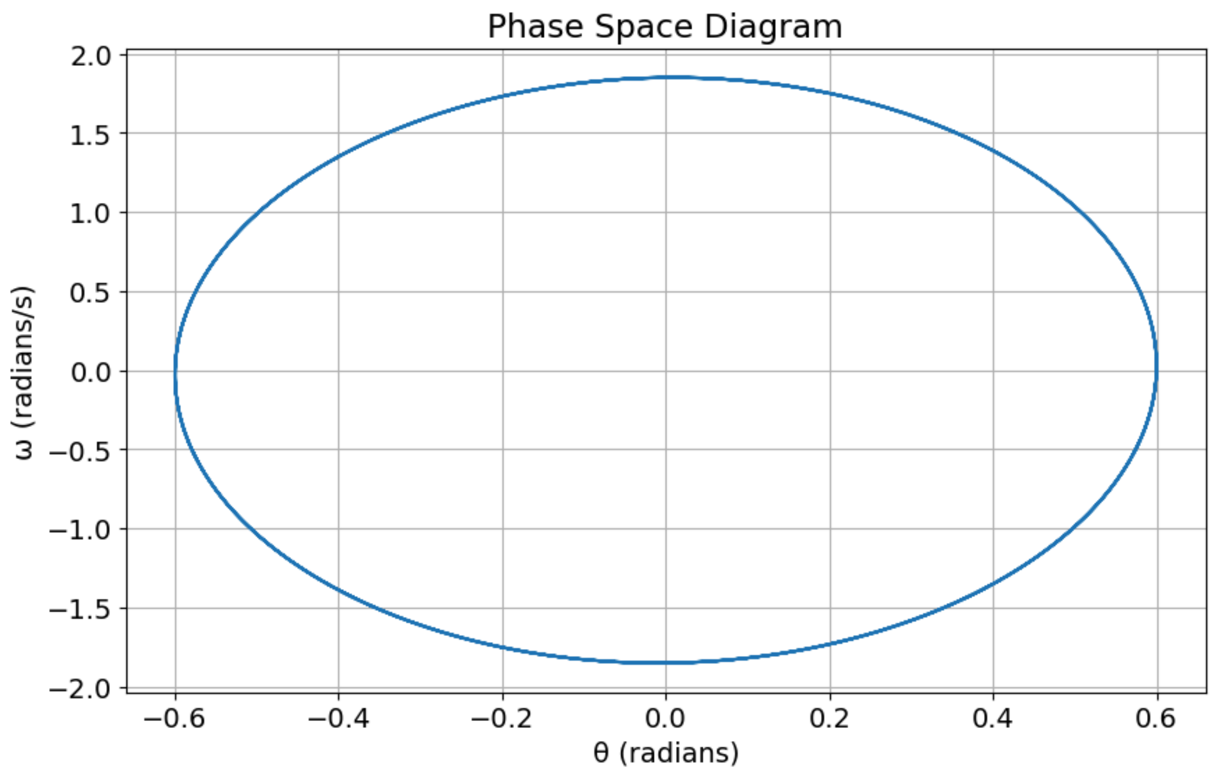
its phase trajectory will be an ellipse with axes of lengths $2A$ and $2A\Omega$.

? Answer (start)

In [20]:
```
###ANSWER HERE
def Force(t, pos, vel, params):
    g = params.get('g', 9.8)
    l1 = params['l1']
    F = -g/l1 * np.sin(pos[0])

    return np.array([F])

def run_simulation(params):
    T = params['T']
    dt = params['dt']
    init_pos = params['initPos']
    init_vel = params['initVel']

    steps = int(T / dt)
    t = np.linspace(0, T, steps)

    pos = np.zeros((steps, len(init_pos)))
    vel = np.zeros((steps, len(init_vel)))

    pos[0] = init_pos
    vel[0] = init_vel

    for i in range(1, steps):
        force = Force(t[i-1], pos[i-1], vel[i-1], params)
        vel[i] = vel[i-1] + force * dt
        pos[i] = pos[i-1] + vel[i] * dt

    return t, pos, vel

params = {
    'g': 9.8,
    'l1': 1.0,
    'dt': 0.01,
    'T': 10.0,
    'initPos': np.array([0.6]),
    'initVel': np.array([0.0])
}

t, pos, vel = run_simulation(params)

plt.figure(figsize=(10, 6))
plt.plot(t, pos[:, 0])
plt.xlabel('Time (s)')
plt.ylabel('θ (radians)')
plt.title('Pendulum Angle vs Time')
plt.grid(True)
plt.show()

x = params['l1'] * np.sin(pos[:, 0])
y = -params['l1'] * np.cos(pos[:, 0])

plt.figure(figsize=(8, 8))
plt.plot(x, y)
plt.plot(0, 0, 'ko')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Pendulum Trajectory')
plt.axis('equal')
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(pos[:, 0], vel[:, 0])
plt.xlabel('θ (radians)')
plt.ylabel('ω (radians/s)')
plt.title('Phase Space Diagram')
plt.grid(True)
plt.show()

positions = pos
```



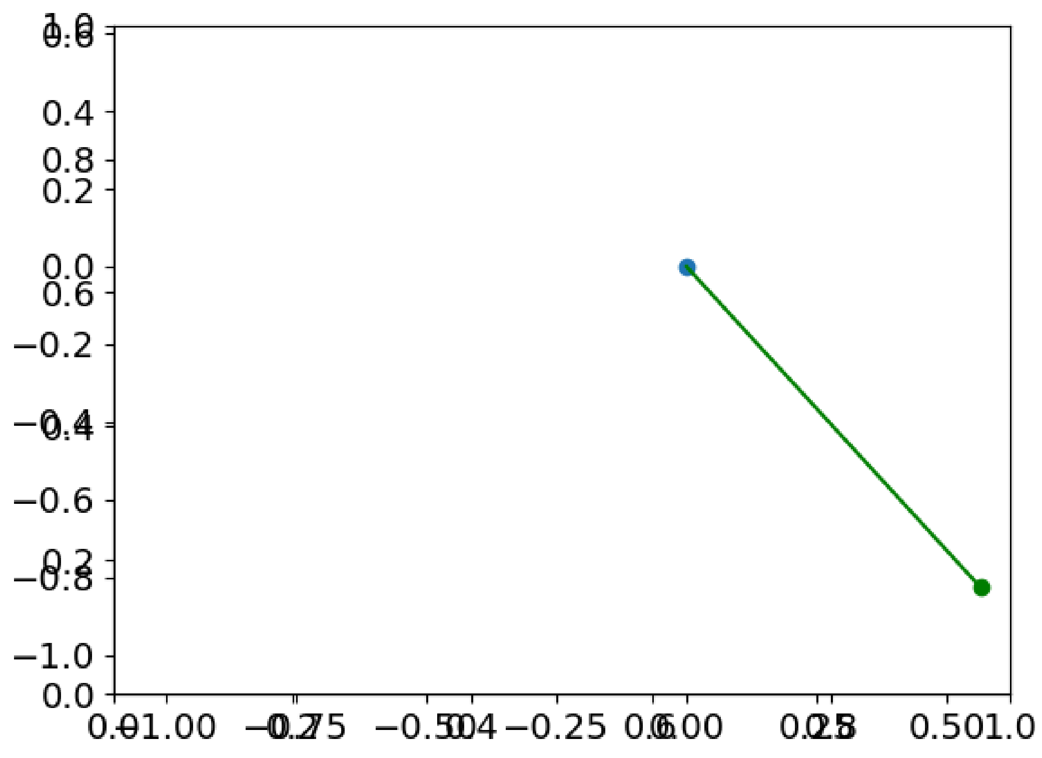

Pendulum Trajectory
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
y
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
x

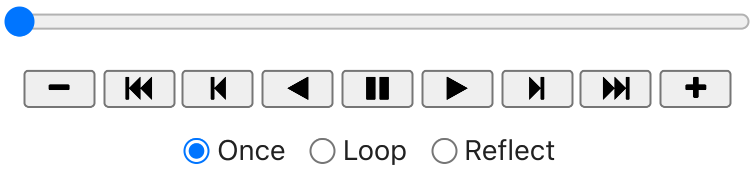


In [21]:
```
#RUN ME TO ANIMATE YOUR CODE
ani=animateMe_singlePendula([positions])
HTML(ani.to_jshtml())
```

```
WARNING:matplotlib.animation:Animation size has reached 20984548 bytes, exce
eding the limit of 20971520.0. If you're sure you want a larger animation em
bedded, set the animation.embed_limit rc parameter to a larger value (in M
B). This and further frames will be dropped.
```

Out[21]:





Answer (end)

Q: Does the phase space diagram match your expectations? Explain

A:Yes because it shows an eliptical trajectory. Which is expected for a pendulum

b. Pendula and the different starting positions

Recall in your intro mechanics class (Physics 211/etc) that you know the analytic solution for a pendulum is

$$\theta(t) = \theta_0 \cos(\Omega t)$$

However, that solution only works for *small angles*.

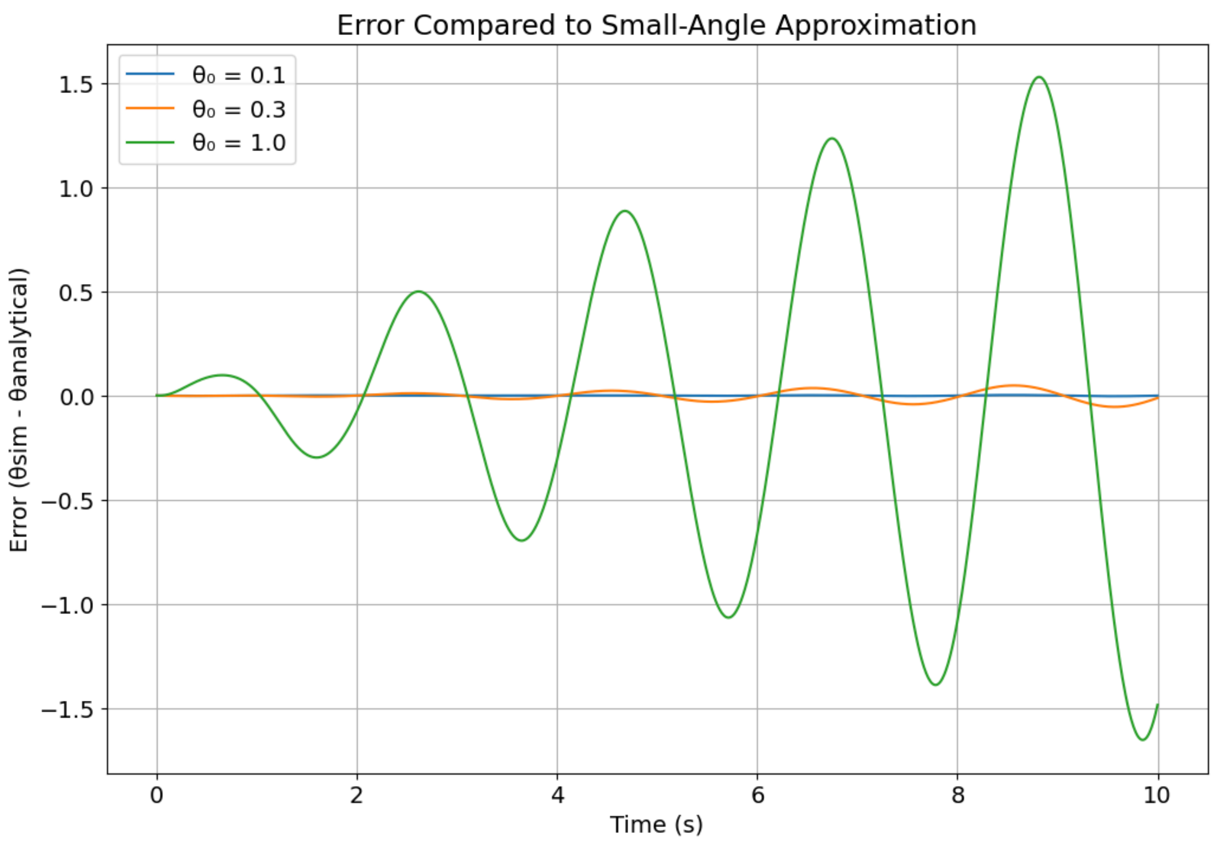
🦉 Let's quickly check this, by looking at the "**error**" (difference) against the 'analytic solution' for $\theta_0 \in \{0.1, 0.3, 1\}$.

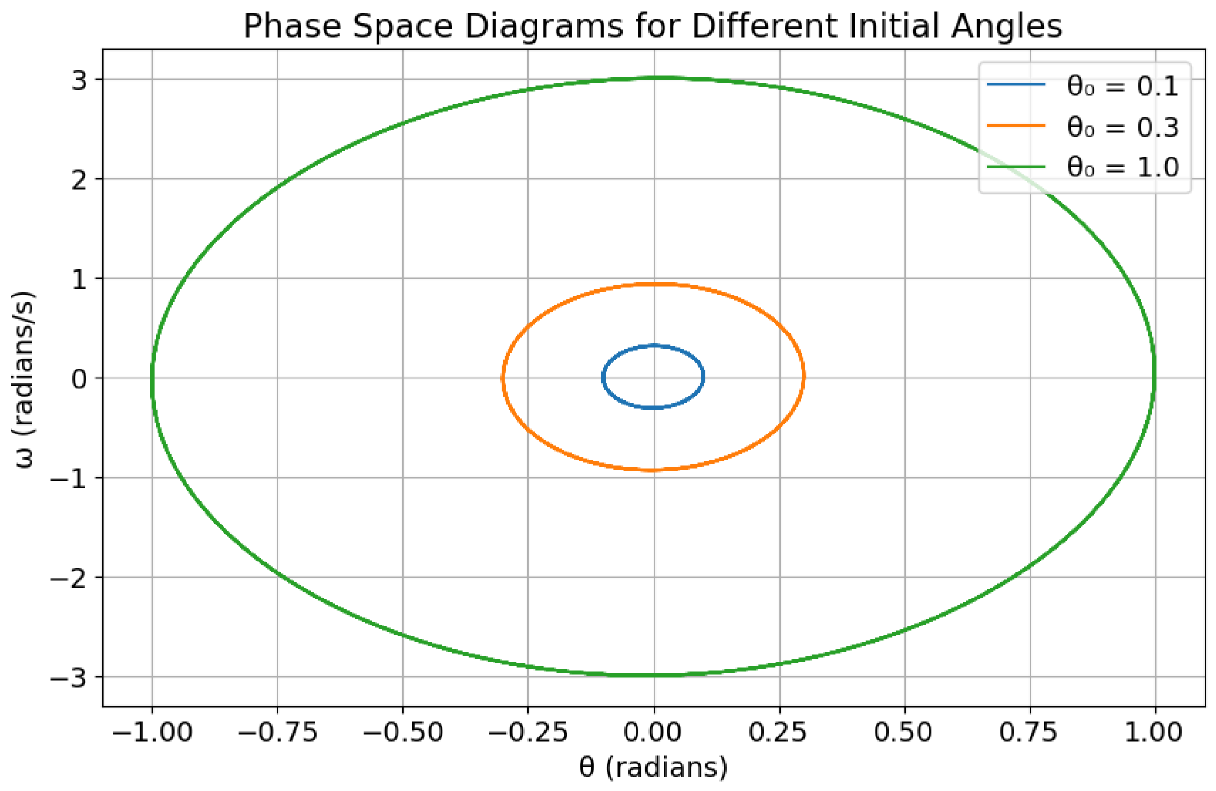
Then plot their phase space diagrams on top of one another. What do you see?

? Answer (start)

In [22]:
```python
###ANSWER HERE
initial_angles = [0.1, 0.3, 1.0]
results = []
position_arrays = []

for theta0 in initial_angles:
    params = {
        'g': 9.8,
        'l1': 1.0,
        'dt': 0.01,
        'T': 10.0,
        'initPos': np.array([theta0]),
        'initVel': np.array([0.0])
    }

    t, pos, vel = run_simulation(params)
    results.append((t, pos, vel))

    position_array = np.column_stack((pos, vel))
    position_arrays.append(position_array)

omega = np.sqrt(params['g'] / params['l1'])
analytical_solutions = []

for theta0 in initial_angles:
    analytical = theta0 * np.cos(omega * results[0][0])
    analytical_solutions.append(analytical)

plt.figure(figsize=(12, 8))
for i, theta0 in enumerate(initial_angles):
    t, pos, vel = results[i]
    error = pos[:, 0] - analytical_solutions[i]
    plt.plot(t, error, label=f'θ₀ = {theta0}')

plt.xlabel('Time (s)')
plt.ylabel('Error (θsim - θanalytical)')
plt.title('Error Compared to Small-Angle Approximation')
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 6))
for i, theta0 in enumerate(initial_angles):
    t, pos, vel = results[i]
    plt.plot(pos[:, 0], vel[:, 0], label=f'θ₀ = {theta0}')

plt.xlabel('θ (radians)')
plt.ylabel('ω (radians/s)')
plt.title('Phase Space Diagrams for Different Initial Angles')
plt.legend()
plt.grid(True)
plt.show()

positions = position_arrays[0]
```

```
positions2 = position_arrays[1]
positions3 = position_arrays[2]
```

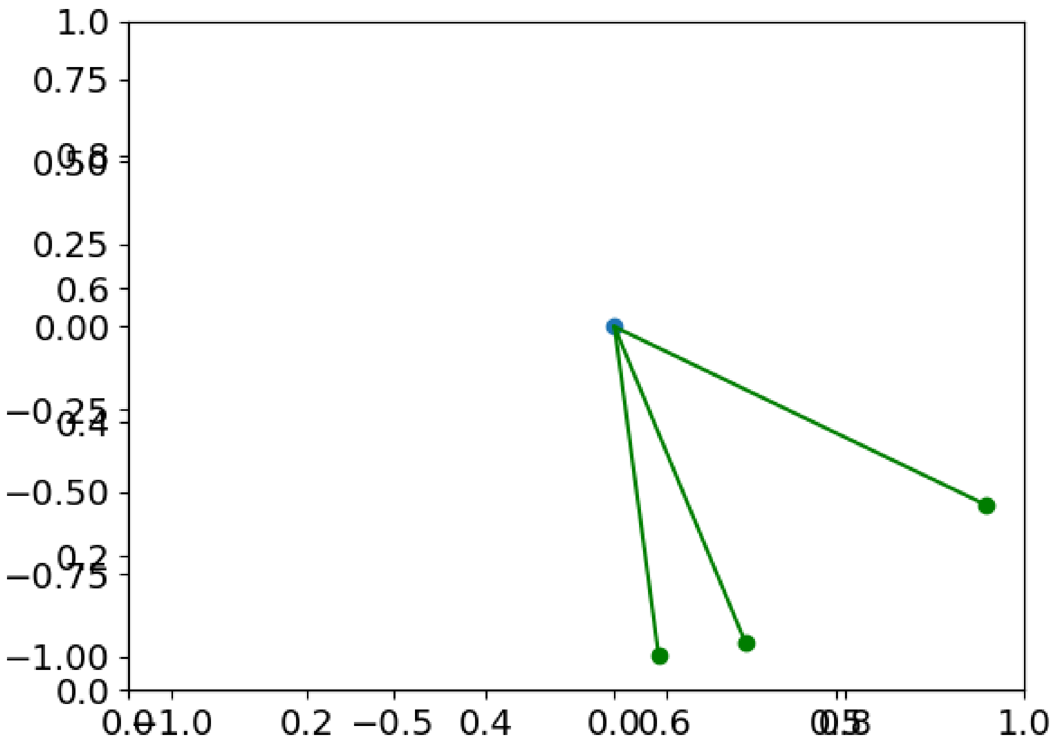


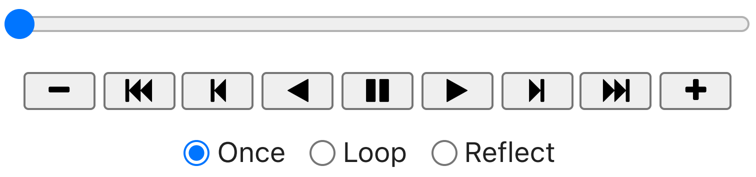


In [23]:
```
ani=animateMe_singlePendula([positions,positions2,positions3])
HTML(ani.to_jshtml())
```

```
WARNING:matplotlib.animation:Animation size has reached 21001969 bytes, exce
eding the limit of 20971520.0. If you're sure you want a larger animation em
bedded, set the animation.embed_limit rc parameter to a larger value (in M
B). This and further frames will be dropped.
```

Out[23]:





✓ Answer (end)

Q: Describe the "error" plot; what's going on?

A:This plot shows the difference between numerical sim and small-angle approximation. The error is minimal for small theta and increases as theta increases (especially for theta=1). This is mainly because the pendulum period increases with amplitude, while the small angle approx assumes const period.

Q: What does the phase space diagram look like now? Do any of these initial conditions change the "simple" behaivor of the pendulum?

A:Still eliptical but now we have three of them. They are closed curves so it is periodic motion. As theta increases, the curves seem to become larger and maybe more stretched. This happens because at larger theta the pendulums motion becomes more non-linear and small angle approx breaks down

c. Damped and Driven and phase plots

The behavior of a “simple” pendulum becomes less simple when you add damping and drive the system.

Suppose we make a pendulum from an object of mass m at the end of a rigid, massless rod of length L. The rod is suspended from a bearing which exerts an angular velocity-dependent drag force on the pendulum. Because we are using a rod, not a string, the pendulum can swing in a full circle if it is moving sufficiently quickly. A motor attached to the pendulum provides a sinusoidally varying torque. Gravity acts on the pendulum in the usual way.

The equation of motion for the pendulum is then

$$\tau = I\alpha = mL^2\frac{d^2\theta}{dt^2} \tag{1}$$
$$= -mgL\sin\theta - \beta\frac{d\theta}{dt} + \gamma\sin\Omega_{\text{ext}}t \tag{2}$$

The first term to the right of the equal sign is the torque due to gravity. The second term is the velocity-dependent damping, while the third term is the external driving torque. The symbols β and γ represent constants.

Do some algebra:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta - \frac{\beta}{mL^2}\frac{d\theta}{dt} + \frac{\gamma}{mL^2}\sin\Omega_{\text{ext}}t$$

or

$$\frac{d^2\theta}{dt^2} + A\frac{d\theta}{dt} + B\sin\theta = C\sin\Omega_{\text{ext}}t$$

where A and B are positive constants. We can rewrite this as a pair of first order equations:

$$\frac{d\theta}{dt} = \omega \qquad\qquad \frac{d\omega}{dt} = -A\omega - B\sin\theta + C\sin\Omega_{\text{ext}}t$$

🦉 Please write a program to calculate (and graph) θ vs. t and ω vs. t for the pendulum. You may find it convenient to write your expression for the force in this way:

```python
def Force(t,pos,vel,params):
    A = params['A']
    B = params['B']
    C = params['C']
    OMEGA = params['OMEGA']
    F = -A*vel - B*np.sin(pos) + C*np.sin(OMEGA * t)
    return F
```

Make sure when you make your plots that all values of θ are $-\pi \leq \theta \leq \pi$. When you produce your graphs make sure that you wrap things so that those are the only values that you see on your graph.

Run your program with the following parameters and be sure to include all the plots in your Jupyter notebook.

Parameter	Version 1	Version 2	Version 3
A (damping)	0	0.1	0.1
B (restoring force)	0.1	1	1
C (external driving force)	0	0.1	2
m	1	1	1
Ω_{ext} (driving frequency)	N/A	2	1.2
θ_0	0	0	0
ω_0	0.1	0.1	0.1
t_{max}	120	120	500
dt	.01	.01	.01

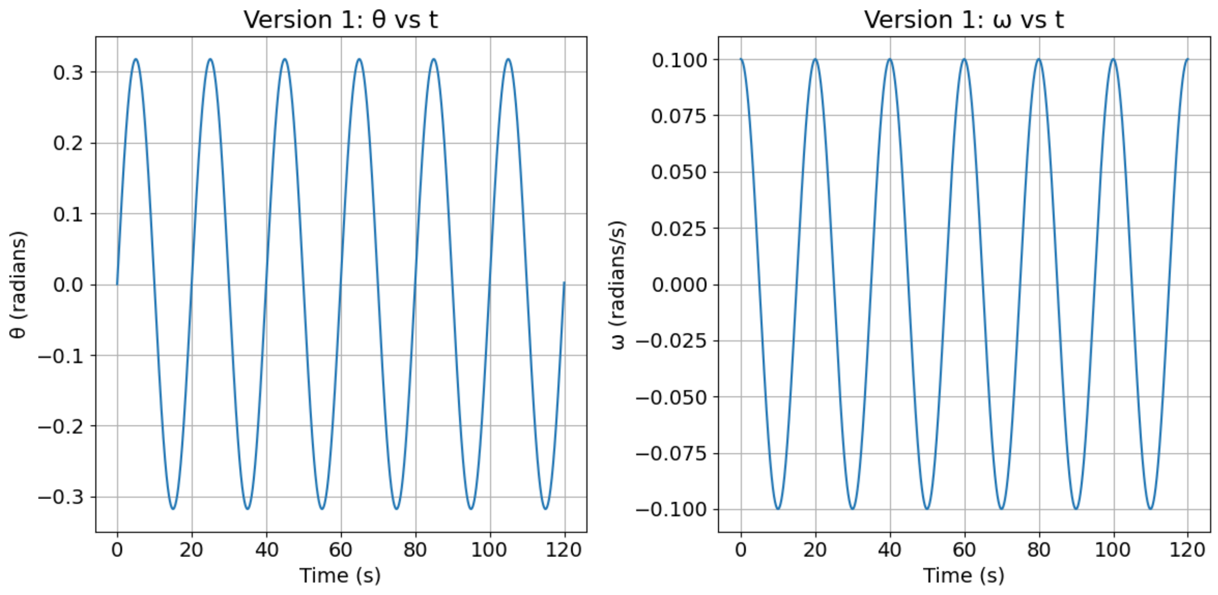
Version 1 is an undamped, undriven pendulum. The amplitude of motion is small enough so that it behaves very much like a harmonic oscillator with oscillation frequency $\Omega = \sqrt{0.1}$. Version 2 is lightly damped, and driven at twice its natural frequency. As the initial motion (which has $\Omega = 1$) dies away, the driving force will cause it to oscillate at the same frequency as the driving frequency.

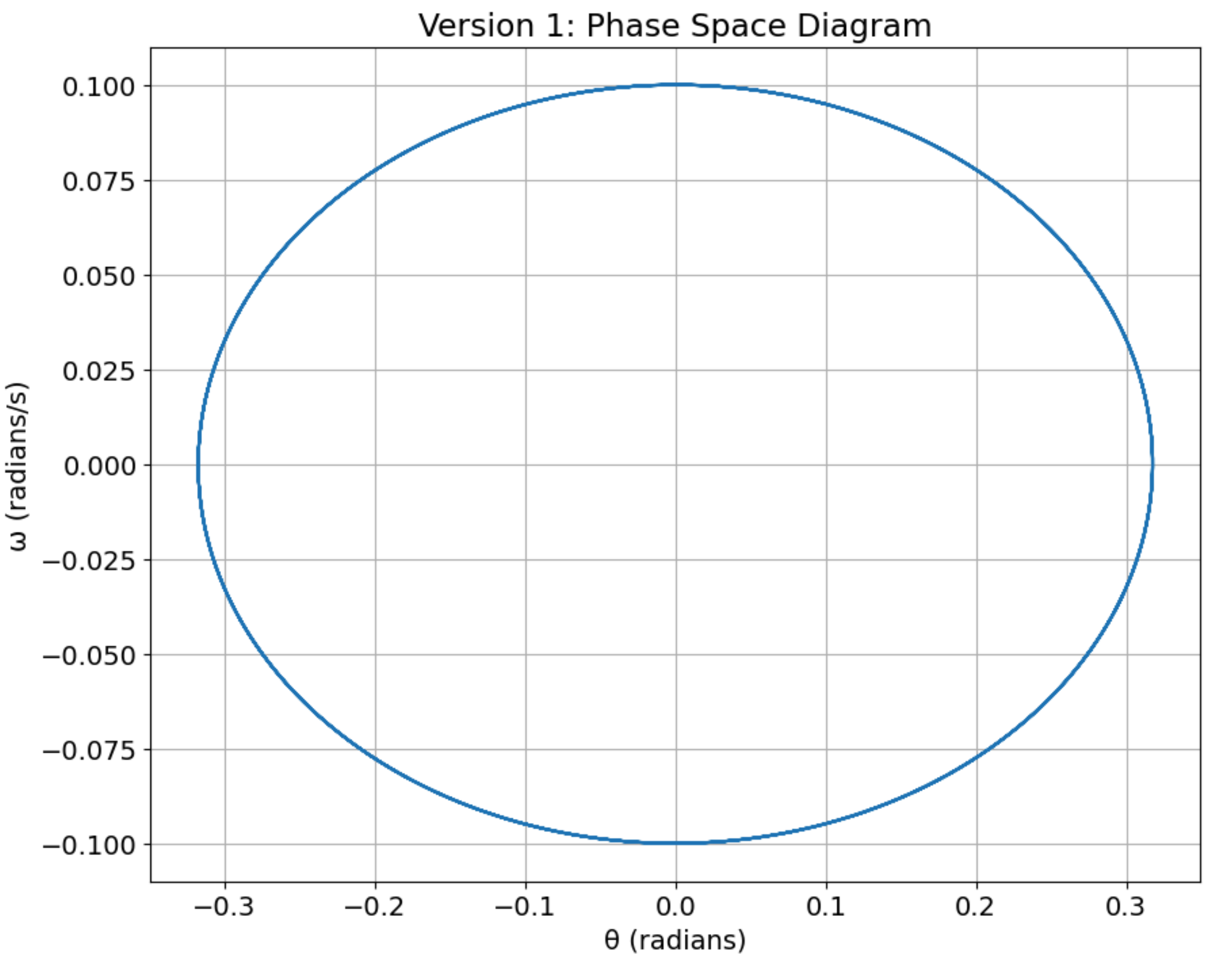
It takes longer for the pendulum to settle down than it did in case 2.

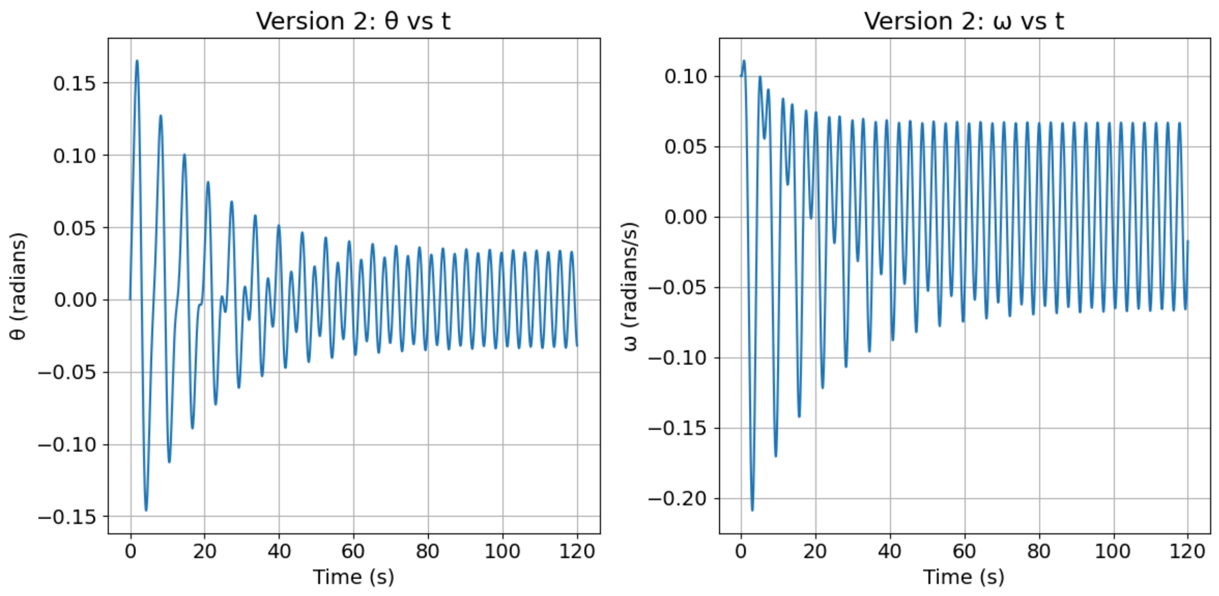
If we had tuned the parameters just right, we might have found a set for which the motion was chaotic, never settling into a periodic mode.

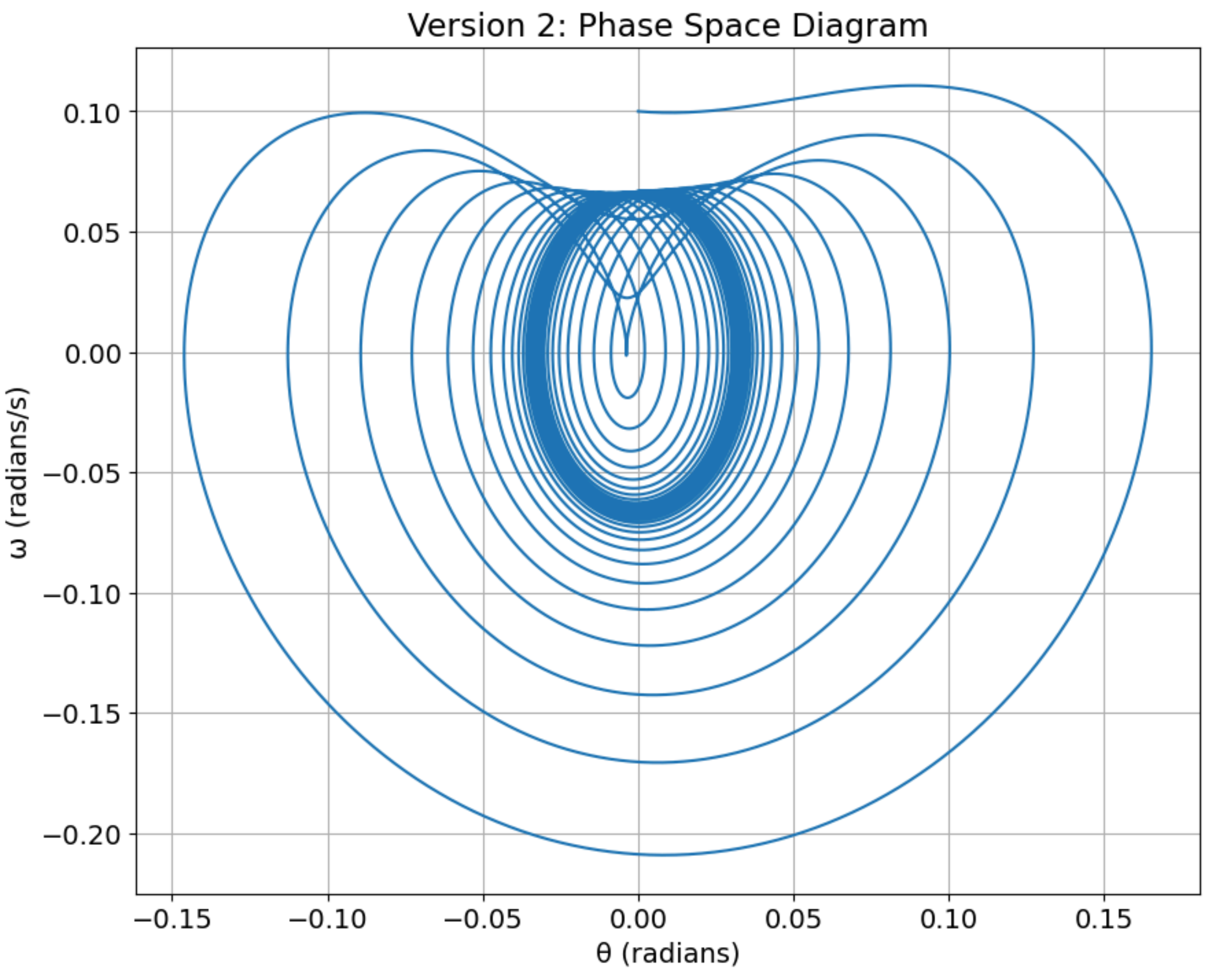
? Answer (start)

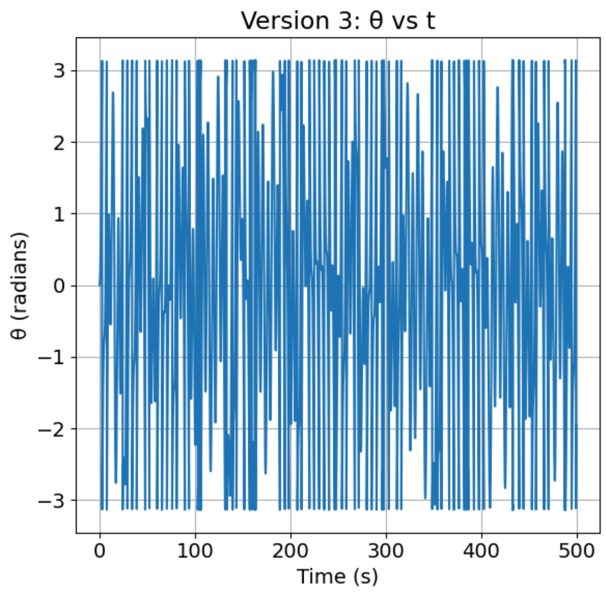
In [24]:
```python
### ANSWER HERE
def Force_damped_driven(t, pos, vel, params):
    A = params['A']
    B = params['B']
    C = params.get('C', 0)
    OMEGA = params.get('OMEGA', 0)

    F = -A * vel[0] - B * np.sin(pos[0]) + C * np.sin(OMEGA * t)

    return np.array([F])

def run_damped_driven_sim(params):
    T = params.get('t_max', params.get('T', 10))
    dt = params['dt']
    init_pos = params.get('theta0', params.get('initPos', np.array([0])))
    init_vel = params.get('omega0', params.get('initVel', np.array([0])))

    steps = int(T / dt)
    t = np.linspace(0, T, steps)

    pos = np.zeros((steps, len(init_pos) if hasattr(init_pos, '__len__') els
    vel = np.zeros((steps, len(init_vel) if hasattr(init_vel, '__len__') els

    pos[0] = init_pos
    vel[0] = init_vel

    for i in range(1, steps):
        force = Force_damped_driven(t[i-1], pos[i-1], vel[i-1], params)
        vel[i] = vel[i-1] + force * dt
        pos[i] = pos[i-1] + vel[i] * dt


        pos[i, 0] = ((pos[i, 0] + np.pi) % (2 * np.pi)) - np.pi

    return t, pos, vel


param_sets = [

    {
        'A': 0,
        'B': 0.1,
        'C': 0,
        'm': 1,
        'OMEGA': 0,
        'theta0': np.array([0]),
        'omega0': np.array([0.1]),
        't_max': 120,
        'dt': 0.01
    },

    {
        'A': 0.1,
        'B': 1,
        'C': 0.1,
        'm': 1,
        'OMEGA': 2,
        'theta0': np.array([0]),
        'omega0': np.array([0.1]),
        't_max': 120,
        'dt': 0.01
    },

    {
        'A': 0.1,
        'B': 1,
        'C': 2,
        'm': 1,
        'OMEGA': 1.2,
        'theta0': np.array([0]),
        'omega0': np.array([0.1]),
        't_max': 500,
        'dt': 0.01
    }
]

for i, params in enumerate(param_sets):
    version = i + 1

    t, pos, vel = run_damped_driven_sim(params)

    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(t, pos[:, 0])
    plt.xlabel('Time (s)')
    plt.ylabel('θ (radians)')
    plt.title(f'Version {version}: θ vs t')
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(t, vel[:, 0])
    plt.xlabel('Time (s)')
    plt.ylabel('ω (radians/s)')
    plt.title(f'Version {version}: ω vs t')
    plt.grid(True)

    plt.tight_layout()
    plt.show()

    plt.figure(figsize=(10, 8))
    plt.plot(pos[:, 0], vel[:, 0])
    plt.xlabel('θ (radians)')
    plt.ylabel('ω (radians/s)')
    plt.title(f'Version {version}: Phase Space Diagram')
    plt.grid(True)
    plt.show()
```

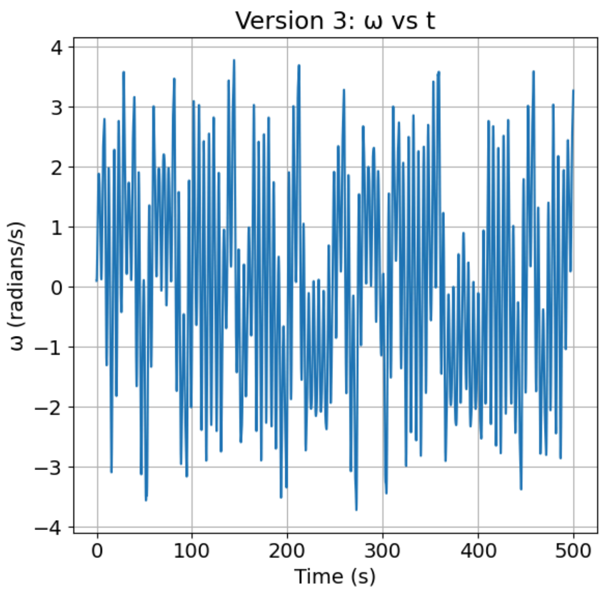


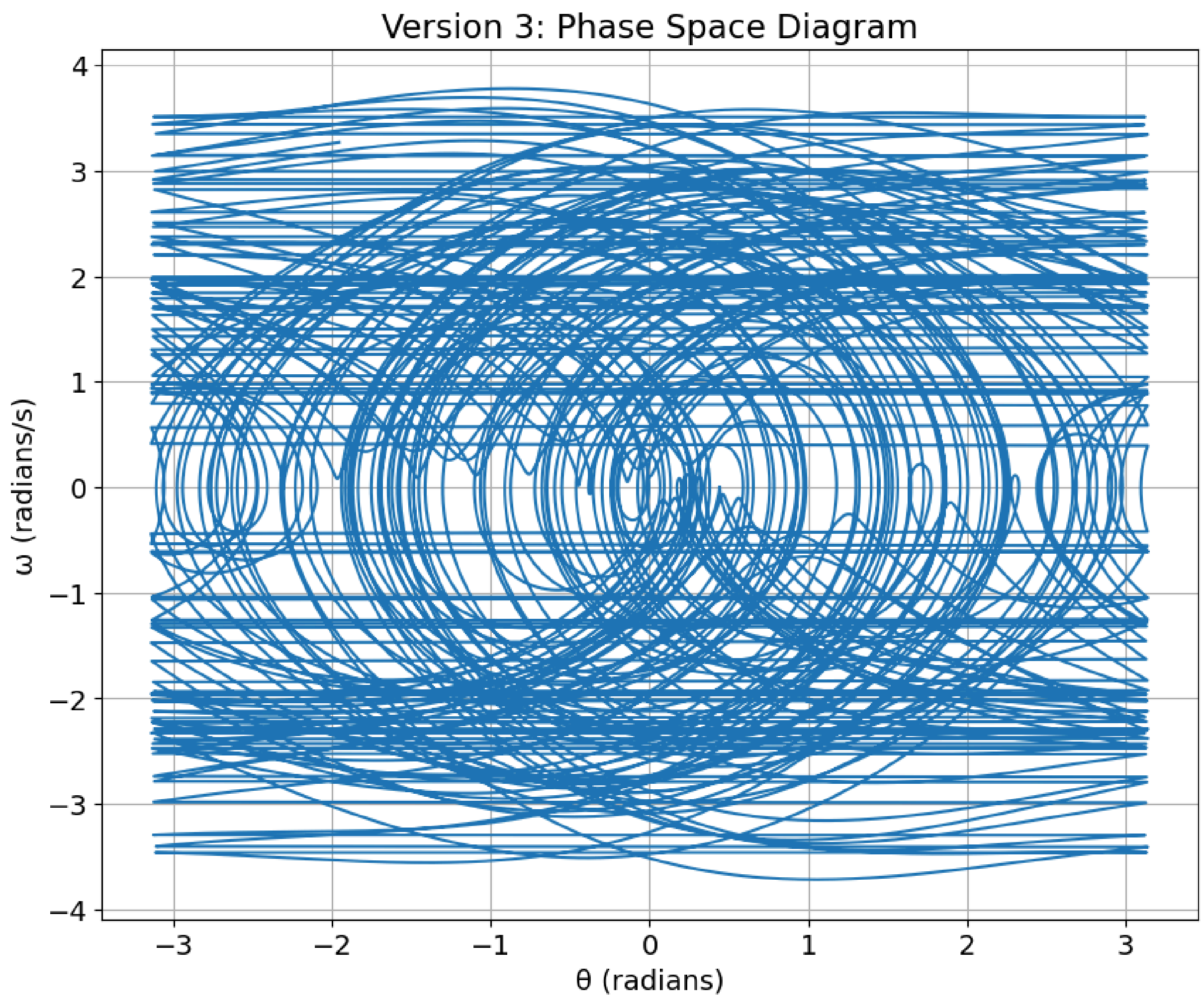

Version 1: Phase Space Diagram
ω (radians/s)
θ (radians)
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3


Version 2: θ vs t
θ (radians)
Time (s)
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
20
40
60
80
100
120
Version 2: ω vs t
ω (radians/s)
Time (s)
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20


Version 2: Phase Space Diagram
ω (radians/s)
θ (radians)
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15


Version 3: θ vs t
θ (radians)
Time (s)
3
2
1
0
−1
−2
−3
0
100
200
300
400
500


Version 3: ω vs t
ω (radians/s)
Time (s)
4
3
2
1
0
−1
−2
−3
−4
0
100
200
300
400
500



✓ Answer (end)

d. Simple pendulum phase space trajectory

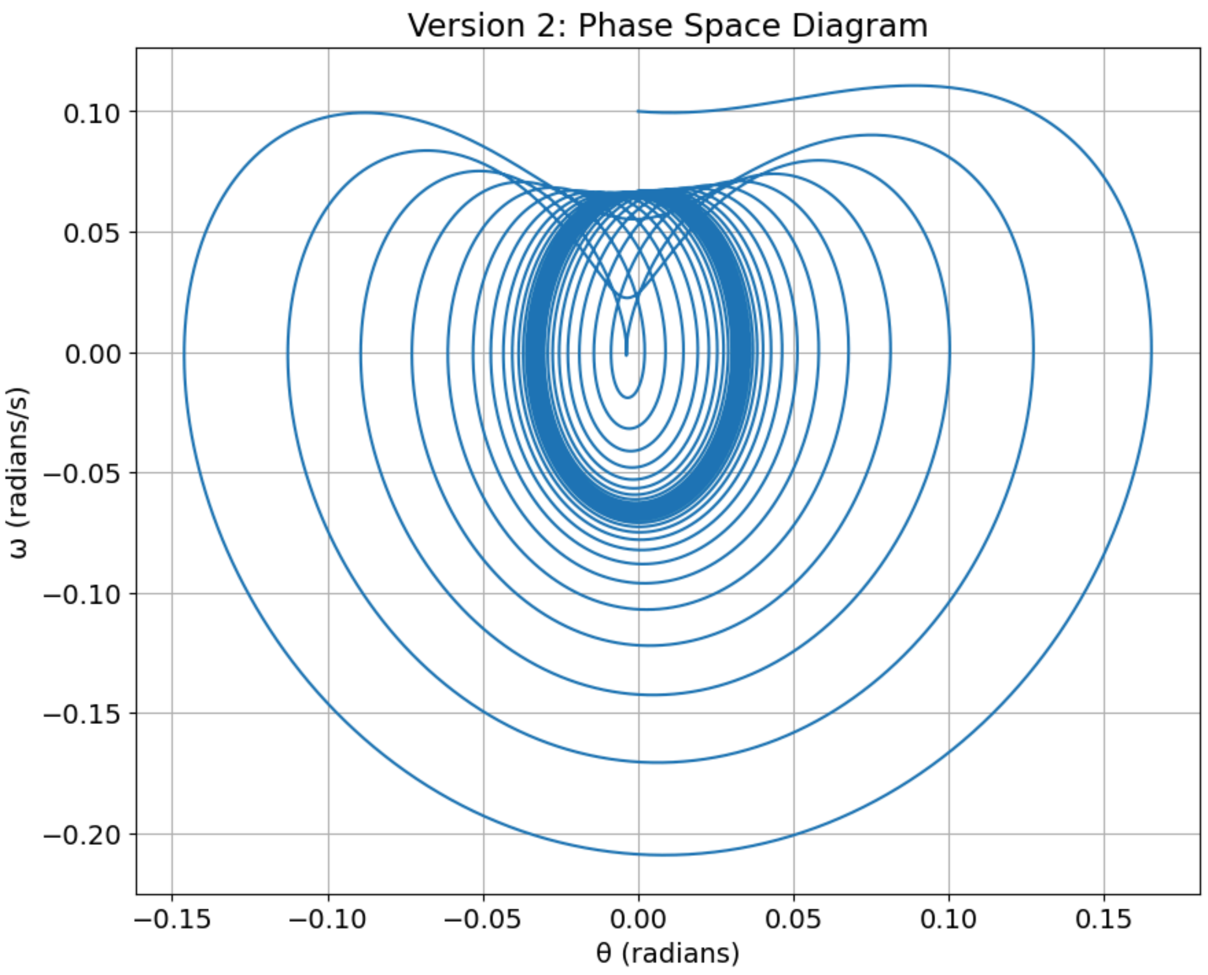
The phase trajectories of driven nonlinear oscillators can be much more complicated than this. 🦉 Please generate phase space diagrams for the simple pendulum using the parameters for version 2 and version 4 (shown below again). Note that $t_{\max} = 500$

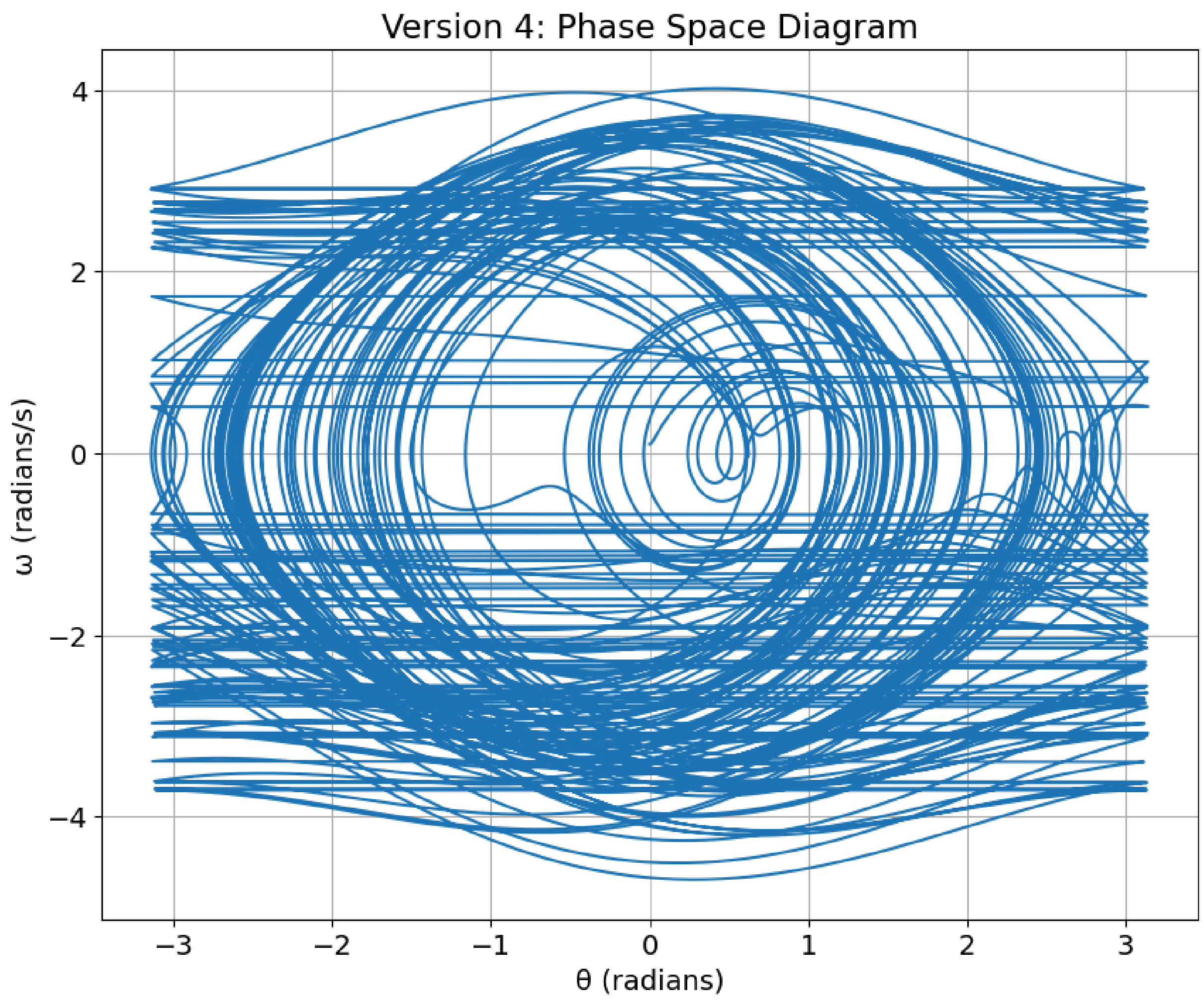
Parameter	Version 2	Version 4
A (damping)	0.1	0.1
B (restoring force)	1	2
C (external driving force)	0.1	2
m	1	1
Ω_{ext} (driving frequency)	2	1.2
θ_0	0	0
ω_0	0.1	0.1
t_{max}	500	500

Parameter	Version 2	Version 4
dt	.01	.01

? Answer (start)

In [33]:
```
###ANSWER HERE
param_set_4 = {
    'A': 0.1,
    'B': 2,
    'C': 2,
    'm': 1,
    'OMEGA': 1.2,
    'theta0': np.array([0]),
    'omega0': np.array([0.1]),
    't_max': 500,
    'dt': 0.01
}

t2, pos2, vel2 = run_damped_driven_sim(param_sets[1])
t4, pos4, vel4 = run_damped_driven_sim(param_set_4)

plt.figure(figsize=(10, 8))
plt.plot(pos2[:, 0], vel2[:, 0])
plt.xlabel('θ (radians)')
plt.ylabel('ω (radians/s)')
plt.title('Version 2: Phase Space Diagram')
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 8))
plt.plot(pos4[:, 0], vel4[:, 0])
plt.xlabel('θ (radians)')
plt.ylabel('ω (radians/s)')
plt.title('Version 4: Phase Space Diagram')
plt.grid(True)
plt.show()

positions2 = np.column_stack((pos2, vel2))
positions4 = np.column_stack((pos4, vel4))
```


Version 2: Phase Space Diagram
ω (radians/s)
θ (radians)
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15

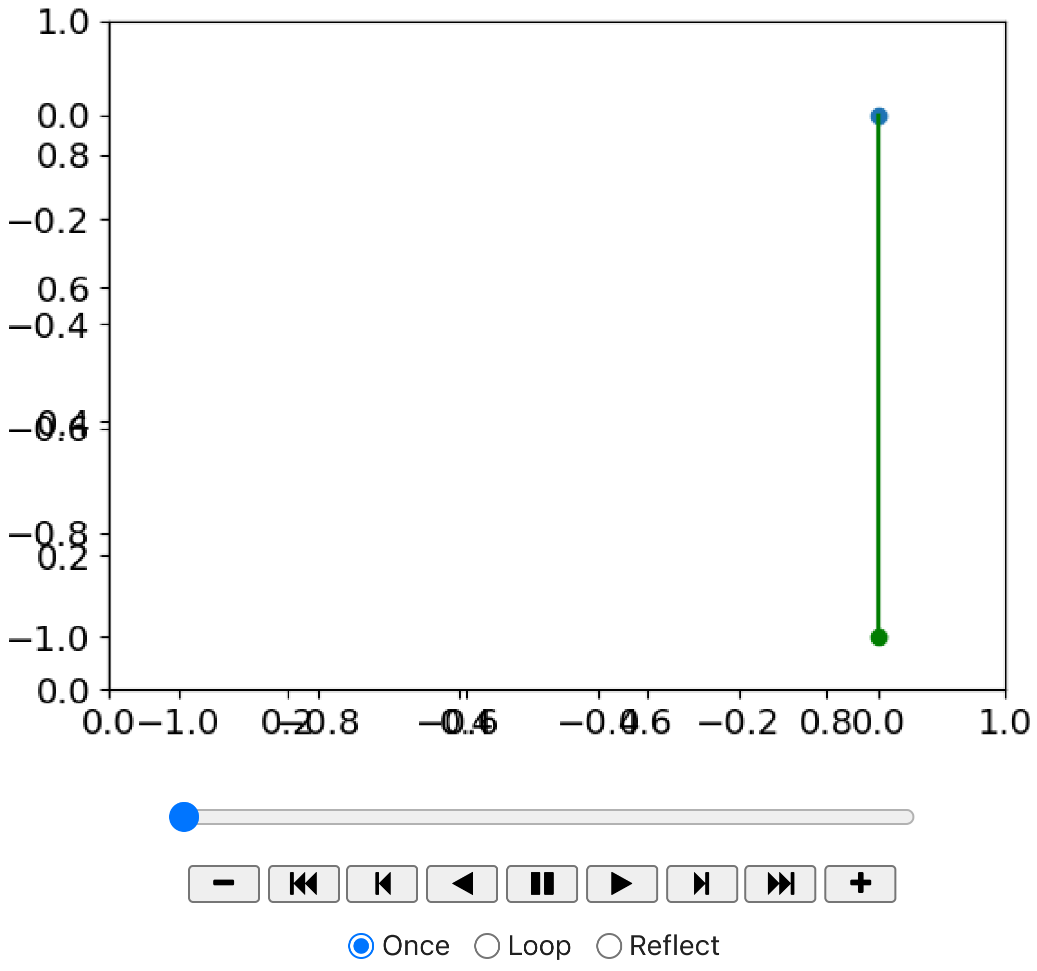


If you want, go ahead and animate these pendula

```
In [34]: l=len(positions2)
         ani=animateMe_singlePendula([positions2[::l//1000]])
         HTML(ani.to_jshtml())
```

```
WARNING:matplotlib.animation:Animation size has reached 20978254 bytes, exce
eding the limit of 20971520.0. If you're sure you want a larger animation em
bedded, set the animation.embed_limit rc parameter to a larger value (in M
B). This and further frames will be dropped.
```

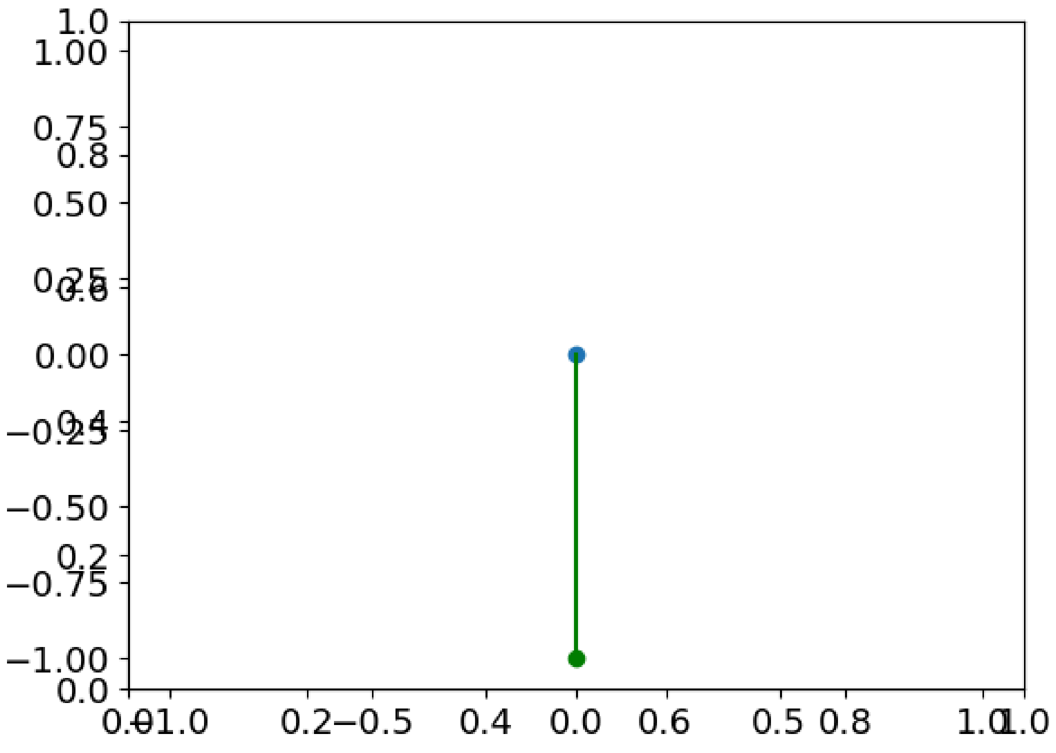
Out[34]:

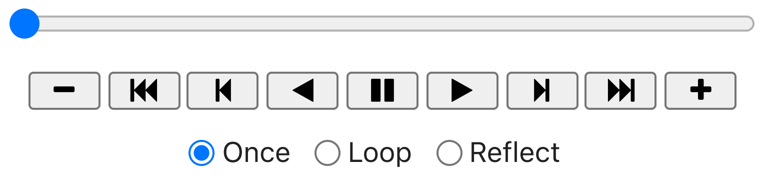


In [35]:
```
l=len(positions4)
ani=animateMe_singlePendula([positions4[::l//5000]])
HTML(ani.to_jshtml())
```

```
WARNING:matplotlib.animation:Animation size has reached 20980282 bytes, exce
eding the limit of 20971520.0. If you're sure you want a larger animation em
bedded, set the animation.embed_limit rc parameter to a larger value (in M
B). This and further frames will be dropped.
```

Out[35]:





Once [✓] Loop Reflect

✓ Answer (end)

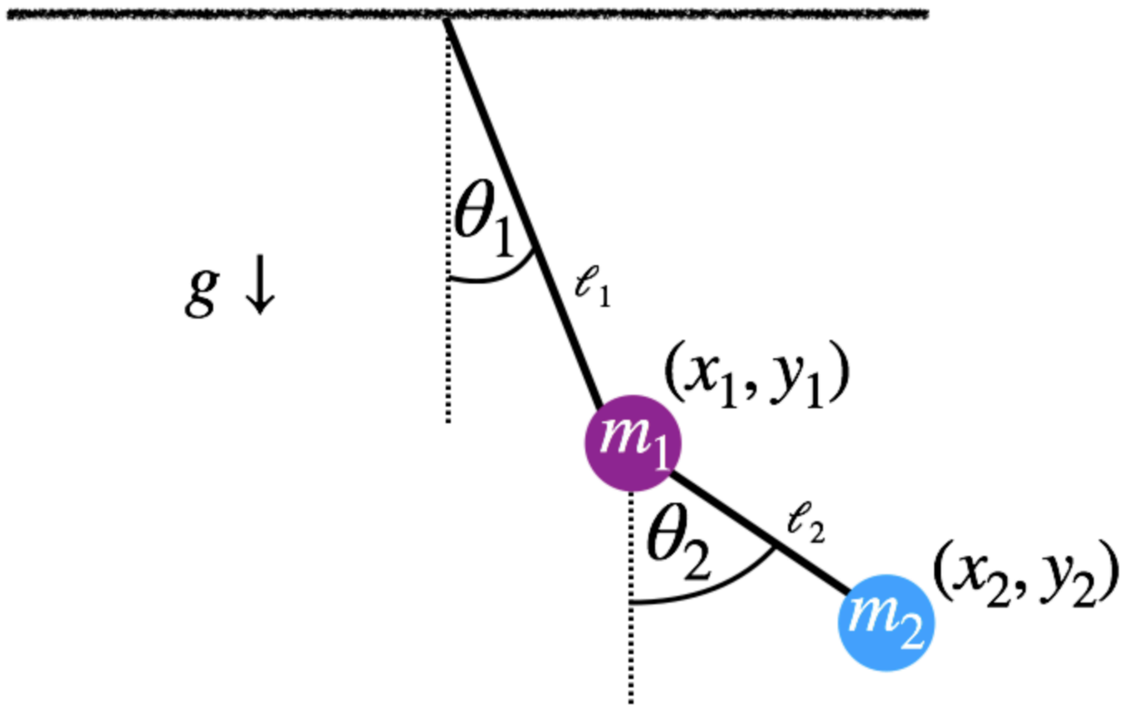
Exercise 2. Double Pendulum

- **List of collaborators:**
- **References you used in developing your code:**

Single (rigid) Pendula are somewhat boring. We can compute most things about them analytically. Double pendula on the other hand are much more exotic. In this exercise you are going to simulate a double pendula.

In [36]:
```
def animateMe_doublePendula(positions):
    positionArray = []
    for pos in positions:
        theta1 = pos[:, 0]
        theta2 = pos[:, 1]
        x1 = params['l1'] * np.sin(theta1)
        y1 = -params['l1'] * np.cos(theta1)
        x2 = x1 + params['l2'] * np.sin(theta2)
        y2 = y1 - params['l2'] * np.cos(theta2)
        l = len(x1)
        pos_xy = np.zeros((l, 4))
        pos_xy[:, 0] = x1
        pos_xy[:, 1] = y1
        pos_xy[:, 2] = x2
        pos_xy[:, 3] = y2
        positionArray.append(pos_xy)

    fig, ax = plt.subplots()
    x_min = np.min([np.min(np.concatenate((p[:,0], p[:,2]))) for p in positi
    x_max = np.max([np.max(np.concatenate((p[:,0], p[:,2]))) for p in positi
    y_min = np.min([np.min(np.concatenate((p[:,1], p[:,3]))) for p in positi
    y_max = np.max([np.max(np.concatenate((p[:,1], p[:,3]))) for p in positi
    y_max = np.max([y_max, 0.1])
    y_max = np.max([y_max, x_max])
    x_max = y_max
    y_min = np.min([y_min, x_min])
    x_min = y_min

    ax = plt.axes(xlim=(x_min, x_max), ylim=(y_min, y_max))
    ax.plot([0], [0], 'o', color='k')

    lines = []
    colorWheel = ['g', 'b', 'r']
    for idx, pos in enumerate(positionArray):
        lines.append(ax.plot([], [], 'o', color=colorWheel[0])[0])
        lines.append(ax.plot([], [], '-', color=colorWheel[0])[0])
        lines.append(ax.plot([], [], 'o', color=colorWheel[0])[0])
        lines.append(ax.plot([], [], '-', color=colorWheel[0])[0])

    def update(i, positionArray, lines):
        for idx, pos in enumerate(positionArray):
            lines[4*idx+0].set_data([pos[i, 0]], [pos[i, 1]])
            lines[4*idx+1].set_data([0, pos[i, 0]], [0, pos[i, 1]])
            lines[4*idx+2].set_data([pos[i, 2]], [pos[i, 3]])
            lines[4*idx+3].set_data([pos[i, 0], pos[i, 2]], [pos[i, 1], pos[
        return lines

    ani = animation.FuncAnimation(fig, update, frames=len(positionArray[0]),
                                  fargs=[positionArray, lines],
                                  interval=20, blit=True, repeat=False)
    plt.close()
    return ani
```

a. Simulating double pendula



Modify your code to simulate a double pendula. Now instead of representing the location of your pendula with one θ you need to use two θ. (Be careful you are dealing with the mass correctly). Again, this can be mainly accomplished by playing with your initial conditions.

Now, you need to get the force to work. The force for a double pendulum is somewhat complicated. To compute the force you either need to:

- derive the force carefully
  - This is a bit annoying because part of the force involves the tension in the rod (i.e. the first mass has a force of $(-T_1 \sin(\theta_1) + T_2 \sin(\theta_2), T_1 \cos\theta_1 - T_2 \cos\theta_2 - m_1 g))$ Because we don't know what T is, we need to do some algebraic manipulation to be able to write $\frac{d^2\theta}{dt^2}$ as a function of $\omega = \frac{d\theta}{dt}$ and θ (which is what we need)
- use the Euler-Lagrange equations. This ia a better approach for systems with constraints and is something you will learn in classical mechanics.

At the moment we will just give you the effective force[1]:

```
f1=-l2/l1 *(m2/(m1+m2))*omega2**2*np.sin(t1-t2)-g/l1*np.sin(t1)
f2=l1/l2 * omega1**2 * np.sin(t1-t2)-g/l2*np.sin(t2)
alpha1=l2/l1*(m2/(m1+m2))*np.cos(t1-t2)
alpha2=l1/l2*np.cos(t1-t2)
den=(1-alpha1*alpha2)
omega1_dot = (f1-alpha1*f2)/den
omega2_dot = (-alpha2*f1+f2)/den
```

Note that `omega1` / `omega2` here mean the velocity or $d\theta_i/dt$, $i = 1, 2$ respectively.

Put this force in your code.

🦉 Run with the following parameters:

```
params={ 'm1':1.0,
         'm2':1.0,
         'l1':1.0,
         'l2':1.0,
         'T' :15.0,
         'g' :9.8,
         'dt':0.01,
         'initPos' : np.array([1,1+0.11]),
         'initVel' : np.array([0.0,0.0])
       }
```

and make a phase space plot of the system for both the first mass and the second mass (on the same plot). Also go ahead and animate it.

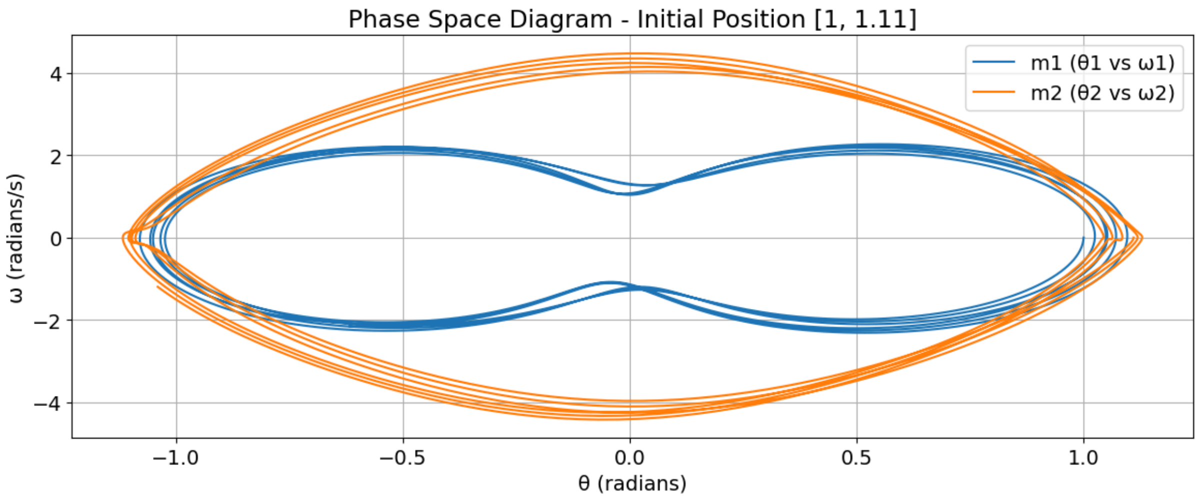
Then switch to use the initial position of `'initPos': np.array([3.14,3.14+0.11]),` . How do these compare?

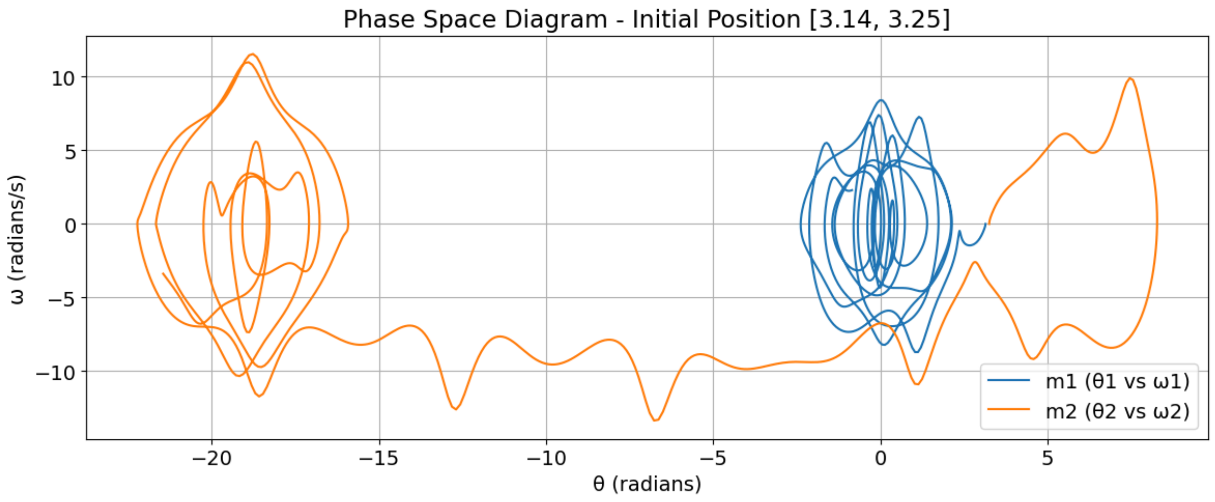
In the animation, you may not want to use all the points.

? Answer (start)

In [37]:
```
### ANSWER HERE
def double_pendulum_force(t, pos, vel, params):
    m1 = params['m1']
    m2 = params['m2']
    l1 = params['l1']
    l2 = params['l2']
    g = params['g']

    t1 = pos[0]
    t2 = pos[1]
    omega1 = vel[0]
    omega2 = vel[1]

    f1 = -l2/l1 * (m2/(m1+m2)) * omega2**2 * np.sin(t1-t2) - g/l1 * np.sin(t
    f2 = l1/l2 * omega1**2 * np.sin(t1-t2) - g/l2 * np.sin(t2)
    alpha1 = l2/l1 * (m2/(m1+m2)) * np.cos(t1-t2)
    alpha2 = l1/l2 * np.cos(t1-t2)
    den = (1 - alpha1*alpha2)
    omega1_dot = (f1 - alpha1*f2) / den
    omega2_dot = (-alpha2*f1 + f2) / den

    return np.array([omega1_dot, omega2_dot])

def run_double_pendulum_sim(params):
    T = params['T']
    dt = params['dt']
    init_pos = params['initPos']
    init_vel = params['initVel']
    steps = int(T / dt)
    t = np.linspace(0, T, steps)

    pos = np.zeros((steps, 2))
    vel = np.zeros((steps, 2))

    pos[0] = init_pos
    vel[0] = init_vel

    for i in range(1, steps):
        force = double_pendulum_force(t[i-1], pos[i-1], vel[i-1], params)
        vel[i] = vel[i-1] + force * dt
        pos[i] = pos[i-1] + vel[i] * dt

    return t, pos, vel

params = {
    'm1': 1.0,
    'm2': 1.0,
    'l1': 1.0,
    'l2': 1.0,
    'T': 15.0,
    'g': 9.8,
    'dt': 0.01,
    'initPos': np.array([1, 1+0.11]),
    'initVel': np.array([0.0, 0.0])
}

t1, pos1, vel1 = run_double_pendulum_sim(params)

params['initPos'] = np.array([3.14, 3.14+0.11])
t2, pos2, vel2 = run_double_pendulum_sim(params)

plt.figure(figsize=(12, 10))

plt.subplot(2, 1, 1)
plt.plot(pos1[:, 0], vel1[:, 0], label='m1 (θ1 vs ω1)')
plt.plot(pos1[:, 1], vel1[:, 1], label='m2 (θ2 vs ω2)')
plt.xlabel('θ (radians)')
plt.ylabel('ω (radians/s)')
plt.title('Phase Space Diagram - Initial Position [1, 1.11]')
plt.legend()
plt.grid(True)

plt.subplot(2, 1, 2)
plt.plot(pos2[:, 0], vel2[:, 0], label='m1 (θ1 vs ω1)')
plt.plot(pos2[:, 1], vel2[:, 1], label='m2 (θ2 vs ω2)')
plt.xlabel('θ (radians)')
plt.ylabel('ω (radians/s)')
plt.title('Phase Space Diagram - Initial Position [3.14, 3.25]')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

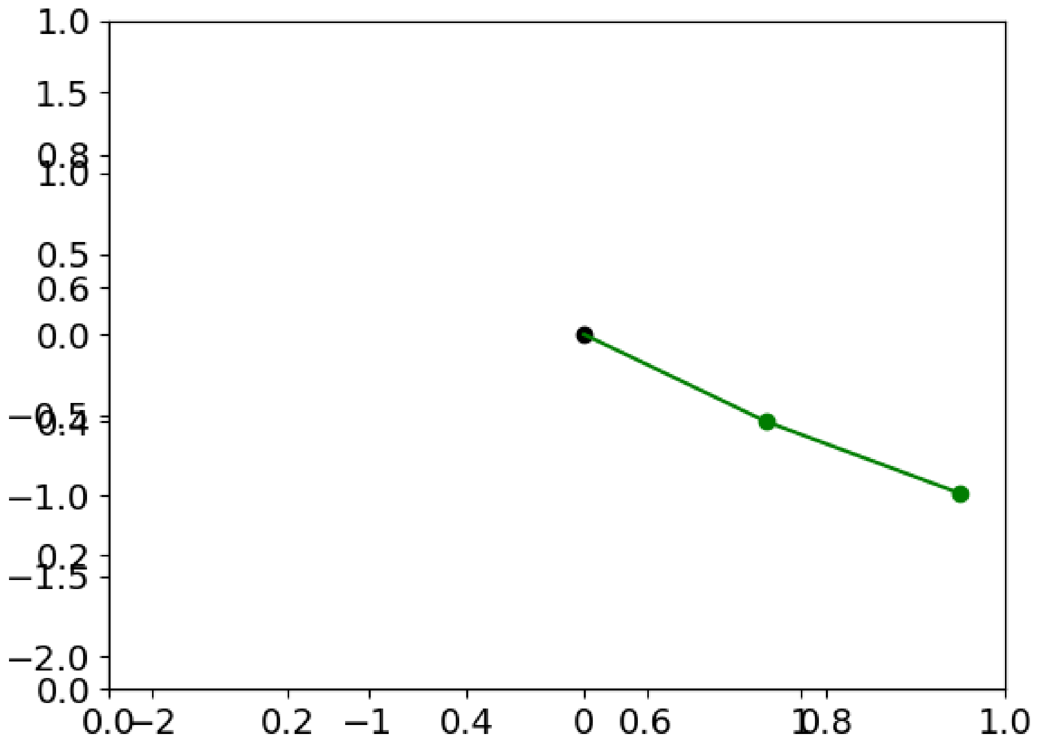
```
pos = pos1
```





```
In [38]: l=len(pos)
         ani=animateMe_doublePendula([pos[::l//100]])
         HTML(ani.to_jshtml())
```

Out[38]:



[✓ Once] Loop Reflect

✓ Answer (end)

Q: Does the inner bob (m_1) act like a simple pendulum? Use the phase space diagram in your answer

A:No it does not. Bob (m1) deviates from the perfect elipses we saw in the simple pendulums. The trajectory has pinching and irregularaties that couple with m2's motion. Also there are multiple loops and crossings, traverses a wide region, and has a complex patern.

Q: How does the behavior of the pendulum change compared to a simple pendulum as the initial angle is increase?

A: As the angle increases the behavior changes a lot. The motion transitions from periodic to chaotic, the angular displacement range expands, and both pendulum bobs become complex with irregular trajectories.

b. Chaos

Our goal now is to see that this double pendula is chaotic. Chaos means that if the system starts with very similar initial conditions, that the result will look very different in the future (long times). Go ahead and run a series of 10 initial conditions except that each one differs from the previous one by an initial angle on the second pendula of 10^{-4} , i.e. `params['initPos'][1]+i*1e-4,params['initPos'][1]+(i+1)*1e-4` Animate them all simultaneously to see what happens.

Hint: Your code should have something like this

```
params={ 'm1':1.0,
         'm2':1.0,
         'l1':1.0,
         'l2':1.0,
         'T' :15.0,
         'g' :9.8,
         'dt':0.01,
         'initPos' : np.array([3.14,3.14+0.11]),
         'initVel' : np.array([0.0,0.0])
       }
for i in range(10):
    pos = params['initPos']+np.array([0,i*1e-4])
    vel = params['initVel']
    run_simulations(pos,vel)
```

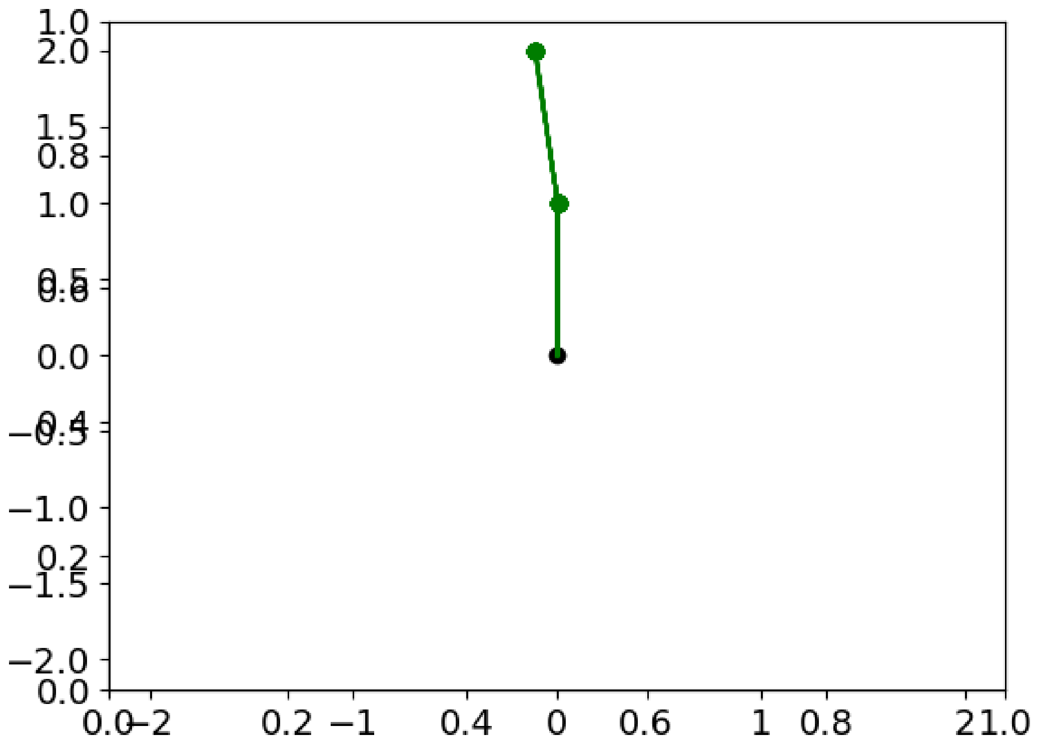
? Answer (start)

In [39]:
```
###ANSWER HERE
params = {
    'm1': 1.0,
    'm2': 1.0,
    'l1': 1.0,
    'l2': 1.0,
    'T': 15.0,
    'g': 9.8,
    'dt': 0.01,
    'initPos': np.array([3.14, 3.14+0.11]),
    'initVel': np.array([0.0, 0.0])
}

allPositions = []
for i in range(10):
    init_pos = params['initPos'] + np.array([0, i*1e-4])
    params['initPos'] = init_pos
    _, pos, _ = run_double_pendulum_sim(params)
    allPositions.append(pos)
```

In [40]:
```
ani=animateMe_doublePendula(allPositions)
HTML(ani.to_jshtml())
```

```
WARNING:matplotlib.animation:Animation size has reached 20992911 bytes, exce
eding the limit of 20971520.0. If you're sure you want a larger animation em
bedded, set the animation.embed_limit rc parameter to a larger value (in M
B). This and further frames will be dropped.
```

Out[40]:



[✓ Once] Loop Reflect

✓ Answer (end)

Acknowledgements:

- Ex 1: George Gollin (original); Bryan Clark, Ryan Levy (modifications)
- Ex 2: Bryan Clark (original)