ME314 Homework 2

Submission instructions

Deliverables that should be included with your submission are shown in **bold** at the end of each problem statement and the corresponding supplemental material. Your homework will be graded IFF you submit a **single** PDF, .mp4 videos of animations when requested and a link to a Google colab file that meet all the requirements outlined below.

- List the names of students you've collaborated with on this homework assignment.
- Include all of your code (and handwritten solutions when applicable) used to complete the problems.
- Highlight your answers (i.e. **bold** and outline the answers) for handwritten or markdown questions and include simplified code outputs (e.g. .simplify()) for python questions.
- Enable Google Colab permission for viewing

- Click Share in the upper right corner
- Under "Get Link" click "Share with..." or "Change"
- Then make sure it says "Anyone with Link" and "Editor" under the dropdown menu

- Make sure all cells are run before submitting (i.e. check the permission by running your code in a private mode)

- Please don't make changes to your file after submitting, so we can grade it!

- Submit a link to your Google Colab file that has been run (before the submission deadline) and don't edit it afterwards!

NOTE: This Juputer Notebook file serves as a template for you to start homework. Make sure you first copy this template to your own Google driver (click "File" -> "Save a copy in Drive"), and then start to edit it.

In [1]:
```
# this code is provided for upgrading sympy to latest version, you don't need to run i
# by yourself, so please leave it commented out
# !pip install --upgrade sympy

# print sympy version for testing, should be 1.6.2
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
import plotly
print(sym.__version__)
```

```
1.11.1
```

In [2]:
```
################################################################################
# If you're using Google Colab, uncomment this section by selecting the whole section
# ctrl+'/' on your and keyboard. Run it before you start programming, this will enable
```

```
# LaTeX "display()" function for you. If you're using the local Jupyter environment, l
##########################################################################################
# def custom_latex_printer(exp,**options):
#     from google.colab.output._publish import javascript
#     url = "https://cdnjs.cloudflare.com/ajax/libs/mathjax/3.1.1/latest.js?config=TeX
#     javascript(url=url)
#     return sym.printing.latex(exp,**options)
# sym.init_printing(use_latex="mathjax",latex_printer=custom_latex_printer)
```

Below are the help functions in previous homeworks, which you may need for this homework.

In [3]:
```
#deleted 'em and added my own helper functions from HW1
```

Helper Functions

In [4]:
```
def compute_EL(lagrangian, q):
    '''
    Helper function for computing the Euler-Lagrange equations for a given system,
    so I don't have to keep writing it out over and over again.

    Inputs:
    - lagrangian: our Lagrangian function in symbolic (Sympy) form
    - q: our state vector [x1, x2, ...], in symbolic (Sympy) form

    Outputs:
    - eqn: the Euler-Lagrange equations in Sympy form
    '''

    # wrap system states into one vector (in SymPy would be Matrix)
    #q = sym.Matrix([x1, x2])
    qd = q.diff(t)
    qdd = qd.diff(t)

    # compute derivative wrt a vector, method 1
    # wrap the expression into a SymPy Matrix
    L_mat = sym.Matrix([lagrangian])
    dL_dq = L_mat.jacobian(q)
    dL_dqdot = L_mat.jacobian(qd)

    #set up the Euler-Lagrange equations
    LHS = dL_dq - dL_dqdot.diff(t)
    RHS = sym.Matrix([0,0]).T
    eqn = sym.Eq(LHS.T, RHS.T)

    return eqn


def solve_EL(eqn, var):
    '''
    Helper function to solve and display the solution for the Euler-Lagrange
    equations.

    Inputs:
    - eqn: Euler-Lagrange equation (type: Sympy Equation())
```

```
    - var: state vector (type: Sympy Matrix). typically a form of q-doubledot
        but may have different terms for

    Outputs:
    - Prints symbolic solutions
    - Returns symbolic solutions in a dictionary
    '''

    soln = sym.solve(eqn, var, dict = True)
    eqns_solved = []

    for i, sol in enumerate(soln):
        for x in var:
            eqn_solved = sym.Eq(x, sol[x])
            eqns_solved.append(eqn_solved)

    return eqns_solved
```

In [5]:

```
def rk4(dxdt, x, t, dt):
    '''
    Applies the Runge-Kutta method, 4th order, to a sample function,
    for a given state q0, for a given step size. Currently only
    configured for a 2-variable dependent system (x,y).
    ==========
    dxdt: a Sympy function that specifies the derivative of the system of interest
    t: the current timestep of the simulation
    x: current value of the state vector
    dt: the amount to increment by for Runge-Kutta
    ======
    returns:
    x_new: value of the state vector at the next timestep
    '''
    k1 = dt * dxdt(t, x)
    k2 = dt * dxdt(t + dt/2.0, x + k1/2.0)
    k3 = dt * dxdt(t + dt/2.0, x + k2/2.0)
    k4 = dt * dxdt(t + dt, x + k3)
    x_new = x + (k1 + 2.0*k2 + 2.0*k3 + k4)/6.0

    return x_new

def euler(dxdt, x, t, dt):
    '''
    Euler's method

    Parameters:
    =============
    dxdt: a Sympy function that specifies the derivative of the system of interest
    t: the current timestep of the simulation
    x: current value of the state vector
    dt: the amount to increment by for Euler
    ======
    returns:
    x_new: value of the state vector at the next timestep
    '''
    x_new = x + dt * dxdt(t, x)
    return x_new
```

```
eul = euler(lambda t, x: x**2, 0.5, 2, 0.1)
assert eul == 0.525, f"Euler val: {res}"
print("assertion passed")

rk = rk4(lambda t, x: x**2, 0.5, 2, 0.1)
assert np.isclose(rk, 0.526315781526278075), f"RK4 value: {rk}" #from an online RK4 sc
print("assertion passed")
```

```
assertion passed
assertion passed
```

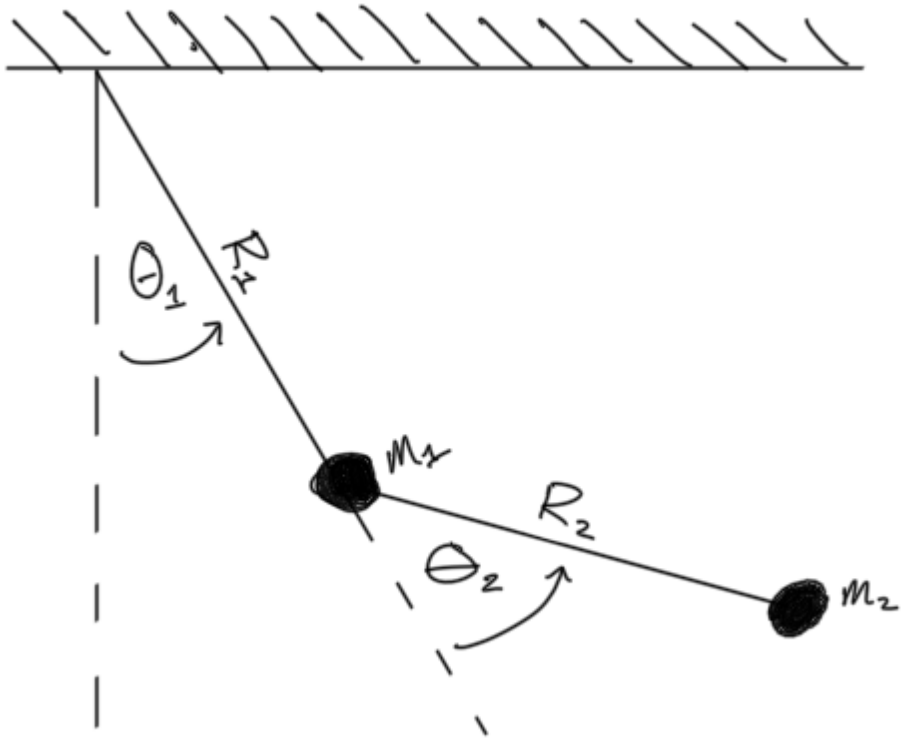
In [6]:

```
def simulate(f, x0, tspan, dt, integrate):
    """
    This function takes in an initial condition x0, a timestep dt,
    a time span tspan consisting of a list [min_time, max_time],
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x0. It outputs a full trajectory simulated
    over the time span of dimensions (xvec_size, time_vec_size).

    Parameters
    ============
    f: Python function
        derivate of the system at a given step x(t),
        it can considered as \dot{x}(t) = func(x(t))
    x0: NumPy array
        initial conditions
    tspan: Python list
        tspan = [min_time, max_time], it defines the start and end
        time of simulation
    dt:
        time step for numerical integration
    integrate: Python function
        numerical integration method used in this simulation

    Return
    ============
    x_traj:
        simulated trajectory of x(t) from t=0 to tf
    """
    N = int((max(tspan)-min(tspan))/dt)
    x = np.copy(x0)
    tvec = np.linspace(min(tspan),max(tspan),N)
    xtraj = np.zeros((len(x0),N))

    for i in range(N):
        t = tvec[i]
        xtraj[:,i]=integrate(f,x,t,dt)
        x = np.copy(xtraj[:,i])
    return xtraj
```

Problem 1 (15pts)

In [7]:

```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/mast
```



You're given a double-pendulum system hanging in gravity is shown in the figure above. With $q = [\theta_1, \theta_2]$ as the system configuration variables, use Python's SymPy package to compute the Lagrangian of the system. Note that we assume that the z-axis is pointing out from the screen/paper and thus the positive direction of rotation is counter-clockwise.

> *Hint 1: We recommend that you compute the positions and their time derivatives (velocities) in x-y coordinates! This will involve using some trigonometry to express the x and y coordinates of each mass in terms of θ_1 and θ_2. Consequently, compute kinetic and potential energy based on that.*
>
> *Hint 2: By convention we will define gravity with positive sign (i.e. $g = 9.8$) for numerical evaluation required in the later problems. As such, be careful with the sign of potential energy! You can always go back here after you verify your results by numerical evaluation in Problem 2 and 3.*

Turn in: Include the code used to symbolically compute Lagrangian and highlight the output of your code which should be the symbolic Lagrangian expression.

In [8]:
```
# You can start your implementation here :)

#1. grab variables
m1, m2, R1, R2, g = sym.symbols(r'm1, m2, R1, R2, g')# constants
t = sym.symbols(r't')

theta1 = sym.Function(r'\theta_1')(t)
```

```
theta2 = sym.Function(r'\theta_2')(t)
theta1d = theta1.diff(t)
theta2d = theta2.diff(t)

consts_dict = {
    m1: 1,
    m2: 2,
    R1: 2,
    R2: 1,
    g: 9.8,
} #all in SI units

#-------------------------#

#intermediate variables for ease of representation
x1 = sym.Function(r'x1')(t)
y1 = sym.Function(r'y1')(t)
x2 = sym.Function(r'x2')(t)
y2 = sym.Function(r'y2')(t)

x1 = R1 * sym.sin(theta1)
y1 = -R1 * sym.cos(theta1)

x2 = x1 + R2 * sym.sin(theta1 + theta2)
y2 = y1 - R2 * sym.cos(theta1 + theta2)

x1d = x1.diff(t)
x2d = x2.diff(t)
y1d = y1.diff(t)
y2d = y2.diff(t)

#-------------------------#

KE1 = 0.5 * m1 * (x1d**2 + y1d**2)
KE2 = 0.5 * m2 * (x2d**2 + y2d**2)

U1 = m1 * g * y1
U2 = m2 * g * y2

lagrangian = (KE1 + KE2) - (U1 + U2)
print('Lagrangian:')
display(lagrangian)
```

```
Lagrangian:
```

$$R_1 g m_1 \cos\left(\theta_1(t)\right) - g m_2 \left(-R_1 \cos\left(\theta_1(t)\right) - R_2 \cos\left(\theta_1(t) + \theta_2(t)\right)\right)$$
$$+ 0.5 m_1 \left(R_1^2 \sin^2\left(\theta_1(t)\right) \left(\frac{d}{dt}\theta_1(t)\right)^2 + R_1^2 \cos^2\left(\theta_1(t)\right) \left(\frac{d}{dt}\theta_1(t)\right)^2 \right)$$
$$+ 0.5 m_2 \left(\left(R_1 \sin\left(\theta_1(t)\right) \frac{d}{dt}\theta_1(t) + R_2 \left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \sin\left(\theta_1(t) + \theta_2(t)\right) \right)^2 + \left(R_1 \cos \right.$$

Problem 2 (15pts)

see Appendix for the full equations

Use Python's SymPy package to compute the Euler-Lagrange equations for the same double-pendulum system in Problem 1 and solve for $\ddot{\theta}_1$ and $\ddot{\theta}_2$.

Turn in: Include the code used to symbolically compute and solve Euler-Lagrange equations. Also include the output of your code, as in the symbolic expression of Euler-Lagrange equations and their solutions (i.e. $\ddot{\theta}_1$ and $\ddot{\theta}_2$).

In [9]:
```
# You can start your implementation here :)

q = sym.Matrix([theta1, theta2])
qd = q.diff(t)
qdd = qd.diff(t)

eqn = compute_EL(lagrangian, q)
eqn = eqn.simplify()
print("Euler-Lagrange equations:")
display(eqn)
```

```
Euler-Lagrange equations:
```

see Appendix for the rest of the equation

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1.0R_1^2 m_1 \frac{d^2}{dt^2}\theta_1(t) - R_1 g m_1 \sin\left(\theta_1(t)\right) - g m_2 \left(R_1 \sin\left(\theta_1(t)\right) + \right. \\ -1.0 m_2 \left(R_1^2 \frac{d^2}{dt^2}\theta_1(t) - 2R_1 R_2 \sin\left(\theta_2(t)\right) \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t) - R_1 R_2 \sin\left(\theta_2(t)\right) \left(\frac{d}{dt}\theta_2(t) \right. \right. \\ \left. \left. \left(\theta_2(t)\right) \frac{d^2}{dt^2}\theta_2(t) + R_2^2 \frac{d^2}{dt^2}\theta_1(t) + R_2^2 \frac{d^2}{dt^2}\theta_2(t) \right. \right. \\ \left. \left. -1.0 R_2 m_2 \left(R_1 \sin\left(\theta_2(t)\right) \left(\frac{d}{dt}\theta_1(t) \right)^2 + R_1 \cos\left(\theta_2(t)\right) \frac{d^2}{dt^2}\theta_1(t) + R_2 \frac{d^2}{dt^2}\theta_1(t) - \right. \right. \end{bmatrix}$$

In [10]:
```
solved_eqns = solve_EL(eqn, qdd)
simplified_eqns = []

print("Solved:")
for eq in solved_eqns:
    eq_new = eq.simplify()
    simplified_eqns.append(eq_new)
    display(eq_new)
```

```
Solved:
```

$$\frac{d^2}{dt^2}\theta_1(t) = \frac{0.5 R_1 m_2 \sin\left(2\theta_2(t)\right) \left(\frac{d}{dt}\theta_1(t) \right)^2 + 1.0 R_2 m_2 \sin\left(\theta_2(t)\right) \left(\frac{d}{dt}\theta_1(t) \right)^2 + 2.0 R_2 m_2 \sin \left(\theta_2(t)\right) \left(\frac{d}{dt}\theta_2(t) \right)^2 - 1.0 g m_1 \sin\left(\theta_1(t)\right) + 0.5 g m_2 \sin\left(\theta_1(t) + 2\theta_2(t)\right) - 0.5 g m_2 \sin}{R_1 \left(m_1 + m_2 \sin^2\left(\theta_2(t)\right) \right)}$$

see Appendix for the rest of the equation

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{2\left(-1.0R_1^2 m_1 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0R_1^2 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0R_1R_2 \sin\left(2\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 0.5R_1R_2 m_2 \sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.5R_1 g m_1 \sin\left(\theta_1(t)\right) + 0.5R_1 g m_2 \sin\left(\theta_1(t) - \theta_2(t)\right) - 0.5R_1 g m_2 \sin\left(\theta_1(t) + \theta_2(t)\right) - 1.0R_2^2 m_2 \sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 1.0R_2^2 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 1.0R_2 g m_1 \sin\left(\theta_1(t)\right) - \left(\theta_1(t)\right)\right)}{R_1R_2 \cdot \left(2m_1 - m_2\cos\left(2\theta_2(t)\right) + m_2\right.}$$

see Appendix for the rest of the equation

Problem 3 (15pts)

Numerically evaluate your solutions for $\ddot{\theta}_1$ and $\ddot{\theta}_2$ from Problem 2 using SymPy's `lambdify()` method, simulate the system for $t \in [0, 5]$ with $m_1 = 1, m_2 = 2, R_1 = 2, R_2 = 1$ and initial condition as $\theta_1 = \theta_2 = -\frac{\pi}{2}, \dot{\theta}_1 = \dot{\theta}_2 = 0$. Plot the simulated trajectories of $\theta_1(t)$ and $\theta_2(t)$ versus time.

> *Hint 1: Feel free to use the provided example code or your implementation in Homework 1.*
>
> *Hint 2: By convention, we will define $g = 9.8$ as a positive constant. If you got some wierd "flipped" trajectory, go back to Problem 1 and check the sign of your gravity potential energy term.*

Turn in: Include the code used for numerical evaluation and simulation as well as the output of your code, i.e. values of $\ddot{\theta}_1$, $\ddot{\theta}_2$ and the plot of $\theta_1(t)$ and $\theta_2(t)$ trajectories versus time. Make sure to label the figure and specify the axis as well as include a legend.

In [11]:
```
# You can start your implementation here :)

theta1dd_sy = simplified_eqns[0]
theta2dd_sy = simplified_eqns[1]

theta1dd_sy = theta1dd_sy.subs(consts_dict)
theta2dd_sy = theta2dd_sy.subs(consts_dict)

print("Theta1dd and Theta1dd with constants substituted in:")
display(theta1dd_sy)
display(theta2dd_sy)

q_ext = sym.Matrix([theta1, theta2, theta1d, theta2d])
theta1dd_np = sym.lambdify(q_ext, theta1dd_sy.rhs)
theta2dd_np = sym.lambdify(q_ext, theta2dd_sy.rhs)
```

```
Theta1dd and Theta1dd with constants substituted in:
```

see Appendix for the rest of the equation

$$\frac{d^2}{dt^2}\theta_1(t) = \frac{9.8\sin\left(\theta_1(t)+2\theta_2(t)\right) - 19.6\sin\left(\theta_1(t)\right) + 2.0\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 4.0\sin\left(\theta_2 + 2.0\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2\right.}{2\cdot\left(2\sin^2\left(\theta_2(t)\right)+1\right)}$$

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{29.4\sin\left(\theta_1(t)-\theta_2(t)\right) - 29.4\sin\left(\theta_1(t)+\theta_2(t)\right) - 9.8\sin\left(\theta_1(t)+2\theta_2(t)\right) + 19.6}{(\theta_2(t))\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 2.0\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 - 4.0\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - \cdot}$$
$$\frac{(2\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2}{4-2\cos\left(2\theta_2(t)\right)}$$

In [12]:
```
# You can start your implementation here :)

def dxdt(t, s):
    '''
    Derivative of our state vector at the given state [x1, x2, x1d, x2d].
    Inputs:
     - xmdd: a lambdified Python function that calculates xmddot as a function
         of xm, theta, xmd, thetad
     - thetadd: ditto for thetaddot
     - q: current value of our state vector

     implicitly, this dxdt() function takes in the 2 functions x1dd and x2dd -
     they are not included in the arguments for compatibility with rk4() and euler()

     Returns: an array [xmd, thetad, xmdd, thetadd]
    '''
    #potential to turn this into a dictionary-type set of functions

    #calculate x1'' and x2'' based on current state values and sympy function
    #x1' and x2' are given
    return np.array([s[2], s[3], theta1dd_np(*s), theta2dd_np(*s)])
```

In [13]:
```
#simulate system using initial conditions
th0 = float(-sym.pi/2.0)
s0 = [th0, th0, 0, 0]
t_span = [0, 5]
dt = 0.01

print("Values of theta1dd and theta2dd at t0 with ICs:")
print(str(theta1dd_np(*s0)) + "s^-2")
print(str(round(theta2dd_np(*s0), 2)) + "s^-2")
```

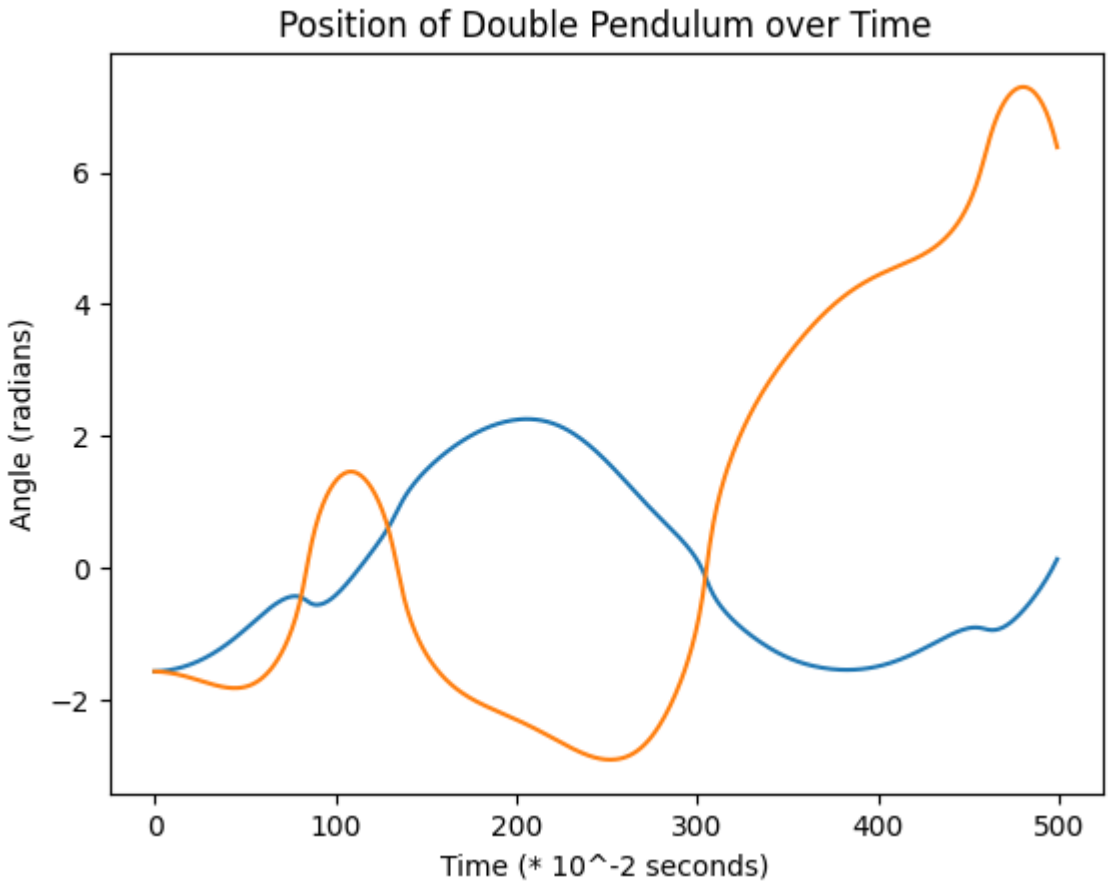
```
Values of theta1dd and theta2dd at t0 with ICs:
4.9s^-2
-4.9s^-2
```

In [14]:
```
#use rk4 for numerical integration
q_array = simulate(dxdt, s0, t_span, dt, rk4)
print('shape of trajectory: ', q_array.shape)
```

```
#plot
theta1_array = q_array[0]
theta2_array = q_array[1]
plt.plot(theta1_array)
plt.plot(theta2_array)

plt.xlabel("Time (* 10^-2 seconds)")
plt.ylabel("Angle (radians)")
plt.title("Position of Double Pendulum over Time")
```

```
shape of trajectory:  (4, 500)
```

Out[14]: Text(0.5, 1.0, 'Position of Double Pendulum over Time')



Verify results by calculating energy of system at each timestep

In [15]:
```
#positions in q[0] and q[1], velocities in q[2] and q[3]
def KE1(s):
    [_, _, theta1d, _] = s
    m1 = 1
    R1 = 2
    return 0.5 * m1 * R1**2 * theta1d**2

def KE2(s):
    [theta1, theta2, theta1d, theta2d] = s
    m2 = 2
    R1 = 2
    R2 = 1

    x1d = R1 * np.cos(theta1) * theta1d
    y1d = R1 * np.sin(theta1) * theta1d
```

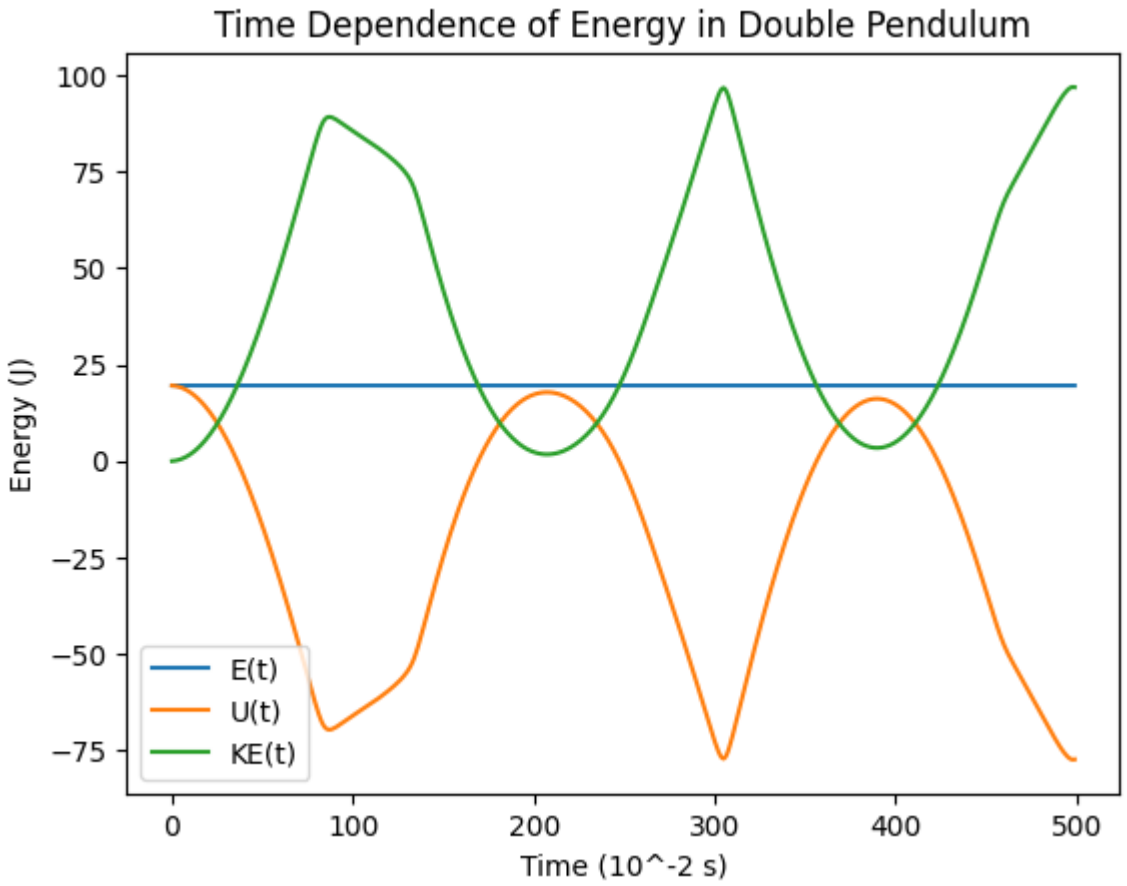
```
    x2d = x1d + R2 * np.cos(theta1 + theta2) * (theta1d + theta2d)
    y2d = y1d + R2 * np.sin(theta1 + theta2) * (theta1d + theta2d)

    return 0.5 * m2 * (x2d**2 + y2d**2)

def U1(s):
    [theta1, _, _, _] = s
    m1 = 1
    R1 = 2
    g = 9.8
    return m1 * g * -R1 * np.cos(theta1)

def U2(s):
    [theta1, theta2, _, _] = s
    m2 = 2
    R1 = 2
    R2 = 1
    g = 9.8
    return -m2 * g * (
        R1 * np.cos(theta1) +
        R2 * np.cos(theta1 + theta2)
    )

def E(s):
    return KE1(s) + KE2(s) + U1(s) + U2(s)

# print(q_array)
# print(np.matrix(q_array))
len(q_array.T)
E_array = [E(s) for s in q_array.T]
KE_array = [KE1(s) + KE2(s) for s in q_array.T]
U_array = [U1(s) + U2(s) for s in q_array.T]
```

In [16]:

```
#plot
plt.figure()
plt.title("Time Dependence of Energy in Double Pendulum")
plt.xlabel("Time (10^-2 s)")
plt.ylabel('Energy (J)')

plt.plot(E_array)
plt.plot(U_array)
plt.plot(KE_array)
plt.legend(["E(t)", "U(t)", "KE(t)"], loc = 'lower left')
```

Out[16]:

```
<matplotlib.legend.Legend at 0x1d229eb43a0>
```



Problem 4 (10pts)

Finally, let's get fancy! Use the function provided below to animate your simulation of the double-pendulum system based on the trajectories you got in Problem 3.

> *Hint 1: If your animation seems to be slow, press "pause" and then press "play" again! This should play animation at normal speed.*

Turn in: Include the code used to generate the animation but note that you don't need to include the animation function! In addition, upload the video of animation through Canvas and make sure that the video format is .mp4. You can use screen capture or record the screen directly with your phone.

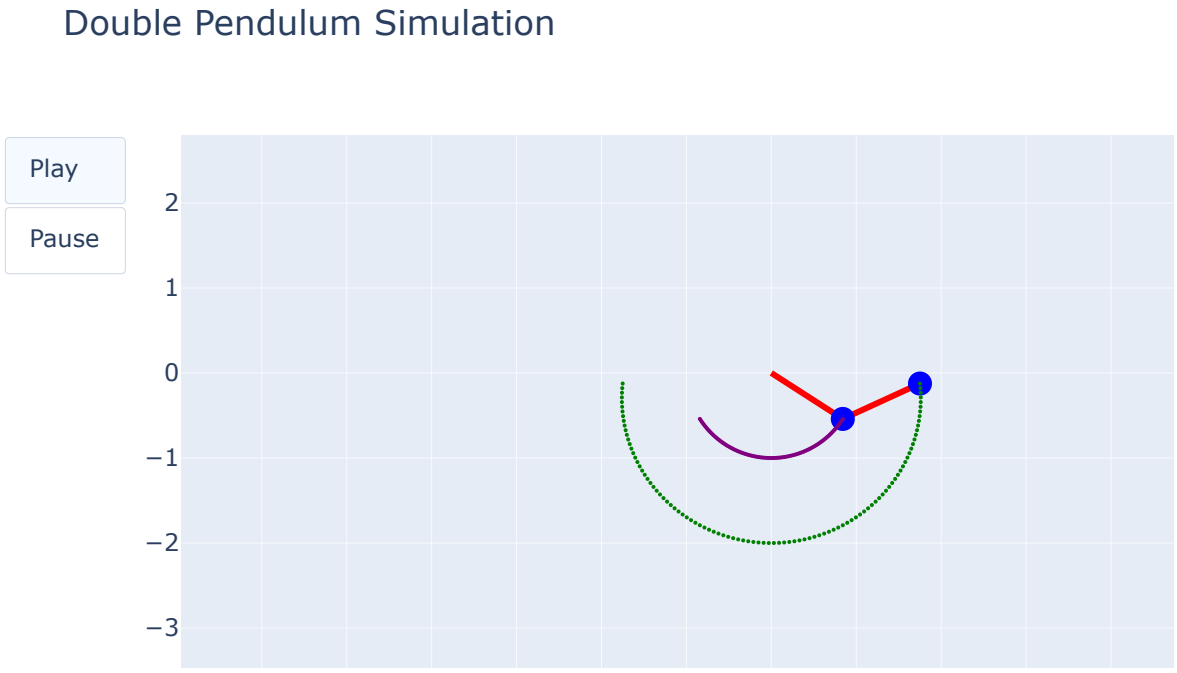
In [17]:
```
def animate_double_pend(theta_array,L1=1,L2=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ===============================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first pendulum
    L2:
```

```
    length of the second pendulum
T:
    length/seconds of animation duration

Returns: None
"""

################################
# Imports required for animation.
from plotly.offline import init_notebook_mode, iplot
from IPython.display import display, HTML
import plotly.graph_objects as go

#######################
# Browser configuration.
def configure_plotly_browser_state():
    import IPython
    display(IPython.core.display.HTML('''
        <script src="/static/components/requirejs/require.js"></script>
        <script>
          requirejs.config({
            paths: {
              base: '/static/base',
              plotly: 'https://cdn.plot.ly/plotly-1.5.1.min.js?noext',
            },
          });
        </script>
        '''))
configure_plotly_browser_state()
init_notebook_mode(connected=False)

###############################################
# Getting data from pendulum angle trajectories.
xx1=L1*np.sin(theta_array[0])
yy1=-L1*np.cos(theta_array[0])
xx2=xx1+L2*np.sin(theta_array[0]+theta_array[1])
yy2=yy1-L2*np.cos(theta_array[0]+theta_array[1])
N = len(theta_array[0]) # Need this for specifying length of simulation

#####################################
# Using these to specify axis limits.
xm=np.min(xx1)-0.5
xM=np.max(xx1)+0.5
ym=np.min(yy1)-2.5
yM=np.max(yy1)+1.5

############################
# Defining data dictionary.
# Trajectories are here.
data=[dict(x=xx1, y=yy1,
           mode='lines', name='Arm',
           line=dict(width=2, color='blue')
          ),
      dict(x=xx1, y=yy1,
           mode='lines', name='Mass 1',
           line=dict(width=2, color='purple')
          ),
```

```
          dict(x=xx2, y=yy2,
               mode='lines', name='Mass 2',
               line=dict(width=2, color='green')
              ),
          dict(x=xx1, y=yy1,
               mode='markers', name='Pendulum 1 Traj',
               marker=dict(color="purple", size=2)
              ),
          dict(x=xx2, y=yy2,
               mode='markers', name='Pendulum 2 Traj',
               marker=dict(color="green", size=2)
              ),
        ]

    ################################
    # Preparing simulation layout.
    # Title and axis ranges are here.
    layout=dict(xaxis=dict(range=[xm, xM], autorange=False, zeroline=False,dtick=1),
                yaxis=dict(range=[ym, yM], autorange=False, zeroline=False,scaleanchor
                title='Double Pendulum Simulation',
                hovermode='closest',
                updatemenus= [{'type': 'buttons',
                               'buttons': [{'label': 'Play','method': 'animate',
                                            'args': [None, {'frame': {'duration': T, '
                                           {'args': [[None], {'frame': {'duration': T,
                                            'transition': {'duration': 0}}],'label': '
                                          ]
                              }]
                )

    ########################################
    # Defining the frames of the simulation.
    # This is what draws the lines from
    # joint to joint of the pendulum.
    frames=[dict(data=[dict(x=[0,xx1[k],xx2[k]],
                            y=[0,yy1[k],yy2[k]],
                            mode='lines',
                            line=dict(color='red', width=3)
                            ),
                       go.Scatter(
                            x=[xx1[k]],
                            y=[yy1[k]],
                            mode="markers",
                            marker=dict(color="blue", size=12)),
                       go.Scatter(
                            x=[xx2[k]],
                            y=[yy2[k]],
                            mode="markers",
                            marker=dict(color="blue", size=12)),
                      ]) for k in range(N)]

    ########################################
    # Putting it all together and plotting.
    figure1=dict(data=data, layout=layout, frames=frames)
    iplot(figure1)

###################################################
```

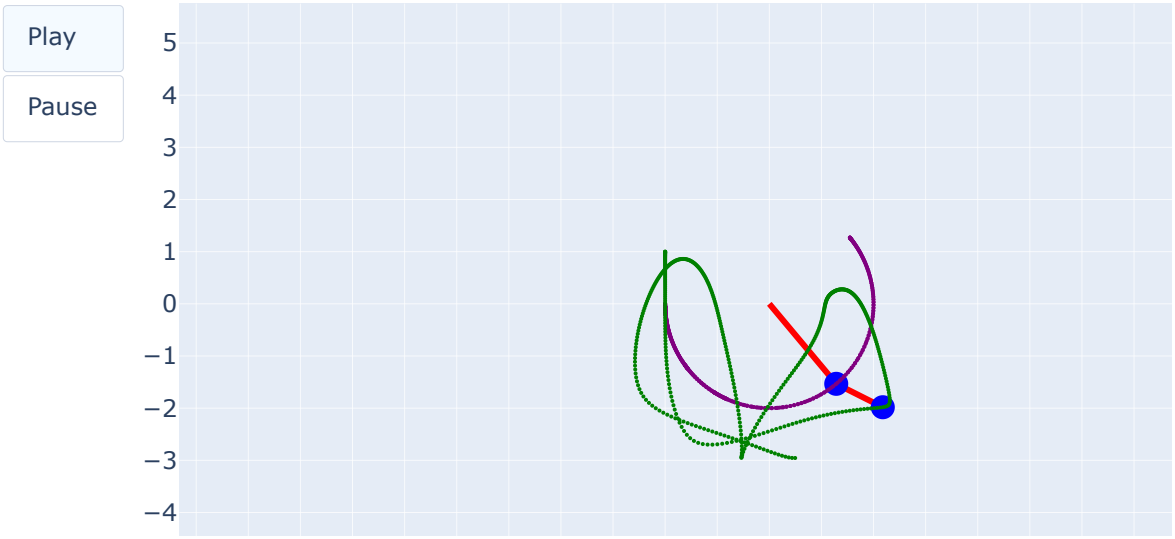
```
# Example of animation

# provide a trajectory of double-pendulum
# (note that this array below is not an actual simulation,
# but lets you see this animation code work)
import numpy as np
sim_traj = np.array([np.linspace(-1, 1, 100), np.linspace(-1, 1, 100)])
print('shape of trajectory: ', sim_traj.shape)

# second, animate!
animate_double_pend(sim_traj,L1=1,L2=1,T=10)
```

```
shape of trajectory:  (2, 100)
```



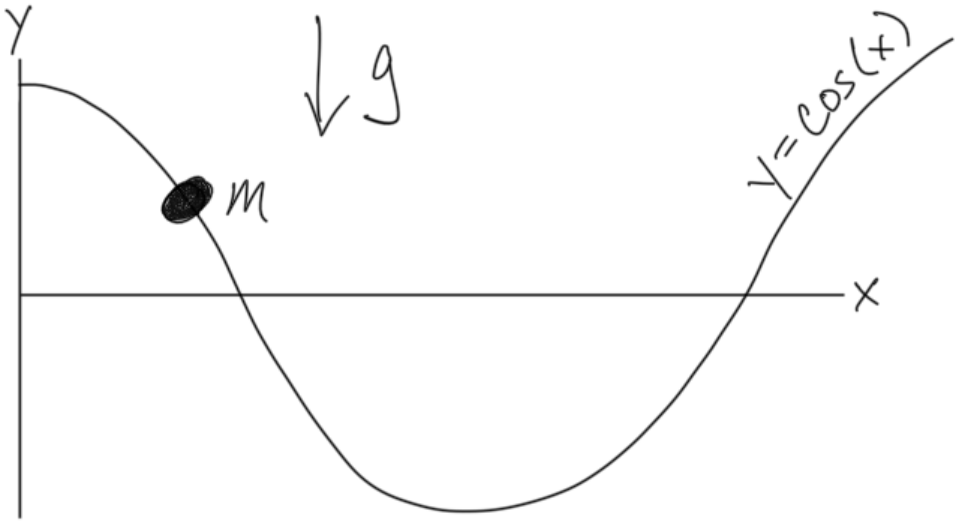
In [18]:
```
# You can start your implementation here
animate_double_pend(q_array,L1=2,L2=1,T=10)
```

Double Pendulum Simulation



Problem 5 (15pts)

In [19]:
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/mast
```



As shown in the figure above, a bead in gravity is constrained to the path $y = \cos(x)$. With the x-y positions of the bead as the system configuration variables, compute the Lagrangian symbolically using SymPy and write down constraint equation $\phi(q) = 0$ of the system as SymPy's symbolic equation.

Turn in: Include the code you used to compute the Lagrangian and generate the constraint equation as well as the output of your code, i.e. the symbolic expression for the Lagrangian and constraint equation.

In [20]:
```
# You can start your implementation here

#grab variables
m, g, lamb = sym.symbols(r'm, g, \lambda')
t = sym.symbols(r't')

#define state vector elements
x = sym.Function(r'x')(t)
y = sym.Function(r'y')(t)

xd = x.diff(t)
yd = y.diff(t)
xdd = xd.diff(t)
ydd = yd.diff(t)

q = sym.Matrix([x, y])
qd = q.diff(t)
qdd = qd.diff(t)

#kinetic + potential energy functions; lagrangian
KE = 0.5 * m * (xd**2 + yd**2)
U = m * g * y
lagrangian = KE - U
print("Lagrangian:")
display(lagrangian)

#constraint function
phi = y - sym.cos(x)
constraint_eq = sym.Eq(phi, 0)
print("Constraint equation:")
display(constraint_eq)
```

```
Lagrangian:
```

$$-gmy(t) + 0.5m\left(\left(\frac{d}{dt}x(t)\right)^2 + \left(\frac{d}{dt}y(t)\right)^2\right)$$

```
Constraint equation:
```

$$y(t) - \cos\left(x(t)\right) = 0$$

Problem 6 (10pts)

Use Python's SymPy's package to solve for the equations of motion ($\ddot{x}$ and $\ddot{y}$) and constraint force λ for the constrained bead system in Problem 5.

Turn in: Include the code used to symbolically solve for the equations of motion and constraint force as well as the output of the code, i.e. the symbolic EOM equations and constraint force.

In [21]:
```
# You can start your implementation here

#constraint equation math
lamb_grad = sym.Matrix([lamb * phi.diff(a) for a in q])
phidd = phi.diff(t).diff(t)

#compute E-L equations
eqn = compute_EL(lagrangian, q)
eq_constrained = sym.Eq(eqn.lhs, lamb_grad)

print("Constrained E-L equations:")
display(eq_constrained)
print("Constraint force:")
display(phidd)
```

```
Constrained E-L equations:
```

$$\begin{bmatrix} -1.0m\frac{d^2}{dt^2}x(t) \\ -gm - 1.0m\frac{d^2}{dt^2}y(t) \end{bmatrix} = \begin{bmatrix} \lambda \sin\left(x(t)\right) \\ \lambda \end{bmatrix}$$

```
Constraint force:
```

$$\sin\left(x(t)\right)\frac{d^2}{dt^2}x(t) + \cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2 + \frac{d^2}{dt^2}y(t)$$

In [22]:
```
#format equations so they're all in one matrix
expr_matrix = eqn.lhs - lamb_grad
phidd_matrix = sym.Matrix([phidd])
expr_matrix = expr_matrix.row_insert(2,phidd_matrix)

print("Matrix of expressions to solve:")
display(expr_matrix)
```

```
Matrix of expressions to solve:
```

$$\begin{bmatrix} -\lambda \sin\left(x(t)\right) - 1.0m\frac{d^2}{dt^2}x(t) \\ -\lambda - gm - 1.0m\frac{d^2}{dt^2}y(t) \\ \sin\left(x(t)\right)\frac{d^2}{dt^2}x(t) + \cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2 + \frac{d^2}{dt^2}y(t) \end{bmatrix}$$

In [23]:
```
#solve E-L equations
q_mod = sym.Matrix([xdd, ydd, lamb])
eqns_solved = solve_EL(expr_matrix, q_mod)

#display EOM
xdd_sy = eqns_solved[0].simplify()
ydd_sy = eqns_solved[1].simplify()
lamb_sy = eqns_solved[2].simplify()
```

```
print("Equations of motion and constraint force lambda:")
display(xdd_sy)
display(ydd_sy)
display(lamb_sy)
```

```
Equations of motion and constraint force lambda:
```

$$\frac{d^2}{dt^2}x(t) = \frac{\left(g - \cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2\right)\sin\left(x(t)\right)}{\sin^2\left(x(t)\right) + 1.0}$$

$$\frac{d^2}{dt^2}y(t) = -\frac{g\sin^2\left(x(t)\right) + \cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2}{\sin^2\left(x(t)\right) + 1.0}$$

$$\lambda = \frac{m\left(-g + \cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2\right)}{\sin^2\left(x(t)\right) + 1.0}$$

Problem 7 (20pts)

Simulate this constrained bead system with $m = 1$ and initial condition as $x = 0.1, y = \cos(0.1), \dot{x} = \dot{y} = 0$, for $t \in [0, 10]$. Animate your simulated trajectory using the provided function below.

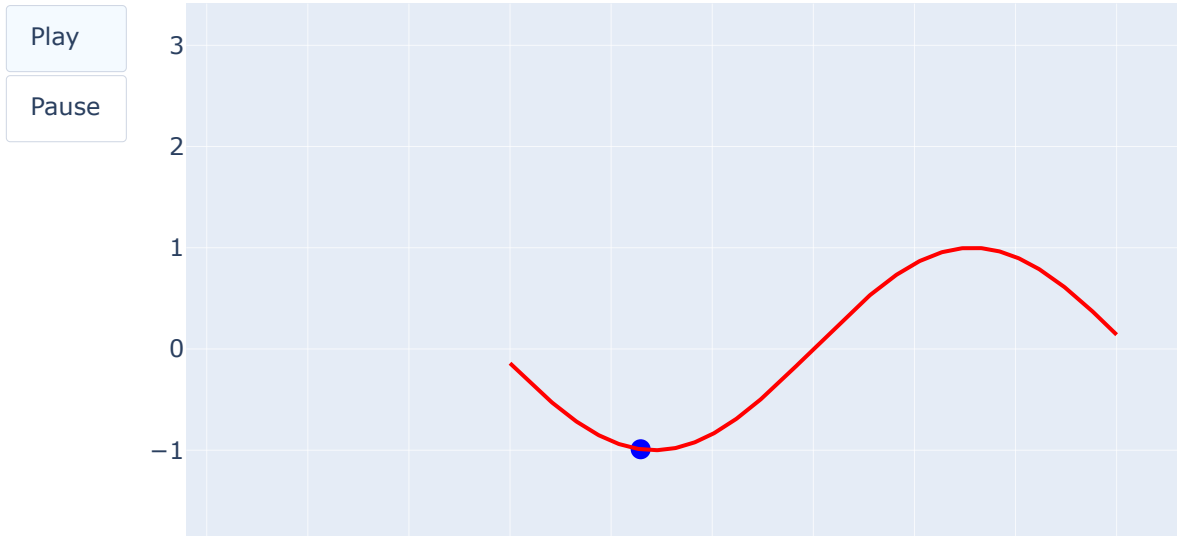
Turn in: Include the code used to generate the animation but note that you don't need to include the animation function! In addition, upload the video of animation through Canvas and make sure that the video format is .mp4. You can use screen capture or record the screen directly with your phone.

In [24]:
```
def animate_bead(xy_array,T=10):
    """
    Function to generate web-based animation of constrained bead system

    Parameters:
    ===============================================
    xy_array:
        trajectory of x and y, should be a NumPy array with
        shape of (2,N)
    T:
        length/seconds of animation duration

    Returns: None
    """

    ################################
    # Imports required for animation.
    from plotly.offline import init_notebook_mode, iplot
    from IPython.display import display, HTML
    import plotly.graph_objects as go
```

```
    ######################
    # Browser configuration.
    def configure_plotly_browser_state():
        import IPython
        display(IPython.core.display.HTML('''
            <script src="/static/components/requirejs/require.js"></script>
            <script>
              requirejs.config({
                paths: {
                  base: '/static/base',
                  plotly: 'https://cdn.plot.ly/plotly-1.5.1.min.js?noext',
                },
              });
            </script>
            '''))
    configure_plotly_browser_state()
    init_notebook_mode(connected=False)

    ################################################
    # Getting data from trajectories.
    N = len(xy_array[0]) # Need this for specifying length of simulation

    ####################################
    # Using these to specify axis limits.
    xm=np.min(xy_array[0])-0.5
    xM=np.max(xy_array[0])+0.5
    ym=np.min(xy_array[1])-0.5
    yM=np.max(xy_array[1])+0.5

    ############################
    # Defining data dictionary.
    # Trajectories are here.
    data=[dict(x=xy_array[0], y=xy_array[1],
               mode='markers', name='bead',
               marker=dict(color="blue", size=10)
              ),
          dict(x=xy_array[0], y=xy_array[1],
               mode='lines', name='trajectory',
               line=dict(width=2, color='red')
              ),
        ]

    ##################################
    # Preparing simulation layout.
    # Title and axis ranges are here.
    layout=dict(xaxis=dict(range=[xm, xM], autorange=False, zeroline=False,dtick=1),
                yaxis=dict(range=[ym, yM], autorange=False, zeroline=False,scaleanchor
                title='Constrained Bead Simulation',
                hovermode='closest',
                updatemenus= [{'type': 'buttons',
                               'buttons': [{'label': 'Play','method': 'animate',
                                            'args': [None, {'frame': {'duration': T, '
                                           {'args': [[None], {'frame': {'duration': T,
                                            'transition': {'duration': 0}}],'label': '
                                          ]
                              }]
              )
```

```
    ########################################
    # Defining the frames of the simulation.
    # This is what draws the bead at each time
    # step of simulation.
    frames=[dict(data=[go.Scatter(
                            x=[xy_array[0][k]],
                            y=[xy_array[1][k]],
                            mode="markers",
                            marker=dict(color="blue", size=10))
                       ]) for k in range(N)]



    ########################################
    # Putting it all together and plotting.
    figure1=dict(data=data, layout=layout, frames=frames)
    iplot(figure1)

#################################################
# Example of animation

# provide a trajectory of constrained bead
# (note that this array below is not an actual simulation,
# but lets you see this animation code work)
import numpy as np
sim_traj = np.array([np.linspace(-3, 3, 600), np.sin(np.linspace(-3, 3, 600))])
print('Shape of trajectory: ', sim_traj.shape)

# second, animate!
animate_bead(sim_traj,T=3)
```

```
Shape of trajectory:  (2, 600)
```



In [25]:
```python
# You can start your implementation here

const_dict = {
    m: 1,
    g: 9.8
}

q_ext = sym.Matrix([x, y, xd, yd])

#subs()titute in variables
xdd_sy = xdd_sy.subs(const_dict)
ydd_sy = ydd_sy.subs(const_dict)
lamb_sy = lamb_sy.subs(const_dict)

print("Xdd and Ydd after substitution:")
display(xdd_sy)
display(ydd_sy)

#lambdify functions
xdd_np = sym.lambdify(q_ext, xdd_sy.rhs)
ydd_np = sym.lambdify(q_ext, ydd_sy.rhs)
```

```
Xdd and Ydd after substitution:
```

$$\frac{d^2}{dt^2}x(t) = \frac{\left(-\cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2 + 9.8\right)\sin\left(x(t)\right)}{\sin^2\left(x(t)\right) + 1.0}$$

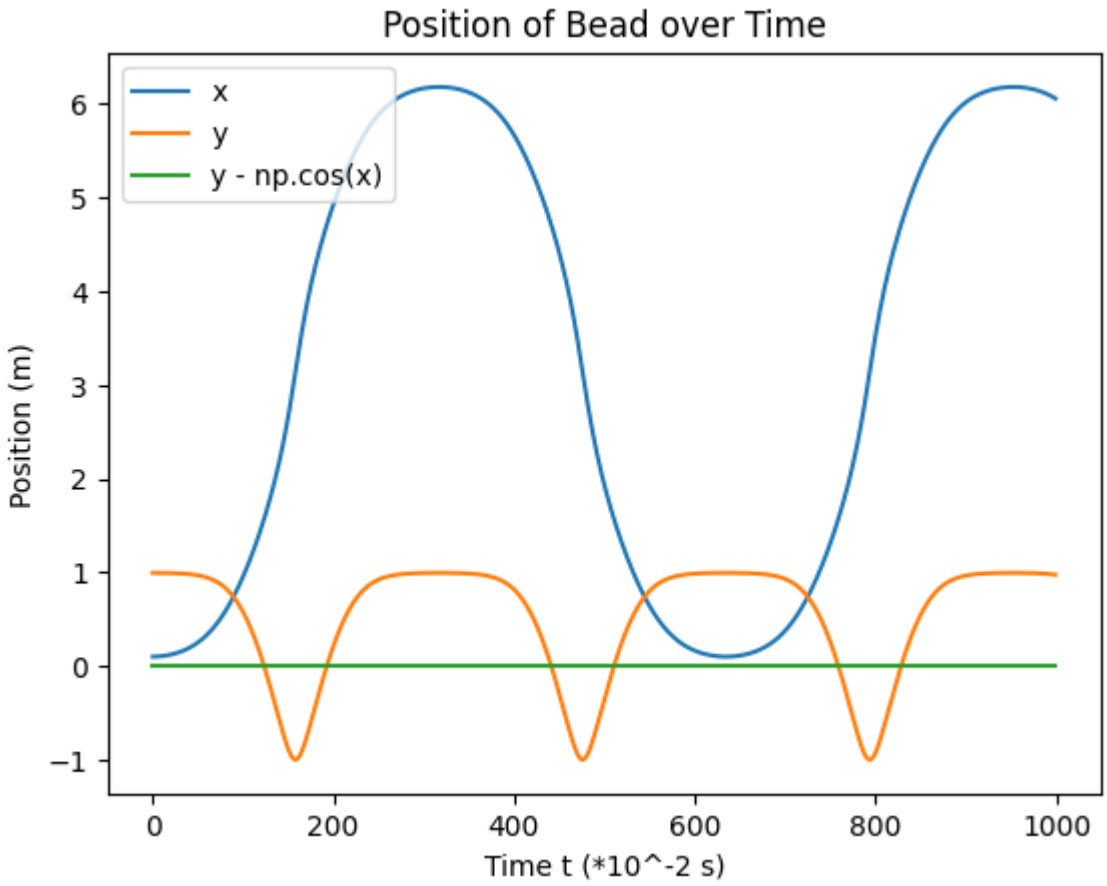
$$\frac{d^2}{dt^2}y(t) = -\frac{9.8\sin^2\left(x(t)\right) + \cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2}{\sin^2\left(x(t)\right) + 1.0}$$

In [26]:
```
def dxdt_p2(t, s):
    '''
    Derivative of our state vector at the given state [x1, x2, x1d, x2d].
    Inputs:
     - xmdd: a lambdified Python function that calculates xmddot as a function
         of xm, theta, xmd, thetad
     - thetadd: ditto for thetaddot
     - q: current value of our state vector

     implicitly, this dxdt() function takes in the 2 functions x1dd and x2dd -
     they are not included in the arguments for compatibility with rk4() and euler()

     Returns: an array [xmd, thetad, xmdd, thetadd]
    '''
    #potential to turn this into a dictionary-type set of functions

    #calculate x1'' and x2'' based on current state values and sympy function
    #x1' and x2' are given
    return np.array([s[2], s[3], xdd_np(*s), ydd_np(*s)])
```

In [27]:
```
#ICs
x0 = [0.1, np.cos(0.1), 0, 0]
t_span = [0, 10]
dt = 0.01

#simulate motion
x_array = simulate(dxdt_p2, x0, t_span, dt, rk4)
print('Shape of trajectory: ',x_array.shape)

x_plt = x_array[0]
y_plt = x_array[1]
diff_plt = [y_plt[i] - np.cos(x_plt[i]) for i in range(len(x_plt))]

#plot so we can see what's going on
plt.figure()
plt.plot(x_plt)
plt.plot(y_plt)
plt.plot(diff_plt)

plt.xlabel("Time t (*10^-2 s)")
plt.ylabel("Position (m)")
plt.title("Position of Bead over Time")
plt.legend(["x","y","y - np.cos(x)"], loc='upper left')
```

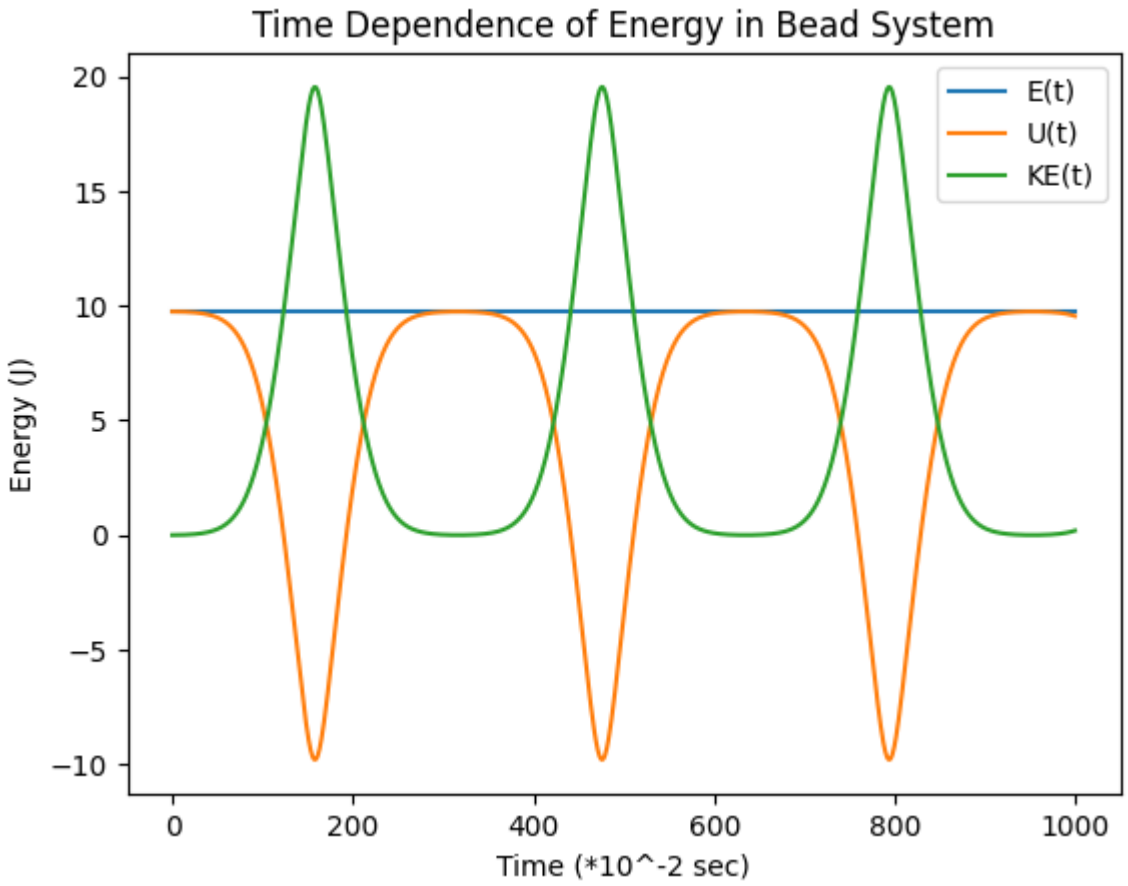
```
Shape of trajectory:  (4, 1000)
```

Out[27]:
```
<matplotlib.legend.Legend at 0x1d21b389000>
```



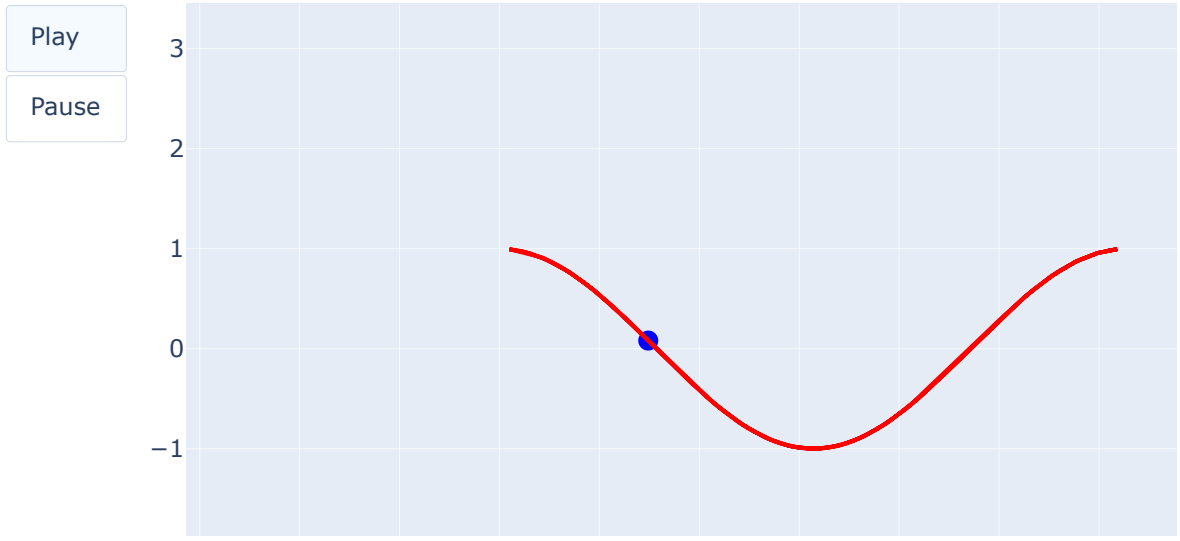
In [28]:
```
#check on the energy in the system at any given time
def U3(s):
    [_, y, _, _] = s
    m = 1
    g = 9.8
    return m*g*y

def KE3(s):
    [x, y, xd, yd] = s
    m = 1
    g = 9.8
    return 0.5 * m * (xd**2 + yd**2)

U_array2 = [U3(s) for s in x_array.T]
KE_array2 = [KE3(s) for s in x_array.T]
E_array2 = [U3(s) + KE3(s) for s in x_array.T]

plt.figure()
plt.plot(E_array2)
plt.plot(U_array2)
plt.plot(KE_array2)

plt.xlabel("Time (*10^-2 sec)")
plt.ylabel("Energy (J)")
plt.title("Time Dependence of Energy in Bead System")
plt.legend(["E(t)","U(t)","KE(t)"])
```

Out[28]:
```
<matplotlib.legend.Legend at 0x1d22e5218a0>
```



In [29]:
```
# second, animate!
print(f"Shape of array: {x_array.shape}")
animate_bead(x_array,T=10)
```

```
Shape of array: (4, 1000)
```



In [ ]: