+ Code + Text

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 Jupyter 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.

People who worked with on this homework

- Stellar Yu
- Ishani Narwankar
- Kyle Wang
- Jihai Zhao

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

# print sympy version for testing, should be 1.6.2
import math
import sympy as sym
import numpy as np
print(sym.__version__)

from IPython.display import display, Markdown
import matplotlib.pyplot as plt
```

```
1.12
```

[2]
```
################################################################################
# If you're using Google Colab, uncomment this section by selecting the whole section and press
# ctrl+'/' on your and keyboard. Run it before you start programming, this will enable the nice
# LaTeX "display()" function for you. If you're using the local Jupyter environment, leave it alone
################################################################################
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-AMS_HTML"
    javascript(url=url)
    return sym.printing.latex(exp,**options)
sym.init_printing(use_latex="mathjax",latex_printer=custom_latex_printer)
```

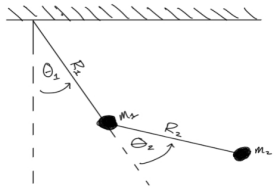
Below are the help functions in previous homeworks, which you may need for this homework.

[3]
```
def integrate(f, xt, dt):
    """
    This function takes in an initial condition x(t) and a timestep dt,
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x(t). It outputs a vector x(t+dt) at the future
    time step.

    Parameters
    ============
    dyn: Python function
        derivate of the system at a given step x(t),
        it can considered as \dot{x}(t) = func(x(t))
    xt: NumPy array
        current step x(t)
    dt:
        step size for integration

    Return
    ============
    new_xt:
        value of x(t+dt) integrated from x(t)
    """
    k1 = dt * f(xt)
    k2 = dt * f(xt+k1/2.)
    k3 = dt * f(xt+k2/2.)
    k4 = dt * f(xt+k3)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

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):
        xtraj[:,i]=integrate(f,x,dt)
        x = np.copy(xtraj[:,i])
    return xtraj
```

Problem 1 (15pts)

Double-click (or enter) to edit

[4]
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/master/dyndoublepend.png' width=500' height='350'></table>"))
```



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.

```
[5] # You can start your implementation here :)
    m1, m2, R1, R2, g = sym.symbols(r'm_1, m_2, R_1, R_2, g')
    t = sym.symbols(r't')
    theta1 = sym.Function(r'\theta_1')(t)
    theta2 = sym.Function(r'\theta_2')(t)

    theta1dot = theta1.diff(t)
    theta2dot = theta2.diff(t)

    p01_vec    = sym.Matrix([R1*sym.sin(theta1), -R1*sym.cos(theta1), 0])
    p12_vec    = sym.Matrix([R2*sym.sin(theta1+theta2), -R2*sym.cos(theta1+theta2), 0])


    display(p01_vec.diff(t))
    display((p01_vec+p12_vec).diff(t))

    display(p01_vec)
    display(p01_vec+p12_vec)

    omega1_vec = sym.Matrix([0, 0, theta1dot])
    omega2_vec = sym.Matrix([0, 0, theta1dot+theta2dot])

    v1_vec = omega1_vec.cross(p01_vec)
    v2_vec = v1_vec + omega2_vec.cross(p12_vec)

    display(Markdown(r'**The linear velocity of the mass 1 $\vec{v}_1$ is:**'))
    display(v1_vec)
    display(Markdown(r'**The linear velocity of the mass 2 $\vec{v}_2$ is:**'))
    display(v2_vec)

    KE_1 = 1/2*m1*v1_vec.dot(v1_vec)
    KE_2 = 1/2*m2*v2_vec.dot(v2_vec)
    KE   = KE_1 + KE_2
    display(Markdown(r'**The Kinetic Energy term is:**'))
    display(KE)

    V_1 = -m1*g*R1*sym.cos(theta1)
    V_2 = -m2*g*(R1*sym.cos(theta1) + R2*sym.cos(theta1+theta2))
    V   = V_1 + V_2
    display(Markdown(r'**The Potential Energy term is:**'))
    display(V)

    L = KE - V
    display(Markdown('**The Lagrangian $L$ of the system is:**'))
    display(sym.Eq(sym.symbols(r'L'), sym.simplify(L)))
```

$$\begin{bmatrix} R_1 \cos(\theta_1(t)) \frac{d}{dt}\theta_1(t) \\ R_1 \sin(\theta_1(t)) \frac{d}{dt}\theta_1(t) \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} R_1 \cos(\theta_1(t)) \frac{d}{dt}\theta_1(t) + R_2 \left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \cos(\theta_1(t) + \theta_2(t)) \\ R_1 \sin(\theta_1(t)) \frac{d}{dt}\theta_1(t) + R_2 \left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \sin(\theta_1(t) + \theta_2(t)) \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} R_1 \sin(\theta_1(t)) \\ -R_1 \cos(\theta_1(t)) \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} R_1 \sin(\theta_1(t)) + R_2 \sin(\theta_1(t) + \theta_2(t)) \\ -R_1 \cos(\theta_1(t)) - R_2 \cos(\theta_1(t) + \theta_2(t)) \\ 0 \end{bmatrix}$$

The linear velocity of the mass 1 $\vec{v}_1$ is:

$$\begin{bmatrix} R_1 \cos(\theta_1(t)) \frac{d}{dt}\theta_1(t) \\ R_1 \sin(\theta_1(t)) \frac{d}{dt}\theta_1(t) \\ 0 \end{bmatrix}$$

The linear velocity of the mass 2 $\vec{v}_2$ is:

$$\begin{bmatrix} R_1 \cos(\theta_1(t)) \frac{d}{dt}\theta_1(t) + R_2 \left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \cos(\theta_1(t) + \theta_2(t)) \\ R_1 \sin(\theta_1(t)) \frac{d}{dt}\theta_1(t) + R_2 \left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \sin(\theta_1(t) + \theta_2(t)) \\ 0 \end{bmatrix}$$

The Kinetic Energy term is:

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

The Potential Energy term is:

$$-R_1 g m_1 \cos(\theta_1(t)) - g m_2 \left(R_1 \cos(\theta_1(t)) + R_2 \cos(\theta_1(t) + \theta_2(t))\right)$$

The Lagrangian L of the system is:

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

Problem 2 (15pts)

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 $\ddot{\theta}_2$).

```
[6] # You can start your implementation here :)
    q     = sym.Matrix([theta1, theta2])
    qdot  = q.diff(t)
    qddot = qdot.diff(t)

    display(Markdown(r'**The configuration $\vec{q}$ is:**'))
    display(q)

    L_mat = sym.Matrix([L])
    dLdq = L_mat.jacobian(q).T
    display(Markdown(r'**<br/>The derivate of $\frac{\partial L}{\partial \vec{q}}$ is:**'))
    display(sym.simplify(dLdq))

    dLdqdot = L_mat.jacobian(qdot).T
    ddLdqdotdt = dLdqdot.diff(t)
    display(Markdown(r'**<br/>The derivate of $\frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\vec{q}}}\right)$ is:**'))
    display(sym.simplify(ddLdqdotdt))

    EL = dLdq - ddLdqdotdt
    EL_eqn = sym.Eq(sym.simplify(EL), sym.Matrix([0, 0]))
    display(Markdown(r'**<br/>The Euler-Lagrange Equation is:**'))
    display(EL_eqn)
    sols = sym.solve(EL_eqn, qddot, dict=True)
    # display(sols)

    display(Markdown(r'**<br/>The symbolic solutions are:**'))
    for sol in sols:
      for v in qddot:
        display(sym.Eq(v, sym.simplify(sol[v])))
```

The configuration $\vec{q}$ is:

$$\begin{bmatrix} \theta_1(t) \\ \theta_2(t) \end{bmatrix}$$

The derivate of $\frac{\partial L}{\partial \vec{q}}$ is:

$$\begin{bmatrix} -g\left(R_1 m_1 \sin(\theta_1(t)) + R_1 m_2 \sin(\theta_1(t)) + R_2 m_2 \sin(\theta_1(t) + \theta_2(t))\right) \\ R_2 m_2 \left(-R_1 \left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \sin(\theta_2(t)) \frac{d}{dt}\theta_1(t) - g \sin(\theta_1(t) + \theta_2(t))\right) \end{bmatrix}$$

The derivate of $\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right)$ is:

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

The Euler-Lagrange Equation is:

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

The symbolic solutions are:

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

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

$$R_1 R_2 \cdot (2m_1 - m_2 \cos(2\theta_1(t)) + m_2)$$

Problem 3 (15pts)

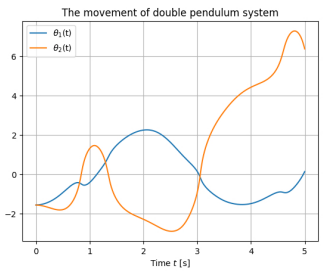
Numerically evaluate your solutions for $\ddot{\theta}_1$ and $\ddot{\theta}_2$ from Problem 2 using SymPy's `lambdify()` method, simulate the system for $t \in [0, 5]$ with $m_1 = 1, m_2 = 2, R_1 = 2, R_2 = 1$ and initial condition as $\theta_1 = \theta_2 = -\frac{\pi}{2}, \dot{\theta}_1 = \dot{\theta}_2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

[7]
```
# You can start your implementation here :)
funcs = []
for sol in sols:
  for v in qddot:
    funcs.append(sym.lambdify([m1, m2, R1, R2, g, theta1, theta2, theta1dot, theta2dot], sol[v]))

def integrate(f, xt, dt):
    """
    This function takes in an initial condition x(t) and a timestep dt,
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x(t). It outputs a vector x(t+dt) at the future
    time step.

    Parameters
    ============
    dyn: Python function
        derivate of the system at a given step x(t),
        it can considered as \dot{x}(t) = func(x(t))
    xt: NumPy array
        current step x(t)
    dt:
        step size for integration

    Return
    ============
    new_xt:
        value of x(t+dt) integrated from x(t)
    """
    k1 = dt * f(xt)
    k2 = dt * f(xt+k1/2.)
    k3 = dt * f(xt+k2/2.)
    k4 = dt * f(xt+k3)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

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):
        xtraj[:,i]=integrate(f,x,dt)
        x = np.copy(xtraj[:,i])
    return xtraj

val_m1 = 1
val_m2 = 2
val_R1 = 2
val_R2 = 1
val_g  = 9.8

val_theta1    = -math.pi/2
val_theta2    = -math.pi/2
val_theta1dot = 0
val_theta2dot = 0

val_theta1ddot = funcs[0](val_m1, val_m2, val_R1, val_R2, val_g, val_theta1, val_theta2, val_theta1dot, val_theta2dot)
val_theta2ddot = funcs[1](val_m1, val_m2, val_R1, val_R2, val_g, val_theta1, val_theta2, val_theta1dot, val_theta2dot)

display(Markdown (r'**The numberic solution is $\ddot{\theta}_1$ = ' + f'{val_theta1ddot}' + r', $\ddot{\theta}_2$ = ' + f'{val_theta2ddot}**'))

# print(val_theta1ddot, val_theta2ddot)

def theta1ddot(theta1, theta2, theta1dot, theta2dot):
  return funcs[0](val_m1, val_m2, val_R1, val_R2, val_g, theta1, theta2, theta1dot, theta2dot)

def theta2ddot(theta1, theta2, theta1dot, theta2dot):
  return funcs[1](val_m1, val_m2, val_R1, val_R2, val_g, theta1, theta2, theta1dot, theta2dot)

def dyn(s):
  theta1 = s[0]
  theta2 = s[1]
  theta1dot = s[2]
  theta2dot = s[3]

  return np.array([theta1dot, theta2dot, theta1ddot(theta1, theta2, theta1dot, theta2dot), theta2ddot(theta1, theta2, theta1dot, theta2dot)])

s0 = np.array([-math.pi/2, -math.pi/2, 0, 0])
traj = simulate(dyn, s0, [0, 5], 0.01, integrate)

t = np.linspace(0, 5, 500)
# print(traj.shape)
plt.plot(t, traj[0], label=r'$\theta_1$(t)')
plt.plot(t, traj[1], label=r'$\theta_2$(t)')
plt.legend()
plt.title("The movement of double pendulum system")
plt.xlabel('Time $t$ [s]')
plt.grid()
plt.show()
```

The numberic solution is $\ddot{\theta}_1$ = 4.9, $\ddot{\theta}_2$ = -4.9



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.

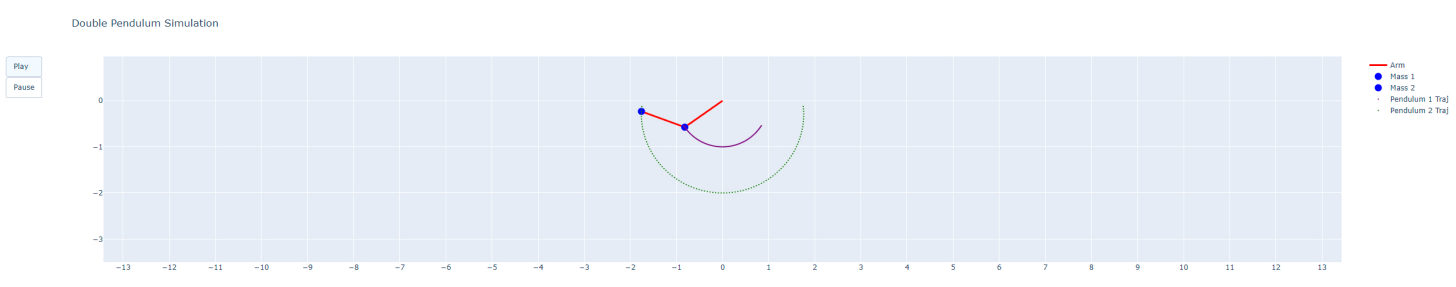
> *Hint 1: If your animation seems to be slow, press "pause" and then press "play" again! This should play animation at normal speed.*

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.

[8]
```
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 = "x",dtick=1),
                title='Double Pendulum Simulation',
                hovermode='closest',
                updatemenus= [{'type': 'buttons',
                               'buttons': [{'label': 'Play','method': 'animate',
                                            'args': [None, {'frame': {'duration': T, 'redraw': False}}]},
                                           {'args': [[None], {'frame': {'duration': T, 'redraw': False}, 'mode': 'immediate',
                                            'transition': {'duration': 0}}],'label': 'Pause','method': 'animate'}
                                           ]
                              }]
                )

    ######################################
    # 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=5)
```

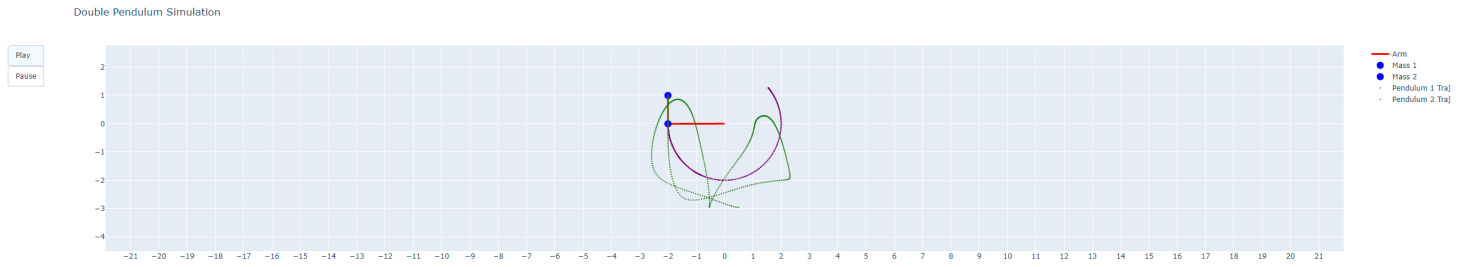
```
shape of trajectory:  (2, 100)
```



[9]
```
# You can start your implementation here
sim_traj = np.array([traj[0], traj[1]])
print('shape of trajectory: ', sim_traj.shape)
```

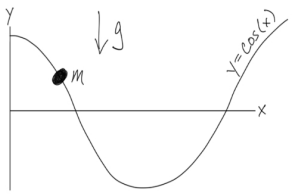
```
# second, animate!
animate_double_pend(sim_traj,L1=2,L2=1,T=5)
```

```
shape of trajectory:  (2, 500)
```



Problem 5 (15pts)

```
[10] from IPython.core.display import HTML
     display(HTML("<table><tr><td><img src='https://github.com/HuchenSun/ME314pngs/raw/master/dynbeadwire.png' width=500' height='350'></table>"))
```



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.

```
[11] from sympy.integrals.transforms import LaplaceTransform
     # You can start your implementation here
     m, g = sym.symbols(r'm, g')
     lam  = sym.symbols(r'\lambda')
     t    = sym.symbols('t')
     x    = sym.Function('x')(t)
     y    = sym.Function('y')(t)
     xdot = x.diff(t)
     ydot = y.diff(t)

     KE = 1/2*m*(xdot**2 + ydot**2)
     V  = m*g*y

     L  = KE - V
     display(Markdown(r'**The Lagrangian of the system is:**'))
     display(sym.Eq(sym.symbols('L'), L))

     q    = sym.Matrix([x, y])
     qdot = q.diff(t)
     display(Markdown(r'**The configuration $\vec{q}$ of the system is**'))
     display(q)

     L_mat = sym.Matrix([L])
     dLdq      = L_mat.jacobian(q).T
     dLdqdot   = L_mat.jacobian(qdot).T
     dLdqdotdt = dLdqdot.diff(t)

     display(Markdown(r'**The derivative $\frac{\partial L}{\partial \vec{q}}$ is:**'))
     display(dLdq)
     display(Markdown(r'**The derivative $\frac{\partial L}{\partial \dot{\vec{q}}}$ is:**'))
     display(dLdqdot)
     display(Markdown(r'**The derivative $\frac{d}{dt}\left (\frac{\partial L}{\partial \dot{\vec{q}}} \right)$ is:**'))
     display(dLdqdotdt)

     display(Markdown(r'**Left hand side of Euler-Lagrange Equation is:**'))
     EL = dLdq - dLdqdotdt
     display(EL)

     Phi = y - sym.cos(x)
     display(Markdown(r'**Constraint $\Phi\left(\vec{q} \right)$ is:**'))
     display(Phi)

     Phi_mat  = sym.Matrix([Phi])
     Phi_grad = Phi_mat.jacobian(q).T
     display(Markdown(r'**The gradient of the constraint $\nabla \Phi(\vec{q})$ is:**'))
     display(Phi_grad)

     cons = lam*Phi_grad
     display(Markdown(r'**Right-hand-side of the Euler-Langgrange Equation $\lambda \nabla \Phi(\vec{q})$ is:**'))
     display(cons)


     EL_eqn = sym.Eq(EL, cons)

     display(Markdown(r'**The constrained Euler-Lagrange Equation is:**'))
     display(EL_eqn)

     Phiddot  = Phi.diff(t).diff(t)
     cons_eqn = sym.Eq(Phiddot, 0)
     display(cons_eqn)
```

The Lagrangian of the system is:

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

The configuration $\vec{q}$ of the system is

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$$

The derivative $\frac{\partial L}{\partial \vec{q}}$ is:

$$\begin{bmatrix} 0 \\ -gm \end{bmatrix}$$

The derivative $\frac{\partial L}{\partial \dot{\vec{q}}}$ is:

$$\begin{bmatrix} 1.0m\frac{d}{dt}x(t) \\ 1.0m\frac{d}{dt}y(t) \end{bmatrix}$$

The derivative $\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\vec{q}}}\right)$ is:

$$\begin{bmatrix} 1.0m\frac{d^2}{dt^2}x(t) \\ 1.0m\frac{d^2}{dt^2}y(t) \end{bmatrix}$$

Left hand side of Euler-Lagrange Equation is:

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

Constraint $\Phi\left(\vec{q}\right)$ is:

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

The gradient of the constraint $\nabla\Phi(\vec{q})$ is:

$$\begin{bmatrix} \sin\left(x(t)\right) \\ 1 \end{bmatrix}$$

Right-hand-side of the Euler-Langgrange Equation $\lambda\nabla\Phi(\vec{q})$ is:

$$\begin{bmatrix} \lambda\sin\left(x(t)\right) \\ \lambda \end{bmatrix}$$

The constrained Euler-Lagrange Equation is:

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

$$\sin\left(x(t)\right)\frac{d^2}{dt^2}x(t) + \cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2 + \frac{d^2}{dt^2}y(t) = 0$$

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.

```
[12] # You can start your implementation here
     eqn_lhs = sym.Matrix([El, Phiddot])
     eqn_rhs = sym.Matrix([cons, sym.Matrix([0])])
     eqn = sym.Eq(eqn_lhs, eqn_rhs)
     display(Markdown(r'**The EOM of the system is**'))
     display(eqn)
     qddot = qdot.diff(t)

     vars = sym.Matrix([qddot, lam])
     sols = sym.solve(eqn, vars, dict=True)

     funcs = []

     display(Markdown(r'**The symbolic solution of the system is**'))
     for sol in sols:
       for v in vars:
         display(sym.Eq(v, sym.simplify(sol[v])))
         funcs.append(sym.lambdify([m, g, x, y, xdot, ydot], sol[v]))
```

The EOM of the system is

$$\begin{bmatrix} -1.0m\frac{d^2}{dt^2}x(t) \\ -gm - 1.0m\frac{d^2}{dt^2}y(t) \\ \sin(x(t))\frac{d^2}{dt^2}x(t) + \cos(x(t))\left(\frac{d}{dt}x(t)\right)^2 + \frac{d^2}{dt^2}y(t) \end{bmatrix} = \begin{bmatrix} \lambda\sin(x(t)) \\ \lambda \\ 0 \end{bmatrix}$$

The symbolic solution of the system is

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

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

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

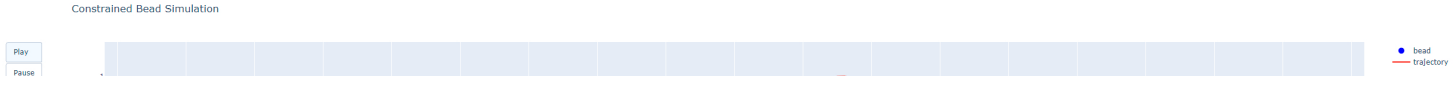
Problem 7 (20pts)

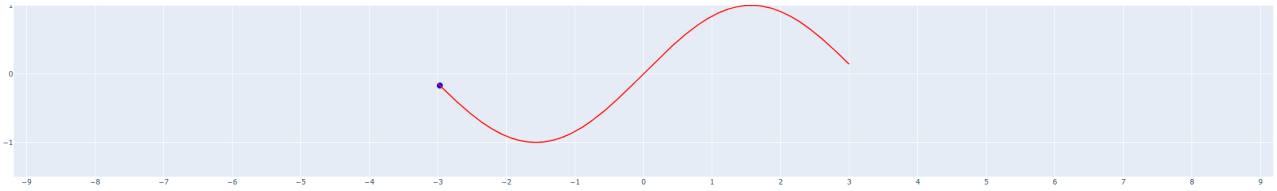
Simulate this constrained bead system with $m = 1$ and initial condition as $x = 0.1, y = \cos(0.1), \dot{x} = \dot{y} = 0$, for $t \in [0, 10]$. Animate your simulated trajectory using the provided function below.

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.

```
[13] 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 = "x",dtick=1),
                   title='Constrained Bead Simulation',
                   hovermode='closest',
                   updatemenus= [{'type': 'buttons',
                                  'buttons': [{'label': 'Play','method': 'animate',
                                               'args': [None, {'frame': {'duration': T, 'redraw': False}}]},
                                              {'args': [[None], {'frame': {'duration': T, 'redraw': False}, 'mode': 'immediate',
                                               'transition': {'duration': 0}}],'label': 'Pause','method': 'animate'}
                                             ]
                                 }]
                  )

       ########################################
       # 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)
```

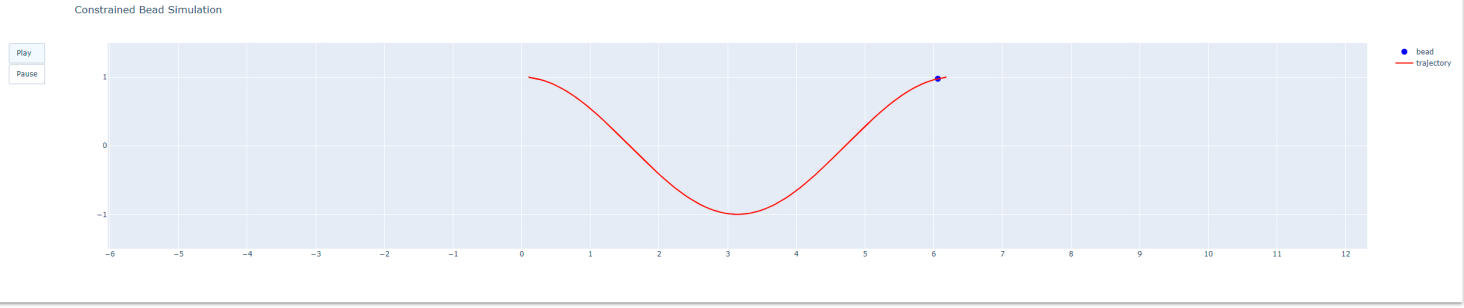
```
Shape of trajectory:  (2, 600)
```





```
# You can start your implementation here
val_m = 1
val_g = 9.8

def xddot(x, y, xdot, ydot):
  return funcs[0](val_m, val_g, x, y, xdot, ydot)

def yddot(x, y, xdot, ydot):
  return funcs[1](val_m, val_g, x, y, xdot, ydot)

def dyn(s):
  x    = s[0]
  y    = s[1]
  xdot = s[2]
  ydot = s[3]

  return np.array([xdot, ydot, xddot(x, y, xdot, ydot), yddot(x, y, xdot, ydot)])
s0 = np.array([0.1, np.cos(0.1), 0, 0])
traj = simulate(dyn, s0, [0, 10], 0.01, integrate)

sim_traj = np.array([traj[0], traj[1]])
print('Shape of trajectory: ', sim_traj.shape)

# second, animate!
animate_bead(sim_traj,T=10)
```

```
Shape of trajectory:  (2, 1000)
```



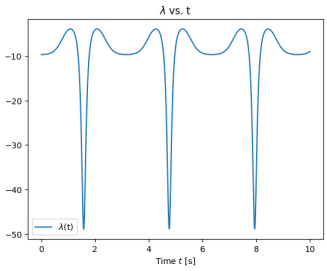
Problem 8 (5pts)

Plot the magnitude of λ vs. time for the constrained bead system. Explain why the magnitude of λ changes over time the way it does (for example, what do peaks correspond to?).

Hint: Recall that $\lambda \nabla \phi(q)$ physically represents the constraint force, that is, the force required to keep the bead on the cosine curve.

Turn in: Plot and explanation of the magnitude of λ over time.

```
[15] # You can start your implementation here
     lam_lst = []

     for i in range(traj.shape[1]):
       val_x    = traj[0][i]
       val_y    = traj[1][i]
       val_xdot = traj[2][i]
       val_ydot = traj[3][i]

       val_lam = funcs[2](val_m, val_g, val_x, val_y, val_xdot, val_ydot)
       lam_lst.append(val_lam)

     plt.plot(np.linspace(0, 10, 1000), lam_lst, label='$\lambda$(t)')
     plt.title('$\lambda$ vs. t')
     plt.xlabel('Time $t$ [s]')
     plt.legend()
     plt.show()
```



The physical meaning of the $\lambda(t)$ is the magnitude of external force required to for the system to satisify the constraints. As the ball moved along the curve, due to the space of the different in curvature of the path and the speed of the ball, the forces required changes. The peaks corresponding to the ball at the bottom of the cosine curve, that requires the most amount of the force to remain on the curve.