ME314 Homework 5 (Template)

*Please note that a **single** PDF file will be the only document that you turn in, which will include your answers to the problems with corresponding derivations and any code used to complete the problems. When including the code, please make sure you also include **code outputs**, and you don't need to include example code. Problems and deliverables that should be included with your submission are shown in **bold**.*

This Juputer Notebook file serves as a template for you to start homework, since we recommend to finish the homework using Jupyter Notebook. You can start with this notebook file with your local Jupyter environment, or upload it to Google Colab. You can include all the code and other deliverables in this notebook Jupyter Notebook supports LaTeX for math equations, and you can export the whole notebook as a PDF file. But this is not the only option, if you are more comfortable with other ways, feel free to do so, as long as you can submit the homework in a single PDF file.

In [153…

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

# import sympy as sym
# 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)
```

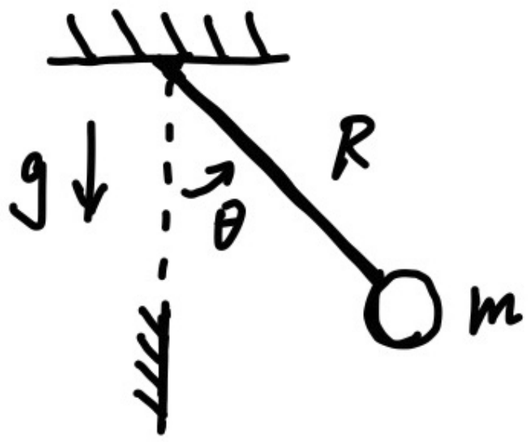
Below are the help functions in previous homeworks, which you may need for this homework.

In [154…

```python
import numpy as np

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
```

In [155…

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



Problem 1 (5pts)

Consider the single pendulum showed above, solve the Euler-Lagrange equations and simulate the system for $t \in [0, 5]$ with $dt = 0.01, R = 1, m = 1, g = 9.8$ and initial condition as $\theta = \frac{\pi}{2}, \dot{\theta} = 0$, and plot θ versus time. Note that in this problem there is no impact involved (ignore the wall at the bottom).

Turn in: A copy of the code used to solve the El-equations and numerically simulate the system. Also include code output, which should be the plot of the trajectory versus time.

In [156…

```
import sympy as sym
from sympy import symbols, Matrix, Eq, Function, sin, cos, solve, simplify
from sympy.abc import t

th = Function(r'\theta')(t)
m = symbols('m')
r = symbols('R')
g = symbols('g')
l = symbols('L')

x = r*sin(th)
y = -r*cos(th)

q = Matrix([th])
qdot = q.diff(t)
qddot = qdot.diff(t)

xdot = x.diff(t)
ydot = y.diff(t)

ke = m*(xdot**2 + ydot**2) / 2
pe = m * g * y

# ke = ke.subs({m:1, r:1, g:9.8})
# pe = pe.subs({m:1, r:1, g:9.8})


L = ke - pe
display(Eq(l, L))
L_mat = Matrix([L])


dLdq = L_mat.jacobian(q).T
dLdqdot = L_mat.jacobian(qdot).T
d_dLdqdot_dt = dLdqdot.diff(t)

el = Eq(sym.simplify(dLdq - d_dLdqdot_dt), Matrix([0]))

# print('================================================')
# print()
# print("dLdq")
# display(dLdq)

# print('================================================')
# print()
# print("dLdqdot")
# display(dLdqdot)

print('================================================')
print()
print("Euler Lagrange equations for single pendulum system")
display(el)
print()
print()

# solve for the equations of motion:
el_soln = solve(el, qddot)

for i in qddot:
    print('================================================')
    print()
    display(sym.simplify(Eq(i, el_soln[i])))
print()
print()
```

$$L = Rgm\cos(\theta(t)) + \frac{m\left(R^2\sin^2(\theta(t))\left(\frac{d}{dt}\theta(t)\right)^2 + R^2\cos^2(\theta(t))\left(\frac{d}{dt}\theta(t)\right)^2\right)}{2}$$

==========

Euler Lagrange equations for single pendulum system

$$\left[-Rm\left(R\frac{d^2}{dt^2}\theta(t) + g\sin(\theta(t))\right)\right] = \left[0\right]$$

==========

$$\frac{d^2}{dt^2}\theta(t) = -\frac{g\sin(\theta(t))}{R}$$

In [157…

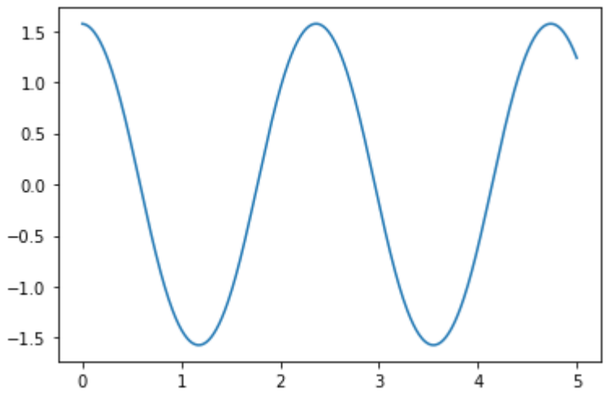
```
from math import pi
from numpy import arange
import numpy as np
import matplotlib.pyplot as plt


el = el.subs({m:1, r:1, g:9.8})

# solve for the equations of motion:
el_soln = solve(el, qddot)

thddot_func = sym.lambdify([th, qdot[0],], el_soln[qddot[0]], modules = sym)


##############################################

def theta_ddot(th, th_dot):
    return thddot_func(th, th_dot)

def dyn(s):
    """
    System dynamics function (extended)

    Parameters
    ============
    s: NumPy array
        s = [theta1, theta2, theta3, theta1dot, theta2dot, theta3dot]

    Return
    ============
    sdot: NumPy array
        time derivative of input state vector,
        sdot = [theta1dot, theta2dot, theta3dot, theta1_ddot, theta2_ddot, theta3_ddot]
    """
    return np.array([s[1], theta_ddot(s[0], s[1])])

# define initial state
s0 = np.array([pi/2, 0])
traj = simulate(dyn, s0, [0,5], 0.01, integrate)
print()
print('shape of traj: ', traj.shape)

print()
print()
```

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

In [158…

```
#plot the theta trajectories
num_pts = int(5/.01)
axis = np.linspace(0,5, num_pts )

print('theta trajectory')
plt.plot(axis, traj[0])
plt.show()

print()
print()
```

```
theta trajectory
```



Problem 2 (10pts)

Now, time for impact! As shown in the figure, there is wall such that the pendulum will hit the wall when $\theta = 0$. Recall that in the course note, to solve the impact update rule, we have two set of equations:

$$\frac{\partial L}{\partial \dot{q}}\Big|_{\tau^-}^{\tau^+} = \lambda \frac{\partial \phi}{\partial q}$$

$$\left[\frac{\partial L}{\partial \dot{q}} \cdot \dot{q} - L(q, \dot{q})\right]\Big|_{\tau^-}^{\tau^+} = 0$$

In this problem you will need to symbolically compute the following three expressions contained the equations above:

$$\frac{\partial L}{\partial \dot{q}}, \quad \frac{\partial \phi}{\partial q}, \quad \frac{\partial L}{\partial \dot{q}} \cdot \dot{q} - L(q, \dot{q})$$

(the third one is the Hamiltonian of the system).

There is one more requirement: all three expressions can be considered as functions of q and $\dot{q}$, you may previously define q and $\dot{q}$ as SymPy's function objects, now you will need to substitute them with dummy symbols (using SymPy's subsititute method). Note that, q and $\dot{q}$ should be two sets of separate symbols.

Turn in: A copy of code used to symbolically compute the three expressions, also include the outputs of your code, which should be the three expressions (make sure there is no SymPy Function(t) left!).

In [ ]:

In [159…

```
#define dummy variables for q and qdot
th_dummy, thdot_dummy = symbols(r'\theta, \dot{\theta}')

q_dummy = {q[0]:th_dummy, qdot[0]:thdot_dummy}

dLdqdot = L_mat.jacobian(qdot).T
p = dLdqdot.subs(q_dummy)
print("dL/dq_dot term:")
display(simplify(p[0]))
print('================================================')
print()

phi = q[0] - pi/2
gradphi = sym.simplify(sym.Matrix([phi]).jacobian(q).T)
gradphi = gradphi.subs(q_dummy)
print("Gradient of our surface along q, dphi_dq:")
display(gradphi)
print('================================================')
print()

Ham = dLdqdot*qdot-L_mat
Ham = Ham.subs(q_dummy)
print("Hamiltonian:")
display(simplify(Ham[0]))
print()
print()
```

dL/dq_dot term:

$R^2\dot{\theta}m$

==

Gradient of our surface along q, dphi_dq:

$[1]$

==

Hamiltonian:

$$\frac{Rm\left(R\dot{\theta}^2 - 2g\cos(\theta)\right)}{2}$$

In [ ]:

Problem 3 (10pts)

Now everything is ready to solve for the impact upate rules. Recall that for those equations to solve, you will need to evaluate them right before and after the impact time at τ^- and τ^+. Here $\dot{q}(\tau^-)$ are actually same as the dummy symbols you defined in Problem 2 (why?), but you will need to define new dummy symbols for $\dot{q}(\tau^+)$. That is to say, $\frac{\partial L}{\partial \dot{q}}$ and $\frac{\partial L}{\partial \dot{q}} \cdot \dot{q} - L(q, \dot{q})$ evaluated at τ^- are those you already had in Problem 2, but you will need to substitute the dummy symbols of $\dot{q}(\tau^+)$ to evaluate them at τ^+.

Based on the information above, define the equations for impact update and solve them for impact update rules. After solving the impact update solution, numerically evalute it as a function using SymPy's lambdify method and test it with $\theta(\tau^-) = 0.01, \dot{\theta}(\tau^-) = 2$. Note that:

1. In your equations and impact update solutions, there should be NO SymPy Function left (except for internal functions like $\sin$ or $\cos$).
2. You may wonder where are $q(\tau^-)$ and $q(\tau^+)$, the question is, do we really need new dummy variables for them?
3. The solution of the impact upate rules, which is obtained by solving the equations for the dummy variables corresponds to $\dot{q}(\tau^+)$ and λ, can be a function of $q(\tau^-)$ or a function of $q(\tau^-)$ and $\dot{q}(\tau^-)$ (q will not be updated during impact, but including it may help you to combine the function into simulation later).

Turn in: A copy of code used to symbolically solve for the impact update rules and evaluate them numerically. Also, include the outputs of your code, which should be the test output of your numerically evaluated impact update function.

In [160…

```
#need to re-compute the main 3 expressions from Problem 2 but now for tau- and tau+
#first start with defining the dummy variables and substituting

#dummy variables
th_minus, thdot_minus = symbols(r'\theta^-, \dot{\theta}^-')
th_plus, thdot_plus = symbols(r'\theta^+, \dot{\theta}^+')
lam = symbols('\lambda')
# display(th_plus, thdot_plus)


q_minus = {q[0]:th_minus, qdot[0]:thdot_minus}
q_plus = {q[0]:th_minus, qdot[0]:thdot_plus}

dLdqdot = L_mat.jacobian(qdot)
p_minus = dLdqdot.subs(q_minus)
p_plus = dLdqdot.subs(q_plus)
print("dL/dq_dot term for p_minus:")
display(simplify(p_minus[0]))
print()
print("dL/dq_dot term for p_plus:")
display(simplify(p_plus[0]))
print('================================================')
print()

phi = q[0] - pi/2
gradphi = sym.simplify(sym.Matrix([phi]).jacobian(q))
gradphi_minus = gradphi.subs(q_minus)
gradphi_plus = gradphi.subs(q_plus)
print("Gradient of our surface along q, dphi_dq for q_minus:")
display(gradphi_minus)
print()
print("Gradient of our surface along q, dphi_dq for q_plus:")
display(gradphi_plus)
print('================================================')
print()

Ham = dLdqdot*qdot-L_mat
Ham_minus = Ham.subs(q_minus)
Ham_plus = Ham.subs(q_plus)
print("Hamiltonian at q(tau-):")
display(simplify(Ham_minus[0]))
print()
print()
print("Hamiltonian at q(tau+):")
display(simplify(Ham_plus[0]))
print()
print()


#compute impact update equations (different in momentum and Hamiltonian terms):
lhs = Matrix([p_plus[0] - p_minus[0], Ham_plus[0] - Ham_minus[0]])
rhs = Matrix([lam * gradphi_minus[0], 0])
impact_eqns = Eq(simplify(lhs), simplify(rhs))

print('================================================')
print("Impact equations in simplified form:")
print()
display(impact_eqns)
print()
print()


#solve for qdot(tau+) and lambda
impact_solns = solve(impact_eqns, [thdot_plus, lam], dict=True)[0]
th_dot_soln = simplify(impact_solns[thdot_plus])
lam_soln = simplify(impact_solns[lam])

#display the thdot_plus and lambda solutions at tau+
```

```
print(" Solutions to impact update rules:  ")
print()
display(Eq(thdot_plus, th_dot_soln))
display(Eq(lam, lam_soln))
print()
print()
```

dL/dq_dot term for p_minus:

$R^2\dot{\theta}^- m$

dL/dq_dot term for p_plus:

$R^2\dot{\theta}^+ m$

==========

Gradient of our surface along q, dphi_dq for q_minus:

$\begin{bmatrix} 1 \end{bmatrix}$

Gradient of our surface along q, dphi_dq for q_plus:

$\begin{bmatrix} 1 \end{bmatrix}$

==========

Hamiltonian at q(tau-):

$$\frac{Rm\left(R\left(\dot{\theta}^-\right)^2 - 2g\cos\left(\theta^-\right)\right)}{2}$$

Hamiltonian at q(tau+):

$$\frac{Rm\left(R\left(\dot{\theta}^+\right)^2 - 2g\cos\left(\theta^-\right)\right)}{2}$$

==========

Impact equations in simplified form:

$$\begin{bmatrix} R^2 m\left(\dot{\theta}^+ - \dot{\theta}^-\right) \\ \frac{R^2 m\left(\left(\dot{\theta}^+\right)^2 - \left(\dot{\theta}^-\right)^2\right)}{2} \end{bmatrix} = \begin{bmatrix} \lambda \\ 0 \end{bmatrix}$$

==========

Solutions to impact update rules:

$\dot{\theta}^+ = -\dot{\theta}^-$

$\lambda = -2R^2\dot{\theta}^- m$

In [161…

```
#numerically solve for th_dot
thdot_update_func = sym.lambdify([th_minus, thdot_minus], th_dot_soln)

#define impact update function
def impact_update(s):
    return np.array([
        s[0],#q will be the same after impact
        thdot_update_func(*s),
    ])

#use given input values to test the impact update function
s = [0.01, 2] #[th, th_dot] = [0.01, 2]

print(f"With input value [th, th_dot] = [0.01, 2], ")
print(f"impact update evaluates to: {impact_update(s)}")

print()
print()
```

```
With input value [th, th_dot] = [0.01, 2],
impact update evaluates to: [ 0.01 -2.  ]
```

Problem 4 (20pts)

Finally, it's time to simulate the impact! In order to combine impact update rules into our previous simulate function, there two more steps:

1. Write a function called "impact_condition", which takes in $s = [q, \dot{q}]$ and returns **True** if s will cause an impact, otherwise the function will return **False**. Hint: you need to use the constraint ϕ in this problem, and note that, since we are doing numerical evaluation, the impact condition will not be perfect, you will need to catch the change of sign at $\phi(s)$ or setup a threshold to decide the condition.
2. Now, with the "impact_condition" function and the numerically evaluated impact update rule for $\dot{q}(\tau^+)$ solved in last problem, find a way to combine them into the previous simulation function, thus it can simulate the impact. Pseudo-code for the simulate function can be found in lecture note 13.

Simulate the system with same parameters and initial condition in Problem 1 for the single pendulum hitting the wall for five times. Plot the trajectory and animate the simulation (you need to modify the animation function by yourself).

Turn in: A copy of the code used to simulate the system. You don't need to include the animation function, but please include other code (impact_condition, simulate, ets.) used for simulating impact. Also, include the plot and a video for animation. The video can be uploaded separately through Canvas, and it should be in ".mp4" format. You can use screen capture or record the screen directly with your phone.

In [162…

```
#define impact function  to be phi = theta - 0
phi = q[0]
print("Function phi:")
display(phi)
phi_func = sym.lambdify([q[0],qdot[0]], phi, modules = sym)

def impact_condition(s, threshold=1e-1):
    phi_val = phi_func(*s)
    if phi_val > -threshold and phi_val < threshold:
        return True
    return False
#use test condition th = 0.01 and thdot = 5
s = [0.01, 5]

print('the impact condition will be at theta={}, theta_dot={}:{}'.format(*s,impact_condition(s, 1e-1)))

print()
print()
```

```
Function phi:
```

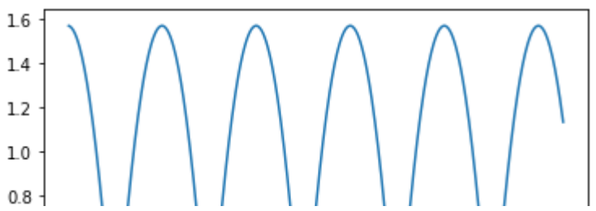
$\theta(t)$

```
the impact condition will be at theta=0.01, theta_dot=5:True
```

In [164…

```
#copy and modify simulate function from above

def simulate_impact(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):
        #first check if impact even happens. the xtraj calculation will change accordingly
        if impact_condition(x) is True:
            x = impact_update(x)
            xtraj[:,i]=integrate(f,x,dt)
        else:
            #if no impact, use original x input
            xtraj[:,i]=integrate(f,x,dt)
        x = np.copy(xtraj[:,i])
    return xtraj

s0 = np.array([pi/2, 0]) # initial values th = pi/2 , th_dot = 0
traj = simulate_impact(dyn, s0, tspan=[0,6], dt=0.01, integrate=integrate)

#plot it
t_space = np.linspace(0,6,600)
th_space = traj[0]

print('Plot of theta with 5 impacts: ')
print('plot will be reattached to bottom of pdf if cut off in this view.')
plt.plot(t_space, th_space)
plt.show()
```

```
Plot of theta with 5 impacts:
plot will be reattached to bottom of pdf if cut off in this view.
```


1.6
1.4
1.2
1.0
0.8

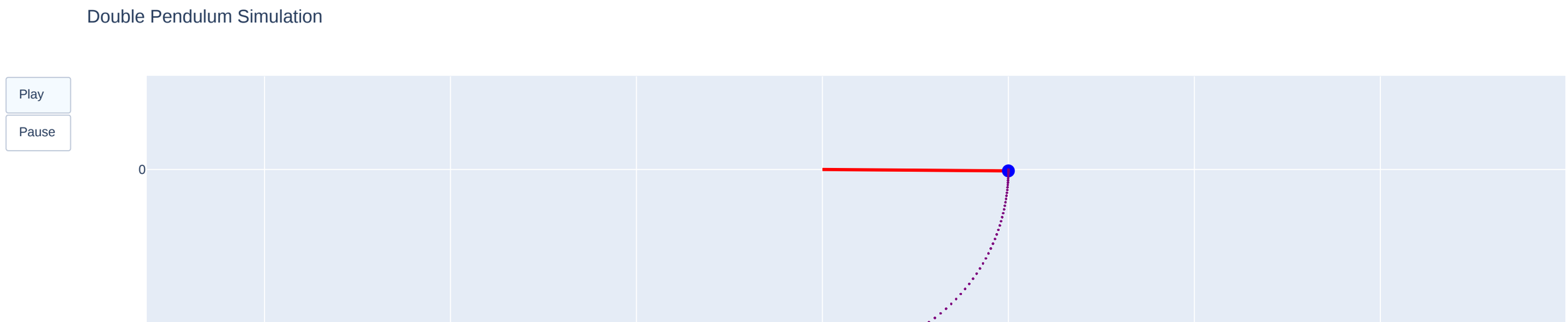
In [84]:

```
#get animation function from previous homeworks

#Animation
def animate_single_pend(theta_array,L1=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ================================================
    theta_array:
        trajectory of theta1, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the 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])
    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)-0.5
    yM=np.max(yy1)+0.5

    ############################
    # 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')
```

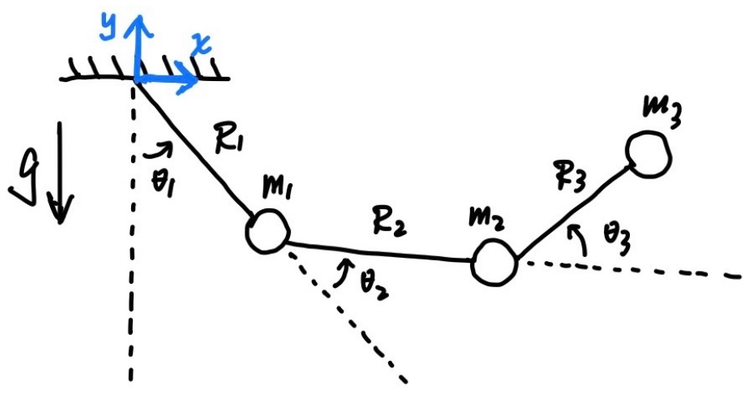
```
      dict(x=xx1, y=yy1,
           mode='markers', name='Pendulum 1 Traj',
           marker=dict(color="purple", size=2)
          ),
    ]

    ################################
    # 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]],
                            y=[0,yy1[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)),
                      ]) for k in range(N)]

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

In [85]:

```
animate_single_pend(traj[0:2],L1=1,T=10)
```



In [86]:
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/master/tripend_constrained.JPG' width=500' height='450'></table>"))
```



Problem 5 (10pts)

For the triple-pendulum system you simulated in last homework there is now a constraint, where the x coordinate of the third pendulum can not be smaller than 0 (but the system configuration is still $q = [\theta_1, \theta_2, \theta_3]$). Note that there is no constraint on the y coordinate of the third pendulum, which means this wall is infinitely high but only apply to the third pendulum. Similar to Problem 2, symbolically compute the following three expressions contained the equations above:

$$\frac{\partial L}{\partial \dot{q}}, \quad \frac{\partial \phi}{\partial q}, \quad \frac{\partial L}{\partial \dot{q}} \cdot \dot{q} - L(q, \dot{q})$$

Again, you will need to subsitute q and $\dot{q}$ with dummy symbols.

In this problem, we have $m_1 = m_2 = m_3 = 1$ and $R_1 = R_2 = R_3 = 1$, thus you don't need to define them as symbols.

Turn in: A copy of code used to symbolically compute the three expressions, also include the outputs of your code, which should be the three expressions (make sure there is no SymPy Function(t) left!).

In [109…

```
#copy over from homework 4 work

th1 = Function(r'theta_1')(t)
th2 = Function(r'theta_2')(t)
th3 = Function(r'theta_3')(t)
m1 = symbols('m_1')
m2 = symbols('m_2')
m3 = symbols('m_3')
r1 = symbols('R_1')
r2 = symbols('R_2')
r3 = symbols('R_3')
g = symbols('g')
l = symbols('L')

x1 = r1*sin(th1)
x2 = x1 + r2*sin(th1 + th2)
x3 = x2 + r3*sin(th1 + th2 + th3)

y1 = -r1*cos(th1)
y2 = y1 - r2*cos(th1 + th2)
y3 = y2 - r3*cos(th1 + th2 + th3)

q = Matrix([th1, th2, th3])
qdot = q.diff(t)
qddot = qdot.diff(t)

x1dot = x1.diff(t)
x2dot = x2.diff(t)
x3dot = x3.diff(t)

y1dot = y1.diff(t)
y2dot = y2.diff(t)
y3dot = y3.diff(t)
```

```
ke = 0.5*m1*(x1dot**2 + y1dot**2) + 0.5*m2*(x2dot**2 + y2dot**2) + 0.5*m3*(x3dot**2 + y3dot**2)
pe = m1*g*y1 + m2*g*y2 + m3*g*y3

ke = ke.subs({m1:1, m2:1, m3:1, r1:1, r2:1, r3:1, g:9.8})
pe = pe.subs({m1:1, m2:1, m3:1, r1:1, r2:1, r3:1, g:9.8})


L = ke - pe
# display(Eq(l, L))
L_mat = Matrix([L])
```

In [105…

```
from math import pi
from numpy import arange
import numpy as np
import matplotlib.pyplot as plt

dLdq = L_mat.jacobian(q).T
dLdqdot = L_mat.jacobian(qdot).T
d_dLdqdot_dt = dLdqdot.diff(t)

el = Eq(dLdq - d_dLdqdot_dt, Matrix([0, 0, 0]))

# print('================================================')
# print()
# print("dLdq")
# display(dLdq)

# print('================================================')
# print()
# print("dLdqdot")
# display(dLdqdot)

# print('================================================')
# print()
# print("Euler Lagrange equations for triple pendulum system")
# display(el)
```

In [106…

```
# solve for the equations of motion:
el_soln = solve(el, qddot)
```

In [107…

```
th1ddot_func = sym.lambdify([th1, th2, th3, qdot[0], qdot[1], qdot[2]], el_soln[qddot[0]], modules = sym)
th2ddot_func = sym.lambdify([th1, th2, th3, qdot[0], qdot[1], qdot[2]], el_soln[qddot[1]], modules = sym)
th3ddot_func = sym.lambdify([th1, th2, th3, qdot[0], qdot[1], qdot[2]], el_soln[qddot[2]], modules = sym)
```

In [110…

```
th1_dummy, th2_dummy, th3_dummy = symbols(r'\theta_1, \theta_2, \theta_3')
th1dot_dummy, th2dot_dummy, th3dot_dummy = symbols(r'\dot{\theta_1}, \dot{\theta_2}, \dot{\theta_3}, ')
subs_q = {q[0]:th1_dummy, q[1]:th2_dummy, q[2]:th3_dummy, qdot[0]:th1dot_dummy, qdot[1]:th2dot_dummy, qdot[2]:th3dot_dummy}


dLdqdot = L_mat.jacobian(qdot)
p = dLdqdot.subs(subs_q)
print("dL/dq_dot term:")
display(simplify(p))
print('================================================')
print()

#determine constraint equation phi for triple-pendulum system
phi = x1 + x2 + x3
gradphi = sym.simplify(sym.Matrix([phi]).jacobian(q))
gradphi = gradphi.subs(subs_q)
print("Gradient of our surface along q, dphi_dq:")
display(gradphi.subs({m1:1, m2:1, m3:1, r1:1, r2:1, r3:1, g:9.8}))
print('================================================')
print()

#compute Hamiltonian
Ham = dLdqdot*qdot - L_mat
Ham = Ham.subs(subs_q)
print("Hamiltonian:")
display(simplify(Ham[0]))
print()
print()
```

```
dL/dq_dot term:
```

$$\left[4.0\theta_1\cos\left(\theta_2\right) + 2.0\theta_1\cos\left(\theta_3\right) + 2.0\theta_1\cos\left(\theta_2+\theta_3\right) + 6.0\theta_1 + 2.0\theta_2\cos\left(\theta_2\right) + 2.0\theta_2\cos\left(\theta_3\right) + 1.0\theta_2\cos\left(\theta_2+\theta_3\right) + 3.0\theta_2 + 1.0\theta_3\cos\left(\theta_3\right) + 1.0\theta_3\cos\left(\theta_2+\theta_3\right) + 1.0\theta_3 \quad 2.0\theta_1\cos\left(\theta_2\right) + 2.0\theta_1\cos\left(\theta_3\right) + 1.0\theta_1\cos\left(\theta_2+\theta_3\right) + 3.0\theta_1 + 2.0\theta_2\cos\left(\theta_3\right) + 3.0\theta_2 + \right.$$

==========

Gradient of our surface along q, dphi_dq:

$$\left[3\cos\left(\theta_1\right) + 2\cos\left(\theta_1+\theta_2\right) + \cos\left(\theta_1+\theta_2+\theta_3\right) \quad 2\cos\left(\theta_1+\theta_2\right) + \cos\left(\theta_1+\theta_2+\theta_3\right) \quad \cos\left(\theta_1+\theta_2+\theta_3\right)\right]$$

==========

Hamiltonian:

$$2.0\dot{\theta}_1^2\cos\left(\theta_2\right) + 1.0\dot{\theta}_1^2\cos\left(\theta_3\right) + 1.0\dot{\theta}_1^2\cos\left(\theta_2+\theta_3\right) + 3.0\dot{\theta}_1^2 + 2.0\dot{\theta}_1\dot{\theta}_2\cos\left(\theta_2\right) + 2.0\dot{\theta}_1\dot{\theta}_2\cos\left(\theta_3\right) + 1.0\dot{\theta}_1\dot{\theta}_2\cos\left(\theta_2+\theta_3\right) + 3.0\dot{\theta}_1\dot{\theta}_2 + 1.0\dot{\theta}_1\dot{\theta}_3\cos\left(\theta_3\right) + 1.0\dot{\theta}_1\dot{\theta}_3\cos\left(\theta_2+\theta_3\right) + 1.0\dot{\theta}_1\dot{\theta}_3 + 1.0\dot{\theta}_2^2\cos\left(\theta_3\right) + 1.5\dot{\theta}_2^2 + 1.0\dot{\theta}_2\dot{\theta}_3\cos\left(\theta_3\right) + 1.0\dot{\theta}_2\dot{\theta}_3 + 0.5\dot{\theta}_3^2 - 29.$$

Problem 6 (10pts)

Similar to Problem 3, now you need to define dummy symbols for $\dot{q}(\tau^+)$, define the equations for impact update rules. Note that you don't need to solve the equations in this problem (in fact it's very time consuming to solve the analytical solution for this one, we will use a trick to get around with it later).

Turn in: A copy of the code used to define the equations for impact update, also print out the equations.

In [117…

```
#define dummy variables for tau+ and tau-
th1_minus, th2_minus, th3_minus = symbols(r'\theta_1^-, \theta_2^-, \theta_3^-')
th1dot_minus, th2dot_minus, th3dot_minus = symbols(r'\dot{\theta_1}^-, \dot{\theta_2}^-, \dot{\theta_3}^-')
th1dot_plus, th2dot_plus, th3dot_plus = symbols(r'\dot{\theta_1}^+, \dot{\theta_2}^+, \dot{\theta_3}^+')

#substitutions
subs_minus = {q[0]:th1_minus, q[1]:th2_minus, q[2]:th3_minus, qdot[0]:th1dot_minus, qdot[1]:th2dot_minus, qdot[2]:th3dot_minus}
subs_plus = {q[0]:th1_minus, q[1]:th2_minus, q[2]:th3_minus, qdot[0]:th1dot_plus, qdot[1]:th2dot_plus, qdot[2]:th3dot_plus}

dLdqdot = L_mat.jacobian(qdot)
p_minus = dLdqdot.subs(subs_minus)
print("p_minus term:")
display(simplify(p_minus.T))
print('================================================')
print()
p_plus = dLdqdot.subs(subs_plus)
print("p_minus term:")
display(simplify(p_plus.T))
print('================================================')
print()

gradphi = sym.simplify(sym.Matrix([phi]).jacobian(q))
gradphi = gradphi.subs(subs_minus)
print("Gradient of our surface along q, dphi_dq:")
display(gradphi.subs({m1:1, m2:1, m3:1, r1:1, r2:1, r3:1, g:9.8}).T)
print('================================================')
print()

#compute Hamiltonian
Ham = dLdqdot*qdot - L_mat
Ham_minus = Ham.subs(subs_minus)
print("Hamiltonian for tau-:")
display(simplify(Ham_minus[0]))
print('================================================')
print()
Ham_plus = Ham.subs(subs_plus)
print("Hamiltonian for tau+:")
display(simplify(Ham_plus[0]))
print()
print()
```

p_minus term:

$$\begin{bmatrix} 4.0\dot{\theta}_1^- \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^- \cos\left(\theta_3^-\right) + 2.0\dot{\theta}_1^- \cos\left(\theta_2^- + \theta_3^-\right) + 6.0\dot{\theta}_1^- + 2.0\dot{\theta}_2^- \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_2^- \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_2^- \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\dot{\theta}_2^- + 1.0\dot{\theta}_3^- \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_3^- \cos\left(\theta_2^- + \theta_3^-\right) + 1.0\dot{\theta}_3^- \\ 2.0\dot{\theta}_1^- \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^- \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^- \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\dot{\theta}_1^- + 2.0\dot{\theta}_2^- \cos\left(\theta_3^-\right) + 3.0\dot{\theta}_2^- + 1.0\dot{\theta}_3^- \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_3^- \\ 1.0\dot{\theta}_1^- \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^- \cos\left(\theta_2^- + \theta_3^-\right) + 1.0\dot{\theta}_1^- + 1.0\dot{\theta}_2^- \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_2^- + 1.0\dot{\theta}_3^- \end{bmatrix}$$

==========

p_minus term:

$$\begin{bmatrix} 4.0\dot{\theta}_1^+ \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^+ \cos\left(\theta_3^-\right) + 2.0\dot{\theta}_1^+ \cos\left(\theta_2^- + \theta_3^-\right) + 6.0\dot{\theta}_1^+ + 2.0\dot{\theta}_2^+ \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_2^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_2^+ \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\dot{\theta}_2^+ + 1.0\dot{\theta}_3^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_3^+ \cos\left(\theta_2^- + \theta_3^-\right) + 1.0\dot{\theta}_3^+ \\ 2.0\dot{\theta}_1^+ \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^+ \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\dot{\theta}_1^+ + 2.0\dot{\theta}_2^+ \cos\left(\theta_3^-\right) + 3.0\dot{\theta}_2^+ + 1.0\dot{\theta}_3^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_3^+ \\ 1.0\dot{\theta}_1^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^+ \cos\left(\theta_2^- + \theta_3^-\right) + 1.0\dot{\theta}_1^+ + 1.0\dot{\theta}_2^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_2^+ + 1.0\dot{\theta}_3^+ \end{bmatrix}$$

==========

Gradient of our surface along q, dphi_dq:

$$\begin{bmatrix} 3\cos\left(\theta_1^-\right) + 2\cos\left(\theta_1^- + \theta_2^-\right) + \cos\left(\theta_1^- + \theta_2^- + \theta_3^-\right) \\ 2\cos\left(\theta_1^- + \theta_2^-\right) + \cos\left(\theta_1^- + \theta_2^- + \theta_3^-\right) \\ \cos\left(\theta_1^- + \theta_2^- + \theta_3^-\right) \end{bmatrix}$$

==========

Hamiltonian for tau-:

$$2.0\left(\dot{\theta}_1^-\right)^2 \cos\left(\theta_2^-\right) + 1.0\left(\dot{\theta}_1^-\right)^2 \cos\left(\theta_3^-\right) + 1.0\left(\dot{\theta}_1^-\right)^2 \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\left(\dot{\theta}_1^-\right)^2 + 2.0\dot{\theta}_1^- \dot{\theta}_2^- \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^- \dot{\theta}_2^- \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^- \dot{\theta}_2^- \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\dot{\theta}_1^- \dot{\theta}_2^- + 1.0\dot{\theta}_1^- \dot{\theta}_3^- \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^- \dot{\theta}_3^- \cos\left(\theta_2^- + \theta_3^-\right) + 1.0\dot{\theta}_1^- \dot{\theta}_3^- + 1.0\left(\dot{\theta}_2^-\right)^2 \cos\left(\theta_3^-\right.$$

==========

Hamiltonian for tau+:

$$2.0\left(\dot{\theta}_1^+\right)^2 \cos\left(\theta_2^-\right) + 1.0\left(\dot{\theta}_1^+\right)^2 \cos\left(\theta_3^-\right) + 1.0\left(\dot{\theta}_1^+\right)^2 \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\left(\dot{\theta}_1^+\right)^2 + 2.0\dot{\theta}_1^+ \dot{\theta}_2^+ \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^+ \dot{\theta}_2^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^+ \dot{\theta}_2^+ \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\dot{\theta}_1^+ \dot{\theta}_2^+ + 1.0\dot{\theta}_1^+ \dot{\theta}_3^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^+ \dot{\theta}_3^+ \cos\left(\theta_2^- + \theta_3^-\right) + 1.0\dot{\theta}_1^+ \dot{\theta}_3^+ + 1.0\left(\dot{\theta}_2^+\right)^2 \cos\left(\theta_3^-\right.$$

In [120…

```
lam = symbols('\lambda')

#compute impact equations and impact update
lhs = Matrix([p_plus[0] - p_minus[0], p_plus[1] - p_minus[1], p_plus[2] - p_minus[2], Ham_plus - Ham_minus])
rhs = Matrix([lam * p_minus[0], lam * p_minus[1], lam * p_minus[2],0])
impact_eqns = sym.Eq(simplify(lhs), simplify(rhs))
print('equations for impact update:')
display(impact_eqns)
```

equations for impact update:

$$4.0\dot{\theta}_1^{+}\cos\left(\theta_2^{-}\right) + 2.0\dot{\theta}_1^{+}\cos\left(\theta_3^{-}\right) + 2.0\dot{\theta}_1^{+}\cos\left(\theta_2^{-} + \theta_3^{-}\right) + 6.0\dot{\theta}_1^{+} - 4.0\dot{\theta}_1^{-}\cos\left(\theta_2^{-}\right) - 2.0\dot{\theta}_1^{-}\cos\left(\theta_3^{-}\right) - 2.0\dot{\theta}_1^{-}\cos\left(\theta_2^{-} +$$

$$2.0\dot{\theta}_1^{+}\cos\left(\theta_2^{-}\right) + 2.0\dot{\theta}_1^{+}\cos\left(\theta_3^{-}\right) + 1.0\dot{\theta}_1^{+}\cos\left(\theta_2^{-} + \theta_3^{-}\right)$$

$$1.0\dot{\theta}_1^{+}\cos\left(\theta_3^{-}\right) + 1.0$$

Problem 7 (15pts)

Since solving the analytical symbolic solution of the impact update rules for the triple-pendulum system is too slow, here we will solve it along within the simulation. The idea is, when the impact happens, substitute the numerical values of q and $\dot{q}$ at that moment into the equations you got in Problem 6, thus you will just need to solve a set equations with most terms being numerical values (which is very fast).

The first thing is to write a function called "impact_update_triple_pend". This function at least takes in the current state of the system $s(t^-) = [q(t^-), \dot{q}(t^-)]$ or $\dot{q}(t^-)$, inside the function you need to substitute in $q(t^-)$ and $\dot{q}(t^-)$, solve for and return $s(t^+) = [q(t^+), \dot{q}(t^+)]$ or $\dot{q}(t^+)$ (which should be numerical values now). This function will replace lambdify, and you can use SymPy's "sym.N()" or "expr.evalf()" methods to convert SymPy expressions into numerical values. Test your function with $\theta_1(\tau^-) = \theta_2(\tau^-) = \theta_3(\tau^-) = 0$ and $\dot{\theta}_1(\tau^-) = \theta_2(\tau^-) = \theta_3(\tau^-) = -1$.

Turn in: A copy of your "impact_update_triple_pend" function, and the test result of your function.

In [125…

```
import numpy as np

def impact_update_triple_pend(s):
        subs_update = {th1_minus:s[0],th2_minus:s[1], th3_minus:s[2],
                       th1dot_minus:s[3],th2dot_minus:s[4],th3dot_minus:s[5]}
        lhs2 = lhs.subs(subs_update)
        rhs2 = rhs.subs(subs_update)
        impact_eqns2 = sym.Eq(lhs2, rhs2)

        display(impact_eqns2)

        impact_solns3_1 = solve(impact_eqns2, [th1dot_plus, th2dot_plus, th3dot_plus, lam],dict=True)[0]
        impact_solns3_2 = solve(impact_eqns2, [th1dot_plus, th2dot_plus, th3dot_plus, lam],dict=True)[1]

        print("solutions")
        display(solve(impact_eqns2, [th1dot_plus, th2dot_plus, th3dot_plus, lam],dict=True))

        if abs(impact_solns3_1[lam]) > abs(impact_solns3_2[lam]):
            impact_solns3 = impact_solns3_1
        else:
            impact_solns3 = impact_solns3_2

        return np.array([s[0], s[1], s[2],
                     float((impact_solns3[th1dot_plus]).evalf()),
                     float((impact_solns3[th2dot_plus]).evalf()),
                     float((impact_solns3[th3dot_plus]).evalf())])

# test function
s2 = [0,0,0,-1,-1,-1]
print("")
print('At [theta1 = {}, theta2 = {}, theta3 = {}, theta1_dot = {}, theta2_dot = {}, theta3_dot = {}]'.format(*s2))
print('impact update is [{},{},{},{},{},{}]'.format(*impact_update_triple_pend(s2)))
```

```
At [theta1 = 0, theta2 = 0, theta3 = 0, theta1_dot = -1, theta2_dot = -1, theta3_dot = -1]
```

$$\begin{bmatrix} \dot{\theta}_1^{+} + 14.0\dot{\theta}_2^{+} + 8.0\dot{\theta}_3^{+} + 3.0 + 25.0 \\ \dot{\theta}_1^{+} + 8.0\dot{\theta}_2^{+} + 5.0\dot{\theta}_3^{+} + 2.0 + 15.0 \\ \dot{\theta}_1^{+} + 3.0\dot{\theta}_2^{+} + 2.0\dot{\theta}_3^{+} + 1.0 + 6.0 \\ -0.5\left(\dot{\theta}_1^{+}\right)^2 + \theta_1\left(14.0\dot{\theta}_1^{+} + 8.0\dot{\theta}_2^{+} + 3.0\dot{\theta}_3^{+}\right) + \theta_2\left(8.0\dot{\theta}_1^{+} + 5.0\dot{\theta}_2^{+} + 2.0\dot{\theta}_3^{+}\right) + \theta_3\left(3.0\dot{\theta}_1^{+} + 2.0\dot{\theta}_2^{+} + 1.0\dot{\theta}_3^{+}\right) - 0.5\left(2\dot{\theta}_1^{+} + \dot{\theta}_2^{+}\right)^2 - 0.5\left(3\dot{\theta}_1^{+} + 2\dot{\theta}_2^{+} + \dot{\theta}_3^{+}\right)^2 - 23.0 \end{bmatrix} = \begin{bmatrix} -25.0\lambda \\ -15.0\lambda \\ -6.0\lambda \\ 0 \end{bmatrix}$$

```
solutions
[{\dot{\theta_1}^+: -1.00000000000000,
  \dot{\theta_2}^+: -1.00000000000000,
```

Problem 8 (15pts)

Similar to the single-pendulum system, you will still to implement a function named "impact_condition_triple_pend" to indicate the moment when impact happens. Again, you need to use the constraint ϕ. After finish the impact condition function, simulate the triple-pendulum system with impact for $t \in [0, 2], dt = 0.01$ with initial condition $\theta_1 = \frac{\pi}{2}, \theta_2 = \frac{\pi}{2}, \theta_3 = -\frac{\pi}{2}$ and $\dot{\theta}_1 = \dot{\theta}_2 = \dot{\theta}_3 = 0$, plot the simulated trajectory versus time and animate your simulated trajectory. You will also need to modify the simulate function in this problem.

Turn in: A copy of code for the impact update function and simulate function, also include the plot of simulated trajectory. The video can be uploaded separately through Canvas, and it should be in ".mp4" format. You can use screen capture or record the screen directly with your phone.

In [140…

```
from sympy import lambdify
phi = phi.subs({m1:1, m2:1, m3:1, r1:1, r2:1, r3:1, g:9.8})
fi = Function('phi')(t)
display(Eq(fi,phi))

#numerically solve constraint function phi
phi_func = lambdify([q[0], q[1], q[2],qdot[0], qdot[1], qdot[2]], phi, modules = sym)

#do same threshold check as in the single pendulum problem after
#solving phi lambdify func
def impact_condition_triple_pend(s, threshold=1e-1):
    phi_val = phi_func(*s)
    if phi_val > -threshold and phi_val < threshold:
        return True
    return False

#tgiven initial condition
s = [pi/2, pi/2, -pi/2, 0,0,0]
print('impact will happen at[theta1 = {}, theta2 = {}, theta3 = {}, theta_1_dot = {}, theta_2_dot = {}, theta_3_dot = {}]:{}'.format(*s,impact_condition_triple_pend(s, 1e-1)))

print()
print()
```

$$\phi(t) = 2\sin\left(\theta_1(t) + \theta_2(t)\right) + \sin\left(\theta_1(t) + \theta_2(t) + \theta_3(t)\right) + 3\sin\left(\theta_1(t)\right)$$

```
impact will happen at[theta1 = 1.5707963267948966, theta2 = 1.5707963267948966, theta3 = -1.5707963267948966, theta_1_dot = 0, theta_2_dot = 0, theta_3_dot = 0]:False
```

In [144…

```
import math

#define integrate function
def integrate(f, xt, dt):
    """4RK4 integration
    """
    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

#define simualate function
def simulate_with_impact_3(f, x0, tspan, dt, integrate):
    """
    simulate with impact
    """
    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):
        if impact_condition_triple_pend(x) is True:
            x=impact_update_triple_pend(x)
            xtraj[:,i]=integrate(f,x,dt)
        else:
            xtraj[:,i]=integrate(f,x,dt)
        x=np.copy(xtraj[:,i])
    return xtraj

def impact_update_triple_pend(s):
        subs_update = {th1_minus:s[0],th2_minus:s[1], th3_minus:s[2],
                       th1dot_minus:s[3],th2dot_minus:s[4],th3dot_minus:s[5]}
        lhs2 = lhs.subs(subs_update)
        rhs2 = rhs.subs(subs_update)
        impact_eqns2 = sym.Eq(lhs2, rhs2)

#         display(impact_eqns2)

        impact_solns3_1 = solve(impact_eqns2, [th1dot_plus, th2dot_plus, th3dot_plus, lam],dict=True)[0]
        impact_solns3_2 = solve(impact_eqns2, [th1dot_plus, th2dot_plus, th3dot_plus, lam],dict=True)[1]

        print("solutions")
#         display(solve(impact_eqns2, [th1dot_plus, th2dot_plus, th3dot_plus, lam],dict=True))

        if abs(impact_solns3_1[lam]) > abs(impact_solns3_2[lam]):
            impact_solns3 = impact_solns3_1
        else:
            impact_solns3 = impact_solns3_2

        return np.array([s[0], s[1], s[2],
                     float((impact_solns3[th1dot_plus]).evalf()),
                     float((impact_solns3[th2dot_plus]).evalf()),
                     float((impact_solns3[th3dot_plus]).evalf())])


def dyn3(s):
    """
    System dynamics function (extended)

    Parameters
    ============
    s: NumPy array
        s = [theta1, theta2, theta3, theta1dot, theta2dot, theta3dot]

    Return
    ============
```

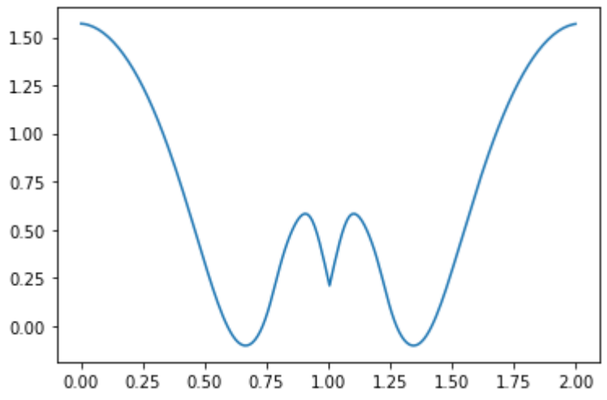
```
        time derivative of input state vector,
        sdot = [theta1dot, theta2dot, theta3dot, theta1_ddot, theta2_ddot, theta3_ddot]
    """
    return np.array([s[3], s[4], s[5],
                     th1ddot_func(s[0], s[1], s[2], s[3], s[4], s[5]),
                     th2ddot_func(s[0], s[1], s[2], s[3], s[4], s[5]),
                     th3ddot_func(s[0], s[1], s[2], s[3], s[4], s[5])])


#simulate the pendulum
# s0 = np.array([pi/2, 0]) # inital values for theta1, theta1dot
# traj2 = simulate_with_impact_3(dyn3, s0, tspan=[0,5], dt=0.01, integrate=integrate)

s00 = np.array([math.pi/2, math.pi/2, -math.pi/2, 0,0,0])
traj3 = simulate_with_impact_3(dyn3, s00, [0,2], 0.01, integrate)
print()
print('shape of traj: ', traj3.shape)
```

```
solutions

shape of traj:  (6, 200)
```

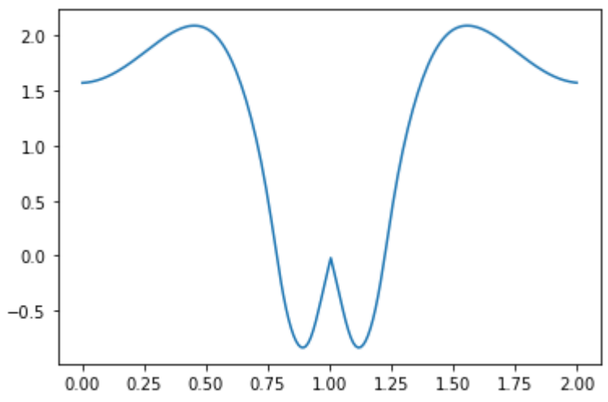
In [149…

```
num_pts = int(2/.01)
print(num_pts)
axis = np.linspace(0,2,num_pts)

print('theta1 trajectory')
plt.plot(axis, traj3[0])
plt.show()
print('theta2 trajectory')
plt.plot(axis, traj3[1])
plt.show()
print('theta3 trajectory')
plt.plot(axis, traj3[2])
plt.show()
```

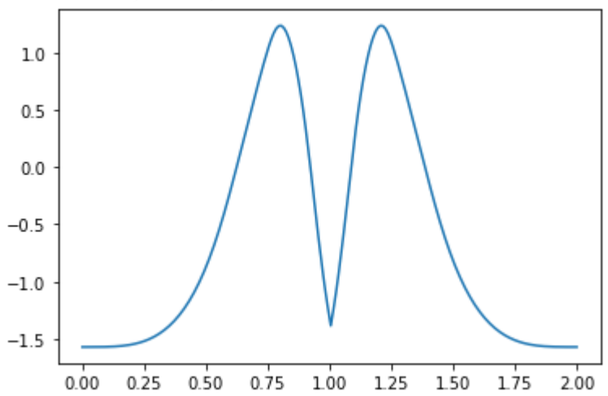
```
200
theta1 trajectory
```




```
theta2 trajectory
```



theta3 trajectory



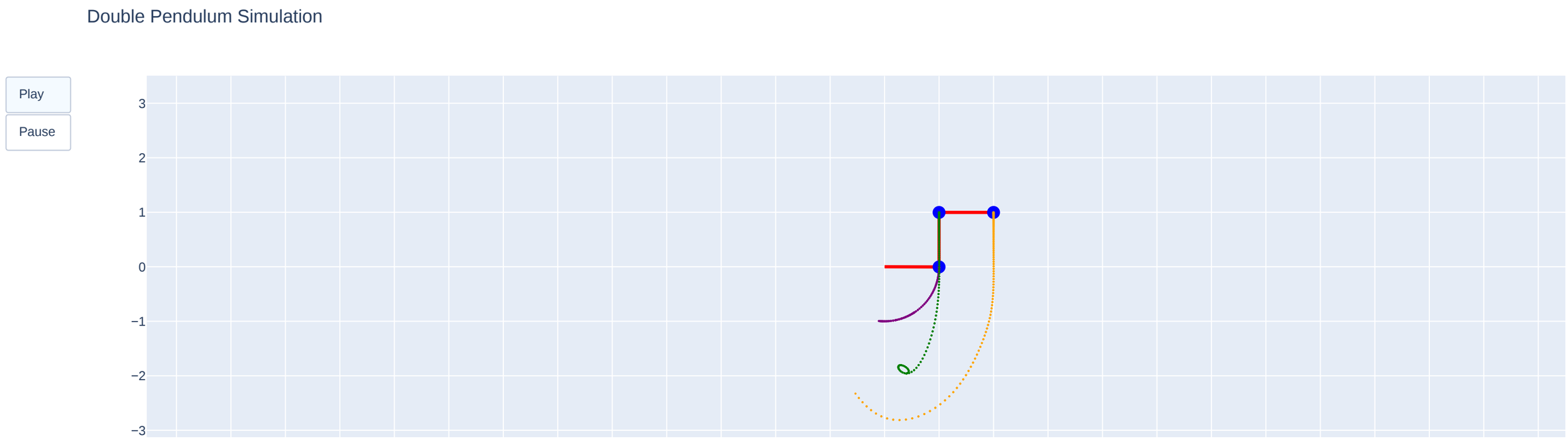
All trajectories where theta1 is blue, theta2 is orange, and theta3 is green

In [150…

```
#copy animation function from previous homework 4
#update for specific use case with impact

def animate_triple_pend(theta_array,L1=1,L2=1,L3=1,T=2):
    """
    Function to generate web-based animation of triple-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
    L3:
        length of the third pendulum
    T:
        length/seconds of animation duration

    Returns: None
    """

    ################################
    # Imports required for animation. (leave this part)
    from plotly.offline import init_notebook_mode, iplot
    from IPython.display import display, HTML
    import plotly.graph_objects as go

    ########################
    # Browser configuration. (leave this part)
    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. (add some code to include the third pendulum)
    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])
    # add some code to include the third pendulum
    xx3=xx2+L3*np.sin(theta_array[0]+theta_array[1]+theta_array[2])
    yy3=yy2-L3*np.cos(theta_array[0]+theta_array[1]+theta_array[2])
    N = len(theta_array[0]) # Need this for specifying length of simulation

    ##################################
    # Using these to specify axis limits. (this needs to be adjusted too)
#     xm=np.min(xx1)-0.5
#     xM=np.max(xx1)+0.5
#     ym=np.min(yy1)-2.5
#     yM=np.max(yy1)+1.5
    xm=np.min(xx3)-0.5
    xM=np.max(xx3)+0.5
    ym=np.min(yy3)-0.5
```

```
    ############################
    # Defining data dictionary. (add some code to include the third pendulum)
    # 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=xx3, y=yy3,
               mode='lines', name='Mass 3',
               line=dict(width=2, color='yellow')
              ),
          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)
              ),
          dict(x=xx3, y=yy3,
               mode='markers', name='Pendulum 3 Traj',
               marker=dict(color="orange", size=2)
              ),
        ]

    ###############################
    # Preparing simulation layout. (leave this part)
    # 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. (add some code to include the third pendulum)
    # This is what draws the lines from
    # joint to joint of the pendulum.
    frames=[dict(data=[dict(x=[0,xx1[k],xx2[k],xx3[k]],
                            y=[0,yy1[k],yy2[k],yy3[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)),
```

```
                        y=[yy3[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)

#################################################
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)

#animate!
animate_triple_pend(traj3,L1=1,L2=1,L3=1,T=2)
```



Problem 9 (5pts)

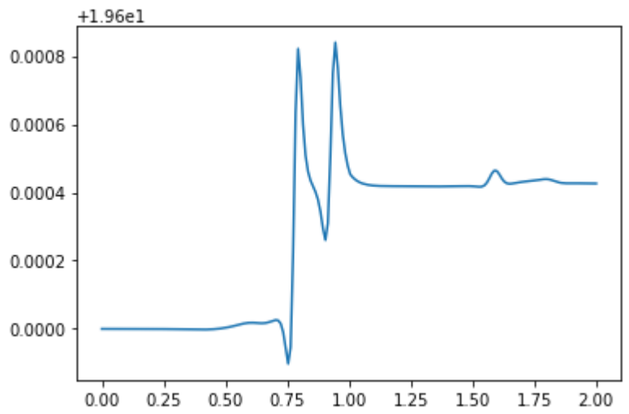
Compute and plot the Hamiltonian of the simulated trajectory for the triple-pendulum system with impact.

Turn in: A copy of code used to compute the Hamiltonian, also include the code output, which should the plot of the Hamiltonian versus time.

In [39]:

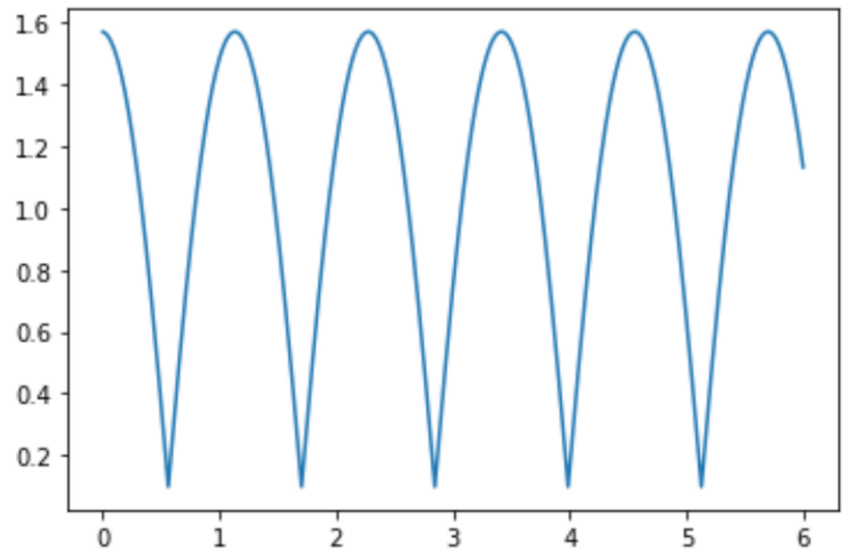
```
Ham = dLdqdot*qdot - L_mat
Ham = Ham.subs({g:9.8})

ham_func = lambdify([q[0], q[1], q[2], qdot[0], qdot[1], qdot[2]], Ham)

H_vals = []
for i in range(num_pts):
    val = ham_func(traj3[0][i], traj3[1][i], traj3[2][i], traj3[3][i], traj3[4][i], traj3[5][i])
    H_vals.append(val)

print("Hamiltonian Plot: ")
plt.plot(axis, H_vals)
plt.show()
```

```
Hamiltonian Plot:
```



In [ ]:

Plot of theta with 5 impacts:
plot will be reattached to bottom of pdf if cut off in this view.



[84]:
```
#get animation function from previous homeworks

#Animation
def animate_single_pend(theta_array,L1=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

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

    T:
        length/seconds of animation duration

    Returns: None
    """

    ################################
    # Imports required for animation.
```

1.7 Problem 7 (15pts)

Since solving the analytical symbolic solution of the impact update rules for the triple-pendulum system is too slow, here we will solve it along within the simulation. The idea is, when the impact happens, substitute the numerical values of q and $\dot{q}$ at that moment into the equations you got in Problem 6, thus you will just need to solve a set equations with most terms being numerical values (which is very fast).

The first thing is to write a function called "impact_update_triple_pend". This function at least takes in the current state of the system $s(t^-) = [q(t^-), \dot{q}(t^-)]$ or $\dot{q}(t^-)$, inside the function you need to substitute in $q(t^-)$ and $\dot{q}(t^-)$, solve for and return $s(t^+) = [q(t^+), \dot{q}(t^+)]$ or $\dot{q}(t^+)$ (which should be numerical values now). This function will replace lambdify, and you can use SymPy's "sym.N()" or "expr.evalf()" methods to convert SymPy expressions into numerical values. Test your function with $\theta_1(\tau^-) = \theta_2(\tau^-) = \theta_3(\tau^-) = 0$ and $\dot{\theta}_1(\tau^-) = \dot{\theta}_2(\tau^-) = \dot{\theta}_3(\tau^-) = -1$.

Turn in: A copy of your "impact_update_triple_pend" function, and the test result of your function.

[125]:
```
import numpy as np

def impact_update_triple_pend(s):
        subs_update = {th1_minus:s[0],th2_minus:s[1], th3_minus:s[2],
                       th1dot_minus:s[3],th2dot_minus:s[4],th3dot_minus:s[5]}
        lhs2 = lhs.subs(subs_update)
        rhs2 = rhs.subs(subs_update)
        impact_eqns2 = sym.Eq(lhs2, rhs2)

        display(impact_eqns2)

        impact_solns3_1 = solve(impact_eqns2, [th1dot_plus, th2dot_plus,␣
↪th3dot_plus, lam],dict=True)[0]
        impact_solns3_2 = solve(impact_eqns2, [th1dot_plus, th2dot_plus,␣
↪th3dot_plus, lam],dict=True)[1]

        print("solutions")
        display(solve(impact_eqns2, [th1dot_plus, th2dot_plus, th3dot_plus,␣
↪lam],dict=True))

        if abs(impact_solns3_1[lam]) > abs(impact_solns3_2[lam]):
            impact_solns3 = impact_solns3_1
        else:
            impact_solns3 = impact_solns3_2

        return np.array([s[0], s[1], s[2],
                    float((impact_solns3[th1dot_plus]).evalf()),
                    float((impact_solns3[th2dot_plus]).evalf()),
                    float((impact_solns3[th3dot_plus]).evalf())])

# test function
```

```
s2 = [0,0,0,-1,-1,-1]
print("")
print('At [theta1 = {}, theta2 = {}, theta3 = {}, theta1_dot = {}, theta2_dot =␣
 ↪{}, theta3_dot = {}]'.format(*s2))
print('impact update is [{},{},{},{},{},{}]'.
 ↪format(*impact_update_triple_pend(s2)))
```

```
At [theta1 = 0, theta2 = 0, theta3 = 0, theta1_dot = -1, theta2_dot = -1,
theta3_dot = -1]
```

$$\begin{bmatrix} 14.0\dot{\theta_1}^+ + 8.0\dot{\theta_2}^+ + 3.0\dot{\theta_3}^+ + 25.0 \\ 8.0\dot{\theta_1}^+ + 5.0\dot{\theta_2}^+ + 2.0\dot{\theta_3}^+ + 15.0 \\ 3.0\dot{\theta_1}^+ + 2.0\dot{\theta_2}^+ + 1.0\dot{\theta_3}^+ + 6.0 \\ -0.5\left(\dot{\theta_1}^+\right)^2 + \dot{\theta_1}^+\left(14.0\dot{\theta_1}^+ + 8.0\dot{\theta_2}^+ + 3.0\dot{\theta_3}^+\right) + \dot{\theta_2}^+\left(8.0\dot{\theta_1}^+ + 5.0\dot{\theta_2}^+ + 2.0\dot{\theta_3}^+\right) + \dot{\theta_3}^+\left(3.0\dot{\theta_1}^+ + 2.0\dot{\theta_2}^+ + 1.0\right. \end{bmatrix}$$

$$\begin{bmatrix} -25.0\lambda \\ -15.0\lambda \\ -6.0\lambda \\ 0 \end{bmatrix}$$

```
solutions

[{\dot{\theta_1}^+: -1.00000000000000,
  \dot{\theta_2}^+: -1.00000000000000,
  \dot{\theta_3}^+: -1.00000000000000,
  \lambda: 0.0},
 {\dot{\theta_1}^+: 1.00000000000000,
  \dot{\theta_2}^+: 1.00000000000000,
  \dot{\theta_3}^+: 1.00000000000000,
  \lambda: -2.00000000000000}]

impact update is [0.0,0.0,0.0,1.0,1.0,1.0]
```

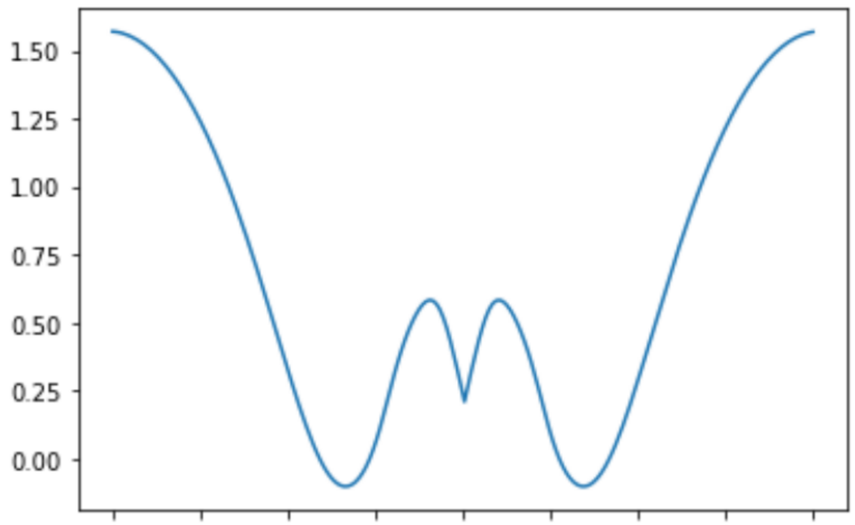
1.8 Problem 8 (15pts)

Similar to the single-pendulum system, you will still to implement a function named "impact_condition_triple_pend" to indicate the moment when impact happens. Again, you need to use the constraint ϕ. After finish the impact condition function, simulate the triple-pendulum system with impact for $t \in [0, 2], dt = 0.01$ with initial condition $\theta_1 = \frac{\pi}{2}, \theta_2 = \frac{\pi}{2}, \theta_3 = -\frac{\pi}{2}$ and $\dot{\theta}_1 = \dot{\theta}_2 = \dot{\theta}_3 = 0$, plot the simulated trajectory versus time and animate your simulated trajectory. You will also need to modify the simulate function in this problem.

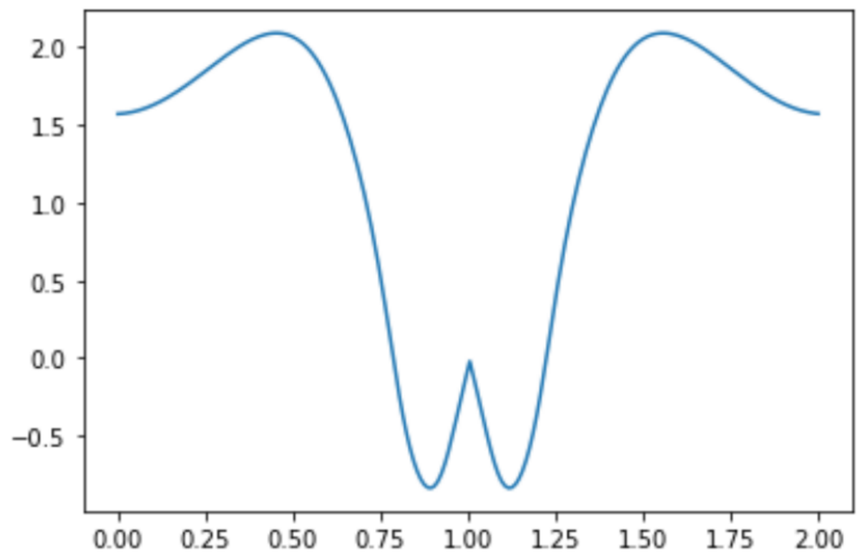
Turn in: A copy of code for the impact update function and simulate function, also include the plot of simulated trajectory. The video can be uploaded separately through Canvas, and it should be in ".mp4" format. You can use screen capture or record the screen directly with your phone.

```
print('theta3 trajectory')
plt.plot(axis, traj3[2])
plt.show()
```

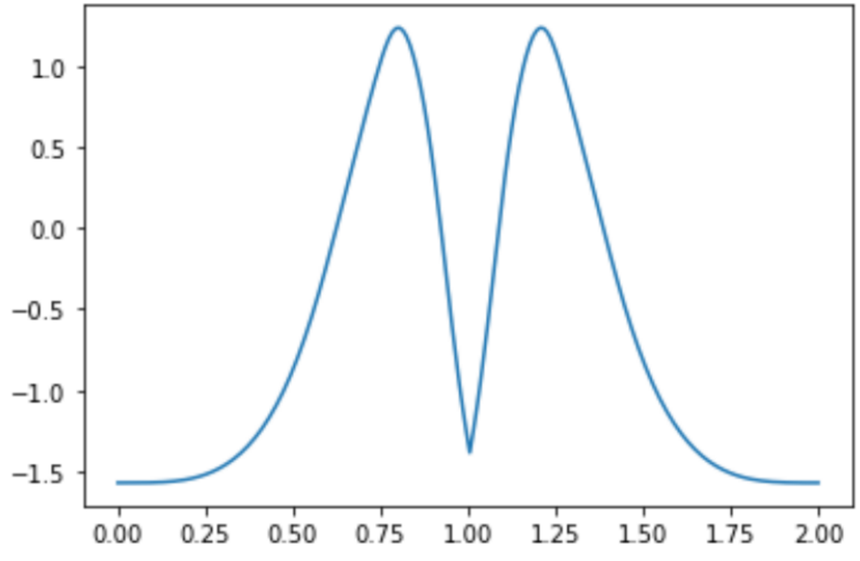
```
200
theta1 trajectory
```



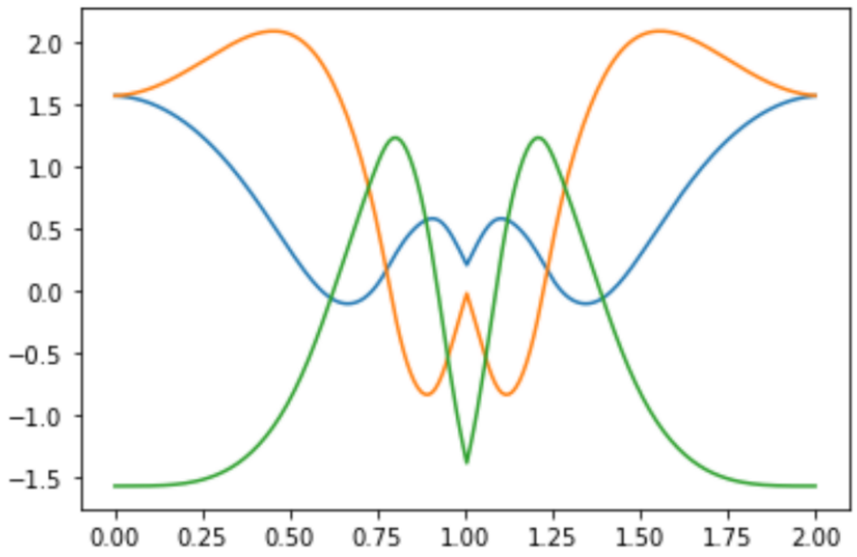

```
theta2 trajectory
```



theta3 trajectory



All trajectories where theta1 is blue, theta2 is orange, and theta3 is green



[150]:
```
#copy animation function from previous homework 4
#update for specific use case with impact

def animate_triple_pend(theta_array,L1=1,L2=1,L3=1,T=2):
    """
    Function to generate web-based animation of triple-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
    L3:
        length of the third pendulum
    T:
        length/seconds of animation duration

    Returns: None
    """

    ################################
    # Imports required for animation. (leave this part)
```