ME314 Homework 5

Collaborated with Valeria Yzaga

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 editing
    - Click Share in the upper right corner
    - Under "Get Link" click "Share with..." or "Change"
    - Then make sure it says "Anyone with Link" and "Editor" under the dropdown menu
- Make sure all cells are run before submitting (i.e. check the permission by running your code in a private mode)
    - Please don't make changes to your file after submitting, so we can grade it!
- Submit a link to your Google Colab file that has been run (before the submission deadline) and don't edit it afterwards!

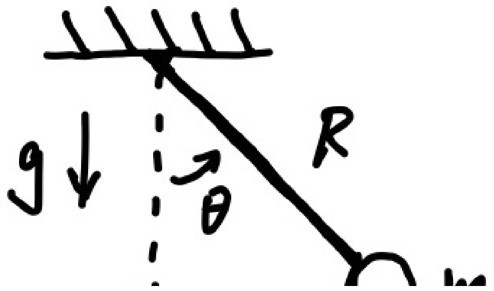
NOTE: This Juputer Notebook file serves as a template for you to start homework. Make sure you first copy this template to your own Google driver (click "File" -> "Save a copy in Drive"), and then start to edit it.

```python
#Import cell
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Markdown, display
```

```python
##############################################################################################
# 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.

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



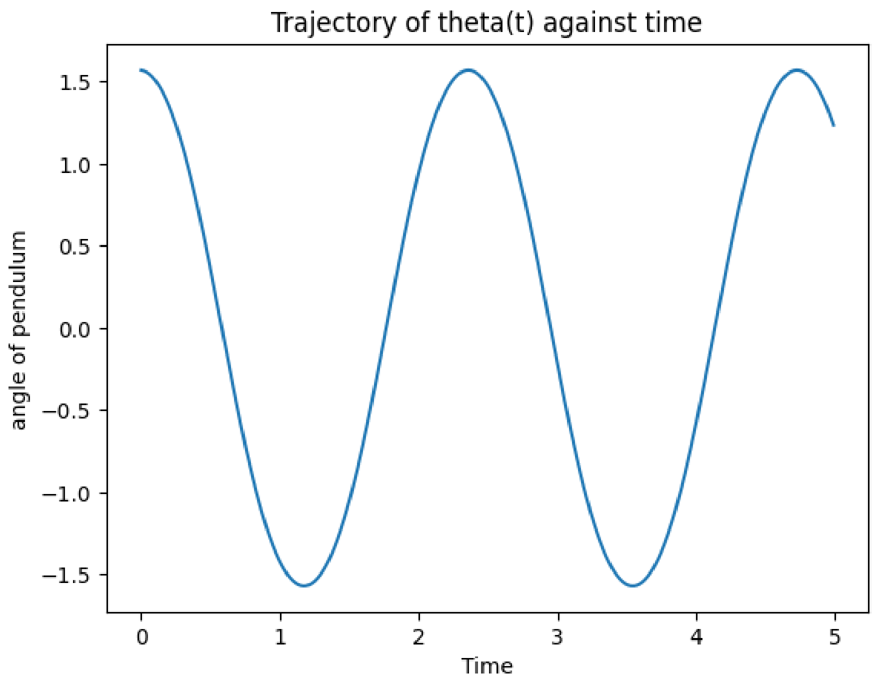
Problem 1 (5pts)

Consider the single pendulum showed above. Solve the Euler-Lagrange equations and simulate the system for $t \in [0, 5]$ with $dt = 0.01, R = 1, m = 1, g = 9.8$ given initial condition as $\theta = \frac{\pi}{2}, \dot{\theta} = 0$. Plot your simulation of the system (i.e. 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.

```
# define symbols, constants
t = sym.symbols('t')
R = sym.symbols('R')
g = sym.symbols('g')
m = sym.symbols('m')

# define functions
theta = sym.Function('theta')(t)

# define derivatives
theta_d = theta.diff(t)
theta_dd = theta_d.diff(t)

# define cartesian
x = R*sym.sin(theta)
y = R*(1 - sym.cos(theta) )

# define derivatives
x_d = x.diff(t)
y_d = y.diff(t)

# energies
KE = 0.5*m*(x_d**2 + y_d**2)
PE = m*g*y

# Lagrangian
Lag = KE - PE

# euler lagrange
eq = sym.Eq( (Lag.diff(theta) - (Lag.diff(theta_d)).diff(t)), 0)
soln = sym.solve(eq, theta_dd)
theta_ddot = soln[0]

# plug constants
theta_ddot_subs = theta_ddot.subs({R:1, m:1, g:9.8})

# lambdify
thetadd_lambdified = sym.lambdify([theta, theta_d],  theta_ddot_subs)

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 theta(t), thetadot(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

##############################################
def thetaddot(theta, theta_d):
  return thetadd_lambdified(theta, theta_d)

```

```python

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

    Parameters
    ============
    s: NumPy array
        s = [theta, thetadot] is the extended system
        state vector, includng the position and
        the velocity of the particle

    Return
    ============
    sdot: NumPy array
        time derivative of input state vector,
        sdot = [thetadot, thetaddot]
    """
    return np.array([s[1], thetaddot(s[0], s[1])])

# define initial state
s0 = np.array([(np.pi)/2, 0])
# simulat from t=0 to 5, since dt=0.01, the returned trajectory
# will have 5/0.01 = 500 time steps, each time step contains extended
traj = simulate(dyn, s0, [0, 5], 0.01, integrate)

plt.plot(np.arange(0, 5, 0.01), traj[0])
plt.title('Trajectory of theta(t) against time')
plt.ylabel('angle of pendulum')
plt.xlabel('Time')
plt.show()



```



Problem 2 (10pts)

Now, time for impact (i.e. don't ignore the vertical wall) ! As shown in the figure above, there is a wall such that the pendulum will hit it when $\theta = 0$. Recall that in the course notes, to solve the impact update rule, we have two set of equations:

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

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

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

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

> Hint 1: The third expression is the Hamiltonian of the system.
>
> Hint 2: All three expressions can be considered as functions of q and $\dot{q}$. If you have previously defined q and $\dot{q}$ as SymPy's function objects, now you will need to substitute them with dummy symbols (using SymPy's substitute method)
>
> Hint 3: 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 your solution output).

```
# define constraint
phi = theta

 # define first equation dL/dq_dot (P)
P = Lag.diff(theta_d)

# define dphi_dq
dphi_dq = phi.diff(theta)

# define Hamiltonian
H = P * theta_d - Lag

# define dummy symbols
theta_s = sym.symbols('theta')
theta_d_s = sym.symbols('thetadot')
# use dummy variables for equations
P_dummy = P.subs({theta:theta_s, theta_d:theta_d_s})
dphi_dq_dummy = dphi_dq.subs({theta:theta_s, theta_d:theta_d_s})
H_dummy = H.subs({theta:theta_s, theta_d:theta_d_s})


# define symbols for display
P_disp = sym.symbols('P')
dphi_disp = sym.symbols('dphi/dq')
H_disp = sym.symbols('H_1')

# display all expressions
display(Markdown("**Expression 1**"))
display(sym.Eq(P_disp, P_dummy.simplify()))
print('')
display(Markdown("**Expression 2**"))
display(sym.Eq(dphi_disp, dphi_dq_dummy.simplify()))
print('')
display(Markdown("**Expression 3**"))
display(sym.Eq(H_disp, H_dummy.expand().simplify()))
```

```
45
```

Expression 1

$P = 1.0R^2m\dot{\theta}$

Expression 2

$dphi/dq = 1$

Expression 3

$H_1 = Rm\left(0.5R\dot{\theta}^2 - 1.0g\cos\left(\theta\right) + 1.0g\right)$

Problem 3 (10pts)

Now everything is ready for you to solve the impact update rules! To solve those equations, you will need to evaluate them right before and after the impact time at τ^- and τ^+.

> Hint 1: Here $\dot{q}(\tau^-)$ is actually same as the dummy symbol you defined in Problem 2 (why?), but you will need to define new dummy symbol 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 evaluate it as a function using SymPy's lambdify method and test it with $\theta(\tau^-) = 0.01, \dot{\theta}(\tau^-) = 2.$

> Hint 2: In your equations and impact update solutions, there should be NO SymPy Function left (except for internal functions like $\sin$ or $\cos$).
>
> Hint 3: You may wonder where are $q(\tau^-)$ and $q(\tau^+)$? The real question at hand is do we really need new dummy variables for them?
>
> Hint 4: The solution of the impact update 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 $\dot{q}(\tau^-)$. While q will not be updated during impact, including it now (as an argument in your lambdify function) 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.

```
# despite having defined expression for q(T-) before, lets do so again to keep things all in this code.

# define lambda
lamb = sym.symbols('lambda')

# define dummy variables
theta_tm = sym.symbols('theta^-')
theta_tp = sym.symbols('theta^+')

# define derivatives of dummy variables
theta_tm_d = sym.symbols('thetadot^-')
theta_tp_d = sym.symbols('thetadot^+')

# Generalized hamiltonian and generalized momentum are defined above
# substituing for tm, tp

# generalized momentum
P_tm = P.subs({theta:theta_tm, theta_d:theta_tm_d})
P_tp = P.subs({theta:theta_tp, theta_d:theta_tp_d})

```

```
# Hamiltonian
H_tm = H.subs({theta:theta_tm, theta_d:theta_tm_d})
H_tp = H.subs({theta:theta_tp, theta_d:theta_tp_d})

# defining the left hand side of impact equations
iu1_lhs = P_tp - P_tm
iu2_lhs = H_tp - H_tm

# define the right hand side of impact equations, note, dphi_dq was defined before
iu1_rhs = lamb*dphi_dq
iu2_rhs = 0

# define theta on the right hand side of equation 1 as theta tau minus
iu1_rhs = iu1_rhs.subs({theta:theta_tm})

# define that q_tp = q_tm
iu1_lhs = iu1_lhs.subs( {theta_tp:theta_tm} )
iu2_lhs = iu2_lhs.subs( {theta_tp:theta_tm} )

# define impact update equations
iu1 = sym.Eq(iu1_lhs, iu1_rhs)
iu2 = sym.Eq(iu2_lhs, iu2_rhs)

display(Markdown("**The Impact Update Equations:**"))
display(iu1.expand().simplify())
print('')
display(iu2.expand().simplify())
print('')

# define what we are solving for, i
i = sym.Matrix([theta_tp_d, lamb])

# solve the impact equations
solved = sym.solve([iu1, iu2], i)

# find impact update rules
rule1 = sym.Eq(theta_tp_d, solved[0][0])
rule2 = sym.Eq(lamb, solved[0][1])

# display impact update rules
display(Markdown("**Display Impact Update Rules:**"))
display(rule1)
print('')
display(rule2)
print('')

# substitute R = 1, m = 1, g = 9.8
rule1_subs = rule1.subs({R:1, m:1, g:9.8})
rule2_subs = rule2.subs({R:1, m:1, g:9.8})

# lambdidy impact update rules so that we can evaluate
iu1 = sym.lambdify([theta_tm, theta_tm_d], rule1_subs.rhs)
iu2 = sym.lambdify([theta_tm, theta_tm_d], rule2_subs.rhs)

# evaluate θ(τ−)=0.01,θ˙(τ−)=2
eval1 = iu1(0.01, 2)
eval2 = iu2(0.01, 2)

display(Markdown("**Numerical Evaluations:**"))
display(sym.Eq(theta_tp_d, eval1))
print('')
display(sym.Eq(lamb, eval2))





```

The Impact Update Equations:

$$\lambda = 1.0R^2 m\left(\dot{\theta}^{+} - \dot{\theta}^{-}\right)$$

$$0.5R^2 m\left(\left(\dot{\theta}^{+}\right)^2 - \left(\dot{\theta}^{-}\right)^2\right) = 0$$

Display Impact Update Rules:

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

$$\lambda = -2.0R^2 m\dot{\theta}^{-}$$

Numerical Evaluations:

$$\dot{\theta}^{+} = -2$$

$$\lambda = -4.0$$

Problem 4 (20pts)

Finally, it's time to simulate the impact! To use impact update rules with 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**.

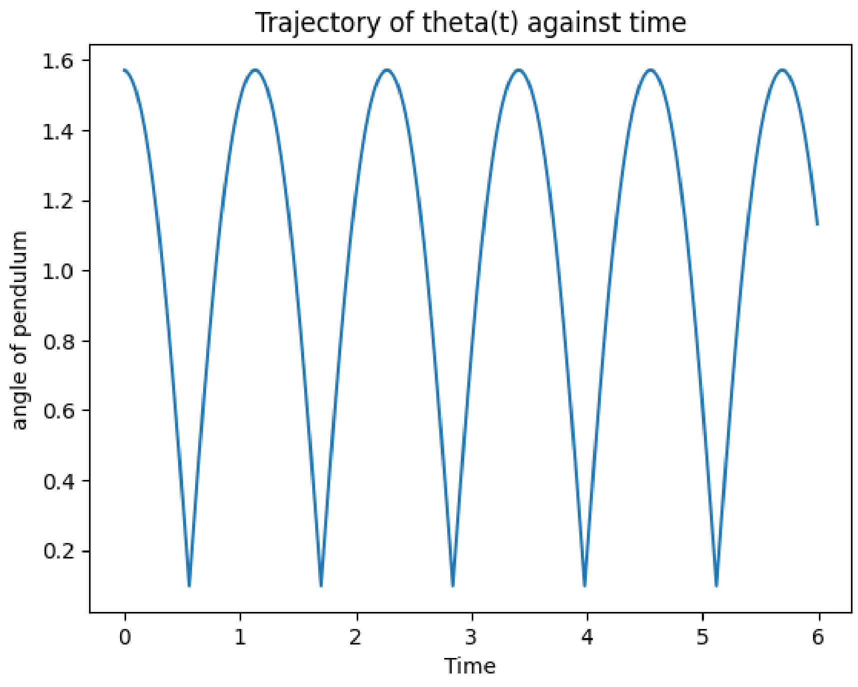
> Hint 1 : you need to use the constraint ϕ in this problem, and note that, since we are doing numerical evaluation, the impact condition will not be perfect, you will need to catch the change of sign at $\phi(s)$ or setup a threshold to decide the condition.

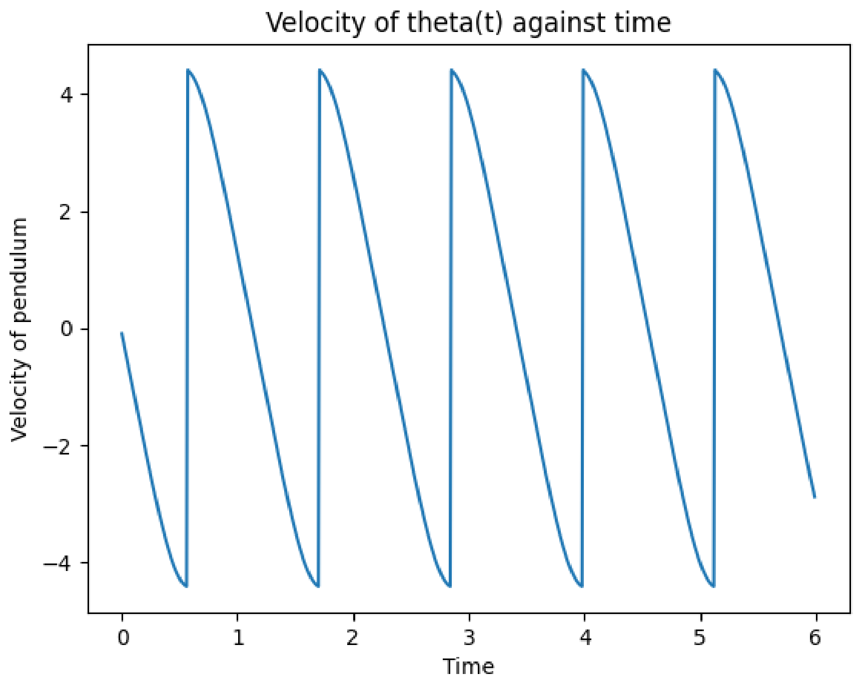
2. Now, with the "impact_condition" function and the numerically evaluated impact update rule for $\dot{q}(\tau^{+})$ solved in last problem, find a way to combine them into the previous simulation function, thus it can simulate the impact. Pseudo-code for the simulate function can be found in lecture 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.

```
phi = theta_s
# lambdify the constraint phi
phi_lamb = sym.lambdify([theta_s, theta_d_s], phi)

# defining the impact condition function that determines if an impact occurs. This is if the state s is withi
def impact_condition(s, threshold=1e-1):
  phi_val = phi_lamb(s[0], s[1])
  # if the phi_val is within the threshold (aka close enough to zero)
  if phi_val > -threshold and phi_val < threshold:
    return True
  return False

def impact_update(s):
  return np.array([s[0], iu1(*s)])


def integrate(f, xt, dt):

  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_with_impact(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):
    # check if there is impact
    if impact_condition(x) is True:
      # to avoid staying inside surface
      if phi_lamb(x[0], x[1]) < 0:
        iu = iu1(xtraj[:,i-1][0], xtraj[:,i-1][1])
        # update state
        theta_update = xtraj[:,i-1][0]
        theta_d_update = iu
        xtraj[:,i] = [theta_update, theta_d_update]

      else:
        # else, its a normal impact
        x = impact_update(x)
        xtraj[:,i] = integrate(f,x,dt)

    else:
      # else, we have normal trajectory
      xtraj[:,i] = integrate(f,x,dt)
    x = np.copy(xtraj[:,i])
  return xtraj


# define dyn
def dyn(s):
  return np.array([s[1], thetadd_lambdified(*s)])


s0 = [np.pi/2, 0]
traj_impact = simulate_with_impact(dyn, s0, tspan=[0,6], dt=0.01, integrate=integrate)

import matplotlib.pyplot as plt
plt.plot(np.arange(0, 6, 0.01), traj_impact[0].T)
plt.title('Trajectory of theta(t) against time')
plt.ylabel('angle of pendulum')
plt.xlabel('Time')
plt.show()

plt.plot(np.arange(0, 6, 0.01), traj_impact[1].T)
plt.title('Velocity of theta(t) against time')
plt.ylabel('Velocity of pendulum')
plt.xlabel('Time')
plt.show()
```

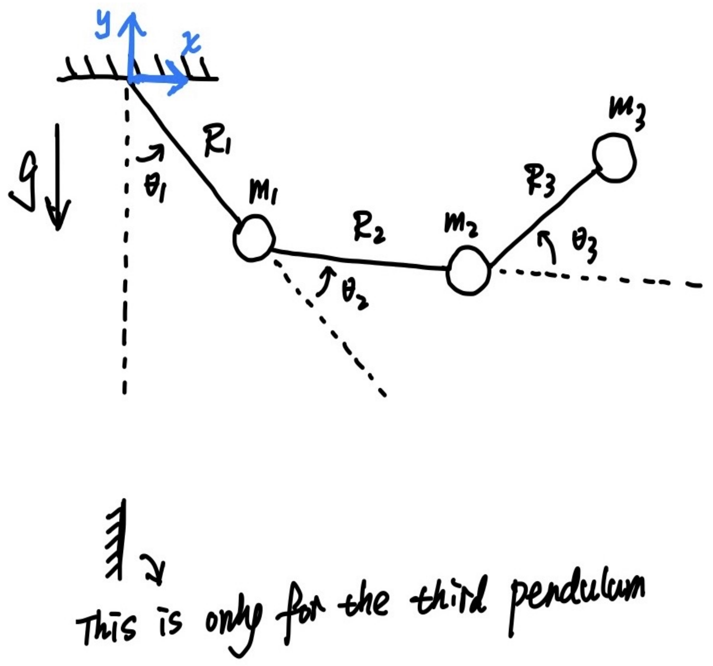




```
def animate_pend(theta_array, L=1, T=5):
    """
    Function to generate web-based animation of pendulum system

    Parameters:
    ================================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L:
        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. Using our equations
    xx1=L*np.sin(theta_array[0])
    yy1= -L*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)-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=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='Pendulum Impact 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
                                            'transition': {'duration': 0}}],'label': 'Pause','method': 'anima
                                         ]
                             }]
              )

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

##################################################
# 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(-(np.pi)/2, 1, 100), np.linspace(-(np.pi)/2, 1, 100)])
#print('shape of trajectory: ', sim_traj.shape)

# just make L2 = 0, that way it cancels as a seocnd pendulum
animate_pend(traj_impact, L=1, T=5)
```

Problem 5 (10pts)

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



We will now consider a constrained triple-pendulum system with the system configuration $q = [\theta_1, \theta_2, \theta_3]$. A constraint is such that *x coordinate* of the third pendulum (i.e. m_3) ONLY can not be smaller than 0 -- there exist a vertical wall high enough for third pendulum impact. Note that there is no constraint on *y coordinate* -- the top ceiling is infinitely high!

Similar to Problem 2, symbolically compute the following three expressions contained the equations above:

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

Use $m_1 = m_2 = m_3 = 1$ and $R_1 = R_2 = R_3 = 1$ as numerical values in the equations (i.e. **do not** define $m_1, m_2, m_3, R_1, R_2, R_3$ as symbols).

> Hint 1: As before, you will need to substitute q and $\dot{q}$ with dummy symbols.

Turn in: Include the code used to symbolically compute the three expressions, as well as code outputs - the resulting three expressions. Make sure there is no SymPy Function(t) left!

```
from sympy import sin, cos

# let us define time
t = sym.symbols('t')

# define constants
g = sym.symbols('g')

# define configuration variables
theta1 = sym.Function('theta_1')(t)
theta2 = sym.Function('theta_2')(t)
theta3 = sym.Function('theta_3')(t)

```

```
# define derivatives of configurations
theta1_d = theta1.diff(t)
theta2_d = theta2.diff(t)
theta3_d = theta3.diff(t)

theta1_dd = theta1_d.diff(t)
theta2_dd = theta2_d.diff(t)
theta3_dd = theta3_d.diff(t)

# define cartesian positions of the pendulums
x1 = sin(theta1)
y1 = -cos(theta1)

x2 = x1 + sin(theta1 + theta2)
y2 = y1 - cos(theta1 + theta2)

x3 = x2 + sin(theta1 + theta2 + theta3)
y3 = y2 - cos(theta1 + theta2 + theta3)

# derivatives of cartesian coordinates
x1_d = x1.diff(t)
y1_d = y1.diff(t)

x2_d = x2.diff(t)
y2_d = y2.diff(t)

x3_d = x3.diff(t)
y3_d = y3.diff(t)

# define KE
KE = 0.5*(x1_d**2 + y1_d**2) + 0.5*(x2_d**2 + y2_d**2) + 0.5*(x3_d**2 + y3_d**2)

# define potential energy
PE = g*y1 + g*y2 + g*y3

# define Lagrangian
L = KE - PE
L = L.subs({g:9.8})

# define constraint
phi = x3

### DETERMINE EXPRESSIONS ###

# define configuration
q = sym.Matrix([theta1, theta2, theta3])
# define derivative of configuration matrix q_d
q_d = sym.Matrix([theta1_d, theta2_d, theta3_d])

# first expressions ∂L/∂q˙ (P)
P1 = L.diff(theta1_d).simplify()
P2 = L.diff(theta2_d).simplify()
P3 = L.diff(theta3_d).simplify()
# defining P as 1xn row matrix
P = sym.Matrix( [[ (P1), (P2), (P3) ]] )

# second expressions (∂ϕ/∂q)
dphi_dq1 = phi.diff(theta1)
dphi_dq2 = phi.diff(theta2)
dphi_dq3 = phi.diff(theta3)
# generate matrix
dphi_dq = sym.Matrix([dphi_dq1, dphi_dq2, dphi_dq3])


# third expressions (H)
H_s = P.dot(q_d) - L

# define dummy variables (s)
# define dummy variables (s)
theta1s = sym.symbols('theta_1')
theta2s = sym.symbols('theta_2')
theta3s = sym.symbols('theta_3')

theta1s_d = sym.symbols('thetadot_1')
theta2s_d = sym.symbols('thetadot_2')
theta3s_d = sym.symbols('thetadot_3')

# redefine P for later purposes
P = sym.Matrix( [ P1, P2, P3] )

# defining all substitutions as dictionary dummy
dummy = {theta1:theta1s, theta2:theta2s, theta3:theta3s, theta1_d:theta1s_d, theta2_d:theta2s_d, theta3_d:the

# performing substitutions on expressions
P_subs = P.subs(dummy)

dphi_dq_subs = sym.Matrix([dphi_dq1.subs(dummy), dphi_dq2.subs(dummy), dphi_dq3.subs(dummy)])

H_subs = H_s.subs(dummy).simplify()

# define symbols for display
P_1 = sym.symbols('P_1')
P_2 = sym.symbols('P_2')
P_3 = sym.symbols('P_3')

dphi = sym.symbols('dphi/dq')

## display expressions
display(Markdown("**Expression 1:**"))
display(P_subs)
print('')
display(Markdown("**Expression 2:**"))
display(dphi_dq_subs)
print('')
display(Markdown("**Expression 3:**"))
display(H_subs)
```

```python




print('')
```

Expression 1:

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

Expression 2:

$$\begin{bmatrix} \cos(\theta_1) + \cos(\theta_1+\theta_2) + \cos(\theta_1+\theta_2+\theta_3) \\ \cos(\theta_1+\theta_2) + \cos(\theta_1+\theta_2+\theta_3) \\ \cos(\theta_1+\theta_2+\theta_3) \end{bmatrix}$$

Expression 3:

$$2.0\dot{\theta}_1^2\cos(\theta_2) + 1.0\dot{\theta}_1^2\cos(\theta_3) + 1.0\dot{\theta}_1^2\cos(\theta_2+\theta_3) + 3.0\dot{\theta}_1^2 + 2.0\dot{\theta}_1\dot{\theta}_2\cos(\theta_2) + 2.0\dot{\theta}_1\dot{\theta}_2\cos(\theta_3) + 1.0\dot{\theta}_1\dot{\theta}_2\cos(\theta_2+\theta_3) + 3.0\dot{\theta}_1\dot{\theta}$$
$$1.0\dot{\theta}_1\dot{\theta}_3\cos(\theta_3) + 1.0\dot{\theta}_1\dot{\theta}_3\cos(\theta_2+\theta_3) + 1.0\dot{\theta}_1\dot{\theta}_3 + 1.0\dot{\theta}_2^2\cos(\theta_3) + 1.5\dot{\theta}_2^2 + 1.0\dot{\theta}_2\dot{\theta}_3\cos(\theta_3) + 1.0\dot{\theta}_2\dot{\theta}_3 + 0.5\dot{\theta}_3^2 - 29.4\cos(\theta_1) -$$
$$19.6\cos(\theta_1+\theta_2) - 9.8\cos(\theta_1+\theta_2+\theta_3)$$

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, and we will use a trick to get around it later!

Turn in: Include a copy of the code used to define the equations for impact update and the code output (i.e. print out of the equations).

```python
# define constants lambda
lamb = sym.symbols('lambda')

# define dummy sumbols for q(Tau-), q_d(Tau-), q(Tau+), q_d(Tau+)
theta1_tm = sym.symbols('theta_1^-')
theta2_tm = sym.symbols('theta_2^-')
theta3_tm = sym.symbols('theta_3^-')

theta1_tm_d = sym.symbols('thetadot_1^-')
theta2_tm_d = sym.symbols('thetadot_2^-')
theta3_tm_d = sym.symbols('thetadot_3^-')

theta1_tp = sym.symbols('theta_1^+')
theta2_tp = sym.symbols('theta_2^+')
theta3_tp = sym.symbols('theta_3^+')

theta1_tp_d = sym.symbols('thetadot_1^+')
theta2_tp_d = sym.symbols('thetadot_2^+')
theta3_tp_d = sym.symbols('thetadot_3^+')

theta1_tp = theta1_tm
theta2_tp = theta2_tm
theta3_tp = theta3_tm

# define all dummy for tau neg
tm = {theta1s:theta1_tm, theta2s:theta2_tm, theta3s:theta3_tm, theta1s_d:theta1_tm_d, theta2s_d:theta2_tm_d, 
# define all dummy for tau plus
tp = {theta1s:theta1_tp, theta2s:theta2_tp, theta3s:theta3_tp, theta1s_d:theta1_tp_d, theta2s_d:theta2_tp_d, 


# define q(T+) = q(T-)

# IMPACT UPDATE EQUATIONS
P_tp = P_subs.subs(tp)
P_tm = P_subs.subs(tm)

H_tp = H_subs.subs(tp)
H_tm = H_subs.subs(tm)

eq1_lhs = P_tp[0] - P_tm[0]
eq2_lhs = P_tp[1] - P_tm[1]
eq3_lhs = P_tp[2] - P_tm[2]

eq4_lhs = H_tp - H_tm

eq1_rhs = lamb*dphi_dq_subs[0].subs(tm)
eq2_rhs = lamb*dphi_dq_subs[1].subs(tm)
eq3_rhs = lamb*dphi_dq_subs[2].subs(tm)
eq4_rhs = 0

## substitute configurations so that they are equal
eq1_lhs = eq1_lhs.expand().simplify()
eq2_lhs = eq2_lhs.expand().simplify()
eq3_lhs = eq3_lhs.expand().simplify()
eq4_lhs = eq4_lhs.expand().simplify()

# Generate Martrix
eqs_lhs = sym.Matrix([eq1_lhs, eq2_lhs, eq3_lhs])
eqs_rhs = sym.Matrix([eq1_rhs, eq2_rhs, eq3_rhs])

# Matrix of Equations
P_eqs = sym.Eq(eqs_lhs, eqs_rhs)
H = sym.Eq(eq4_lhs, eq4_rhs)

## DISPLAY IMPACT UPDATES ###
display(Markdown("**Impact Update Equations:**"))
print('')
print('Impact Update Eq1-Eq3')
display(P_eqs)
print('')
print('Impact Update Eq4')
display(H)
```

Impact Update Equations:

```
Impact Update Eq1-Eq3
```

$$\begin{bmatrix} 4.0\dot{\theta}_1^+ \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^+ \cos\left(\theta_3^-\right) + 2.0\dot{\theta}_1^+ \cos\left(\theta_2^- + \theta_3^-\right) + 6.0\dot{\theta}_1^+ - 4.0\dot{\theta}_1^- \cos\left(\theta_2^-\right) - 2.0\dot{\theta}_1^- \cos\left(\theta_3^-\right) - 2.0\dot{\theta}_1^- \cos\left(\theta_2^- + \theta_3^-\right) - 6. \\ 2.0\dot{\theta}_1^+ \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^+ \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\dot{\theta} \\ 1.0\dot{\theta}_1^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^+ \cos \end{bmatrix}$$

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

```
Impact Update Eq4
```

$$2.0\left(\dot{\theta}_1^+\right)^2 \cos\left(\theta_2^-\right) + 1.0\left(\dot{\theta}_1^+\right)^2 \cos\left(\theta_3^-\right) + 1.0\left(\dot{\theta}_1^+\right)^2 \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\left(\dot{\theta}_1^+\right)^2 + 2.0\dot{\theta}_1^+\dot{\theta}_2^+ \cos\left(\theta_2^-\right) + 2.0\dot{\theta}_1^+\dot{\theta}_2^+ \cos\left(\theta_3^-\right) +$$
$$1.0\dot{\theta}_1^+\dot{\theta}_2^+ \cos\left(\theta_2^- + \theta_3^-\right) + 3.0\dot{\theta}_1^+\dot{\theta}_2^+ + 1.0\dot{\theta}_1^+\dot{\theta}_3^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_1^+\dot{\theta}_3^+ \cos\left(\theta_2^- + \theta_3^-\right) + 1.0\dot{\theta}_1^+\dot{\theta}_3^+ - 2.0\left(\dot{\theta}_1^-\right)^2 \cos\left(\theta_2^-\right) - 1.0\left(\dot{\theta}_1^-\right)^2 \mathrm{co}$$
$$1.0\left(\dot{\theta}_1^-\right)^2 \cos\left(\theta_2^- + \theta_3^-\right) - 3.0\left(\dot{\theta}_1^-\right)^2 - 2.0\dot{\theta}_1^-\dot{\theta}_2^- \cos\left(\theta_2^-\right) - 2.0\dot{\theta}_1^-\dot{\theta}_2^- \cos\left(\theta_3^-\right) - 1.0\dot{\theta}_1^-\dot{\theta}_2^- \cos\left(\theta_2^- + \theta_3^-\right) - 3.0\dot{\theta}_1^-\dot{\theta}_2^- - 1.0\dot{\theta}_1^-\dot{\theta}_3^- \mathrm{co}$$
$$1.0\dot{\theta}_1^-\dot{\theta}_3^- \cos\left(\theta_2^- + \theta_3^-\right) - 1.0\dot{\theta}_1^-\dot{\theta}_3^- + 1.0\left(\dot{\theta}_2^+\right)^2 \cos\left(\theta_3^-\right) + 1.5\left(\dot{\theta}_2^+\right)^2 + 1.0\dot{\theta}_2^+\dot{\theta}_3^+ \cos\left(\theta_3^-\right) + 1.0\dot{\theta}_2^+\dot{\theta}_3^+ - 1.0\left(\dot{\theta}_2^-\right)^2 \cos\left(\theta_3^-\right) - 1.5$$
$$1.0\dot{\theta}_2^-\dot{\theta}_3^- \cos\left(\theta_3^-\right) - 1.0\dot{\theta}_2^-\dot{\theta}_3^- + 0.5\left(\dot{\theta}_3^+\right)^2 - 0.5\left(\dot{\theta}_3^-\right)^2 = 0$$

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 $\dot{\theta}_1(\tau^-) = \dot{\theta}_2(\tau^-) = \dot{\theta}_3(\tau^-) = -1$.

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

```
def impact_update_triple_pend(s):
  P_eq_lhs = (P_eqs.lhs).subs({theta1_tm:s[0], theta2_tm:s[1], theta3_tm:s[2], theta1_tm_d:s[3], theta2_tm_d:s
  P_eq_rhs = (P_eqs.rhs).subs({theta1_tm:s[0], theta2_tm:s[1], theta3_tm:s[2], theta1_tm_d:s[3], theta2_tm_d:s
  H_eq = eq4_lhs.subs({theta1_tm:s[0], theta2_tm:s[1], theta3_tm:s[2], theta1_tm_d:s[3], theta2_tm_d:s[4], the

  # generate matrix of updates equations (4 equations)
  iu_lhs = sym.Matrix([P_eq_lhs[0], P_eq_lhs[1], P_eq_lhs[2], H_eq])
  iu_rhs = sym.Matrix([P_eq_rhs[0], P_eq_rhs[1], P_eq_rhs[2], 0])
  iu_eqs = sym.Eq(iu_lhs, iu_rhs)
  # define what we are solving for (4 unkowns)
  a = [theta1_tp_d, theta2_tp_d, theta3_tp_d, lamb]
  # solve
  soln = sym.solve(iu_eqs, a)
  # define solutions (the second one where lambda is not equal to zero)
  thet1_tp_d = soln[0][0]
  thet2_tp_d = soln[0][1]
  thet3_tp_d = soln[0][2]
  lam = soln[0][3]
  if lam == 0:
    thet1_tp_d = soln[1][0]
    thet2_tp_d = soln[1][1]
    thet3_tp_d = soln[1][2]
    lam = soln[1][3]

  l1 = sym.lambdify([theta1_tm, theta2_tm, theta3_tm, theta1_tm_d, theta2_tm_d, theta3_tm_d], thet1_tp_d)
  l2 = sym.lambdify([theta1_tm, theta2_tm, theta3_tm, theta1_tm_d, theta2_tm_d, theta3_tm_d], thet2_tp_d)
  l3 = sym.lambdify([theta1_tm, theta2_tm, theta3_tm, theta1_tm_d, theta2_tm_d, theta3_tm_d], thet3_tp_d)

  return [s[0], s[1], s[2], l1(*s), l2(*s), l3(*s)]

# test s = [0, 0, 0, -1, -1, -1]
display(Markdown("**Test Values:**"))
display(impact_update_triple_pend(s = [0, 0, 0, -1, -1, -1]))
```

Test Values:

$$[0,\ 0,\ 0,\ -1.0,\ -1.0,\ 11.0]$$

Problem 8 (15pts)

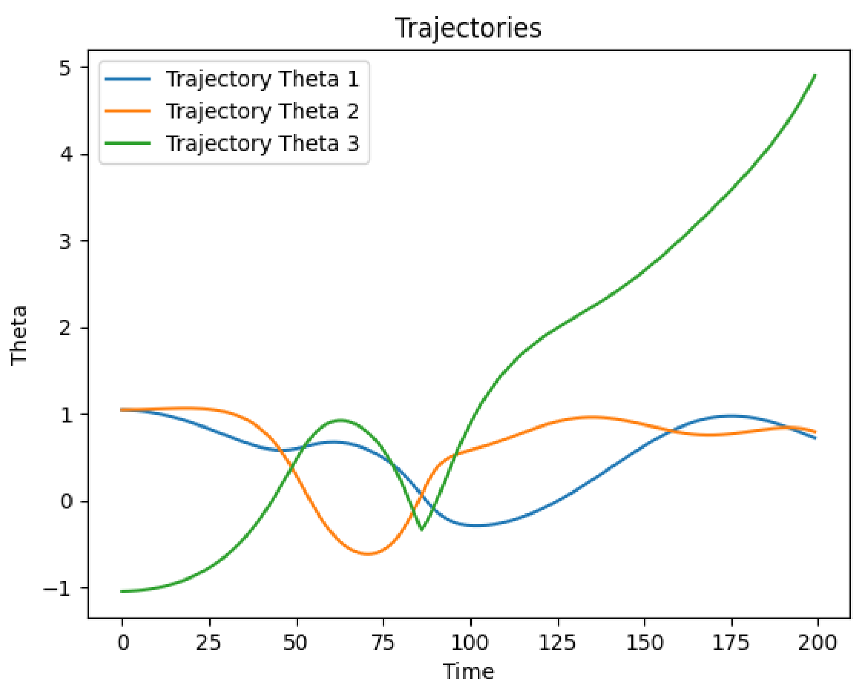
Similar to the single-pendulum system, you will still want to implement a function named "impact_condition_triple_pend" to indicate the moment when impact happens. Again, you need to use the constraint ϕ. After obtaining the impact condition function, simulate the triple-pendulum system with impact for $t \in [0, 2], dt = 0.01$ with initial condition $\theta_1 = \frac{\pi}{3}, \theta_2 = \frac{\pi}{3}, \theta_3 = -\frac{\pi}{3}$ and $\dot{\theta}_1 = \dot{\theta}_2 = \dot{\theta}_3 = 0$. Plot the simulated trajectory versus time and animate your simulated trajectory.

> Hint 1: You will need to modify the simulate function!

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

```
## SOLVING THE EULER LAGRANGE

# define configuration variables
theta1 = sym.Function('theta_1')(t)
theta2 = sym.Function('theta_2')(t)
theta3 = sym.Function('theta_3')(t)

# define derivatives of configurations
theta1_d = theta1.diff(t)
theta2_d = theta2.diff(t)
theta3_d = theta3.diff(t)

theta1_dd = theta1_d.diff(t)
theta2_dd = theta2_d.diff(t)
theta3_dd = theta3_d.diff(t)

EL1_lhs = L.diff(theta1) - (L.diff(theta1_d)).diff(t)
EL2_lhs = L.diff(theta2) - (L.diff(theta2_d)).diff(t)
EL3_lhs = L.diff(theta3) - (L.diff(theta3_d)).diff(t)

lhs = sym.Matrix([EL1_lhs.expand().simplify(), EL2_lhs.expand().simplify(), EL3_lhs.expand().simplify()])
rhs = sym.Matrix([0, 0, 0])

eqs = sym.Eq(lhs, rhs)
r = sym.Matrix([theta1_dd, theta2_dd, theta3_dd])

soln = sym.solve(eqs, r)

soln1 = soln[theta1_dd]
soln2 = soln[theta2_dd]
soln3 = soln[theta3_dd]

## Lambdify
l1 = sym.lambdify([theta1, theta2, theta3, theta1_d, theta2_d, theta3_d], soln1)
l2 = sym.lambdify([theta1, theta2, theta3, theta1_d, theta2_d, theta3_d], soln2)
l3 = sym.lambdify([theta1, theta2, theta3, theta1_d, theta2_d, theta3_d], soln3)
```

```
phi = x3.subs({theta1:theta1s, theta2:theta2s, theta3:theta3s, theta1_d:theta1s_d, theta2_d:theta2s_d, theta3_
# lambdify the constraint phi
phi_lamb_trip = sym.lambdify([theta1s, theta2s, theta3s, theta1s_d, theta2s_d, theta3s_d], phi)

# defining the impact condition function that determines if an impact occurs. This is if the state s is withi
def impact_condition_triple_pend(s, threshold=1e-1):
  phi_val = phi_lamb_trip(*s)
  # if the phi_val is within the threshold (aka close enough to zero)
  if phi_val > -threshold and phi_val < threshold:
    return True
  return False

# define dyn
def dyn(s):
  return np.array([s[3], s[4], s[5], l1(*s), l2(*s), l3(*s)])


def integrate(f, xt, dt):
  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_with_impact_triple(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):
    # check if there is impact
    if impact_condition_triple_pend(x) is True:
      # # to avoid staying inside surface
      # if phi_lamb(x[0], x[1]) < 0:
      #   iu = iu1(xtraj[:,i-1][0], xtraj[:,i-1][1])
      #   # update state
      #   theta_update = xtraj[:,i-1][0]
      #   theta_d_update = iu
      #   xtraj[:,i] = [theta_update, theta_d_update]

      # else:
        # else, its a normal impact
      x = impact_update_triple_pend(x)
      xtraj[:,i] = integrate(f,x,dt)
    else:
      # else, we have normal trajectory
      xtraj[:,i] = integrate(f,x,dt)
    x = np.copy(xtraj[:,i])
  return xtraj


s0 = np.array([(np.pi)/3, (np.pi)/3, -(np.pi)/3, 0, 0, 0])

traj_impact = simulate_with_impact_triple(dyn, s0, tspan=[0,2], dt=0.01, integrate=integrate)

# Plot Trajectories #
import numpy as np
import matplotlib.pyplot as plt


# Plotting
plt.plot(np.arange(traj_impact.shape[1]), traj_impact[0], label='Trajectory Theta 1')  # Plot Trajectory 1
plt.plot(np.arange(traj_impact.shape[1]), traj_impact[1], label='Trajectory Theta 2')  # Plot Trajectory 2
plt.plot(np.arange(traj_impact.shape[1]), traj_impact[2], label='Trajectory Theta 3')  # Plot Trajectory 3

# Add labels and title
plt.xlabel('Time')
plt.ylabel('Theta')
plt.title('Trajectories')

# Add legend
plt.legend()

# Display the plot
plt.show()
```



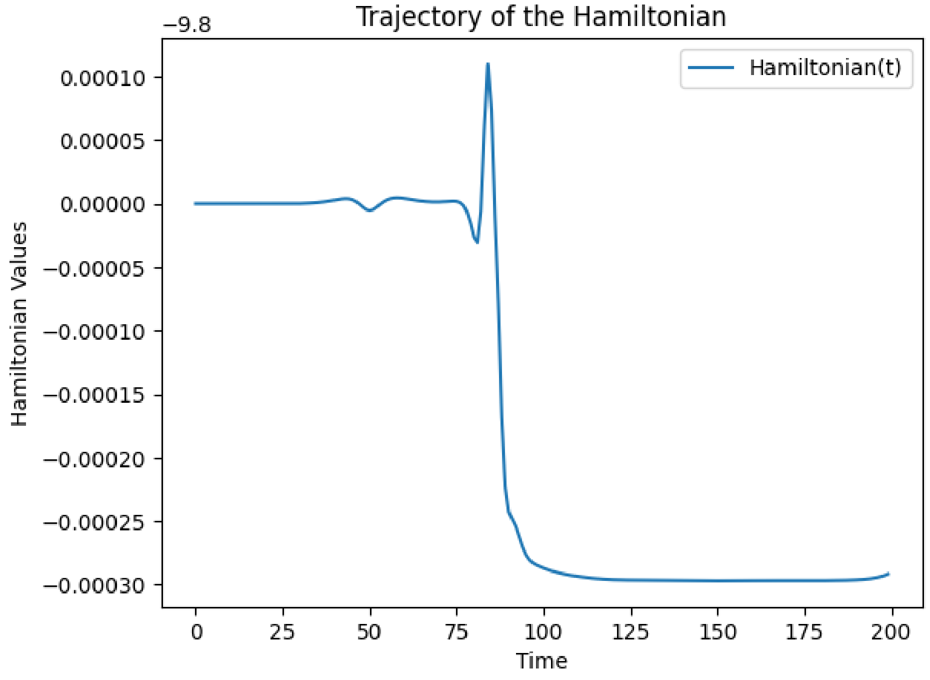
```
def animate_triple_pend(theta_array, L1=1, L2=1, L3=1, T=2):
    """
    Function to generate web-based animation of triple-pendulum system

    Parameters:
    ================================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (3,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.
    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])
    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.
    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=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="yellow", 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='Triple 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
                                            'transition': {'duration': 0}}],'label': 'Pause','method': 'anima
                                          ]
                              }]
               )

    ########################################
    # 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],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)),
                       go.Scatter(
                            x=[xx3[k]],
                            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)

animate_triple_pend(traj_impact)
```

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.

```
## Define Trajectories
theta1_traj = traj_impact[0]
theta2_traj = traj_impact[0]
theta3_traj = traj_impact[0]
theta1_d_traj = traj_impact[0]
theta2_d_traj = traj_impact[0]
theta3_d_traj = traj_impact[0]

# lambdify the hamiltonian
H_l = sym.lambdify([theta1, theta2, theta3, theta1_d, theta2_d, theta3_d], H_s)

def Hamiltonian(theta1, theta2, theta3, theta1_d, theta2_d, theta3_d):
  return H_l(theta1, theta2, theta3, theta1_d, theta2_d, theta3_d)

Hvec = []

# iterating over the time step and determining the hamiltonian at each time step
for i in range(traj_impact.shape[1]):
  H_value = list(row[i] for row in traj_impact)
  Hvec.append(Hamiltonian(*H_value))


plt.plot(np.arange(traj_impact.shape[1]), Hvec, label='Hamiltonian(t)')
plt.title("Trajectory of the Hamiltonian")
plt.ylabel('Hamiltonian Values')
plt.xlabel("Time")
plt.legend()
plt.show()


```



●