ME314 Homework 5 (Solutions)

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

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

In [1]:
```python
#Import cell
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```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)
```

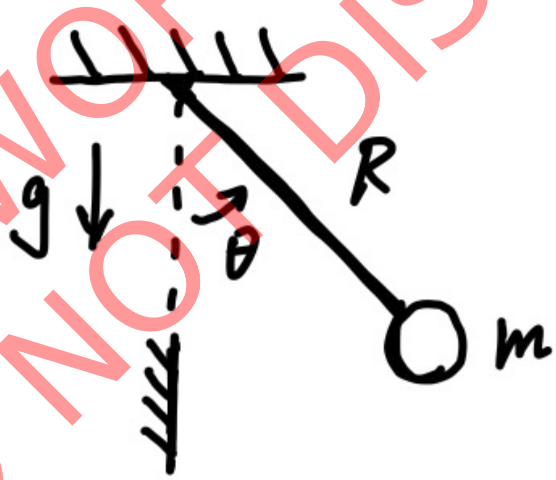
Below are the help functions in previous homeworks, which you may need for this homework.

In [3]:
```python
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
```

```
In [4]: def simulate(f, x0, tspan, dt, integrate):
    """
    This function takes in an initial condition x0, a timestep dt,
    a time span tspan consisting of a list [min_time, max_time],
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x0. It outputs a full trajectory simulated
    over the time span of dimensions (xvec_size, time_vec_size).

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

    Return
    ============
    x_traj:
        simulated trajectory of x(t) from t=0 to tf
    """
    N = int((max(tspan)-min(tspan))/dt)
    x = np.copy(x0)
    tvec = np.linspace(min(tspan),max(tspan),N)
    xtraj = np.zeros((len(x0),N))
    for i in range(N):
        xtraj[:,i]=integrate(f,x,dt)
        x = np.copy(xtraj[:,i])
    return xtraj
```

Problem 1 (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 $\theta = \frac{\pi}{2}, \dot{\theta} = 0$. 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.

In [5]:
```python
# define symbols
t, m, R, g = sym.symbols('t, m, R, g')
theta = sym.Function(r'\theta')(t)
thetadot = theta.diff(t)
thetaddot = thetadot.diff(t)

# define xy position of the pendulum
px = R * sym.sin(theta)
py = -R * sym.cos(theta)
pxdot = px.diff(t)
pydot = py.diff(t)

# Lagrangian
KE = 0.5 * m * (pxdot**2 + pydot**2)
PE = m*g*py
L = KE - PE
print('Lagrangian:')
display(L)

# EL-equation(s)
dLdq = L.diff(theta)
dLdqdot = L.diff(thetadot)
d_dLdqdot_dt = dLdqdot.diff(t)
el_eqns = sym.Eq(d_dLdqdot_dt - dLdq, 0)
print('EL-equations:')
display(el_eqns)

# solve for equations of motion
el_solns = sym.solve(el_eqns, thetaddot, dict=True)
thetaddot_sol = el_solns[0][thetaddot]
print('solution for thetaddot:')
display(thetaddot_sol)

# lambdify
thetaddot_sol = thetaddot_sol.subs({R:1, m:1, g:9.8})
thetaddot_func = sym.lambdify([theta, thetadot], thetaddot_sol)
def pend_dyn(s):
    return np.array([s[1], thetaddot_func(*s)])

# simulate
s0 = np.array([np.pi/2, 0])
print('test pend_dyn(-pi/2, 0): ', pend_dyn(s0))
traj = simulate(pend_dyn, s0, tspan=[0,5], dt=0.01, integrate=integrate)
print('traj.shape: ', traj.shape)

# plot
plt.plot(np.arange(traj.shape[1]), traj[0])

plt.show()
```

Lagrangian:

$$Rgm\cos(\theta(t)) + 0.5m\left(R^2\sin^2(\theta(t))\left(\frac{d}{dt}\theta(t)\right)^2 + R^2\cos^2(\theta(t))\left(\frac{d}{dt}\theta(t)\right)^2\right)$$

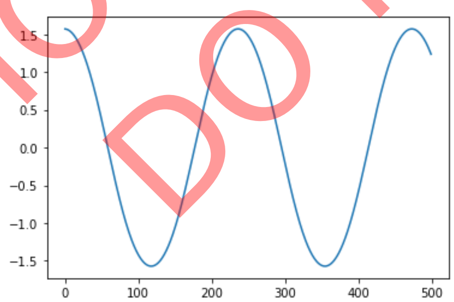
EL-equations:

$$Rgm\sin(\theta(t)) + 0.5m\left(2R^2\sin^2(\theta(t))\frac{d^2}{dt^2}\theta(t) + 2R^2\cos^2(\theta(t))\frac{d^2}{dt^2}\theta(t)\right) = 0$$

solution for thetaddot:

$$-\frac{g\sin(\theta(t))}{R}$$

```
test pend_dyn(-pi/2, 0):  [ 0.  -9.8]
traj.shape:  (2, 500)
```



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:

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

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

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

> 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 subsititute 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).

In [6]:
```
# first define the constraint
phi = theta
dphidq = phi.diff(theta)

# then compute the Hamiltonian
H = dLdqdot * thetadot - L

# define dummy symbols
thetaSym = sym.symbols(r'\theta')
thetadotSym = sym.symbols(r'\dot{\theta}')
thetaddotSym = sym.symbols(r'\ddot{\theta}')
subs_dict = {theta:thetaSym, thetadot:thetadotSym, thetaddot:thetaddotSym}

# substitute
dLdqdot_Sym = dLdqdot.subs(subs_dict).simplify()
dphidq_Sym = dphidq.subs(subs_dict).simplify()
H_Sym = H.subs(subs_dict).simplify()
print('print to check there is no Function(t) left:')
display(dLdqdot_Sym, dphidq_Sym, H_Sym)
```

```
print to check there is no Function(t) left:
```

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

1

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

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, should be function of $q(\tau^-)$ and $\dot{q}(\tau^-)$.

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 [7]:
```python
# define new dummy variables for tau+ and lambda
thetadotSymPlus = sym.symbols(r'\dot{\theta}_{+}')
lamb = sym.symbols(r'\lambda')

# dLdq evaluated at tau+, why we don't need to substitute theta(tau+)?
dLdqdot_SymPlus = dLdqdot_Sym.subs({thetadotSym: thetadotSymPlus})

# H evaluated at tau+
H_SymPlus = H_Sym.subs({thetadotSym: thetadotSymPlus})

# left hand side of the equations
lhs = sym.Matrix([dLdqdot_SymPlus - dLdqdot_Sym, H_SymPlus - H_Sym])
rhs = sym.Matrix([lamb * dphidq_Sym, 0])
impact_eqns = sym.Eq(lhs, rhs)
print('equations for impact update:')
display(impact_eqns.simplify())

# solve it for impact update
impact_solns = sym.solve(impact_eqns, [thetadotSymPlus, lamb], dict=True)
print('impact update rules: thetadot(tau+) = ')
display(impact_solns[0][thetadotSymPlus]) # there are two solutions, one is not useful

# lambdify that solution
# even though in this problem the impact update solution doesn't involve q, but that's
# not always the case, always include it when numerically evaluating the solution.
impact_func = sym.lambdify([thetaSym, thetadotSym], impact_solns[0][thetadotSymPlus])
print( 'test numerically evaluted impact update rule: ', impact_func(0.01, 2.0) )
```

```
equations for impact update:
```

$$\begin{bmatrix} \lambda \\ 0 \end{bmatrix} = \begin{bmatrix} 1.0R^2 m \left(-\dot{\theta} + \dot{\theta}_+\right) \\ 0.5R^2 m \left(-\dot{\theta}^2 + \dot{\theta}_+^2\right) \end{bmatrix}$$

```
impact update rules: thetadot(tau+) =
```

$$-\dot{\theta}$$

```
test numerically evaluted impact update rule:  -2.0
```

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.

1. 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.

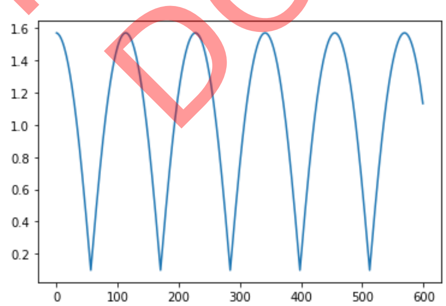
In [8]:

```python
# first, numerically evaluate phi
phi_func = sym.lambdify([theta, thetadot], phi)

# define impact condition
def impact_condition(s, phi_func, threshold):
    if -threshold < phi_func(*s) and phi_func(*s) < threshold:
        return True
    else:
        return False

# define a new simulate function
def simulate_impact_singlepend(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):
        # decide whether impact condition is satisfied
        if impact_condition(x, phi_func, 1e-1) is True:
            print('impact!')
            x[1] = impact_func(*x) # update qdot based on impact rule
            xtraj[:,i]=integrate(f,x,dt) # then simulate/integrate
        else:
            xtraj[:,i]=integrate(f,x,dt)
        x = np.copy(xtraj[:,i])
    return xtraj
```

In [9]:

```python
# test simulation
s0 = np.array([np.pi/2, 0])
print('test pend_dyn(pi/2, 0): ', pend_dyn(s0))
traj = simulate_impact_singlepend(pend_dyn, s0, tspan=[0,6], dt=0.01, integrate=integrate)
print('traj.shape: ', traj.shape)

# plot
plt.plot(np.arange(traj.shape[1]), traj[0])
plt.show()
```

```
test pend_dyn(pi/2, 0):  [ 0.  -9.8]
impact!
impact!
impact!
impact!
impact!
traj.shape:  (2, 600)
```

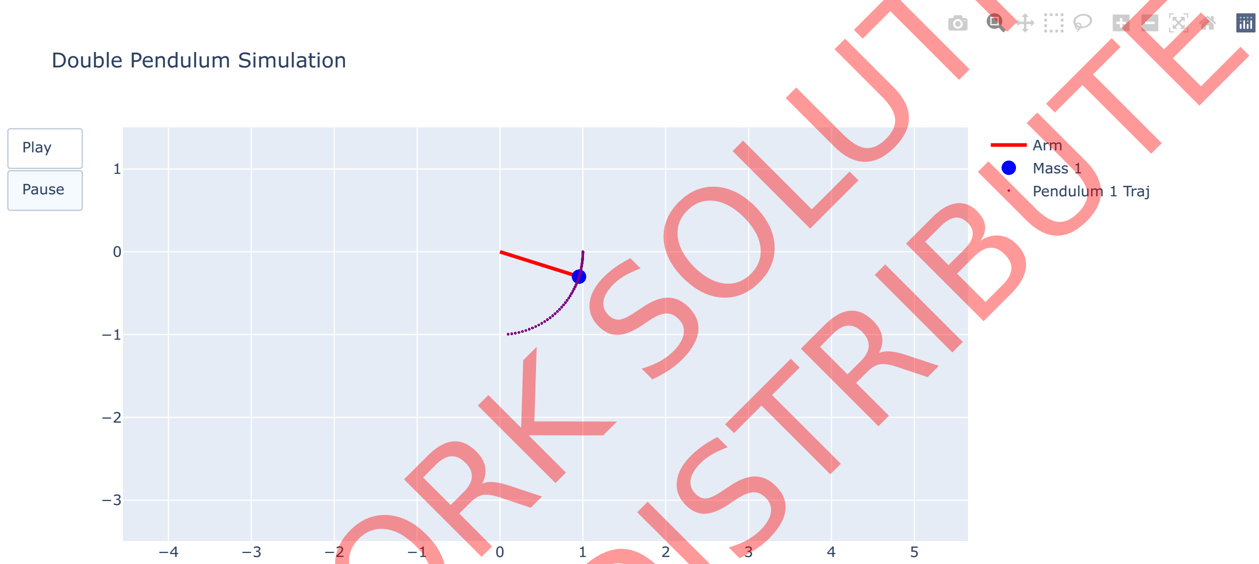


In [10]:
```python
def animate_single_pend(theta_array,L1=1,T=10):
    """
    Function to generate web-based animation of single-pendulum system

    Parameters:
    ================================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first pendulum
    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)
    yy1=-L1*np.cos(theta_array)
    print(yy1.shape)
    N = len(theta_array) # 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='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 [11]:
```
# animate simulation
animate_single_pend(traj[0])
```

```
(600,)
```



Problem 5 (10pts)

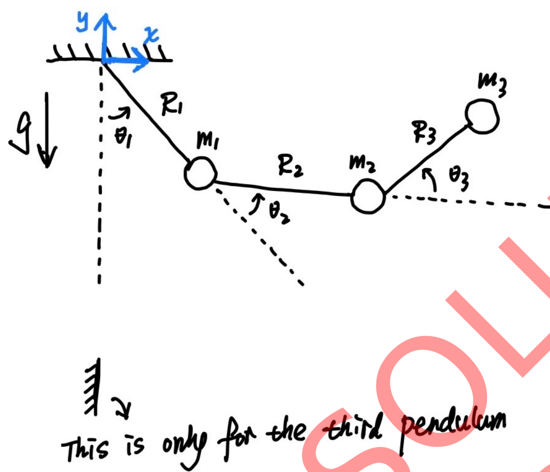
We will now consider a constrained triple-pendulum system with the system configuration $q = [\theta_1, \theta_2, \theta_3]$. A constraint is such that *x coordinate* of the third pendulum (i.e. m_3) ONLY can not be smaller than 0 -- there exist a vertical wall high enough for third pendulum impact. 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 subsitute 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!

In [12]:

```
########################################################################
# this is code for simulating unconstrained triple-pendulum system
########################################################################
# constants
# m1, m2, m3, R1, R2, R3, g = symbols(r'm_1, m_2, m_3, R_1, R_2, R_3, g')
m1 = 1
m2 = 1
m3 = 1
R1 = 1
R2 = 1
R3 = 1
g = 9.8

# state variables
# this way of definition could save the name "theta" for dummy variables later
# you can also just name these by hand as in the last homework solutions
q = sym.Matrix([sym.Function(r'\theta_'+str(i+1))(t) for i in range(3)])
qdot = q.diff(t)
qddot = qdot.diff(t)

# coordinate transfer
x1 = R1 * sym.sin(q[0])
y1 = -R1 * sym.cos(q[0])
x2 = x1 + R2 * sym.sin(q[0]+q[1])
y2 = y1 - R2 * sym.cos(q[0]+q[1])
x3 = x2 + R3 * sym.sin(q[0]+q[1]+q[2])
y3 = y2 - R3 * sym.cos(q[0]+q[1]+q[2])
x1d = x1.diff(t)
y1d = y1.diff(t)
x2d = x2.diff(t)
y2d = y2.diff(t)
x3d = x3.diff(t)
y3d = y3.diff(t)

# Lagrangian
# Rational(1,2) is same as 1/2, but it's better for printing out
KE = sym.Rational(1,2)*m1*(x1d**2+y1d**2) + sym.Rational(1,2)*m2*(x2d**2+y2d**2) + sym.Rational(1,2)*m3*(x3d**2+y3d**2)
V = m1*g*y1 + m2*g*y2 + m3*g*y3
L = KE - V # DO NOT PRINT OUT Lagrangian
L = sym.simplify(L)

# EL-equations
L = sym.Matrix([L])

dLdq = L.jacobian(q)
dLdq = sym.simplify(dLdq)

dLdqdot = L.jacobian(qdot)
dLdqdot = sym.simplify(dLdqdot)

d_dLdqdot_dt = dLdqdot.diff(t)
d_dLdqdot_dt = sym.simplify(d_dLdqdot_dt)

rhs = sym.simplify(d_dLdqdot_dt.T - dLdq.T)
el_eqns = sym.Eq(rhs, sym.Matrix([0, 0, 0]))

# solve EL-equations
el_solns = sym.solve(el_eqns, qddot, dict=True)
```

In [13]:

```
# split solutions
th1ddot = el_solns[0][qddot[0]]
th1ddot_func = sym.lambdify([q[0], q[1], q[2], qdot[0], qdot[1], qdot[2]], th1ddot)
th2ddot = el_solns[0][qddot[1]]
th2ddot_func = sym.lambdify([q[0], q[1], q[2], qdot[0], qdot[1], qdot[2]], th2ddot)
th3ddot = el_solns[0][qddot[2]]
th3ddot_func = sym.lambdify([q[0], q[1], q[2], qdot[0], qdot[1], qdot[2]], th3ddot)

# lambdify
th1ddot_func = sym.lambdify([q[0], q[1], q[2], qdot[0], qdot[1], qdot[2]], th1ddot)
th2ddot_func = sym.lambdify([q[0], q[1], q[2], qdot[0], qdot[1], qdot[2]], th2ddot)
th3ddot_func = sym.lambdify([q[0], q[1], q[2], qdot[0], qdot[1], qdot[2]], th3ddot)

# define dynamics
def dyn_triple_pendulum(s):
    sdot = np.array([
        s[3],
        s[4],
        s[5],
        th1ddot_func(*s),
        th2ddot_func(*s),
        th3ddot_func(*s),
    ])
    return sdot
```

In [14]:
```python
# define dummy variables
th1, th2, th3 = sym.symbols(r'\theta_1, \theta_2, \theta_3')
th1dot, th2dot, th3dot = sym.symbols(r'\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3')
th1ddot, th2ddot, th3ddot = sym.symbols(r'\ddot{\theta}_1, \ddot{\theta}_2, \ddot{\theta}_3')
dummy_dict = {q[0]:th1, q[1]:th2, q[2]:th3,
              qdot[0]:th1dot, qdot[1]:th2dot, qdot[2]:th3dot,
              qddot[0]:th1ddot, qddot[1]:th2ddot, qddot[2]:th3ddot}

# define phi
phi_Sym = sym.Matrix([x3.subs(dummy_dict)])

# compute Hamiltonian
H = dLdqdot * qdot - L
H = sym.simplify(H)

# compute expressions
dLdqdot_Sym = dLdqdot.subs(dummy_dict)
dphidq_Sym = phi_Sym.jacobian([th1,th2,th3])
H_Sym = H.subs(dummy_dict)

#Expression 1: dLdqdot
dLdqdot_Sym = dLdqdot.subs(dummy_dict)
print('Expression 1: dLdqdot')
display(sym.simplify(dLdqdot_Sym).T)

#Expression 2: dphidq
print('Expression 2: dphidq')
display(sym.simplify(dphidq_Sym).T)

#Expression 3: Hamiltonian
print('Expression 3: Hamiltonian')
display(sym.simplify(H_Sym))
```

Expression 1: dLdqdot

$$\begin{bmatrix} 4.0\dot{\theta}_1\cos(\theta_2) + 2\dot{\theta}_1\cos(\theta_3) + 2\dot{\theta}_1\cos(\theta_2+\theta_3) + 6.0\dot{\theta}_1 + 2.0\dot{\theta}_2\cos(\theta_2) + 2.0\dot{\theta}_2\cos(\theta_3) + \dot{\theta}_2\cos(\theta_2+\theta_3) + 3.0\dot{\theta}_2 + \dot{\theta}_3\cos(\theta_3) + \dot{\theta}_3\cos(\theta_2+\theta_3) + \dot{\theta}_3 \\ 2.0\dot{\theta}_1\cos(\theta_2) + 2.0\dot{\theta}_1\cos(\theta_3) + \dot{\theta}_1\cos(\theta_2+\theta_3) + 3.0\dot{\theta}_1 + 2\dot{\theta}_2\cos(\theta_3) + 3.0\dot{\theta}_2 + \dot{\theta}_3\cos(\theta_3) + \dot{\theta}_3 \\ 1.0\dot{\theta}_1\cos(\theta_3) + 1.0\dot{\theta}_1\cos(\theta_2+\theta_3) + 1.0\dot{\theta}_1 + 1.0\dot{\theta}_2\cos(\theta_3) + 1.0\dot{\theta}_2 + 1.0\dot{\theta}_3 \end{bmatrix}$$

Expression 2: dphidq

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

Expression 3: Hamiltonian

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

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).

In [15]:
```python
# define dummy symbols for tau+
lamb = sym.symbols(r'\lambda')
th1dotPlus, th2dotPlus, th3dotPlus = sym.symbols(r'\dot{\theta}_{1+}, \dot{\theta}_{2+}, \dot{\theta}_{3+}')
impact_dict = {th1dot:th1dotPlus, th2dot:th2dotPlus, th3dot:th3dotPlus}

# evaluate the expressions at tau+
dLdqdot_SymPlus = dLdqdot_Sym.subs(impact_dict)
dLdqdot_SymPlus = sym.simplify(dLdqdot_SymPlus)
dphidq_SymPlus = dphidq_Sym.subs(impact_dict)
dphidq_SymPlus = sym.simplify(dphidq_SymPlus)
H_SymPlus = H_Sym.subs(impact_dict)
H_SymPlus = sym.simplify(H_SymPlus)

# define impact equations
# be careful with the dimension of each variable here!
lhs = sym.Matrix([dLdqdot_SymPlus[0]-dLdqdot_Sym[0], dLdqdot_SymPlus[1]-dLdqdot_Sym[1], dLdqdot_SymPlus[2]-dLdqdot_Sym[2], H_
SymPlus[0] - H_Sym[0]])
rhs = sym.Matrix([lamb * dphidq_Sym[0], lamb * dphidq_Sym[1], lamb * dphidq_Sym[2], 0])
impact_eqns = sym.Eq(lhs, rhs)
impact_eqns = sym.simplify(impact_eqns)
```

In [16]:
```python
display(impact_eqns)
```

$$\left[\begin{matrix}\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)}\\0\end{matrix}\right] = \left[\begin{matrix}- 4.0 \dot{\theta}_{1} \cos{\left(\theta_{2} \right)} - 2 \dot{\theta}_{1} \cos{\left(\theta_{3} \right)} - 2 \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)} - \dot{\theta}_{2} \cos{\left(\theta_{2} + \theta_{3} \right)} - 3.0 \dot{\theta}_{2} - \dot{\theta}_{3} \cos{\left(\theta_{3} \right)} - \dot{\theta}_{3} \cos{\left(\theta_{2} + \theta_{3} \right)} - \dot{\theta}_{3} + 4.0 \dot{\theta}_{1+} \cos{\left(\theta_{2} \right)} + 2 \dot{\theta}_{1+} \cos{\left(\theta_{3} \right)} + 2 \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)} + \dot{\theta}_{2+} \cos{\left(\theta_{2} + \theta_{3} \right)} + 3.0 \dot{\theta}_{2+} + \dot{\theta}_{3+} \cos{\left(\theta_{3} \right)} + \dot{\theta}_{3+} \cos{\left(\theta_{2} + \theta_{3} \right)} + \dot{\theta}_{3+}\\- 2.0 \dot{\theta}_{1} \cos{\left(\theta_{2} \right)} - 2.0 \dot{\theta}_{1} \cos{\left(\theta_{3} \right)} - \dot{\theta}_{1} \cos{\left(\theta_{2} + \theta_{3} \right)} - 3.0 \dot{\theta}_{1} - 2 \dot{\theta}_{2} \cos{\left(\theta_{3} \right)} - 3.0 \dot{\theta}_{2} - \dot{\theta}_{3} \cos{\left(\theta_{3} \right)} - \dot{\theta}_{3} + 2.0 \dot{\theta}_{1+} \cos{\left(\theta_{2} \right)} + 2.0 \dot{\theta}_{1+} \cos{\left(\theta_{3} \right)} + \dot{\theta}_{1+} \cos{\left(\theta_{2} + \theta_{3} \right)} + 3.0 \dot{\theta}_{1+} + 2 \dot{\theta}_{2+} \cos{\left(\theta_{3} \right)} + 3.0 \dot{\theta}_{2+} + \dot{\theta}_{3+} \cos{\left(\theta_{3} \right)} + \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} + 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+}\\- 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} + 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}\end{matrix}\right]$$

In [45]:
```python
# reprint each row individually in case wrapping is confusing
for idx in range(4):
    print('Row {}:'.format(idx))
    display(sym.Eq(rhs[idx],lhs[idx]))
```

Row 0:

$$\lambda \left(\cos{\left(\theta_{1} \right)} + \cos{\left(\theta_{1} + \theta_{2} \right)} + \cos{\left(\theta_{1} + \theta_{2} + \theta_{3} \right)}\right) = - 4.0 \dot{\theta}_{1} \cos{\left(\theta_{2} \right)} - 2 \dot{\theta}_{1} \cos{\left(\theta_{3} \right)} - 2 \dot{\theta}_{1} \cos{\left(\theta_{2} + \theta_{3} \right)} - 6.0 \dot{\theta}_{1} - 2.0 \dot{\theta}_{2} \cos{\left(\theta_{2} \right)} - 2.0 \dot{\theta}_{2} \cos{\left(\theta_{3} \right)} - \dot{\theta}_{2} \cos{\left(\theta_{2} + \theta_{3} \right)} - 3.0 \dot{\theta}_{2} - \dot{\theta}_{3} \cos{\left(\theta_{3} \right)} - \dot{\theta}_{3} \cos{\left(\theta_{2} + \theta_{3} \right)} - \dot{\theta}_{3} + 4.0 \dot{\theta}_{1+} \cos{\left(\theta_{2} \right)} + 2 \dot{\theta}_{1+} \cos{\left(\theta_{3} \right)} + 2 \dot{\theta}_{1+} \cos{\left(\theta_{2} + \theta_{3} \right)} + 6.0 \dot{\theta}_{1+} + 2.0 \dot{\theta}_{2+} \cos{\left(\theta_{2} \right)} + 2.0 \dot{\theta}_{2+} \cos{\left(\theta_{3} \right)} + \dot{\theta}_{2+} \cos{\left(\theta_{2} + \theta_{3} \right)} + 3.0 \dot{\theta}_{2+} + \dot{\theta}_{3+} \cos{\left(\theta_{3} \right)} + \dot{\theta}_{3+} \cos{\left(\theta_{2} + \theta_{3} \right)} + \dot{\theta}_{3+}$$

Row 1:

$$\lambda \left(\cos{\left(\theta_{1} + \theta_{2} \right)} + \cos{\left(\theta_{1} + \theta_{2} + \theta_{3} \right)}\right) = - 2.0 \dot{\theta}_{1} \cos{\left(\theta_{2} \right)} - 2.0 \dot{\theta}_{1} \cos{\left(\theta_{3} \right)} - \dot{\theta}_{1} \cos{\left(\theta_{2} + \theta_{3} \right)} - 3.0 \dot{\theta}_{1} - 2 \dot{\theta}_{2} \cos{\left(\theta_{3} \right)} - 3.0 \dot{\theta}_{2} - \dot{\theta}_{3} \cos{\left(\theta_{3} \right)} - \dot{\theta}_{3} + 2.0 \dot{\theta}_{1+} \cos{\left(\theta_{2} \right)} + 2.0 \dot{\theta}_{1+} \cos{\left(\theta_{3} \right)} + \dot{\theta}_{1+} \cos{\left(\theta_{2} + \theta_{3} \right)} + 3.0 \dot{\theta}_{1+} + 2 \dot{\theta}_{2+} \cos{\left(\theta_{3} \right)} + 3.0 \dot{\theta}_{2+} + \dot{\theta}_{3+} \cos{\left(\theta_{3} \right)} + \dot{\theta}_{3+}$$

Row 2:

$$\lambda \cos{\left(\theta_{1} + \theta_{2} + \theta_{3} \right)} = - 1.0 \dot{\theta}_{1} \cos{\left(\theta_{3} \right)} - 1.0 \dot{\theta}_{1} \cos{\left(\theta_{2} + \theta_{3} \right)} - 1.0 \dot{\theta}_{1} - 1.0 \dot{\theta}_{2} \cos{\left(\theta_{3} \right)} - 1.0 \dot{\theta}_{2} - 1.0 \dot{\theta}_{3} + 1.0 \dot{\theta}_{1+} \cos{\left(\theta_{3} \right)} + 1.0 \dot{\theta}_{1+} \cos{\left(\theta_{2} + \theta_{3} \right)} + 1.0 \dot{\theta}_{1+} + 1.0 \dot{\theta}_{2+} \cos{\left(\theta_{3} \right)} + 1.0 \dot{\theta}_{2+} + 1.0 \dot{\theta}_{3+}$$

Row 3:

$$0 = - 2.0 \dot{\theta}_{1}^{2} \cos{\left(\theta_{2} \right)} - 1.0 \dot{\theta}_{1}^{2} \cos{\left(\theta_{3} \right)} - 1.0 \dot{\theta}_{1}^{2} \cos{\left(\theta_{2} + \theta_{3} \right)} - 3.0 \dot{\theta}_{1}^{2} - 2.0 \dot{\theta}_{1} \dot{\theta}_{2} \cos{\left(\theta_{2} \right)} - 2.0 \dot{\theta}_{1} \dot{\theta}_{2} \cos{\left(\theta_{3} \right)} - 1.0 \dot{\theta}_{1} \dot{\theta}_{2} \cos{\left(\theta_{2} + \theta_{3} \right)} - 3.0 \dot{\theta}_{1} \dot{\theta}_{2} - 1.0 \dot{\theta}_{1} \dot{\theta}_{3} \cos{\left(\theta_{3} \right)} - 1.0 \dot{\theta}_{1} \dot{\theta}_{3} \cos{\left(\theta_{2} + \theta_{3} \right)} - 1.0 \dot{\theta}_{1} \dot{\theta}_{3} - 1.0 \dot{\theta}_{2}^{2} \cos{\left(\theta_{3} \right)} - 1.5 \dot{\theta}_{2}^{2} - 1.0 \dot{\theta}_{2} \dot{\theta}_{3} \cos{\left(\theta_{3} \right)} - 1.0 \dot{\theta}_{2} \dot{\theta}_{3} - 0.5 \dot{\theta}_{3}^{2} + 2.0 \dot{\theta}_{1+}^{2} \cos{\left(\theta_{2} \right)} + 1.0 \dot{\theta}_{1+}^{2} \cos{\left(\theta_{3} \right)} + 1.0 \dot{\theta}_{1+}^{2} \cos{\left(\theta_{2} + \theta_{3} \right)} + 3.0 \dot{\theta}_{1+}^{2} + 2.0 \dot{\theta}_{1+} \dot{\theta}_{2+} \cos{\left(\theta_{2} \right)} + 2.0 \dot{\theta}_{1+} \dot{\theta}_{2+} \cos{\left(\theta_{3} \right)} + 1.0 \dot{\theta}_{1+} \dot{\theta}_{2+} \cos{\left(\theta_{2} + \theta_{3} \right)} + 3.0 \dot{\theta}_{1+} \dot{\theta}_{2+} + 1.0 \dot{\theta}_{1+} \dot{\theta}_{3+} \cos{\left(\theta_{3} \right)} + 1.0 \dot{\theta}_{1+} \dot{\theta}_{3+} \cos{\left(\theta_{2} + \theta_{3} \right)} + 1.0 \dot{\theta}_{1+} \dot{\theta}_{3+} + 1.0 \dot{\theta}_{2+}^{2} \cos{\left(\theta_{3} \right)} + 1.5 \dot{\theta}_{2+}^{2} + 1.0 \dot{\theta}_{2+} \dot{\theta}_{3+} \cos{\left(\theta_{3} \right)} + 1.0 \dot{\theta}_{2+} \dot{\theta}_{3+} + 0.5 \dot{\theta}_{3+}^{2}$$

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.

In [17]:
```
def impact_update_triple_pend(s, impact_eqns, sym_list):
    subs_dict = {m1:1, m2:1, m3:1, R1:1, R2:1, R3:1, g:9.8,
                 th1:s[0], th2:s[1], th3:s[2],
                 th1dot:s[3], th2dot:s[4], th3dot:s[5]}
    new_impact_eqns = impact_eqns.subs(subs_dict)
    impact_solns = sym.solve(new_impact_eqns, [th1dotPlus, th2dotPlus, th3dotPlus, lamb], dict=True)
    if len(impact_solns) == 1:
        print("Damn only one solution ...")
    else:
        for sol in impact_solns:
            lamb_sol = sol[lamb]
            if abs(lamb_sol) < 1e-06:
                pass # it means it's a false solution
            else:
                return np.array([
                    s[0],
                    s[1],
                    s[2],
                    float(sym.N(sol[sym_list[0]])),
                    float(sym.N(sol[sym_list[1]])),
                    float(sym.N(sol[sym_list[2]])),
                ])

s_test = np.array([0.0, 0.0, 0.0, -1.0, -1.0, -1.0])
impact_update_triple_pend(s_test, impact_eqns, [th1dotPlus, th2dotPlus, th3dotPlus])
```

Out[17]:
```
array([ 0.,  0.,  0., -1., -1., 11.])
```

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.

In [18]:
```
# define impact condition function
phi_func = sym.lambdify([th1, th2, th3, th1dot, th2dot, th3dot], phi_Sym)
def impact_condition_triple_pend(s, phi_func, threshold):
    if phi_func(*s) < threshold and phi_func(*s) > -threshold:
        return True
    else:
        return False
print('test impact condition function:', impact_condition_triple_pend([0.0, 0.0, 0.01, 0.2, 0.2, 0.2], phi_func, 0.01))
```

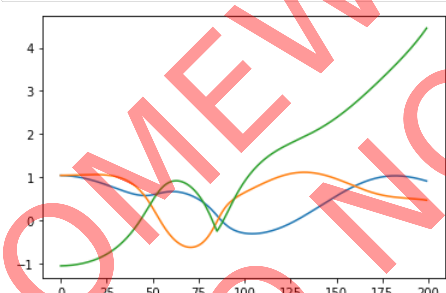
```
test impact condition function: True
```

In [19]:
```python
# define a new simulate function
def simulate_impact_triple_pend(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):
        # decide whether impact condition is satisfied
        if impact_condition_triple_pend(x, phi_func, 1e-1) is True:
            print('impact!', i)
            x = impact_update_triple_pend(x, impact_eqns, [th1dotPlus, th2dotPlus, th3dotPlus])
            xtraj[:,i]=integrate(f,x,dt) # then simulate/integrate
        else:
            xtraj[:,i]=integrate(f,x,dt)
        x = np.copy(xtraj[:,i])
    return xtraj
```

In [20]:
```python
# simulate
s0 = np.array([np.pi/3, np.pi/3, -np.pi/3, 0.0, 0.0, 0.0])
traj = simulate_impact_triple_pend(dyn_triple_pendulum, s0, tspan=[0,2], dt=0.01, integrate=integrate)
```

```
impact! 86
```

In [21]:
```python
# plot
import matplotlib.pyplot as plt
plt.plot(np.arange(traj.shape[1]), traj[0:3].T)
plt.show()
```

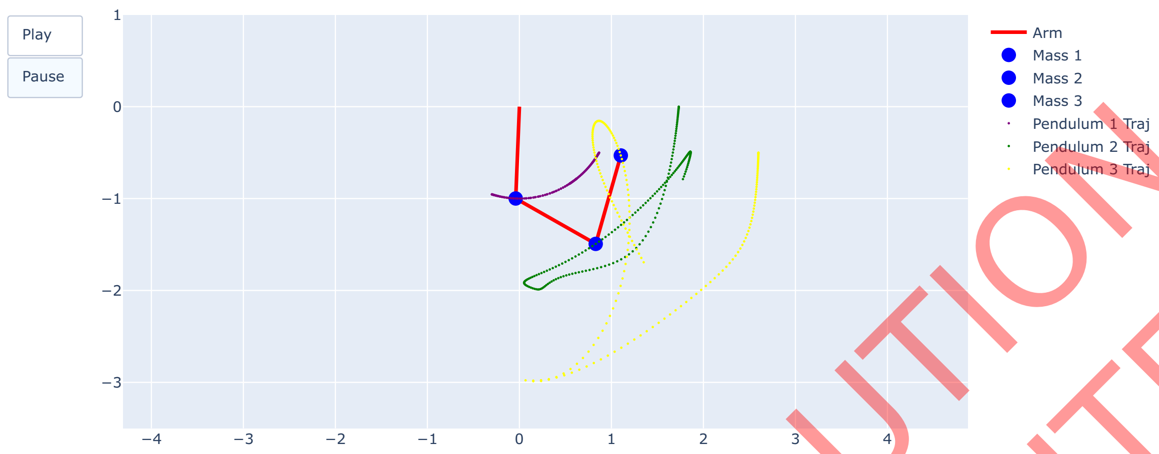


In [22]:
```python
# animate
def animate_double_pend(theta_array, L1=1, L2=1, L3=1, T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ================================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (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)
              ),
```

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

    ########################################
    # Defining the frames of the simulation.
    # This is what draws the lines from
    # joint to joint of the pendulum.
    frames=[dict(data=[dict(x=[0,xx1[k],xx2[k],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)

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

# # provide a trajectory of double-pendulum
# # (note that this array below is not an actual simulation,
# # but lets you see this animation code work)
# import numpy as np
# sim_traj = np.array([np.linspace(-1, 1, 100) for _ in range(3)])
# print('shape of trajectory: ', sim_traj.shape)

# # second, animate!
# animate_double_pend(sim_traj, L1=1, L2=1, L3=1, T=10)
```

In [23]:
```
animate_double_pend(traj, L1=1, L2=1, L3=1, T=10)
```

Double Pendulum Simulation



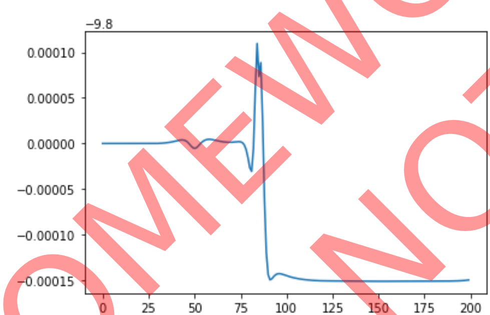
Problem 9 (5pts)

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 [24]:
```
H_func = sym.lambdify([th1, th2, th3, th1dot, th2dot, th3dot], H_Sym)
H_list = []
for s in traj.T:
    H_list.append(H_func(*s)[0][0])
plt.plot(np.arange(len(H_list)), H_list)

# for t in [94, 261, 301, 411, 435, 479]:
#     plt.axvline(t, linestyle='--', color='r')
```

Out[24]:
```
[<matplotlib.lines.Line2D at 0x7f61d16c6fd0>]
```



In [ ]: