ME314 Homework 3 (Template)

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

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

In [1]:

```
# ##################################################################################################
# # If you're using Google Colab, uncomment this section by selecting the whole section and press
# # ctrl+'/' on your and keyboard. Run it before you start programming, this will enable the nice
# # LaTeX "display()" function for you. If you're using the local Jupyter environment, leave it alone
# ##################################################################################################

# import sympy as sym
# def custom_latex_printer(exp,**options):
#     from google.colab.output._publish import javascript
#     url = "https://cdnjs.cloudflare.com/ajax/libs/mathjax/3.1.1/latest.js?config=TeX-AMS_HTML"
#     javascript(url=url)
#     return sym.printing.latex(exp,**options)
# sym.init_printing(use_latex="mathjax",latex_printer=custom_latex_printer)
```

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

In [2]:

```python
import numpy as np

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

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

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

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

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

    Return
    ============
    x_traj:
        simulated trajectory of x(t) from t=0 to tf
    """
    N = int((max(tspan)-min(tspan))/dt)
    x = np.copy(x0)
    tvec = np.linspace(min(tspan),max(tspan),N)
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ================================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first pendulum
    L2:
        length of the second pendulum
    T:
        length/seconds of animation duration

    Returns: None
    """

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

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

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

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

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

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

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

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

Problem 1 (10pts)

Let $f : \mathbb{R}^2 \to \mathbb{R}$ with $f(x, y) = -\cos(x + y)\cos(x - y)$. Show that $(x, y) = (0, 0)$ satisfies both the necessary and sufficient conditions to be a local minimizer of f.

Hint 1: You will need to take the first- and second-order derivative of f with respect to $[x, y]$.

Turn in: A scanned (or photograph from your phone or webcam) copy of your hand written solution. You can also use LaTeX. If you use SymPy, then you just need to include a copy of code and the code outputs, with notes that explain why the code outputs indicate the necessary and sufficient conditions.

In [187…

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

x = symbols('x')
y = symbols('y')

q = Matrix([x, y])

f = -cos(x+y)*cos(x-y)
f = Matrix([f])
fdot = f.jacobian(q).T
fddot = fdot.jacobian(q).T

print('==================================================')
print()
print('first derivative of f(x,y): ')
display(fdot)


print('==================================================')
print()
print('second derivative of f(x,y): ')
display(fddot)

eqn1 = Eq(fdot, Matrix([0, 0]))
eqn2 = Eq(fddot, Matrix([0, 0]))

eqn1_solve = solve(eqn1, Matrix([0, 0]))
eqn2_solve = solve(eqn2, Matrix([0, 0]))

print('==================================================')
print()
print('solving first derivative of f(x,y) at (x,y) = (0,0): ')
display(eqn1_solve)

print('==================================================')
print()
print('solving second derivative of f(x,y) at (x,y) = (0,0): ')
display(eqn2_solve)
```

```
==================================================

first derivative of f(x,y):
```

$$\begin{bmatrix} \sin(x-y)\cos(x+y) + \sin(x+y)\cos(x-y) \\ -\sin(x-y)\cos(x+y) + \sin(x+y)\cos(x-y) \end{bmatrix}$$

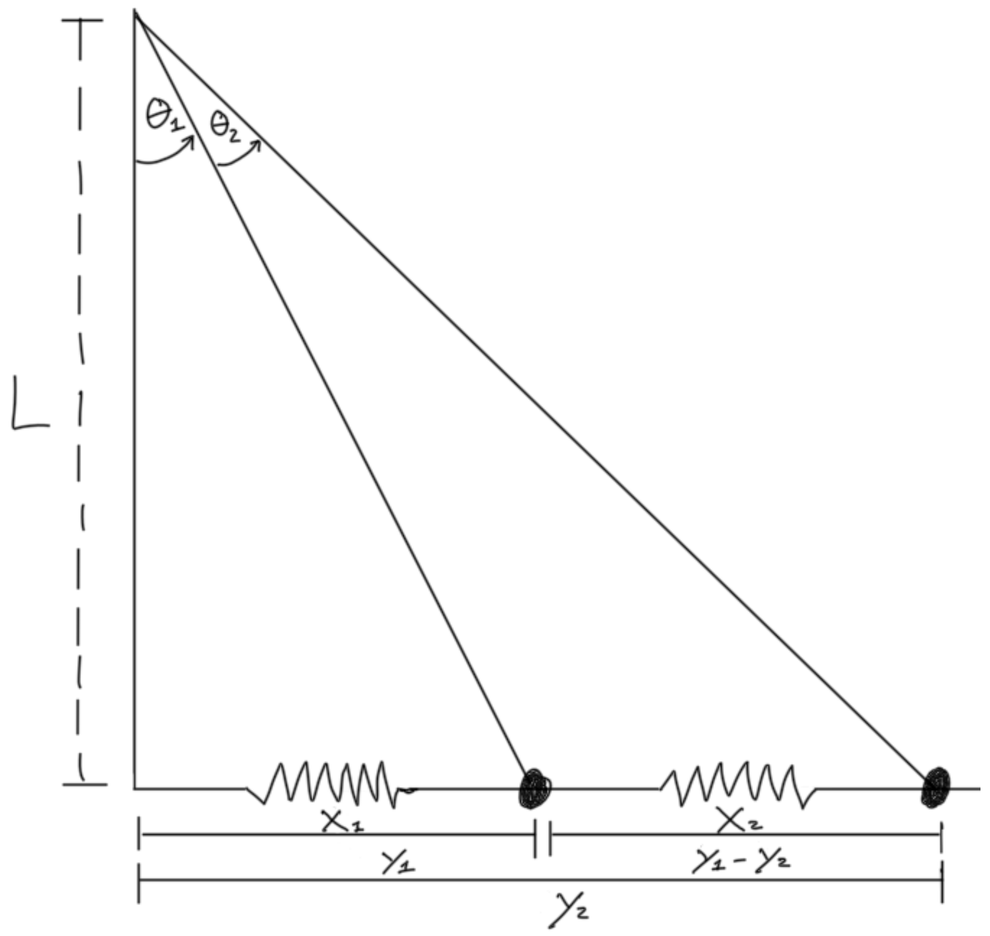
```
==================================================

second derivative of f(x,y):
```

$$\begin{bmatrix} -2\sin(x-y)\sin(x+y) + 2\cos(x-y)\cos(x+y) & 0 \\ 0 & 2\sin(x-y)\sin(x+y) + 2\cos(x-y)\cos(x+y) \end{bmatrix}$$

```
==================================================

solving first derivative of f(x,y) at (x,y) = (0,0):
{0: 0}
==================================================
```

In [3]:

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



Problem 2 (20pts)

Compute the equations of motion for the two-mass-spring system (shown above) in $\theta = (\theta_1, \theta_2)$ coordinates. The first mass with mass m_1 is the one close to the wall, and the second mass is with mass m_2. Assume that there is a spring of spring constant k_1 between the first mass and the wall and a spring of spring constant k_2 between the first mass and the second mass.

Turn in: A copy of the code used to symbolically solve for the equations of motion, also include the outputs of the code, which should be the equations of motion.

In [4]:

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

m1, m2, L, k1, k2 = symbols(r'm_1, m_2, L, k_1, k_2')
th1 = Function(r'\theta_1')(t)
th2 = Function(r'\theta_2')(t)


x1 = L*tan(th1)
x2 = L*tan(th1+th2) + x1
x1dot = x1.diff(t)
x2dot = x2.diff(t)

KE = 0.5*m1*x1dot**2 + 0.5*m2*x2dot**2
PE = 0.5*k1*x1**2 + 0.5*k2*(L*tan(th1+th2) - x1)**2

L = KE - PE
L = Matrix([L])

#make vector q containing theta1 and theta2. This will make deriving the Jacobian easier
q = Matrix([th1, th2])
qdot = q.diff(t)
qddot = qdot.diff(t)


#take derivative of the Lagrangian wrt q = [theta1, theta2]
dLdq = L.jacobian(q).T

#symbolically compute dL/dq_dot terms of Euler-Lagrange equations
dLdq_dot = L.jacobian(qdot).T

#take time derivative of dL/dq_dot
ddt_dL_dqdot = dLdq_dot.diff(t)

#combine the previous terms to get lhs of Euler-Langrange equations
eL = Eq(dLdq - ddt_dL_dqdot, Matrix([0, 0]))


#symbolically solve for the x_ddot and t_ddot variables contained in vector q_ddot
eL_solved = solve(eL, [qddot[0], qddot[1]]) #symbollically solve for each term in the E-L vector



#display the solution for x_ddot and t_ddot
print('=================================================')
print()
print('Equations of motion: ')
for a in eL_solved:
    print()
    display(Eq(a, eL_solved[a]))
```

```
=================================================

Equations of motion: 
```

$$\frac{d^2}{dt^2}\theta_1(t) = -\frac{k_1\tan\left(\theta_1(t)\right)}{m_1\tan^2\left(\theta_1(t)\right)+m_1} + \frac{2.0k_2\tan\left(\theta_1(t)+\theta_2(t)\right)}{m_1\tan^2\left(\theta_1(t)\right)+m_1} - \frac{2.0k_2\tan\left(\theta_1(t)\right)}{m_1\tan^2\left(\theta_1(t)\right)+m_1} - \frac{2.0m_1\tan^3\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1\tan^2\left(\theta_1(t)\right)+m_1} - \frac{2.0m_1\tan\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1\tan^2\left(\theta_1(t)\right)+m_1}$$

$$\frac{d^2}{dt^2}\theta_1(t) = -\frac{k_1 \tan\left(\theta_1(t)\right)}{m_1 \tan^2\left(\theta_1(t)\right) + m_1} + \frac{2.0 k_2 \tan\left(\theta_1(t) + \theta_2(t)\right)}{m_1 \tan^2\left(\theta_1(t)\right) + m_1} - \frac{2.0 k_2 \tan\left(\theta_1(t)\right)}{m_1 \tan^2\left(\theta_1(t)\right) + m_1} - \frac{2.0 m_1 \tan^3\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1 \tan^2\left(\theta_1(t)\right) + m_1}$$
$$- \frac{2.0 m_1 \tan\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1 \tan^2\left(\theta_1(t)\right) + m_1}$$

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{k_1 m_2 \tan^2\left(\theta_1(t) + \theta_2(t)\right)\tan\left(\theta_1(t)\right)}{m_1 m_2 \tan^2\left(\theta_1(t) + \theta_2(t)\right)\tan^2\left(\theta_1(t)\right) + m_1 m_2 \tan^2\left(\theta_1(t) + \theta_2(t)\right) + m_1 m_2 \tan^2\left(\theta_1(t)\right) + m_1 m_2} + \frac{k_1 m_2 \tan^3\left(\theta_1(t)\right)}{m_1 m_2 \tan^2\left(\theta_1(t) + \theta_2(t)\right)\tan^2\left(\theta_1(t)\right) + m_1 m_2 \tan^2\left(\theta_1(t) + \theta_2(t)\right) + m_1 m_2 \tan^2\left(\theta_1(t)\right) + m_1 m_2} + \frac{2.0k}{m_1 m_2 \tan^2\left(\theta_1(t) + \theta_2(t)\right)\tan^2\left(\theta_1(t)\right) + m}$$

$$
\begin{aligned}
\frac{d^2}{dt^2}\theta_2(t) = {} & \frac{k_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& + \frac{k_1 m_2 \tan^3(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& + \frac{2.0 k_1 m_2 \tan(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{k_2 m_1 \tan(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{k_2 m_1 \tan(\theta_1(t)+\theta_2(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& + \frac{k_2 m_1 \tan^3(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& + \frac{k_2 m_1 \tan(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{2.0 k_2 m_2 \tan^3(\theta_1(t)+\theta_2(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& + \frac{2.0 k_2 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{2.0 k_2 m_2 \tan(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{4.0 k_2 m_2 \tan(\theta_1(t)+\theta_2(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& + \frac{2.0 k_2 m_2 \tan^3(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& + \frac{4.0 k_2 m_2 \tan(\theta_1(t))}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{2.0 m_1 m_2 \tan^3(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{4.0 m_1 m_2 \tan^3(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t)}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{2.0 m_1 m_2 \tan^3(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))\left(\frac{d}{dt}\theta_2(t)\right)^2}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2} \\
& - \frac{2.0 m_1 m_2 \tan^3(\theta_1(t)+\theta_2(t))\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t)) + m_1 m_2 \tan^2(\theta_1(t)+\theta_2(t)) + m_1 m_2 \tan^2(\theta_1(t)) + m_1 m_2}
\end{aligned}
$$

Problem 3 (10pts)

For the same two-spring-mass system in Problem 2, show by example that Newton's equations do not hold in an arbitrary choice of coordinates (but they do, of course, hold in Cartesian coordinates). Your example should be implemented using Python's SymPy package.

Hint 1: In other words, you need to find a set of coordinates $q = [q_1, q_2]$*, and compute the equations of motion* $(F = ma = m\ddot{q})$*, showing that these equations of motion do not make the same prediction as Newton's laws in the Cartesian inertially fixed frame (where they are correct).*

Hint 2: Newton's equations don't hold in non-inertia coordinates. For the x_1, x_2 *and* y_1, y_2 *coordinates shown in the image, one of them is non-inertia coordinate.*

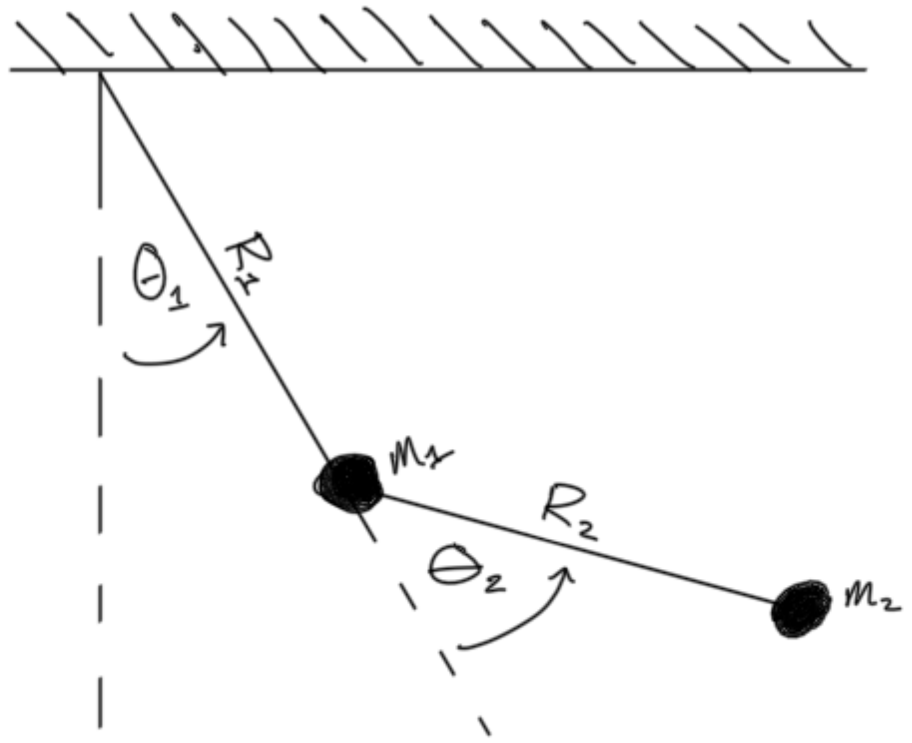
Turn in: A copy of code you used to symbolically compute the equations of motion to show that Newton's equations don't hold. Also, include the output of the code, which should be the equations of motion under the chosen set of coordinates (indicate what coordinate you choose in the comments).

In [ ]:

```
```

In [5]:

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



Problem 4 (10pts)

For the same double-pendulum system hanging in gravity in Homework 2, take $q = [\theta_1, \theta_2]$ as the system configuration variables, with $R_1 = R_2 = 1, m_1 = m_2 = 1$. Symbolically compute the Hamiltonian of this system using Python's SymPy package.

Turn in: A copy of the code used to symbolically compute the Hamiltonian of the system, also include the output of the code, which should the Hamiltonian of the system.

In [ ]:

In [7]:

```
from sympy import Eq, Matrix, Inverse, Identity
import sympy as sym

m1 = symbols('m_1')
m2 = symbols('m_2')
r1 = symbols('R_1')
r2 = symbols('R_2')
g = symbols('g')
l = symbols('L')
h = symbols('H')

th1 = Function(r'\theta_1')(t)
th2 = Function(r'\theta_2')(t)
dth1 = th1.diff(t)
dth2 = th2.diff(t)
ddth1 = dth1.diff(t)
ddth2 = dth2.diff(t)

x1 = r1*sym.sin(th1)
y1 = -r1*sym.cos(th1)
x2 = x1 + r2*sym.sin(th1 + th2)
y2 = y1 + r2*-sym.cos(th1 + th2)

x1dot = x1.diff(t)
y1dot = y1.diff(t)
x2dot = x2.diff(t)
y2dot = y2.diff(t)

k = 0.5*m1*(x1dot**2 + y1dot**2) + 0.5*m2*(x2dot**2 + y2dot**2)
p = m1*g*r1*cos(th1) + m2*g*(r1*cos(th1) + r2*cos(th1 + th2))
L = k - p

print("The Lagrange equation: L = KE - PE: ")
display(Eq(l, L))

L = Matrix([L])
q = Matrix([th1, th2])
qdot = q.diff(t)
qddot = qdot.diff(t)
# display(L)

#p = dL/dxdot
p = L.jacobian(qdot)
# display(p)

# display(p*qdot)

# compute the Hamiltonian: H = 0.5*p*I^(-1)*p^T + PE
H = p*qdot - L
H = H.subs({m1:1, m2:1, r1:1, r2:1, g:9.8})
print('==================================================')
print()
print("The Hamiltonian equation: H = p*qdot - L: ")
display(Eq(h, H[0]))
```

```
The Lagrange equation: L = KE - PE:
```

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

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

```
==================================================

The Hamiltonian equation: H = p*qdot - L:
```

$H = 0.5\Big(2\Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right)\sin(\theta_1(t) + \theta_2(t)) + \sin(\theta_1(t))\frac{d}{dt}\theta_1(t)\Big)\sin(\theta_1(t) + \theta_2(t)) + 2\Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right)\cos(\theta_1(t) + \theta_2(t)) + \cos(\theta_1(t))\frac{d}{dt}\theta_1(t)\Big)\cos(\theta_1(t) + \theta_2(t))\Big)\frac{d}{dt}\theta_2(t) - 0.5\Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right)\sin(\theta_1(t) + \theta_2(t)) + \sin(\theta_1(t))\frac{d}{dt}\theta$

$$H$$
$$= 0.5 \Big(2 \Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \sin(\theta_1(t) + \theta_2(t)) + \sin(\theta_1(t)) \frac{d}{dt}\theta_1(t) \Big) \sin(\theta_1(t) + \theta_2(t))$$
$$+ 2 \Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \cos(\theta_1(t) + \theta_2(t)) + \cos(\theta_1(t)) \frac{d}{dt}\theta_1(t) \Big) \cos(\theta_1(t) + \theta_2(t)) \Big) \frac{d}{dt}\theta_2(t)$$
$$- 0.5 \Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \sin(\theta_1(t) + \theta_2(t)) + \sin(\theta_1(t)) \frac{d}{dt}\theta_1(t) \Big)^2 - 0.5 \Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \cos(\theta_1(t) + \theta_2(t)) + \cos(\theta_1(t)) \frac{d}{dt}\theta_1(t) \Big)^2$$
$$+ \Big(0.5 \Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \sin(\theta_1(t) + \theta_2(t)) + \sin(\theta_1(t)) \frac{d}{dt}\theta_1(t) \Big) (2 \sin(\theta_1(t) + \theta_2(t)) + 2 \sin(\theta_1(t)))$$
$$+ 0.5 \Big(\left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right) \cos(\theta_1(t) + \theta_2(t)) + \cos(\theta_1(t)) \frac{d}{dt}\theta_1(t) \Big) (2 \cos(\theta_1(t) + \theta_2(t)) + 2 \cos(\theta_1(t))) + 1.0 \sin^2(\theta_1(t)) \frac{d}{dt}\theta_1(t) + 1.0 \cos^2$$
$$(\theta_1(t)) \frac{d}{dt}\theta_1(t) \Big) \frac{d}{dt}\theta_1(t) - 0.5 \sin^2(\theta_1(t)) \left(\frac{d}{dt}\theta_1(t)\right)^2 + 9.8 \cos(\theta_1(t) + \theta_2(t)) - 0.5 \cos^2(\theta_1(t)) \left(\frac{d}{dt}\theta_1(t)\right)^2 + 19.6 \cos(\theta_1(t))$$

Problem 5 (10pts)

Simulate the double-pendulum system in Problem 4 with initial condition $\theta_1 = \theta_2 = -\frac{\pi}{2}, \dot{\theta}_1 = \dot{\theta}_2 = 0$ for $t \in [0, 10]$ and $dt = 0.01$. Numerically evaluate the Hamiltonian of this system from the simulated trajectory, and plot it.

Hint 1: The Hamiltonian can be numerically evaluated as a function of $\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2$, which means for each time step in the simulated trajectory, you can compute the Hamiltonian for this time step, and store it in a list or array for plotting later. This doesn't need to be done during the numerical simulation, after you have the simulated the trajectory you can access each time step within another loop.

Turn in: A copy of the code used to numerically evaluate and plot the Hamiltonian, also include the output of the code, which should be the plot of Hamiltonian.

In [56]:

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

# k = 0.5*m1*(x1dot**2 + y1**2) + 0.5*m2*(x2dot**2 + y2dot**2)
# p = m1*g*r1*cos(th1) + m2*g*(r1*cos(th1) + r2*cos(th1 + th2))
# L = k - p
# L = Matrix([L])

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

el = Eq(sym.simplify(d_dLdqdot_dt - dLdq), Matrix([0 for _ in range(q.shape[0])]))
print('==================================================')
print()
print("Euler Lagrange equations for double pendulum system")
display(el)

#solve for the equations of motion:
el_soln = solve(el, qddot)
# soln_th1ddot = el_soln[qddot[0]]
# soln_th2ddot = el_soln[qddot[1]]
for i in qddot:
    print('==================================================')
    print()
    display(Eq(i, el_soln[i]))
print('==================================================')
print()

th1ddot_soln = el_soln[qddot[0]].subs({m1:1, m2:1, r1:1, r2:1, g:9.8})
th2ddot_soln = el_soln[qddot[1]].subs({m1:1, m2:1, r1:1, r2:1, g:9.8})

th1ddot_func = sym.lambdify([q[0], q[1], qdot[0], qdot[1]], th1ddot_soln)
th2ddot_func = sym.lambdify([q[0], q[1], qdot[0], qdot[1]], th2ddot_soln)

def dyn(s):
    return np.array([
        s[2],
        s[3],
        th1ddot_func(*s),
        th2ddot_func(*s),

    ])

#initial condition
s0 = np.array([-pi/2, pi/2, 0, 0])

traj = simulate(dyn, s0, [0,10], 0.01, integrate)[0:4]
print('shape of traj: ', traj.shape)

plt.title("plot of equations of motion")
plt.plot(np.arange(1000), traj[0])
plt.plot(np.arange(1000), traj[1])
plt.show()
```

```
==================================================

Euler Lagrange equations for double pendulum system
```

$$\left[\begin{matrix}R_1^2 m_1 \frac{d^2}{dt^2}\theta_1(t) - R_1 g m_1 \sin\left(\theta_1(t)\right) - g m_2\left(R_1\sin\left(\theta_1(t)\right) + R_2\sin\left(\theta_1(t)+\theta_2(t)\right)\right) + m_2\left(R_1^2\frac{d^2}{dt^2}\theta_1(t) - 2R_1R_2\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - R_1R_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 2R_1R_2\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_1(t) + R_1R_2\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_2(t) + R_2^2\frac{d^2}{dt^2}\theta_1(t) + R_2^2\frac{d^2}{dt^2}\theta_2(t)\right)\\ 1.0R_2m_2\left(R_1\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + R_1\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_1(t) + R_2\frac{d^2}{dt^2}\theta_1(t) + R_2\frac{d^2}{dt^2}\theta_2(t) - g\sin\left(\theta_1(t)+\theta_2(t)\right)\right)\end{matrix}\right]$$

$$\left[\begin{matrix}R_1^2 m_1 \frac{d^2}{dt^2}\theta_1(t) - R_1 g m_1 \sin\left(\theta_1(t)\right) - g m_2\left(R_1\sin\left(\theta_1(t)\right) + R_2\sin\left(\theta_1(t)+\theta_2(t)\right)\right) \\ + m_2\left(R_1^2\frac{d^2}{dt^2}\theta_1(t) - 2R_1R_2\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - R_1R_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 2R_1R_2\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_1(t) + R_1R_2\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_2(t)\right. \\ \left. + R_2^2\frac{d^2}{dt^2}\theta_1(t) + R_2^2\frac{d^2}{dt^2}\theta_2(t)\right) \\ 1.0R_2m_2\left(R_1\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + R_1\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_1(t) + R_2\frac{d^2}{dt^2}\theta_1(t) + R_2\frac{d^2}{dt^2}\theta_2(t) - g\sin\left(\theta_1(t)+\theta_2(t)\right)\right)\end{matrix}\right] = \left[\begin{matrix}0\\0\end{matrix}\right]$$

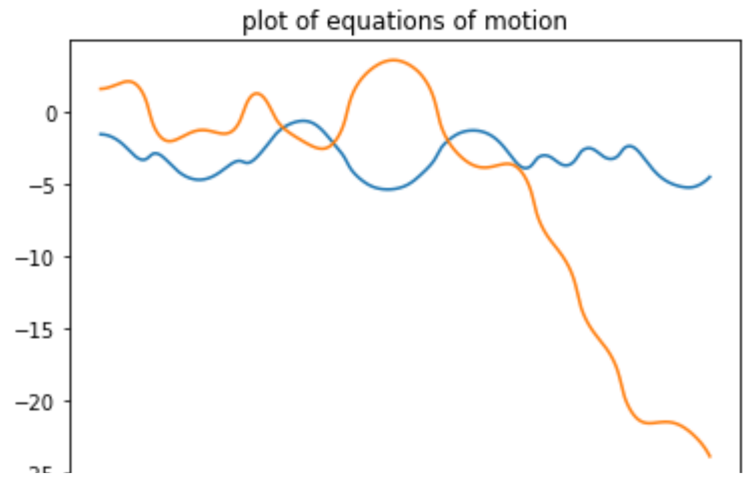
```
==================================================
```

$$\frac{d^2}{dt^2}\theta_1(t) = -\frac{R_1m_2\sin\left(\theta_2(t)\right)\cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{-R_1m_1 + R_1m_2\cos^2\left(\theta_2(t)\right) - R_1m_2} - \frac{R_2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{-R_1m_1 + R_1m_2\cos^2\left(\theta_2(t)\right) - R_1m_2} - \frac{2.0R_2m_2\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t)}{-R_1m_1 + R_1m_2\cos^2\left(\theta_2(t)\right) - R_1m_2} - \frac{R_2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2}{-R_1m_1 + R_1m_2\cos^2\left(\theta_2(t)\right) - R_1m_2} - \frac{gm_1\sin\left(\theta_1(t)\right)}{-R_1m_1 + R_1m_2\cos^2\left(\theta_2(t)\right) - R_1m_2} + \frac{gm_2\sin\left(\theta_1(t)+\theta_2(t)\right)\cos\left(\theta_2(t)\right.}{-R_1m_1 + R_1m_2\cos^2\left(\theta_2(t)\right) - R_1}$$

```
==================================================
```

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{R_1^2m_1\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{-R_1R_2m_1 + R_1R_2m_2\cos^2\left(\theta_2(t)\right) - R_1R_2m_2} + \frac{R_1^2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{-R_1R_2m_1 + R_1R_2m_2\cos^2\left(\theta_2(t)\right) - R_1R_2m_2} + \frac{2.0R_1R_2m_2\sin\left(\theta_2(t)\right)\cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{-R_1R_2m_1 + R_1R_2m_2\cos^2\left(\theta_2(t)\right) - R_1R_2m_2} + \frac{2.0R_1R_2m_2\sin\left(\theta_2(t)\right)\cos\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t)}{-R_1R_2m_1 + R_1R_2m_2\cos^2\left(\theta_2(t)\right) - R_1R_2m_2} + \frac{R_1R_2m_2\sin\left(\theta_2(t)\right)\cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2}{-R_1R_2m_1 + R_1R_2m_2\cos^2\left(\theta_2(t)\right) - R_1R_2m_2}$$

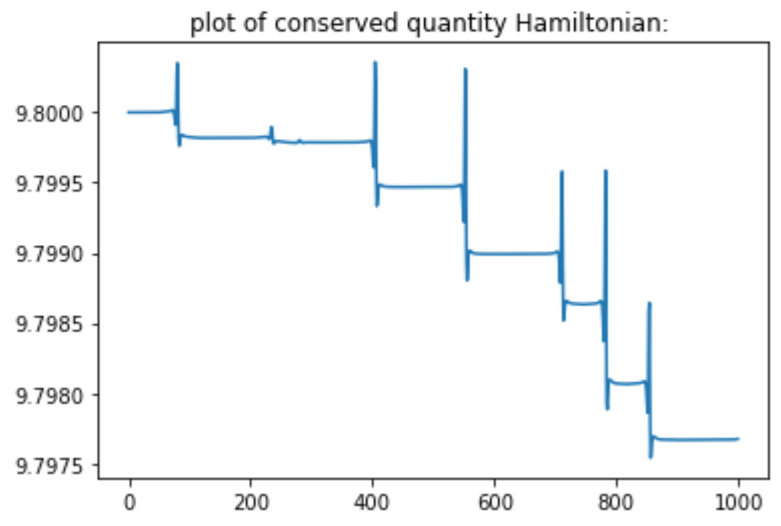
```
==================================================

shape of traj:  (4, 1000)
```



In [150…

```
H_func = sym.lambdify([q[0], q[1], qdot[0], qdot[1]], H)
H_eval = []

traj = simulate(dyn, s0, [0,10], 0.01, integrate)[0:4]
print(f'shape of traj:  {traj.shape}\n')

print(f"Simulated trajectory values from t = [0,10] with dt = 0.01 \nEach row corresponds to row vector of traj values for th1, th2, th1dot, and th2dot:\n{traj}")
traj = traj.T
print(f"\nlength of trajectory vector: {len(traj)}")

for i in range(len(traj)):
    H_eval.append(H_func(traj[i][0], traj[i][1], traj[i][2], traj[i][3])[0][0])

print(f"\nH_eval len: {len(H_eval)}")
# print(H_eval)


plt.title("plot of conserved quantity Hamiltonian: ")
plt.plot(np.arange(1000), H_eval)
plt.show()
```

```
shape of traj:  (4, 1000)

Simulated trajectory values from t = [0,10] with dt = 0.01
Each row corresponds to row vector of traj values for th1, th2, th1dot, and th2dot:
[[ -1.57128633  -1.57275633  -1.57520633 ...  -4.61128232  -4.56929051
   -4.52537021]
 [  1.57128621   1.57275441   1.57519661 ... -23.69603776 -23.77562369
  -23.85928156]
 [ -0.098       -0.19600015  -0.29400114 ...   4.10839603   4.29264957
    4.49467016]
 [  0.09795198   0.19561599   0.2927046  ...  -7.76975231  -8.15451262
   -8.58540685]]

length of trajectory vector: 1000

H_eval len: 1000
```



Problem 6 (15pts)

In the previously provided code for simulation, the numerical integration is a forth-order Runge–Kutta integration. Now, write down your own numerical integration function using Euler's method, and use your numerical integration function to simulate the same double-pendulum system with same parameters and initial condition in Problem 4. Compute and plot the Hamiltonian from the simulated trajectory, what's the difference between two plots?

Hint 1: You will need to implement a new {\tt integrate()} function. This function takes in three inputs: a function $f(x)$ representing the dynamics of the system state x (you can consider it as $\dot{x} = f(x)$), current state x (for example $x(t)$ if t is the current time step), and integration step length dt. This function should output $x(t + dt)$, for which the analytical solution is $x(t + dt) = x(t) + \int_t^{t+dt} f(x(\tau))d\tau$. Thus, you need to think about how to numerically evaluate this integration using Euler's method.

Hint 2: The implemented function should have the same input-output structure as the previous one.

Hint 3: After you implement the new integration function, you can use the same helper function $simulate()$ for simulation. You just need to input replace the integration function name as the new one (for example, your new function can be named as $euler\_integrate()$). Please carefully read the comments in the $simulate()$ function. Below is the template/example of how to implement the new integration function and use it for simulation.

In [162…

```python
###########################################################################
# Below is an example of how to implement a new integration
# function and use it for simulation

###########################################################################
# This is the same "simulate()" function we provided
# in previous homework.
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

###########################################################################
# This is the same "integrate()" function we provided in previous homework.
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
    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

##########################################################################
# This is where you implement your new integration function for this
# problem. Please make sure the new integration function has the same
# input-output structure as the old one.
def euler_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)
    """
    pass # you can start your implementation here
    new_xt = xt + f(xt)*dt
    return new_xt

##########################################################################
# In this example, we're going to simulate a particle falling in gravity,
# and assume that we already have the equations of motion (which you have
# to use Euler-Lagrange equations to solve for in the homework, but you
# should have that in last homework)
import numpy as np

def xddot(x, xdot):
    return -9.8

def dyn(s):
    return np.array([s[1], xddot(s[0], s[1])])

# define initial condition
s0 = np.array([10, 0])

##########################################################################
# We first use the old integration function to simulate the system, please
# carefully read the comments inside the "simulate()" function
traj = simulate(f=dyn, x0=s0, tspan=[0,10], dt=0.01, integrate=integrate)
# note that, here I pass the function arguments explicitly using the so-called
# "new_integrate" yet.
print(traj.shape)
```

```
(2, 1000)
```

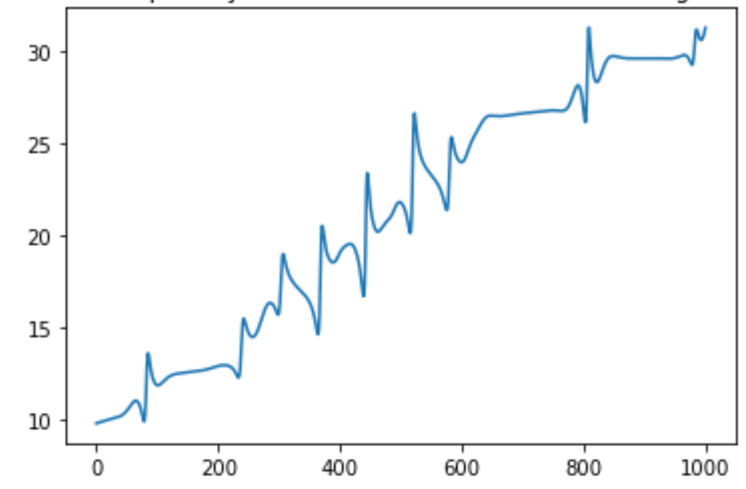
Turn in: A copy of you numerical integration function (you only need to include the code for your new integration function), and the resulting plot of Hamiltonian.

In [167…

```
def dyn(s):
    return np.array([
        s[2],
        s[3],
        th1ddot_func(*s),
        th2ddot_func(*s),

    ])


s0 = np.array([-pi/2, pi/2, 0, 0])

H_func = sym.lambdify([q[0], q[1], qdot[0], qdot[1]], H)
H_eval = []

traj = simulate(dyn, s0, [0,10], 0.01, euler_integrate)[0:4]
print(f'shape of traj:  {traj.shape}\n')

print(f"Simulated trajectory values from t = [0,10] with dt = 0.01 \nEach row corresponds to row vector of traj values for th1, th2, th1dot, and th2dot:\n{traj}")
traj = traj.T
print(f"\nlength of trajectory vector: {len(traj)}")

for i in range(len(traj)):
    H_eval.append(H_func(traj[i][0], traj[i][1], traj[i][2], traj[i][3])[0][0])

print(f"\nH_eval len: {len(H_eval)}")
# print(H_eval)


plt.title("plot of conserved quantity Hamiltonian with custom euler-integration method: ")
plt.plot(np.arange(1000), H_eval)
plt.show()
```

```
shape of traj:  (4, 1000)

Simulated trajectory values from t = [0,10] with dt = 0.01
Each row corresponds to row vector of traj values for th1, th2, th1dot, and th2dot:
[[ -1.57079633  -1.57177633  -1.57373633 ...  -9.91233574  -9.97473185
  -10.03520157]
 [  1.57079633   1.57177633   1.57373537 ...  20.22740645  20.24483009
   20.25537523]
 [ -0.098       -0.196       -0.29400019 ...  -6.23961053  -6.04697222
   -5.83192825]
 [  0.098        0.19590396   0.29342395 ...   1.74236465   1.0545138
    0.3685558 ]]

length of trajectory vector: 1000

H_eval len: 1000
```

plot of conserved quantity Hamiltonian with custom euler-integration method:



Problem 7 (20pts)

For the same double-pendulum you simulated in Problem 4 with same parameters and initial condition, now add a constraint to the system such that the distance between the second pendulum and the origin is fixed at $\sqrt{2}$. Simulate the system with same parameters and initial condition, and animate the system with the same animate function provided in homework 2.

Hint 1: What do you think the equations of motion should look like? Think about how the system will behave after adding the constraint. With no double, you can solve this problem using ϕ and all the following results for constrained Euler-Lagrange equations, however, if you really understand this constrained system, things might be much easier, and you can actually treat it as an unconstrained system.

Turn in: A copy of code used to numerically evaluate, simulate and animate the system. Also, upload the video of animation separately through Canvas, the video should be in ".mp4" format, and you can use screen capture or record the screen directly with your phone.

```
In [ ]:
```

Problem 8 (5pts)

For the same system with same constraint in Problem 6, simulate the system with initial condition $\theta_1 = \theta_2 = -\frac{\pi}{4}$, which actually violates the constraint! Simulate the system and see what happen, what do you think is the actual influence after adding this constraint?

Turn in: Your thoughts about the actual effect of the constraint in this system (you don't need to include any code for this problem).

```
In [ ]:
```

```
In [ ]:
```