ME314 Homework 3

Submission instructions

Deliverables that should be included with your submission are shown in **bold** at the end of each problem statement and the corresponding supplemental material. **Your homework will be graded IFF you submit a single PDF, .mp4 videos of animations when requested and a link to a Google colab file that meet all the requirements outlined below.**

- List the names of students you've collaborated with on this homework assignment.
- Include all of your code (and handwritten solutions when applicable) used to complete the problems.
- Highlight your answers (i.e. **bold** and outline the answers) for handwritten or markdown questions and include simplified code outputs (e.g. .simplify()) for python questions.
- Enable Google Colab permission for viewing

- Click Share in the upper right corner
- Under "Get Link" click "Share with..." or "Change"
- Then make sure it says "Anyone with Link" and "Editor" under the dropdown menu

- Make sure all cells are run before submitting (i.e. check the permission by running your code in a private mode)

- Please don't make changes to your file after submitting, so we can grade it!

- Submit a link to your Google Colab file that has been run (before the submission deadline) and don't edit it afterwards!

NOTE: This 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]:
```
#IMPORT CELL
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
import time
```

In [2]:
```
##############################################################################################
# If you're using Google Colab, uncomment this section by selecting the whole section
# ctrl+'/' on your and keyboard. Run it before you start programming, this will enable
# LaTeX "display()" function for you. If you're using the local Jupyter environment, l
##############################################################################################

# 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
```

```
#     javascript(url=url)
#     return sym.printing.latex(exp,**options)
# sym.init_printing(use_latex="mathjax",latex_printer=custom_latex_printer)
```

In [3]:

```
#helper functions I've used in past HW:

def compute_EL(lagrangian, q):
    '''
    Helper function for computing the Euler-Lagrange equations for a given system,
    so I don't have to keep writing it out over and over again.

    Inputs:
    - lagrangian: our Lagrangian function in symbolic (Sympy) form
    - q: our state vector [x1, x2, ...], in symbolic (Sympy) form

    Outputs:
    - eqn: the Euler-Lagrange equations in Sympy form
    '''

    # wrap system states into one vector (in SymPy would be Matrix)
    #q = sym.Matrix([x1, x2])
    qd = q.diff(t)
    qdd = qd.diff(t)

    # compute derivative wrt a vector, method 1
    # wrap the expression into a SymPy Matrix
    L_mat = sym.Matrix([lagrangian])
    dL_dq = L_mat.jacobian(q)
    dL_dqdot = L_mat.jacobian(qd)

    #set up the Euler-Lagrange equations
    LHS = dL_dq - dL_dqdot.diff(t)
    RHS = sym.Matrix([0,0]).T
    eqn = sym.Eq(LHS.T, RHS.T)

    return eqn


def solve_EL(eqn, var):
    '''
    Helper function to solve and display the solution for the Euler-Lagrange
    equations.

    Inputs:
    - eqn: Euler-Lagrange equation (type: Sympy Equation())
    - var: state vector (type: Sympy Matrix). typically a form of q-doubledot
        but may have different terms

    Outputs:
    - Prints symbolic solutions
    - Returns symbolic solutions in a dictionary
    '''

    soln = sym.solve(eqn, var, dict = True)
    eqns_solved = []
```

```
    for i, sol in enumerate(soln):
        for x in list(sol.keys()):
            eqn_solved = sym.Eq(x, sol[x])
            eqns_solved.append(eqn_solved)

    return eqns_solved
```

In [4]:
```
def rk4(dxdt, x, t, dt):
    '''
    Applies the Runge-Kutta method, 4th order, to a sample function,
    for a given state q0, for a given step size. Currently only
    configured for a 2-variable dependent system (x,y).
    ==========
    dxdt: a Sympy function that specifies the derivative of the system of interest
    t: the current timestep of the simulation
    x: current value of the state vector
    dt: the amount to increment by for Runge-Kutta
    ======
    returns:
    x_new: value of the state vector at the next timestep
    '''
    k1 = dt * dxdt(t, x)
    k2 = dt * dxdt(t + dt/2.0, x + k1/2.0)
    k3 = dt * dxdt(t + dt/2.0, x + k2/2.0)
    k4 = dt * dxdt(t + dt, x + k3)
    x_new = x + (k1 + 2.0*k2 + 2.0*k3 + k4)/6.0

    return x_new

rk = rk4(lambda t, x: x**2, 0.5, 2, 0.1)
assert np.isclose(rk, 0.526315781526278075), f"RK4 value: {rk}" #from an online RK4 sc
print("assertion passed")
```

```
assertion passed
```

In [5]:
```
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):
        t = tvec[i]
        xtraj[:,i]=integrate(f,x,t,dt)
        x = np.copy(xtraj[:,i])
    return xtraj
```

Problem 1 (10pts)

Let $f : \mathbb{R}^2 \to \mathbb{R}$ with $f(x, y) = \sin(x + y)\sin(x - y)$. Show that $(x, y) = (0, \pi/2)$ 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, include a copy of your code and all the outputs. Regardless of the format you choose, explain why your result satisfies the necessary and sufficient conditions.**

In [6]:
```
#new method: find the gradient and show that it's zero to show extremized;
#find the Hessian and show it's positive definite to show minimuzed
x, y, t = sym.symbols(r'x,y,t')
f = sym.sin(x + y) * sym.sin(x - y)

subs_dict = {
    x : 0,
    y : sym.pi/2
}

q = sym.Matrix([x, y])
grad = f.diff(q).simplify()
grad_eval = grad.subs(subs_dict)

print("Gradient:")
display(grad)
print("Gradient evaulated at (x,y) = (0, pi/2):")
display(grad_eval)
```

```
Gradient:
```

$$\begin{bmatrix} \sin(2x) \\ -\sin(2y) \end{bmatrix}$$

```
Gradient evaulated at (x,y) = (0, pi/2):
```

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

In [7]:
```python
#calculate Hessian matrix by taking 2nd derivatives wrt. each variable
hessian = sym.zeros(len(grad), len(q))
for i in range(len(grad)):
    for j in range(len(q)):
        hessian[i,j] = grad[i].diff(q[j])

#use eigenvalues to determine whether positive definite
hessian_eval = hessian.subs(subs_dict)
eigv = list(hessian_eval.eigenvals().keys())

print("Hessian matrix:")
display(hessian)
print("Hessian evaluated at (x,y) = (0, pi/2):")
display(hessian_eval)
print(f"Eigenvalues: {eigv}")
```

```
Hessian matrix:
```

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

```
Hessian evaluated at (x,y) = (0, pi/2):
```

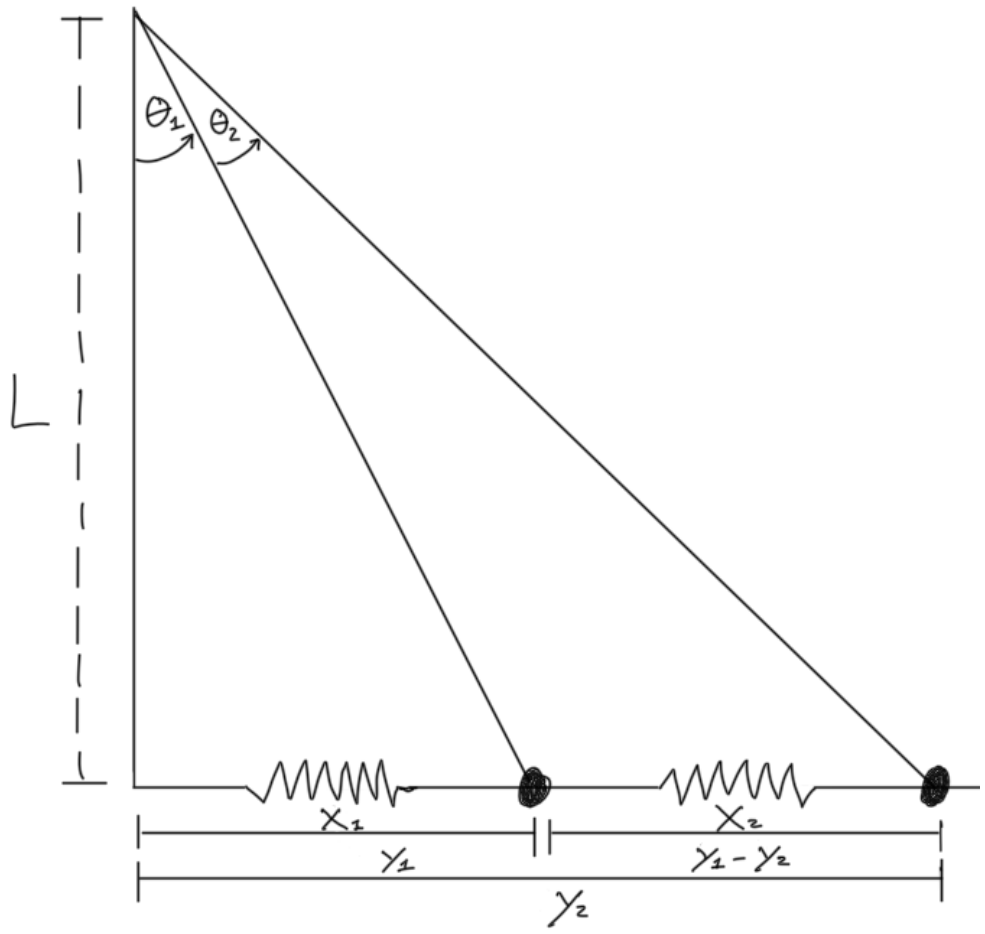
$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

```
Eigenvalues: [2]
```

The point $(x, y) = (0, \pi/2)$ is a local extremizer of $f(x, y) = \sin(x + y)\sin(x - y)$ because grad(f) evaluates to [0, 0]. This point is also a local minimizer of f(x,y) because the Hessian matrix evaluated at this point has only positive eigenvalues.

Problem 2 (20pts)

In [8]:
```python
#@title
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/mast
```



Compute the equations of motion for the two-mass-spring system (shown above) in $\theta = (\theta_1, \theta_2)$ coordinates. The first sphere with mass m_1 is the one close to the wall, and the second sphere has 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: Include the code used to symbolically solve for the equations of motion and the code output, which should be the equations of motion. Make sure you are using *SimPy*'s `.simplify()` functionality when printing your output.

In [9]:
```
#grab variables
m1, m2, k1, k2, L = sym.symbols(r'm_1, m_2, k_1, k_2, L')
t = sym.symbols(r't')

#define variables
theta1 = sym.Function(r'\theta_1')(t)
theta2 = sym.Function(r'\theta_2')(t)

theta1d = theta1.diff(t)
theta1dd = theta1d.diff(t)

theta2d = theta2.diff(t)
theta2dd = theta2d.diff(t)
```

```
#let y coords be an intermediate for calculating equations
y1 = L*sym.tan(theta1)
y2 = L*sym.tan(theta1 + theta2)

y1d = y1.diff(t)
y2d = y2.diff(t)

#print y1 and y2 in terms of theta
print("Representations y1(theta) and y1d:")
display(y1)
display(y1d)
print("Representations y1(theta) and y1d:")
display(y2)
display(y2d)
```

```
Representations y1(theta) and y1d:
```

$$L \tan\left(\theta_1(t)\right)$$

$$L \left(\tan^2\left(\theta_1(t)\right) + 1\right) \frac{d}{dt}\theta_1(t)$$

```
Representations y1(theta) and y1d:
```

$$L \tan\left(\theta_1(t) + \theta_2(t)\right)$$

$$L \left(\tan^2\left(\theta_1(t) + \theta_2(t)\right) + 1\right) \left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right)$$

In [10]:

```
#kinetic and potential energy functions
KE1 = 0.5 * m1 * y1d**2
KE2 = 0.5 * m2 * y2d**2
KE = KE1 + KE2

U1 = 0.5 * k1 * y1**2
U2 = 0.5 * k2 * (y2 - y1) **2
U = U1 + U2

print("Energy functions:")
display(KE)
display(U)

#Lagrangian
lagrangian = KE - U
print("\nLagrangian:")
display(lagrangian)

# print("\nSimplified:")
# lagrangian = lagrangian.simplify()
# display(lagrangian)
```

```
Energy functions:
```

$$0.5L^2m_1\left(\tan^2\left(\theta_1(t)\right) + 1\right)^2\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5L^2m_2\left(\tan^2\left(\theta_1(t) + \theta_2(t)\right) + 1\right)^2\left(\frac{d}{dt}\theta_1(t) + \frac{d}{d}\right.$$

$$0.5L^2k_1\tan^2\left(\theta_1(t)\right) + 0.5k_2\left(L\tan\left(\theta_1(t) + \theta_2(t)\right) - L\tan\left(\theta_1(t)\right)\right)^2$$

```
Lagrangian:
```

$$-0.5L^2k_1\tan^2(\theta_1(t)) + 0.5L^2m_1\left(\tan^2(\theta_1(t)) + 1\right)^2\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5L^2m_2\left(\tan^2(\theta_1(t) + \theta_2\right.$$

In [92]:
```
#compute E-L equations
t1 = time.time()

q = sym.Matrix([theta1, theta2])
eqn = compute_EL(lagrangian, q)
eqn = eqn.simplify()

t2 = time.time()
print(f"Elapsed: {t2 - t1}")
```

```
Elapsed: 175.88956689834595
```

In [93]:
```
print("Euler-Lagrange Equations:")
display(eqn)
```

```
Euler-Lagrange Equations:
```

$$\begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix} L^2\left(-\frac{1.0k_1\tan(\theta_1(t))}{\cos^2(\theta_1(t))} - 1.0k_2\left(\tan(\theta_1(t)+\theta_2(t)) - \tan(\theta_1(t))\right)\left(\tan^2(\theta_1(t)+\theta_2(t))\right.\right. \\ -1.0m \\ L^2\left(\tan^2(\theta_1(t)+\theta_2(t))+1\right)\left(-1.0k_2\left(\tan(\theta_1(t)+\theta_2(t)) - \tan(\theta_1(t))\right) - 2.0m_2\right. \end{bmatrix}$$

In [13]:
```
t1 = time.time()

qd = q.diff(t)
qdd = qd.diff(t)
eqns_solved = solve_EL(eqn, qdd)

print("Solved:")
for i, eq in enumerate(eqns_solved):
    print(f"Equation {i+1}:")
    eqns_solved[i] = eq.simplify()
    display(eqns_solved[i])

t2 = time.time()
print(f"Elapsed: {t2 - t1}")
```

```
Solved:
Equation 1:
```

$$\frac{d^2}{dt^2}\theta_1(t) = \frac{-0.5k_1\sin(2\theta_1(t))\tan(\theta_1(t))\tan(\theta_2(t)) + 0.5k_1\sin(2\theta_1(t)) - 1.0k_2\tan(\theta_2(t))}{m_1\left(\tan(\theta_1(t))\tan\right.}$$

```
Equation 2:
```

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{\frac{k_1 m_2 \sin\left(2\theta_1(t)\right)}{2} + k_1 m_2 \cos^2\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)\tan\left(\theta_1(t)\right) - k_2 m_1 \tan\left(\theta_1(t\right.}{\left.\left(\theta_1(t)\right)\tan^3\left(\theta_1(t)+\theta_2(t)\right) + k_2 m_2 \cos^4\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)\tan^3\left(\theta_1(t)\right) + \right.}$$
$$\left.\left(\theta_1(t)+\theta_2(t)\right) + k_2 m_2 \cos^4\left(\theta_1(t)\right)\tan\left(\theta_1(t)\right) - 2.0 m_1 m_2 \tan^3\left(\theta_1(t)+\theta_2(t)\right)\left(\frac{d}{dt}\right.\right.$$
$$\left.\left(\theta_1(t)+\theta_2(t)\right)\tan\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 2.0 m_1 m_2 \tan\left(\theta_1(t)+\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - \right.$$

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: Include the 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. Make sure toindicate what coordinate you choose in the comments.

Coordinates I chose to show Newton's Equations don't hold: (x1, x2)

In [52]:
```
#new version: force calculations in x and y frames
#this mirrors what I calculated in written homework, but

#grab variables
m1, m2, k1, k2 = sym.symbols(r'm_1, m_2, k_1, k_2')
t = sym.symbols(r't')

#define state variables
x1 = sym.Function(r'x_1')(t)
x2 = sym.Function(r'x_2')(t)
x1d = x1.diff(t)
x2d = x2.diff(t)
x1dd = x1d.diff(t)
x2dd = x2d.diff(t)

#define spring forces
Fs1 = k1 * x1
Fs2 = k2 * x2
```

```
#write equations of motion for 2 masses. in "x" reference frame, we assume ...
#position of mass 2 is given by x2, which is not an inertial reference frame
sum_Fm1_x = sym.Eq(x1dd, 1/m1 * (Fs1 - Fs2))
sum_Fm2_x = sym.Eq(x2dd, 1/m2 * Fs2)

print("Equations of motion in X frame:")
display(sum_Fm1_x)
display(sum_Fm2_x)
```

Equations of motion in X frame:

$$\frac{d^2}{dt^2}x_1(t) = \frac{k_1 x_1(t) - k_2 x_2(t)}{m_1}$$

$$\frac{d^2}{dt^2}x_2(t) = \frac{k_2 x_2(t)}{m_2}$$

In [51]:
```
#find accelerations in y frame as well
y1 = sym.Function(r'y_1')(t)
y2 = sym.Function(r'y_2')(t)
y1d = y1.diff(t)
y2d = y2.diff(t)
y1dd = y1d.diff(t)
y2dd = y2d.diff(t)

Fs1 = k1 * y1
Fs2 = k2 * (y2 - y1)

#write equations of motion for 2 masses
sum_Fm1_y = sym.Eq(y1dd, 1/m1 * (Fs1 - Fs2))
sum_Fm2_y = sym.Eq(y2dd, 1/m2 * Fs2)

print("Equations of motion in Y frame:")
display(sum_Fm1_y)
display(sum_Fm2_y)
```

Equations of motion in Y frame:

$$\frac{d^2}{dt^2}y_1(t) = \frac{k_1 y_1(t) - k_2\left(-y_1(t) + y_2(t)\right)}{m_1}$$

$$\frac{d^2}{dt^2}y_2(t) = \frac{k_2\left(-y_1(t) + y_2(t)\right)}{m_2}$$

In [89]:
```
#show that the acceleration equations are unequal by comparing
#y1dd and y2dd of the x system after the coord change

subs_dict = {
    x1: y1,
    x2: y2 - y1
}

x1dd_coord_change = sum_Fm1_x.subs(subs_dict).rhs
x1dd_coord_change = sym.Eq(y1dd, x1dd_coord_change)

x2dd_coord_change = sum_Fm2_x.subs(subs_dict).rhs + x1dd_coord_change.rhs
x2dd_coord_change = sym.Eq(y2dd, x2dd_coord_change)
```

Applying the coordinate change from x to y coordinates, we get that x2dd = y2dd - y1dd. I attempted to have Sympy rearrange the x2dd-coord-change equation to have y2dd on the left side, and to sub in y1dd in terms of other variables (based on the y1dd we found in the x1dd-coord-change equation). I had to do this manually, as I was unable to have Sympy substitute y1dd for an equation (it gave me an error).

In [91]:
```
print("\nAcceleration equations in x frame:")
display(x1dd_coord_change)
display(x2dd_coord_change)

print("\nAcceleration equations in y frame:")
display(sum_Fm1_y)
display(sum_Fm2_y)
```

```
Acceleration equations in x frame:
```

$$\frac{d^2}{dt^2} y_1(t) = \frac{k_1 y_1(t) - k_2 \left(-y_1(t) + y_2(t)\right)}{m_1}$$

$$\frac{d^2}{dt^2} y_2(t) = \frac{k_2 \left(-y_1(t) + y_2(t)\right)}{m_2} + \frac{k_1 y_1(t) - k_2 \left(-y_1(t) + y_2(t)\right)}{m_1}$$

```
Acceleration equations in y frame:
```

$$\frac{d^2}{dt^2} y_1(t) = \frac{k_1 y_1(t) - k_2 \left(-y_1(t) + y_2(t)\right)}{m_1}$$

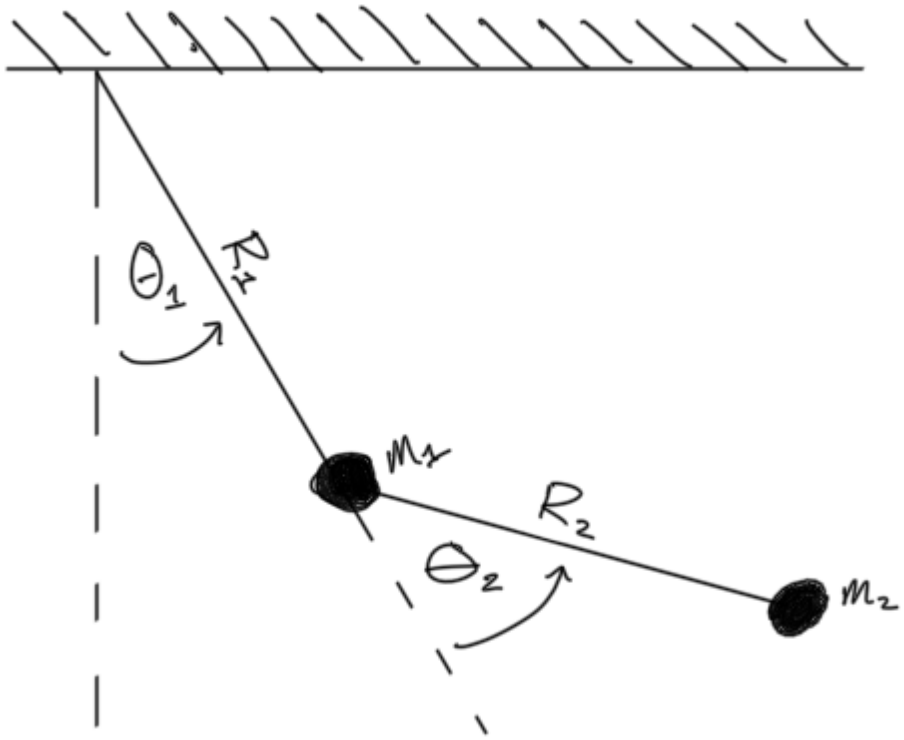
$$\frac{d^2}{dt^2} y_2(t) = \frac{k_2 \left(-y_1(t) + y_2(t)\right)}{m_2}$$

If Newton's second law, F = ma, held for every possible reference frame, we would expect the accelerations of the mass in x and y coordinates to be the same. The accelerations are different for x and y coordinates, which means Newton's laws don't hold for all possible reference frames.

The fact that the results are different is due to the fact that the X coordinate frame is non-inertial, and is accelerating according to the motion of mass m1. Only the y-coordinate frame works for Newton's Laws in this system.

Problem 4 (10pts)

In [19]:
```
#@title
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/mast
```



For the same double-pendulum system hanging in gravity in Homework 2 (shown above), 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: Include the code used to symbolically compute the Hamiltonian of the system and the code output, which should the Hamiltonian of the system. Make sure you are using *SimPy*'s `.simplify()` functionality when printing your output.

In [20]:
```
#grab variables
R1, R2, m1, m2, g, lamb = sym.symbols(r'R_1, R_2, m_1, m_2, g, \lambda')
t = sym.symbols(r't')

#define state variables
theta1 = sym.Function(r'\theta_1')(t)
theta2 = sym.Function(r'\theta_2')(t)

#intermediate variables to simplify energy calculations
x1 = R1 * sym.sin(theta1)
y1 = -R1 * sym.cos(theta1)
x2 = x1 + R2 * sym.sin(theta1 + theta2)
y2 = y1 - R2 * sym.cos(theta1 + theta2)

x1d = x1.diff(t)
x2d = x2.diff(t)
y1d = y1.diff(t)
y2d = y2.diff(t)
```

```
#define KE, U, and lagrangian
KE1 = 0.5 * m1 * (x1d**2 + y1d**2)
KE2 = 0.5 * m2 * (x2d**2 + y2d**2)
KE_sys = KE1 + KE2

U1 = m1 * g * y1
U2 = m2 * g * y2
U_sys = U1 + U2

lagrangian4 = KE_sys - U_sys
```

In [21]:
```
#check values of things
print("Lagrangian:")
display(lagrangian4)
print("Lagrangian, simplified:")
lagrangian4_sim = lagrangian4.simplify()
display(lagrangian4_sim)
```

```
Lagrangian:
```

$$R_1 g m_1 \cos\left(\theta_1(t)\right) - g m_2 \left(-R_1 \cos\left(\theta_1(t)\right) - R_2 \cos\left(\theta_1(t) + \theta_2(t)\right)\right) + 0.5 m_1 \left(R_1^2 \sin^2\left(\theta_1(t)\right)\right.$$

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

```
Lagrangian, simplified:
```

$$0.5 R_1^2 m_1 \left(\frac{d}{dt}\theta_1(t)\right)^2 + R_1 g m_1 \cos\left(\theta_1(t)\right) + g m_2 \left(R_1 \cos\left(\theta_1(t)\right) + R_2 \cos\left(\theta_1(t) + \theta_2(t)\right)\right) + 0$$

In [22]:
```
#hamiltonian: H = pqdot - lagrangian
#p = dL/dqdot
q = sym.Matrix([theta1, theta2])
qd = q.diff(t)
qdd = qd.diff(t)

p = lagrangian4_sim.diff(qd).T
pqd = (p*qd)[0]
ham = pqd - lagrangian4_sim
ham = ham.simplify()

print("Hamiltonian:")
display(ham)
```

```
Hamiltonian:
```

$$0.5 R_1^2 m_1 \left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5 R_1^2 m_2 \left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0 R_1 R_2 m_2 \cos\left(\theta_2(t)\right) \left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0 R_1$$

$$+ 1.0 R_2^2 m_2 \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t) + 0.5 R_2^2 m_2 \left(\frac{d}{dt}\theta_2(t)\right)^2 - 1.0 R_2 g m_2 \cos\left(\theta_1(t) + \theta_2(t)\right)$$

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: Include the code used to numerically evaluate and plot the Hamiltonian, as well as the code output, which should be the plot of Hamiltonian. Make sure you label the plot with axis labels, legend and a title.

In [23]:
```
#set up euler-lagrange equations and solve
eqn = compute_EL(lagrangian4_sim, q)
eqns_solved = solve_EL(eqn, qdd)

t1 = time.time()

eqns_simplified = []
print("Solved:")
for eq in eqns_solved:
    eq_simpl = eq.simplify()
    eqns_simplified.append(eq_simpl)
    display(eq_simpl)

t2 = time.time()
print(f"Elapsed: {t2 - t1}")
```

```
Solved:
```

$$\frac{d^2}{dt^2}\theta_1(t) = \frac{0.5R_1m_2\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0R_2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2.0R_2m_2\sin}{2\left(-1.0R_1^2m_1\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0R_1^2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0R_1R_2\right.}$$

$$\left.(\theta_1(t) - \theta_2(t)) - 0.5R_1gm_1\sin\left(\theta_1(t) + \theta_2(t)\right) + 0.5R_1gm_2\sin\left(\theta_1(t) - \theta_2(t)\right) - 0\right.$$

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{(\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2 + 1.0R_2gm_1\sin\left(\theta_1(t)\right) - 0.5R_2gm_2\sin\left(\theta_1(t) + 2\theta_2(t)\right) + 0.5I}{}$$

```
Elapsed: 62.892900466918945
```

Get expressions for accelerations qdd of each mass in theta coordinates. Use these acceleration equations to simulate motion.

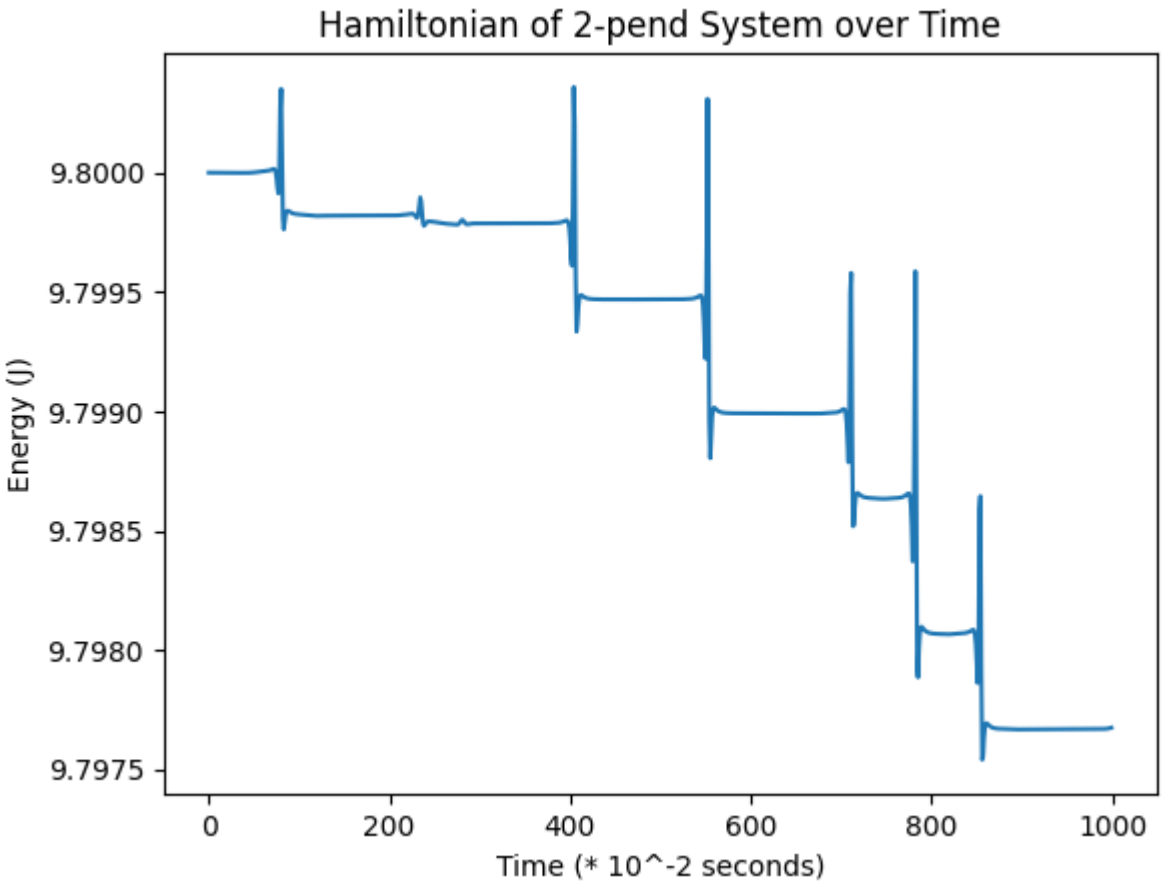
In [24]:
```
theta1dd_sy = eqns_simplified[0]
theta2dd_sy = eqns_simplified[1]

consts_dict = {
    R1: 1,
    R2: 1,
    m1: 1,
    m2: 1,
    g: 9.8
}

theta1dd_sy = theta1dd_sy.subs(consts_dict).rhs
theta2dd_sy = theta2dd_sy.subs(consts_dict).rhs

theta1d = theta1.diff(t)
theta2d = theta2.diff(t)
q_ext = sym.Matrix([theta1, theta2, theta1d, theta2d])

theta1dd_np = sym.lambdify(q_ext, theta1dd_sy)
theta2dd_np = sym.lambdify(q_ext, theta2dd_sy)
```

In [25]:
```
#simulate with given ICs

ICs = [-np.pi/2, -np.pi/2, 0, 0]
t_span = [0, 10]
dt = 0.01

def dxdt(t, s):
    '''
    Derivative of our state vector at the given state.

    Inputs:
        t: current time
        s: current state

    Notes:
        indeces [0,1] of output: 1st derivative of position
        indeces [2,3] of output: 2nd derivative of position
        q: current value of our state vector

     Returns: an array of derivatives of the state vector
    '''
    return np.array([s[2], s[3], theta1dd_np(*s), theta2dd_np(*s)])

q_array = simulate(dxdt, ICs, t_span, dt, rk4)

theta1_array = q_array[0]
theta2_array = q_array[1]
plt.plot(theta1_array)
plt.plot(theta2_array)

plt.xlabel("Time (* 10^-2 seconds)")
plt.ylabel("Angle (radians)")
plt.title("Position of Double Pendulum over Time")
```

Out[25]:
```
Text(0.5, 1.0, 'Position of Double Pendulum over Time')
```

In [26]:
```
#plot hamiltonian over time
ham = ham.subs(consts_dict)
ham_np = sym.lambdify(q_ext, ham)

#apply lambdified function to entire array
ham_array = [ham_np(*s) for s in q_array.T]

plt.plot(ham_array)

plt.xlabel("Time (* 10^-2 seconds)")
plt.ylabel("Energy (J)")
plt.title("Hamiltonian of 2-pend System over Time")
```

Out[26]:
```
Text(0.5, 1.0, 'Hamiltonian of 2-pend System over Time')
```



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* `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.*

Turn in: Include you numerical integration function (you only need to include the code for your new integration function), and the resulting plot of Hamiltonian. Make sure you label the plot appropriately with axis labels, legend and a title.

In [27]:

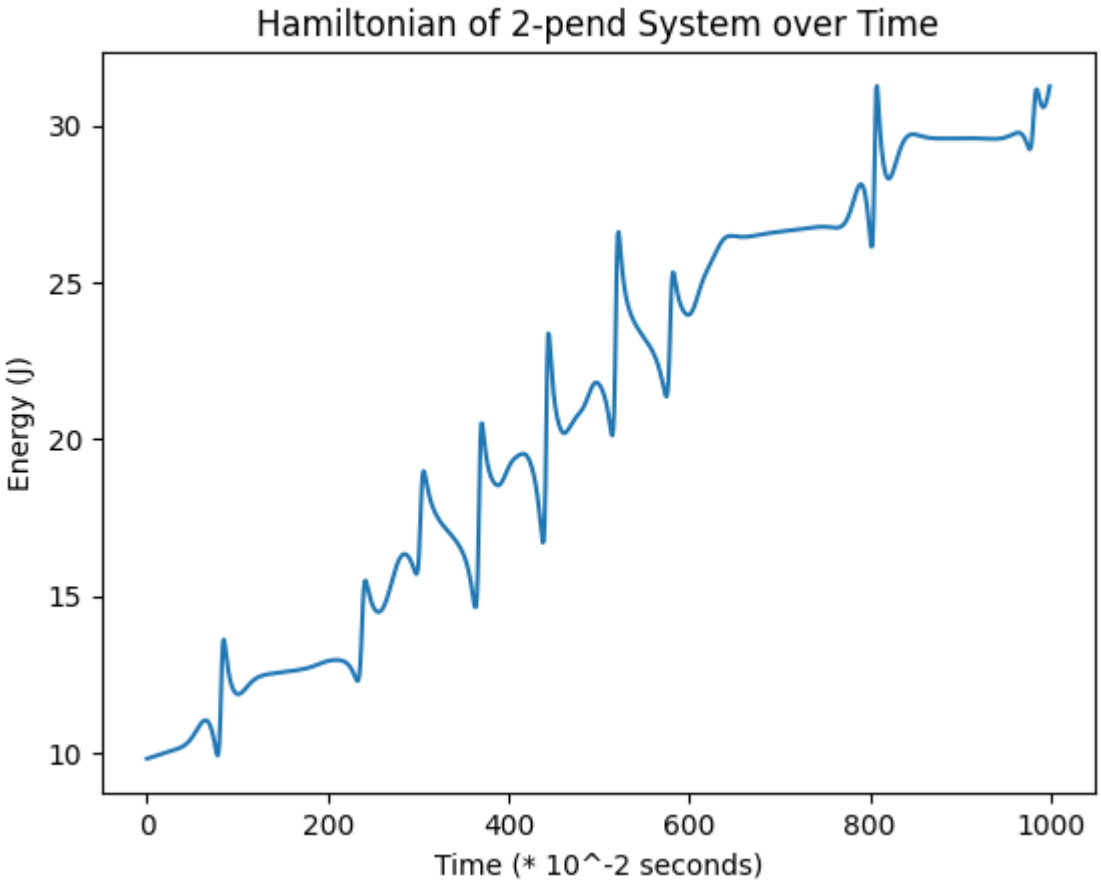
```
##################
# This is the same "simulate()" function we provided
# in previous homework.
def simulate_prob6(f, x0, tspan, dt, integrate):
    """
    See simulate() above. Difference between this function and that one
    is the usage of integrate(f, x, dt) rather than integrate(f, x, t, dt).
    """
    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 where you implement your new integration function for this problem.
def new_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)
    """
    x_new = xt + dt * f(xt)
    return x_new


def dxdt_new(s):
```

```
    '''
    See "dxdt(t, s). This function has no "t" input, as current time is not used to ca
    derivatives of the state vector at the next timestep.
    '''
    return np.array([s[2], s[3], theta1dd_np(*s), theta2dd_np(*s)])

###############
# You can start your simulation implementation here.

traj = simulate_prob6(f=dxdt_new, x0=ICs, tspan=[0,10], dt=0.01, integrate=new_integra
```

Calculate the Hamiltonian of the system at each timestep. Use the values of (q, qd) calculated at each timestep via simulate()

In [28]:
```
x_array = traj[0]
y_array = traj[1]

#apply lambdified function to entire array
ham_array = [ham_np(*s) for s in traj.T]

plt.plot(ham_array)

plt.xlabel("Time (* 10^-2 seconds)")
plt.ylabel("Energy (J)")
plt.title("Hamiltonian of 2-pend System over Time")
```

Out[28]:
```
Text(0.5, 1.0, 'Hamiltonian of 2-pend System over Time')
```



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

In [29]:

```
def solve_constrained_EL(lamb, phi, q, lhs):
    """Now uses just the LHS of the constrained E-L equations,
    rather than the full equation form"""

    qd = q.diff(t)
    qdd = qd.diff(t)

    phidd = phi.diff(t).diff(t)
    lamb_grad = sym.Matrix([lamb * phi.diff(a) for a in q])
    q_mod = qdd.row_insert(2, sym.Matrix([lamb]))

    #format equations so they're all in one matrix
    expr_matrix = lhs - lamb_grad
    phidd_matrix = sym.Matrix([phidd])
    expr_matrix = expr_matrix.row_insert(2,phidd_matrix)

    print("Expressions to be solved:")
    display(expr_matrix)
    print("Variables to solve for:")
    display(q_mod)

    #solve E-L equations
    eqns_solved = solve_EL(expr_matrix, q_mod)
    return eqns_solved
```

NOTE:

this code block below uses the Lagrangian calculated in problem 4. Make sure to run that code block first.

In [30]:
```
#constraint equation
phi = x2**2 + y2**2 - 2
phi = phi.simplify()
print("Constraint equation:")
display(phi)

t1 = time.time()

#left-hand side of constrained Euler-Lagrange equations
eqn = compute_EL(lagrangian4_sim, q)
eqn = eqn.simplify()
lhs = eqn.rhs #left side of the Sympy equation is [0,0]T

eqns_solved = solve_constrained_EL(lamb, phi, q, lhs)
print("Solved:")

eq_simpl = []
for i, eq in enumerate(eqns_solved):

    #no need to simplify lambda
    if i == 2:
        break

    eq_new = eq.simplify()
    eq_simpl.append(eq_new)
    display(eq_new)

t2 = time.time()
print(f"Elapsed: {t2 - t1}")
```

Constraint equation:

$$R_1^2 + 2R_1R_2\cos\left(\theta_2(t)\right) + R_2^2 - 2$$

Expressions to be solved:

$$\left[\begin{array}{r} -1.0R_1^2m_1\frac{d^2}{dt^2}\theta_1(t) - R_1gm_1\sin\left(\theta_1(t)\right) - gm_2\left(I \\ -1.0m_2\left(R_1^2\frac{d^2}{dt^2}\theta_1(t) - 2R_1R_2\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - R_1R_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 2I \\ 2R_1R_2\lambda\sin\left(\theta_2(t)\right) - 1.0R_2m_2\left(R_1\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + R_1\cos\left(\theta_2(t)\right. \\ -2R_1R_2\sin\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_2(t) - 2R_1I \end{array}\right]$$

Variables to solve for:

$$\left[\begin{array}{c} \frac{d^2}{dt^2}\theta_1(t) \\ \frac{d^2}{dt^2}\theta_2(t) \\ \lambda \end{array}\right]$$

Solved:

$$\frac{d^2}{dt^2}\theta_1(t) = \frac{1.0R_1R_2m_2\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + \frac{0.5R_1R_2m_2\left(\frac{d}{dt}\theta_2(t)\right)^2}{\sin\left(\theta_2(t)\right)} - 0.5R_1gm_1\sin\left(\theta_1(t)\right)}{0.5R_1^2m_1 + 0.5R_1^2m_2 + 1.0R_1R_2n}$$

$$\frac{d^2}{dt^2}\theta_2(t) = -\frac{\left(\frac{d}{dt}\theta_2(t)\right)^2}{\tan\left(\theta_2(t)\right)}$$

```
Elapsed: 943.4625804424286
```

In [31]:
```
#2016 seconds is 33.5 minutes, but on my end it was really more like 48 min
#revised: 2min
```

In [32]:
```
consts_dict = {
    R1: 1,
    R2: 1,
    m1: 1,
    m2: 1,
    g: 9.8
}

t1 = time.time()
theta1dd_sy = eq_simpl[0].subs(consts_dict).simplify()
theta2dd_sy = eq_simpl[1].subs(consts_dict).simplify()

print("Equations of motion:")
display(theta1dd_sy)
display(theta2dd_sy)
t2 = time.time()

print(f"Elapsed: {t2 - t1} seconds")
```

```
Equations of motion and constraint force lambda:
```

$$\frac{d^2}{dt^2}\theta_1(t) = \frac{-3.26666666666667\sin\left(\theta_1(t)+\theta_2(t)\right) - 6.53333333333333\sin\left(\theta_1(t)\right) + 0.6666}{0.6666666666}$$

$$\frac{d^2}{dt^2}\theta_2(t) = -\frac{\left(\frac{d}{dt}\theta_2(t)\right)^2}{\tan\left(\theta_2(t)\right)}$$

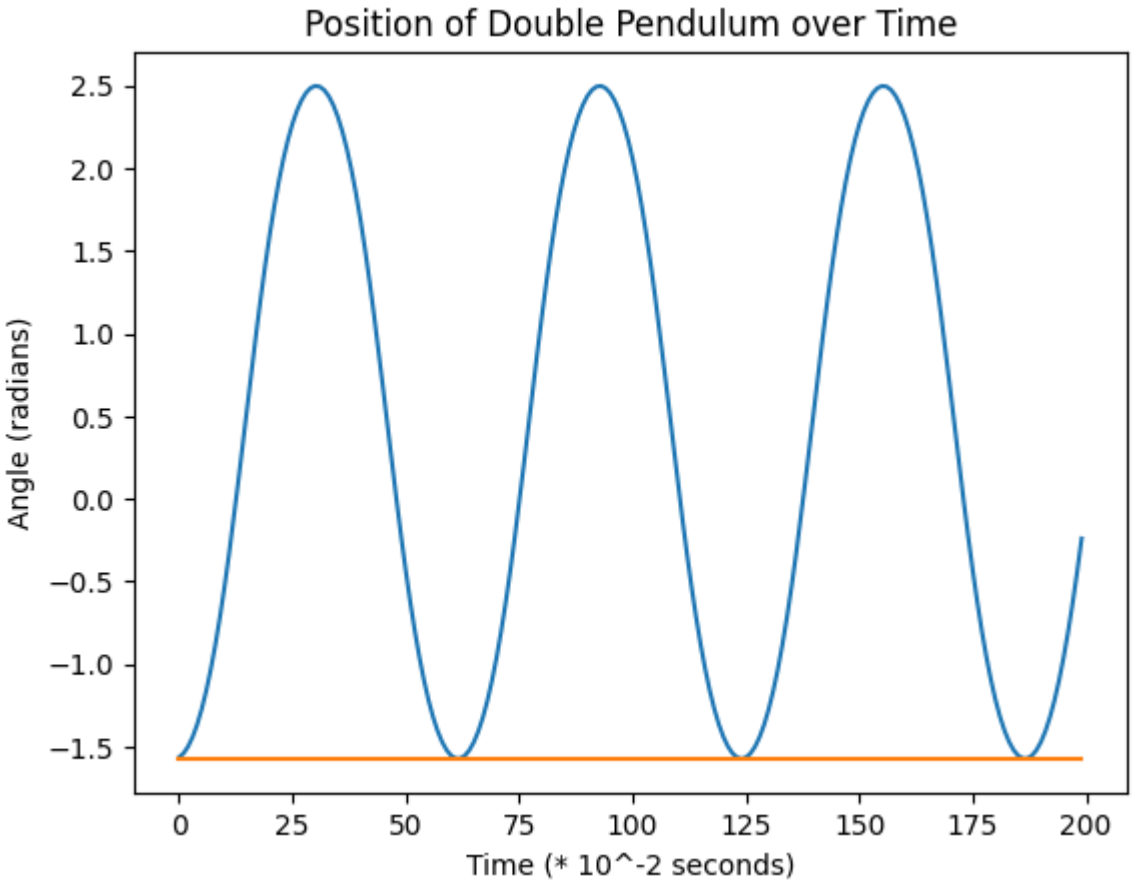
```
Elapsed: 60.74681615829468 seconds
```

In [34]:
```
q_ext = sym.Matrix([theta1, theta2, theta1d, theta2d])
theta1dd_np = sym.lambdify(q_ext, theta1dd_sy.rhs)
theta2dd_np = sym.lambdify(q_ext, theta2dd_sy.rhs)

#simulate motion of the system over time
ICs = [-np.pi/2, -np.pi/2, 0, 0]
t_span = [0, 10]
dt = 0.01

#version with Euler integration and the new simulate function
#traj = simulate_prob6(dxdt_new, x0=ICs, tspan=[0,10], dt=0.01, integrate=new_integrat
traj = simulate(dxdt, x0=ICs, tspan=[0,10], dt=0.05, integrate=rk4)
```

Plot motion of graph. We expect theta1 to be changing in a sinusoid and theta2 to not be changing

In [35]:
```
theta1_array = traj[0]
theta2_array = traj[1]
plt.plot(theta1_array)
plt.plot(theta2_array)

plt.xlabel("Time (* 10^-2 seconds)")
plt.ylabel("Angle (radians)")
plt.title("Position of Double Pendulum over Time")
```

Out[35]:
```
Text(0.5, 1.0, 'Position of Double Pendulum over Time')
```



Plot energy in the system over time to check validity of results.

In [36]:
```
#positions in q[0] and q[1], velocities in q[2] and q[3]

#pull constants from dictionary; use different var names
m1c = consts_dict[m1]
m2c = consts_dict[m2]
R1c = consts_dict[R1]
R2c = consts_dict[R2]
gc = consts_dict[g]

def KE1(s):
    [_, _, theta1d, _] = s
    m1 = m1c
    R1 = R1c
    return 0.5 * m1 * R1**2 * theta1d**2
```

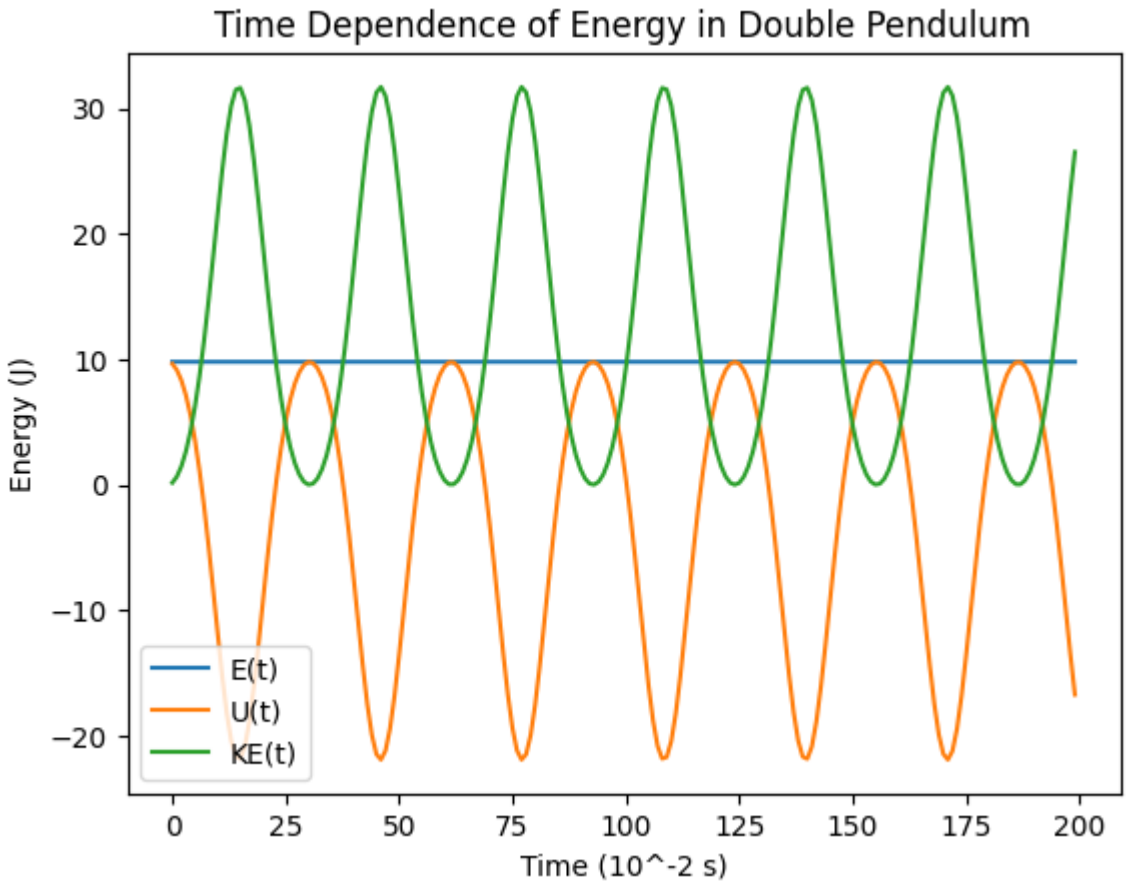
```
def KE2(s):
    [theta1, theta2, theta1d, theta2d] = s
    m2 = m2c
    R1 = R1c
    R2 = R2c

    x1d = R1 * np.cos(theta1) * theta1d
    y1d = R1 * np.sin(theta1) * theta1d
    x2d = x1d + R2 * np.cos(theta1 + theta2) * (theta1d + theta2d)
    y2d = y1d + R2 * np.sin(theta1 + theta2) * (theta1d + theta2d)

    return 0.5 * m2 * (x2d**2 + y2d**2)

def U1(s):
    [theta1, _, _, _] = s
    m1 = m1c
    R1 = R1c
    g = gc
    return m1 * g * -R1 * np.cos(theta1)

def U2(s):
    [theta1, theta2, _, _] = s
    m2 = m2c
    R1 = R1c
    R2 = R2c
    g = gc
    return -m2 * g * (
        R1 * np.cos(theta1) +
        R2 * np.cos(theta1 + theta2)
    )

def E(s):
    return KE1(s) + KE2(s) + U1(s) + U2(s)

# print(q_array)
# print(np.matrix(q_array))
E_array = [E(s) for s in traj.T]
KE_array = [KE1(s) + KE2(s) for s in traj.T]
U_array = [U1(s) + U2(s) for s in traj.T]
```

In [37]:

```
#plot
plt.figure()
plt.title("Time Dependence of Energy in Double Pendulum")
plt.xlabel("Time (10^-2 s)")
plt.ylabel('Energy (J)')

plt.plot(E_array)
plt.plot(U_array)
plt.plot(KE_array)
plt.legend(["E(t)", "U(t)", "KE(t)"], loc = 'lower left')
```

Out[37]:

```
<matplotlib.legend.Legend at 0x2850c56bac0>
```



Animate the trajectory with the same function as homework 2.

In [38]:
```
def animate_double_pend(theta_array,L1=1,L2=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

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

    Returns: None
    """

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

    ######################
    # Browser configuration.
```

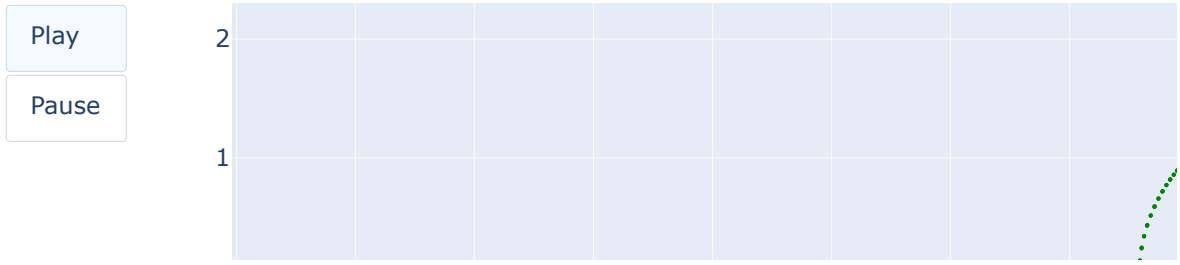
```
def configure_plotly_browser_state():
    import IPython
    display(IPython.core.display.HTML('''
        <script src="/static/components/requirejs/require.js"></script>
        <script>
          requirejs.config({
            paths: {
              base: '/static/base',
              plotly: 'https://cdn.plot.ly/plotly-1.5.1.min.js?noext',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

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

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

############################
# Defining data dictionary.
# Trajectories are here.
data=[dict(x=xx1, y=yy1,
           mode='lines', name='Arm',
           line=dict(width=2, color='blue')
          ),
      dict(x=xx1, y=yy1,
           mode='lines', name='Mass 1',
           line=dict(width=2, color='purple')
          ),
      dict(x=xx2, y=yy2,
           mode='lines', name='Mass 2',
           line=dict(width=2, color='green')
          ),
      dict(x=xx1, y=yy1,
           mode='markers', name='Pendulum 1 Traj',
           marker=dict(color="purple", size=2)
          ),
      dict(x=xx2, y=yy2,
           mode='markers', name='Pendulum 2 Traj',
           marker=dict(color="green", size=2)
          ),
    ]

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

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

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

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

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

In [39]:

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

# second, animate!
animate_double_pend(traj, L1 = consts_dict[R1], L2 = consts_dict[R2],T=10)
```

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



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. Note that you don't need to include any code for this problem.

In [40]:
```
#code below
```

My thoughts: in the mechanical model of the 2-mass system, there are two constraints acting on mass 2. The first is tension in the link between mass 1 and 2, constraining that mass 2 must stay at a fixed radius from mass 1. The second constraint, the distance constraint from mass 2 to the origin, acts like another link between mass 2 and the origin. The triangular shape that these 3 points make constrains the motion of mass 2 relative to mass 1 such that $\dot{\theta_2}$ equals zero.

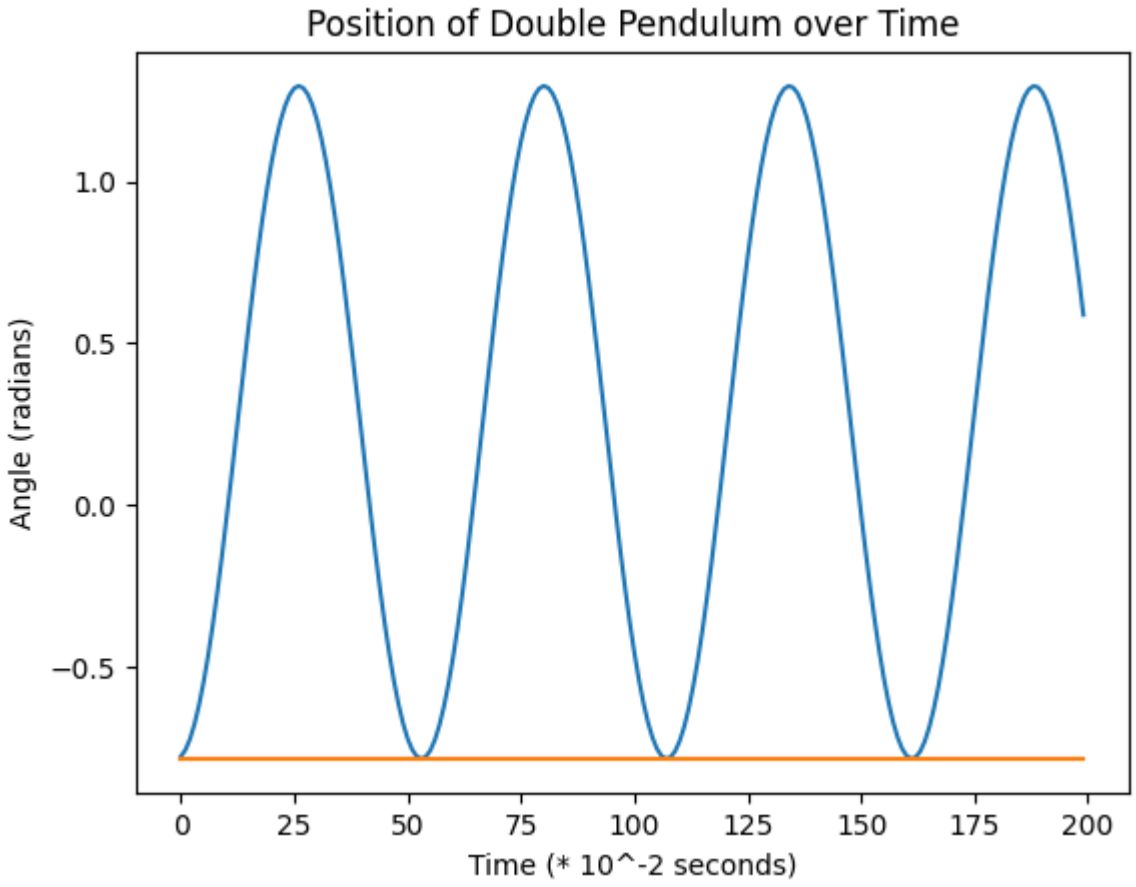
The ICs $\theta_1 = \theta_2 = -\frac{\pi}{4}$ violate the distance constraint, but d/dt(theta2) still equals zero when the simulation is run. I think the actual influence of the constraint is to enforce $\dot{\theta_2} = 0$ and to

keep mass 2 at a fixed radius from the origin.

In [41]:
```
ICs = [-np.pi/4, -np.pi/4, 0, 0]
t_span = [0, 10]
dt = 0.01

#version with Euler integration and the new simulate function
#traj = simulate_prob6(dxdt_new, x0=ICs, tspan=[0,10], dt=0.01, integrate=new_integrat
traj = simulate(dxdt, x0=ICs, tspan=[0,10], dt=0.05, integrate=rk4)
```

In [42]:
```
theta1_array = traj[0]
theta2_array = traj[1]
plt.plot(theta1_array)
plt.plot(theta2_array)

plt.xlabel("Time (* 10^-2 seconds)")
plt.ylabel("Angle (radians)")
plt.title("Position of Double Pendulum over Time")
```

Out[42]:
```
Text(0.5, 1.0, 'Position of Double Pendulum over Time')
```



In [43]:
```
animate_double_pend(traj, L1 = consts_dict[R1], L2 = consts_dict[R2],T=10)
```

Double Pendulum Simulation