ME314 Homework 3 Solutions

In [1]:
```
#import necessary packages
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# ###############################################################################################
# # 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]:
```
def simulate(f, x0, tspan, dt, integrate):
    """
    This function takes in an initial condition x0, a timestep dt,
    a time span tspan consisting of a list [min_time, max_time],
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x0. It outputs a full trajectory simulated
    over the time span of dimensions (xvec_size, time_vec_size).

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

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

In [5]:
```
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 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 = "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)]

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

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.

The necessary condition for $s = [x, y]$ is that the derivative of f with respect to s (Jacobian) is a zero vector at the point. The sufficient condition, however, is based on the second-order derivative (Hessian), which is a square matrix. The Hessian matrix needs to be positive-definite, which means the eigen values need to be greater than 0. You can compute both Jacocbian and Hessian with SymPy:

In [6]:
```
# Relevant variables and function
x,y = sym.symbols('x, y')
fun1 = sym.sin(x-y)*sym.sin(x+y)

# Looking at it's derivative w.r.t x and yf
D_fun = sym.Matrix([fun1]).jacobian([x,y])
print('Jacobian:')
display(D_fun)

# Evaluate at (0,pi/2)
print("\nEvaluating Jacobian at (0,pi/2):")
display(D_fun.subs({x:0, y:sym.pi/2}))

# Hessian
D2_fun = sym.trigsimp(D_fun.jacobian([x,y]))
print('\nHessian:')
display(D2_fun)

# Eigenvalues at (0,pi/2)
print("\nEvaluating Hessian at (0,pi/2):")
display(D2_fun.subs({x:0,y:sym.pi/2}))
print('\nThe Hessian has two eigenvalues, both positive, so it satisfies the sufficient condition for a local minimizer.')
```

```
Jacobian:
```

$$\left[\sin(x-y)\cos(x+y)+\sin(x+y)\cos(x-y) \quad \sin(x-y)\cos(x+y)-\sin(x+y)\cos(x-y)\right]$$

```
Evaluating Jacobian at (0,pi/2):
```

$$\left[0 \quad 0\right]$$

```
Hessian:
```

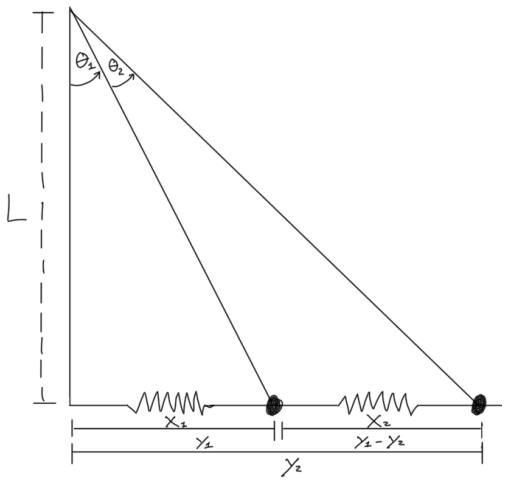
$$\begin{bmatrix} 2\cos(2x) & 0 \\ 0 & -2\cos(2y) \end{bmatrix}$$

```
Evaluating Hessian at (0,pi/2):
```

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

```
The Hessian has two eigenvalues, both positive, so it satisfies the sufficient condition for a local minimizer.
```

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 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 [7]:
```
t, L, m1, m2, k1, k2 = sym.symbols(r't, L, m_1, m_2, k_1, k_2')

th1 = sym.Function(r'\theta_1')(t)
th2 = sym.Function(r'\theta_2')(t)
q = sym.Matrix([th1, th2])
# a clever way of doing this: q = sym.Matrix([Function(r'\theta_'+str(i))(t) for i in range(2)])
qdot = q.diff(t)
qddot = qdot.diff(t)

# we derive the Lagrangian through y coordinates
y1 = L * sym.tan(q[0])
y2 = L * sym.tan(q[1]+q[0])
y1dot = y1.diff(t)
y2dot = y2.diff(t)
KE = 0.5*m1*y1dot**2 + 0.5*m2*y2dot**2
PE = 0.5*k1*y1**2 + 0.5*k2*(y2-y1)**2
L = KE - PE

# EL-equations
dLdq = sym.Matrix([L]).jacobian(q).T
dLdqdot = sym.Matrix([L]).jacobian(qdot).T
d_dLdqdot_dt = dLdqdot.diff(t)
el_eqns = sym.Eq(d_dLdqdot_dt-dLdq,sym.Matrix([0, 0]))
# display(el_eqns)

# solve for equations of motion
el_solns = sym.solve(el_eqns, qddot, dict=True)[0]
display(sym.simplify(sym.Eq(qddot[0], el_solns[qddot[0]])))
display(sym.simplify(sym.Eq(qddot[1], sym.trigsimp(el_solns[qddot[1]]))))
```

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

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{\begin{array}{l} k_1 m_2 \left(1 - \cos\left(2\theta_1(t)\right)\right)^2 \sin\left(\theta_2(t)\right) - 0.5 k_1 m_2 \sin\left(2\theta_1(t) - \theta_2(t)\right) + 1.5 k_1 m_2 \sin \\ \left(2\theta_1(t) + \theta_2(t)\right) + 0.25 k_1 m_2 \sin\left(4\theta_1(t) - \theta_2(t)\right) + 0.25 k_1 m_2 \sin\left(4\theta_1(t) + \theta_2(t)\right) \\ - 2.0 k_1 m_2 \sin\left(\theta_2(t)\right) + 1.0 k_2 m_1 \sin\left(2\theta_1(t) + \theta_2(t)\right) - 1.0 k_2 m_1 \sin\left(2\theta_1(t) + 3\theta_2(t)\right) \\ - 2.0 k_2 m_1 \sin\left(\theta_2(t)\right) + 1.0 k_2 m_2 \sin\left(2\theta_1(t) - \theta_2(t)\right) - 1.0 k_2 m_2 \sin\left(2\theta_1(t) + \theta_2(t)\right) \\ - 2.0 k_2 m_2 \sin\left(\theta_2(t)\right) - 8.0 m_1 m_2 \sin\left(2\theta_1(t) + \theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 4.0 m_1 m_2 \sin \\ \left(2\theta_1(t) + \theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 - 8.0 m_1 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 8.0 m_1 m_2 \sin \end{array}}{\begin{array}{l} \left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 4.0 m_1 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 \\ m_1 m_2 \left(-\sin\left(2\theta_1(t) - \theta_2(t)\right)\tan\left(\theta_2(t)\right) - \sin\left(2\theta_1(t) + \theta_2(t)\right)\tan\left(\theta_2(t)\right) + 2\cos \\ \left(2\theta_1(t)\right)\cos\left(\theta_2(t)\right) + 2\cos\left(\theta_2(t)\right)\right) \end{array}}$$

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.*

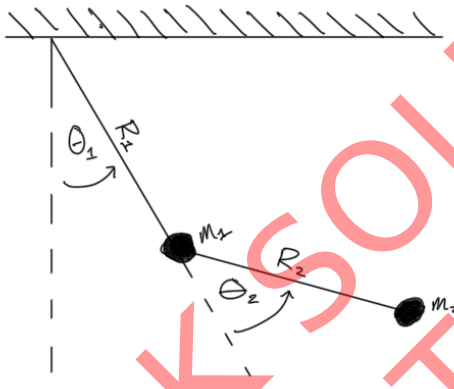
Turn in: Include the code you used to symbolically compute the equations of motion to show that Newton's equations don't hold. Also, include the output of the code, which should be the equations of motion under the chosen set of coordinates. Make sure toindicate what coordinate you choose in the comments.

Consider what $F = ma = m\ddot{q}(t)$ would say in x coordinates:

$$\ddot{x}_1(t) = -k_1 x_1(t) + k_2 x_2(t)$$
$$\ddot{x}_2(t) = -k_2 x_2(t)$$

This is not the same as what you saw in class and is *not* equivalent to Newton's equations in Cartesian coordinates.

Problem 4 (10pts)



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 [8]:
```
# constants
t, g, m1, m2, R1, R2 = sym.symbols(r't, g, m_1, m_2, R_1, R_2')

# states
th1 = sym.Function(r'\theta_1')(t)
th2 = sym.Function(r'\theta_2')(t)
q = sym.Matrix([th1, th2])
qdot = q.diff(t)
qddot = qdot.diff(t)

p1x = R1 * sym.sin(q[0])
p1y = R1 * -sym.cos(q[0])
p2x = p1x + R2 * sym.sin(q[0]+q[1])
p2y = p1y + R2 * -sym.cos(q[0]+q[1])

p1xdot = p1x.diff(t)
p1ydot = p1y.diff(t)
p2xdot = p2x.diff(t)
p2ydot = p2y.diff(t)

# Lagrangian
ke = 0.5 * m1 * (p1xdot**2+p1ydot**2) + 0.5 * m2 * (p2xdot**2+p2ydot**2)
pe = -m1*g*R1*sym.cos(q[0]) - m2*g*(R1*sym.cos(q[0])+R2*sym.cos(q[0]+q[1]))

L = ke - pe
print('Lagrangian:')
display(L)
```

```
Lagrangian:
```

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

In [9]:
```
# EL-equations
L = sym.Matrix([L])
dLdq = L.jacobian(q).T
dLdqdot = L.jacobian(qdot).T
d_dLdqdot_dt = dLdqdot.diff(t)
el_eqns = sym.Eq(sym.simplify(d_dLdqdot_dt - dLdq), sym.Matrix([0, 0]))
print('Euler-Lagrange Equations:')
display(el_eqns)

# solve and lambdify
soln = sym.solve(el_eqns, qddot)
th1ddot_soln = soln[qddot[0]]
th1ddot_soln = th1ddot_soln.subs({m1:1, m2:1, R1:1, R2:1, g:9.8})
th2ddot_soln = soln[qddot[1]]
th2ddot_soln = th2ddot_soln.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)
```

```
Euler-Lagrange Equations:
```

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

In [10]:
```
H = dLdqdot.T * qdot - L
H = sym.simplify(H)
H = H.subs({m1:1, m2:1, R1:1, R2:1, g:9.8})
H_func = sym.lambdify([q[0], q[1], qdot[0], qdot[1]], H)
display (H.simplify())
```

$$\begin{bmatrix} -9.8\cos(\theta_1(t)+\theta_2(t)) - 19.6\cos(\theta_1(t)) + 1.0\cos(\theta_2(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0\cos \\ (\theta_2(t)) \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t) + 1.5\left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0 \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t) + 0.5\left(\frac{d}{dt}\theta_2(t)\right)^2 \end{bmatrix}$$

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.*

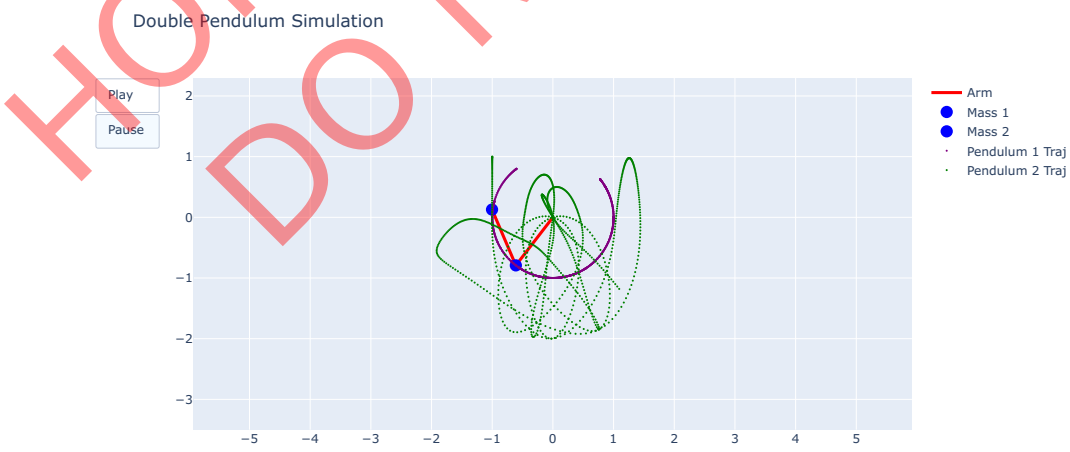
Turn in: Include the code used to numerically evaluate and plot the Hamiltonian, as well as the code output, which should be the plot of Hamiltonian. Make sure you label the plot with axis labels, legend and a title.

In [11]:
```
def dyn(s):
    return np.array([
        s[2],
        s[3],
        th1ddot_func(*s),
        th2ddot_func(*s)
    ])

s0 = np.array([-np.pi/2, -np.pi/2, 0, 0])
traj = simulate(dyn, s0, [0, 10], 0.01, integrate)
print('shape of traj: ', traj.shape)
```

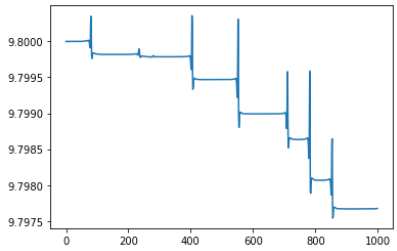
```
shape of traj:  (4, 1000)
```

In [12]:
```
animate_double_pend(traj,L1=1,L2=1,T=10)
```



In [13]:
```
H_traj = []
for t in traj.T:
    Ht = H_func(*t)[0][0]
    H_traj.append(Ht)
H_traj = np.array(H_traj)

plt.plot(np.arange(1000), H_traj)
plt.show()
```



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.

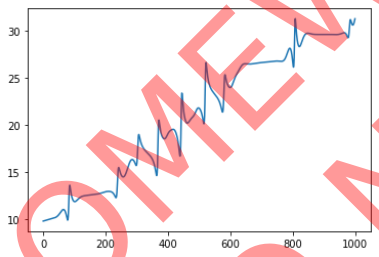
In [14]:
```
def euler_integrate(f, xt, dt):
    return xt + f(xt)*dt

s0 = np.array([-np.pi/2, -np.pi/2, 0, 0])
traj2 = simulate(dyn, s0, [0, 10], 0.01, euler_integrate)

H_traj2 = []
for t in traj2.T:
    Ht = H_func(*t)[0][0]
    H_traj2.append(Ht)
H_traj2 = np.array(H_traj2)

import matplotlib.pyplot as plt

plt.plot(np.arange(1000), H_traj2)
plt.show()
```



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 [15]:
```
t, g, m1, m2, R1, R2, lamb = sym.symbols(r't, g, m_1, m_2, R_1, R_2, \lambda')

th1 = sym.Function(r'\theta_1')(t)
th2 = sym.Function(r'\theta_2')(t)
q = sym.Matrix([th1, th2])
qdot = q.diff(t)
qddot = qdot.diff(t)

p1x = R1 * sym.sin(q[0])
p1y = R1 * -sym.cos(q[0])
p2x = p1x + R2 * sym.sin(q[0]+q[1])
p2y = p1y + R2 * -sym.cos(q[0]+q[1])

p1xdot = p1x.diff(t)
p1ydot = p1y.diff(t)
p2xdot = p2x.diff(t)
p2ydot = p2y.diff(t)

ke = 0.5 * m1 * (p1xdot**2+p1ydot**2) + 0.5 * m2 * (p2xdot**2+p2ydot**2)
pe = -m1*g*R1*sym.cos(q[0]) - m2*g*(R1*sym.cos(q[0])+R2*sym.cos(q[0]+q[1]))
phi = p2x**2 + p2y**2 - 2

L = ke - pe
print('Lagrangian:')
display(L)
```

Lagrangian:

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

In [16]:
```
lhs1 = L.diff(qdot[0]).diff(t) - L.diff(q[0])
lhs2 = L.diff(qdot[1]).diff(t) - L.diff(q[1])
lhs3 = (sym.Matrix([phi]).jacobian(q) * qdot).diff(t)
lhs = sym.Matrix([lhs1, lhs2, lhs3])

rhs1 = lamb * phi.diff(q[0])
rhs2 = lamb * phi.diff(q[1])
rhs3 = 0
rhs = sym.Matrix([rhs1, rhs2, rhs3])

el_eqns = sym.Eq(sym.simplify(lhs), sym.simplify(rhs))
print('Euler-Lagrange Equations with Constriants:')
display(el_eqns)

soln = sym.solve(el_eqns, [qddot[0], qddot[1], lamb])
th1ddot_soln = soln[qddot[0]]
th1ddot_soln = th1ddot_soln.subs({m1:1, m2:1, R1:1, R2:1, g:9.8})
th2ddot_soln = soln[qddot[1]]
th2ddot_soln = th2ddot_soln.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)
```

Euler-Lagrange Equations with Constriants:

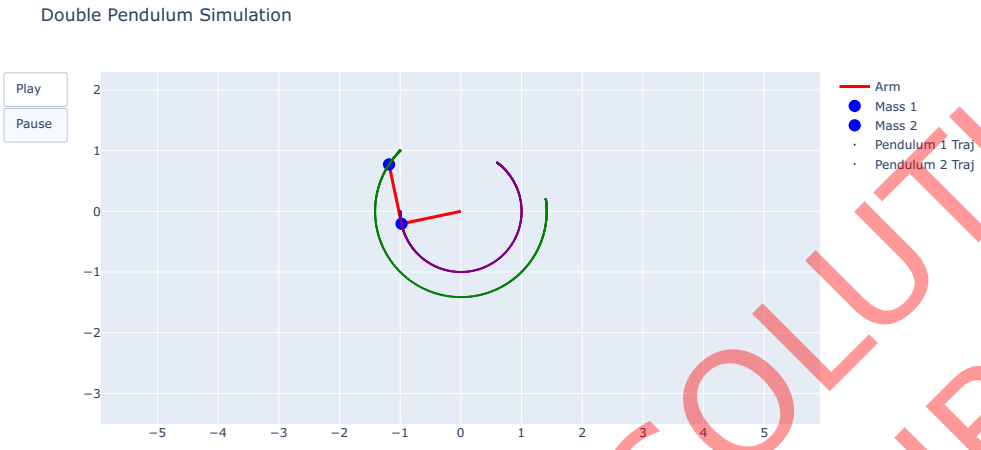
$$\begin{bmatrix} R_1^2 m_1 \frac{d^2}{dt^2}\theta_1(t) + R_1 g m_1 \sin\left(\theta_1(t)\right) + g m_2 \left(R_1 \sin\left(\theta_1(t)\right) + R_2 \sin\left(\theta_1(t) + \theta_2(t)\right)\right) \\ + m_2\left(R_1^2 \frac{d^2}{dt^2}\theta_1(t) - 2R_1 R_2 \sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t)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

In [17]:
```
def dyn(s):
    return np.array([
        s[2],
        s[3],
        th1ddot_func(*s),
        th2ddot_func(*s)
    ])

s0 = np.array([-np.pi/2, -np.pi/2, 0, 0])
traj = simulate(dyn, s0, [0, 10], 0.01, integrate)[0:2]
print('shape of traj: ', traj.shape)

animate_double_pend(traj, L1=1, L2=1, T=10)
```

```
shape of traj:  (2, 1000)
```



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.

What the constraint actually does is to fix the angle betweeen the first the second pendulums, but it doesn't indicate which value it should fix. So even though our configuration in this problem violates the constraint, the system can still be simulated.

In [18]:
```
s0 = np.array([-np.pi/4, -np.pi/4, 0, 0])
traj = simulate(dyn, s0, [0, 10], 0.01, integrate)[0:2]
print('shape of traj: ', traj.shape)

animate_double_pend(traj, L1=1, L2=1, T=10)
```

```
shape of traj:  (2, 1000)
```