ME314 Homework 2 (Solution)

*Please note that a **single PDF file** will be the only document that you turn in that will be graded - with exception of .mp4 videos of animations as part of the code output when requested. It should include all of your answers to the problems with corresponding derivations and the code used to complete the problems. When including the code, please make sure you also include corresponding code outputs and note that you don't need to include example code. Deliverables that should be included with your submission are shown in **bold** at the end of each problem statement and the corresponding supplemental material.*

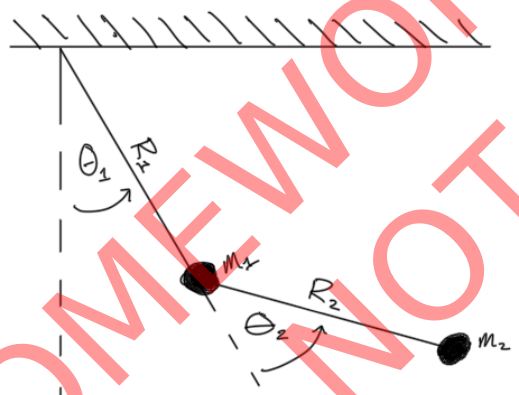
*We recommend you use Jupyter Notebook to complete your homework as (1) it allows you to include all the code and other deliverables in one file, (2) it supports $\LaTeX$ for math equations, and (3) you can export the whole notebook as a single PDF file. This template serves as a starting point and is for you to use if you choose to do so. Note that you can use this notebook file with your local Jupyter environment or upload it to Google Colab. However, 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]:
```
#Import python packages
import sympy as sym
import numpy as np
sym.__version__
from IPython.display import display, Math, Latex
from IPython.core.display import display_html
from sympy import *
init_session(quiet=True)
init_printing()
```

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)
```

Problem 1 (15pts)

In [3]:
```
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>"))
```



You're given a double-pendulum system hanging in gravity is shown in the figure above. With $q = [\theta_1, \theta_2]$ as the system configuration variables, use Python's SymPy package to compute the Lagrangian of the system. Note that we assume that the z-axis is pointing out from the screen/paper and thus the positive direction of rotation is counter-clockwise.

Hint 1: We recommend that you compute the positions and their time derivatives (velocities) in x-y coordinates! This will involve using some trigonometry to express the x and y coordinates of each mass in terms of θ_1 and θ_2. Consequently, compute kinetic and potential energy based on that.

Hint 2: By convention we will define gravity with positive sign (i.e. $g = 9.8$) for numerical evaluation required in the later problems. As such, be careful with the sign of potential energy! You can always go back here after you verify your results by numerical evaluation in Problem 2 and 3.

Turn in: A copy of the code used to symbolically compute Lagrangian and the output of your code which should be the symbolic Lagrangian expression.

In [4]:
```
t, g, m1, m2, R1, R2 = sym.symbols(r't, g, m_1, m_2, R_1, R_2')

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)

# Updated kinetic and potential energy formulation
ke = 0.5 * m1 * (p1xdot**2+p1ydot**2) + 0.5 * m2 * (p2xdot**2+p2ydot**2)
pe = m1*g*p1y + m2*g*p2y

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

Lagrangian:

$$0.5R_1^2m_1\left(\frac{d}{dt}\theta_1(t)\right)^2 + R_1gm_1\cos\left(\theta_1(t)\right) + gm_2\left(R_1\cos\left(\theta_1(t)\right) + R_2\cos\left(\theta_1(t)+\theta_2(t)\right)\right)$$
$$+ 0.5m_2\left(R_1^2\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2R_1R_2\cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2R_1R_2\cos\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + R_2^2\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2R_2^2\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + R_2^2\left(\frac{d}{dt}\theta_2(t)\right)^2\right)$$

Problem 2 (15pts)

Use Python's SymPy package to compute the Euler-Lagrange equations for the same double-pendulum system in Problem 1 and solve them for $\ddot{\theta}_1$ and $\ddot{\theta}_2$.

Turn in: A copy of the code used to symbolically compute and solve Euler-Lagrange equations. Also include the output of your code which should be the symbolic expression of Euler-Lagrange equations and solutions (i.e. $\ddot{\theta}_1$ and $\ddot{\theta}_2$).

In [5]:
```
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(sym.simplify(el_eqns))

soln = sym.solve(el_eqns, qddot)
th1ddot_soln = soln[qddot[0]]
print("\nth1_ddot Symbolic Solution:")
display(sym.simplify(soln[qddot[0]]))

th2ddot_soln = soln[qddot[1]]
print("\nth2_ddot Symbolic Solution:")
display(sym.simplify(soln[qddot[1]]))
```

Euler-Lagrange Equations:

$$\begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix} R_1^2m_1\frac{d^2}{dt^2}\theta_1(t) + R_1gm_1\sin\left(\theta_1(t)\right) + gm_2\left(R_1\sin\left(\theta_1(t)\right) + R_2\sin\left(\theta_1(t)+\theta_2(t)\right)\right) \\ + m_2\left(R_1^2\frac{d^2}{dt^2}\theta_1(t) - 2R_1R_2\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - R_1R_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 2R_1R_2\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_1(t) + R_1R_2\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_2(t)\right. \\ \left. + R_2^2\frac{d^2}{dt^2}\theta_1(t) + R_2^2\frac{d^2}{dt^2}\theta_2(t)\right) \\ 1.0R_2m_2\left(R_1\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + R_1\cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_1(t) + R_2\frac{d^2}{dt^2}\theta_1(t) + R_2\frac{d^2}{dt^2}\theta_2(t) + g\sin\left(\theta_1(t)+\theta_2(t)\right)\right) \end{bmatrix}$$

th1_ddot Symbolic Solution:

$$\frac{0.5R_1m_2\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0R_2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2.0R_2m_2\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + 1.0R_2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 - 1.0gm_1\sin\left(\theta_1(t)\right) + 0.5gm_2\sin\left(\theta_1(t)+2\theta_2(t)\right) - 0.5gm_2\sin\left(\theta_1(t)\right)}{R_1\left(m_1+m_2\sin^2\left(\theta_2(t)\right)\right)}$$

th2_ddot Symbolic Solution:

$$\frac{2\left(-1.0R_1^2m_1\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0R_1^2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0R_1R_2m_2\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0R_1R_2m_2\sin\left(2\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 0.5R_1R_2m_2\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.5R_1gm_1\sin\left(\theta_1(t)-\theta_2(t)\right) - 0.5R_1gm_1\sin\left(\theta_1(t)+\theta_2(t)\right) + 0.5R_1gm_2\sin\left(\theta_1(t)-\theta_2(t)\right) - 0.5R_1gm_2\sin\left(\theta_1(t)+\theta_2(t)\right) - 1.0R_2^2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 2.0R_2^2m_2\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 1.0R_2^2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 1.0R_2gm_1\sin\left(\theta_1(t)\right) - 0.5R_2gm_2\sin\left(\theta_1(t)+2\theta_2(t)\right) + 0.5R_2gm_2\sin\left(\theta_1(t)\right)\right)}{R_1R_2\left(2m_1 - m_2\cos\left(2\theta_2(t)\right) + m_2\right)}$$

Problem 3 (15pts)

Numerically evaluate your solutions for $\ddot{\theta}_1$ and $\ddot{\theta}_2$ from Problem 2 using SymPy's `lambdify()` method, simulate the system for $t \in [0, 5]$ with $m_1 = 1, m_2 = 2, R_1 = 2, R_2 = 1$ and initial condition as $\theta_1 = \theta_2 = -\frac{\pi}{2}, \dot{\theta}_1 = \dot{\theta}_2 = 0$. Plot the simulated trajectories of $\theta_1(t)$ and $\theta_2(t)$ versus time.

Hint 1: Feel free to use the provided example code or your implementation in Homework 1.

Hint 2: By convention, we will define $g = 9.8$ as a positive constant. If you got some wierd "flipped" trajectory, go back to Problem 1 and check the sign of your gravity potential energy term.

Turn in: A copy of the code used for numerical evaluation and simulation as well as the output of your code which should include the plot of trajectories.

In [6]:
```
# gravity is defined positive (g=9.8) as previously stated in problem 1
th1ddot_soln = th1ddot_soln.subs({m1:1, m2:2, R1:2, R2:1, g:9.8})
th2ddot_soln = th2ddot_soln.subs({m1:1, m2:2, R1:2, 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)

print('th1_ddot: {:.2f}'.format(th1ddot_func(-np.pi/2,-np.pi/2, 0, 0)))
print('\nth2_ddot: {:.2f}'.format(th2ddot_func(-np.pi/2,-np.pi/2, 0, 0)))
```

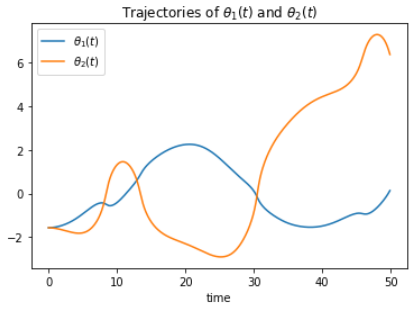
```
th1_ddot: 4.90

th2_ddot: -4.90
```

In [ ]:
```
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 [ ]:
```
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 [7]:
```
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, 5], 0.01, integrate)[0:2]
print('Shape of traj: ', traj.shape)
import matplotlib.pyplot as plt

plt.plot(np.arange(500)*0.1, traj[0:2].T)
plt.title(r'Trajectories of $\theta_1(t)$ and $\theta_2(t)$')
plt.xlabel("time")
plt.legend((r'$\theta_1(t)$', r'$\theta_2(t)$'), loc='upper left')
plt.show()
```

```
Shape of traj:  (2, 500)
```



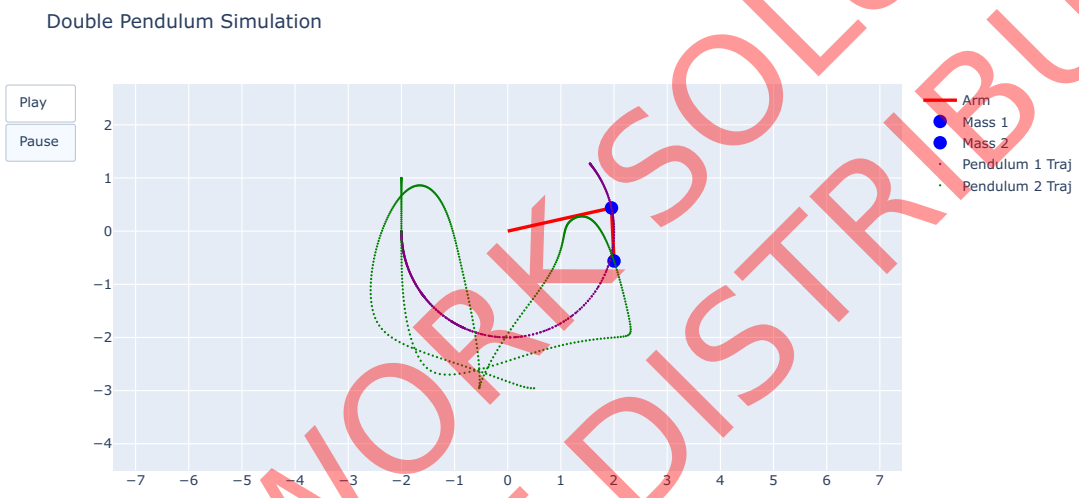
Problem 4 (10pts)

Finally, let's get fancy! Use the function provided below to animate your simulation of the double-pendulum system based on the trajectories you got in Problem 3.

In [8]:
```
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'}
                                          ]
                               }]
```

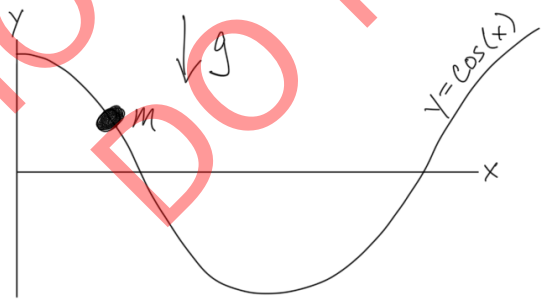
```
            )

    ######################################
    # 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)
```

In [9]:
```
# Simulation animation of the double-pendulum system
animate_double_pend(traj, L1=2, L2=1, T=10)
```



Problem 5 (15pts)

In [10]:
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/master/dynbeadwire.png' width=500' height='350'></table>"))
```



As shown in the figure above, a bead in gravity is constrained to the path $y = \cos(x)$. With the x-y positions of the bead as the system configuration variables, compute the Lagrangian symbolically using SymPy and write down constraint equation $\phi(q) = 0$ of the system as SymPy's symbolic equation.

Turn in: A copy of the code you used to compute the Lagrangian and generate the constraint equation. Include the output of your code which should be the symbolic expression for the Lagrangian and constraint equation.

In [11]:
```
t, m, g, lamb = sym.symbols(r't, m, g, \lambda')

x = sym.Function('x')(t)
y = sym.Function('y')(t)
q = sym.Matrix([x, y])
qdot = q.diff(t)
qddot = qdot.diff(t)


ke = 0.5 * m * (qdot[0]**2 + qdot[1]**2)
pe = m * g * q[1]
phi = q[1] - sym.cos(q[0])

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

print('Constraint Equation:')
display(sym.Eq(q[1], sym.cos(q[0])))
phi = q[1] - sym.cos(q[0])
```

```
Lagrangian:
```

$$-gmy(t) + 0.5m\left(\left(\frac{d}{dt}x(t)\right)^2 + \left(\frac{d}{dt}y(t)\right)^2\right)$$

```
Constraint Equation:
```

$$y(t) = \cos\left(x(t)\right)$$

Problem 6 (10pts)

Use Python's SymPy's package to solve for the equations of motion ($\ddot{x}$ and $\ddot{y}$) and constraint force λ for the constrained bead system in Problem 5.

Turn in: A copy of the code used to symbolically solve for the equations of motion and constraint force. Include the output of the code, which should be the symoblic EOM equations and constraint force.

In [12]:
```
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 Constraints:')
display(el_eqns)

soln = sym.solve(el_eqns, [qddot[0], qddot[1], lamb])

print('\033[1m\n Equations Of Motion (EOM)\033[0m')
xddot_soln = soln[qddot[0]]
print('\n x_ddot')
display(xddot_soln)
xddot_soln = xddot_soln.subs({m:1, g:9.8})
print('\ny_ddot')
yddot_soln = soln[qddot[1]]
display(yddot_soln)
yddot_soln = yddot_soln.subs({m:1, g:9.8})

xddot_func = sym.lambdify([q[0], q[1], qdot[0], qdot[1]], xddot_soln)
yddot_func = sym.lambdify([q[0], q[1], qdot[0], qdot[1]], yddot_soln)
```

```
Euler-Lagrange Equations with Constraints:
```

$$\begin{bmatrix} 1.0m\frac{d^2}{dt^2}x(t) \\ m\left(g + 1.0\frac{d^2}{dt^2}y(t)\right) \\ \sin\left(x(t)\right)\frac{d^2}{dt^2}x(t) + \cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2 + \frac{d^2}{dt^2}y(t) \end{bmatrix} = \begin{bmatrix} \lambda\sin\left(x(t)\right) \\ \lambda \\ 0 \end{bmatrix}$$

Equations Of Motion (EOM)

```
x_ddot
```

$$\frac{g\sin\left(x(t)\right)}{\sin^2\left(x(t)\right) + 1.0} - \frac{\sin\left(x(t)\right)\cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2}{\sin^2\left(x(t)\right) + 1.0}$$

```
y_ddot
```

$$-\frac{g\sin^2\left(x(t)\right)}{\sin^2\left(x(t)\right) + 1.0} - \frac{\cos\left(x(t)\right)\left(\frac{d}{dt}x(t)\right)^2}{\sin^2\left(x(t)\right) + 1.0}$$

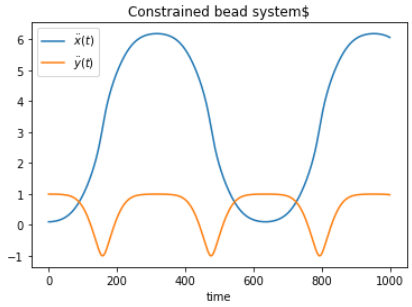
Problem 7 (20pts)

Simulate this constrained bead system with $m = 1$ and initial condition as $x = 0.1, y = \cos(0.1), \dot{x} = \dot{y} = 0$, for $t \in [0, 10]$. Animate your simulated trajectory using the provided function below.

In [13]:
```
def dyn(s):
    return np.array([
        s[2],
        s[3],
        xddot_func(*s),
        yddot_func(*s)
    ])

s0 = np.array([0.1, np.cos(0.1), 0, 0])
traj = simulate(dyn, s0, [0, 10], 0.01, integrate)[0:2]
print('shape of traj: ', traj.shape)
```

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

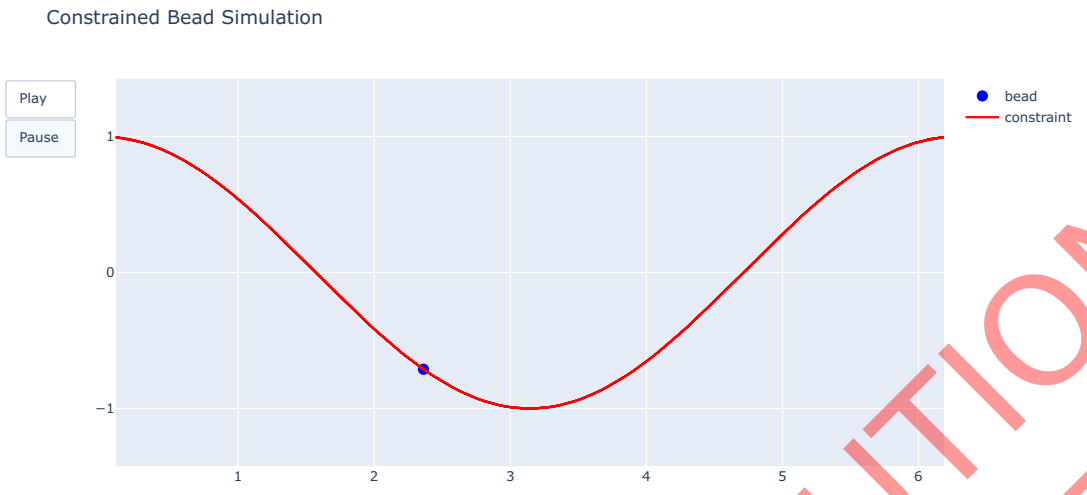
In [14]:
```
from matplotlib import pyplot as plt

plt.plot(np.arange(1000), traj[0])
plt.plot(np.arange(1000), traj[1])

plt.title(r'Constrained bead system$')
plt.xlabel("time")
plt.legend((r'$\ddot{x}(t)$', r'$\ddot{y}(t)$'), loc='upper left')
plt.show()
```



In [15]:

```
def animate_bead(xy_array,T=10):
    """
    Function to generate web-based animation of constrained bead system

    Parameters:
    ===============================================
    xy_array:
        trajectory of x and y, should be a NumPy array with
        shape of (2,N)
    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 trajectories.
    N = len(xy_array[0]) # Need this for specifying length of simulation

    ###################################
    # Using these to specify axis limits.
    xm=np.min(xy_array[0])-0.5
    xM=np.max(xy_array[0])+0.5
    ym=np.min(xy_array[1])-0.5
    yM=np.max(xy_array[1])+0.5

    ############################
    # Defining data dictionary.
    # Trajectories are here.
    data=[dict(x=xy_array[0], y=xy_array[1],
               mode='markers', name='bead',
               marker=dict(color="blue", size=10)
              ),
          dict(x=xy_array[0], y=xy_array[1],
               mode='lines', name='constraint',
               line=dict(width=2, color='red')
              ),
        ]

    ##############################
    # 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='Constrained Bead 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 bead at each time
    # step of simulation.
    frames=[dict(data=[go.Scatter(
                            x=[xy_array[0][k]],
                            y=[xy_array[1][k]],
                            mode="markers",
                            marker=dict(color="blue", size=10))
                      ]) for k in range(N)]



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

In [16]:
```
animate_bead(traj)
```



In [ ]:
```

```