ME314 Homework 4 Solutions

In [1]:
```
#import necessary packages
import numpy as np
import sympy as sym
from sympy.calculus import euler_equations # we try SymPy's built-in method in this problem
from math import sqrt, pi
import matplotlib.pyplot as plt
```

In [ ]:
```
Below are the help functions in previous homeworks, which you may need for this homework.
```

In [2]:
```
def integrate(f, xt, dt):
    """
    This function takes in an initial condition x(t) and a timestep dt,
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x(t). It outputs a vector x(t+dt) at the future
    time step.

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

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

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

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

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


Problem 1

Take the bead on a hoop example shown in the image above, model it using a torque input τ (about the vertical z axis) instead of a velocity input ω. You will need to add a configuration variable ψ that is the rotation about the z axis, so that the system configuration vector is $q = [\theta, \psi]$. Use Python's SymPy package to compute the equations of motion for this system in terms of θ, ψ.

Hint 1: Note that this should be a Lagrangian system with an external force.

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

In [3]:
```
# symbols
t, m, R, tau, g = sym.symbols(r't, m, R, \tau, g')
theta_sym, psi_sym, thetadot, psidot, thetaddot, psiddot = sym.symbols(r'\theta, \psi, \dot{\theta}, \dot{\psi}, \ddot{\theta}, \ddot{\psi}')

# states
theta = sym.Function(r'\theta')
psi = sym.Function(r'\psi')
q = sym.Matrix([theta(t), psi(t)])
qdot = q.diff(t)
qddot = qdot.diff(t)
qddot_sym = sym.Matrix([thetaddot, psiddot])
subs_dict = {q[0]:theta_sym, q[1]:psi_sym, qdot[0]:thetadot, qdot[1]:psidot, qddot[0]:thetaddot, qddot[1]:psiddot}

# Lagrangian
v1 = R * qdot[0]
v2 = qdot[1] * R * sym.sin(q[0])
KE = sym.Rational(1,2) * m * (v1**2 + v2**2)
V = m*g*R*(1-sym.cos(q[0]))
Lagrangian = KE-V
print("Lagrangian:")
display(Lagrangian.subs(subs_dict))

# Euler-Lagrangian Equations with External Force
# force = tau / R*sin(q[0])
force = tau
el_eqs = euler_equations(Lagrangian, q, [t])
el_eqs = sym.Eq(sym.Matrix([-eq.lhs for eq in el_eqs]),sym.Matrix([0, force]))
el_eqs = el_eqs.subs(subs_dict)
print("Euler-Lagrangian Equations with External Force")
display(el_eqs)

# Solve for Equations of Motion
eqs_mt = sym.solve(el_eqs, qddot_sym)
print("Equations of Motion:")
display(sym.Eq(qddot_sym[0], eqs_mt[qddot_sym[0]]))
display(sym.Eq(qddot_sym[1], eqs_mt[qddot_sym[1]]))
```

```
Lagrangian:
```

$$-Rgm\left(1-\cos\left(\theta\right)\right)+\frac{m\left(R^{2}\dot{\psi}^{2}\sin^{2}\left(\theta\right)+R^{2}\dot{\theta}^{2}\right)}{2}$$

```
Euler-Lagrangian Equations with External Force
```

$$\begin{bmatrix}R^{2}\ddot{\theta}m-R^{2}\dot{\psi}^{2}m\sin\left(\theta\right)\cos\left(\theta\right)+Rgm\sin\left(\theta\right)\\R^{2}m\left(\ddot{\psi}\sin\left(\theta\right)+2\dot{\psi}\dot{\theta}\cos\left(\theta\right)\right)\sin\left(\theta\right)\end{bmatrix}=\begin{bmatrix}0\\\tau\end{bmatrix}$$

```
Equations of Motion:
```

$$\ddot{\theta}=\dot{\psi}^{2}\sin\left(\theta\right)\cos\left(\theta\right)-\frac{g\sin\left(\theta\right)}{R}$$

$$\ddot{\psi}=-\frac{2\dot{\psi}\dot{\theta}\cos\left(\theta\right)}{\sin\left(\theta\right)}+\frac{\tau}{R^{2}m\sin^{2}\left(\theta\right)}$$

Problem 2 (30pts)

Consider a point mass in 3D space under the forces of gravity and a radial spring from the origin. The system's Lagrangian is:

$$L=\frac{1}{2}m(\dot{x}^2+\dot{y}^2+\dot{z}^2)-\frac{1}{2}k(x^2+y^2+z^2)-mgz$$

Consider the following rotation matrices, defining rotations about the z, y, and x axes respectively:

$$R_\theta=\begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix},\quad R_\psi=\begin{bmatrix}\cos\psi & 0 & \sin\psi\\ 0 & 1 & 0\\ -\sin\psi & 0 & \cos\psi\end{bmatrix},\quad R_\phi=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\phi & -\sin\phi\\ 0 & \sin\phi & \cos\phi\end{bmatrix}$$

and answer the following three questions:

1. Which, if any, of the transformations $q_\theta = R_\theta q$, $q_\psi = R_\psi q$, or $q_\phi = R_\phi q$ keeps the Lagrangian fixed (invariant)? Is this invariance global or local?
2. Use small angle approximations to linearize your transformation(s) from the first question. The resulting new transformation(s) should have the form $q_\epsilon = q + \epsilon G(q)$. Compute the difference in the Lagrangian $L(q_\epsilon, \dot{q}_\epsilon) - L(q, \dot{q})$ through this/these transformation(s).
3. Apply Noether's theorem to determine a conserved quantity. What does this quantity represent physically? Is there any rationale behind its conservation?

You can solve this problem by hand or use Python's SymPy to do the symbolic computation for you.

Hint 1: For question (1), try to imagine how this system looks. Even though the x, y, and z axes seem to have the same influence on the system, rotation around some axes will influence the Lagrangian more than others will.

Hint 2: Global invariance here means for any magnitude of rotation the Lagrangian will remain fixed.

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

Part (1)

In [4]:
```
# symbols
m, k, g = sym.symbols(r'm, k, g')
th, psi, phi = sym.symbols(r'\theta, \psi, \phi')
x_sym, xdot, xddot = sym.symbols(r'x, \dot{x}, \ddot{x}')
y_sym, ydot, yddot = sym.symbols(r'y, \dot{y}, \ddot{y}')
z_sym, zdot, zddot = sym.symbols(r'z, \dot{z}, \ddot{z}')

# states
x = sym.Function('x')
y = sym.Function('y')
z = sym.Function('z')
q = sym.Matrix([x(t), y(t), z(t)])

# substitution dictionary
subs_dict = {
    x(t):x_sym ,
    x(t).diff(t):xdot ,
    x(t).diff(t).diff(t):xddot ,
    y(t):y_sym ,
    y(t).diff(t):ydot ,
    y(t).diff(t).diff(t):yddot ,
    z(t):z_sym ,
    z(t).diff(t):zdot ,
    z(t).diff(t).diff(t):zddot
}

# rotation matrices
R_th = sym.Matrix([[sym.cos(th), -sym.sin(th), 0],[sym.sin(th), sym.cos(th), 0],[0, 0, 1]])
R_psi = sym.Matrix([[sym.cos(psi), 0, sym.sin(psi)],[0, 1, 0],[-sym.sin(psi), 0, sym.cos(psi)]])
R_phi = sym.Matrix([[1, 0, 0],[0, sym.cos(phi), -sym.sin(phi)],[0, sym.sin(phi), sym.cos(phi)]])

# state variables after rotation
q_th = R_th * q
q_psi = R_psi * q
q_phi = R_phi * q

# a function to return Lagrangian given a set of configuration variables
def CalcLagrangian(q):
  qdot = q.diff(t)
  return sym.Rational(1,2) * m*(qdot[0]**2+qdot[1]**2+qdot[2]**2) - sym.Rational(1,2)*k*(q[0]**2+q[1]**2+q[2]**2) - m*g*q[2]

# Lagrangian after rotaion
L_th = CalcLagrangian(q_th)
print("Lagrangian after rotation around z-axis")
display(sym.simplify(L_th.subs(subs_dict)))

L_psi = CalcLagrangian(q_psi)
print("Lagrangian after rotation around y-axis")
display(sym.simplify(L_psi.subs(subs_dict)))

L_phi = CalcLagrangian(q_phi)
print("Lagrangian after rotation around x-axis")
display(sym.simplify(L_phi.subs(subs_dict)))
```

```
Lagrangian after rotation around z-axis
```

$$-gmz - \frac{k\left(x^2 + y^2 + z^2\right)}{2} + \frac{m\left(\dot{x}^2 + \dot{y}^2 + \dot{z}^2\right)}{2}$$

```
Lagrangian after rotation around y-axis
```

$$gm\left(x\sin\left(\psi\right) - z\cos\left(\psi\right)\right) - \frac{k\left(x^2 + y^2 + z^2\right)}{2} + \frac{m\left(\dot{x}^2 + \dot{y}^2 + \dot{z}^2\right)}{2}$$

```
Lagrangian after rotation around x-axis
```

$$-gm\left(y\sin\left(\phi\right) + z\cos\left(\phi\right)\right) - \frac{k\left(x^2 + y^2 + z^2\right)}{2} + \frac{m\left(\dot{x}^2 + \dot{y}^2 + \dot{z}^2\right)}{2}$$

Shown in the results above, we can see that transformation $q_\theta = R_\theta q$ keeps Lagrangian fixed. This invariance is global.

Part (2)

In [5]:
```
# small angle approximation
q_th_lin = q_th.subs({sym.sin(th):th, sym.cos(th):1})
q_psi_lin = q_psi.subs({sym.sin(psi):psi, sym.cos(psi):1})
q_phi_lin = q_phi.subs({sym.sin(phi):phi, sym.cos(phi):1})

# linearization
G_th = (q_th_lin-q)/th
display(sym.Symbol(r'q_{\theta} = q + G_{\theta} \theta , \text{where $G_{\theta}$ is:}'), G_th.subs(subs_dict))
G_psi = (q_psi_lin-q)/psi
display(sym.Symbol(r'q_{\psi} = q + G_{\psi} \psi , \text{where $G_{\psi}$ is:}'), G_psi.subs(subs_dict))
G_phi = (q_phi_lin-q)/phi
display(sym.Symbol(r'q_{\phi} = q + G_{\phi} \phi , \text{where $G_{\phi}$ is:}'), G_phi.subs(subs_dict))
```

$q_\theta = q + G_\theta \theta$, where G_θ is:

$$\begin{bmatrix} -y \\ x \\ 0 \end{bmatrix}$$

$q_\psi = q + G_\psi \psi$, where G_ψ is:

$$\begin{bmatrix} z \\ 0 \\ -x \end{bmatrix}$$

$q_\phi = q + G_\phi \phi$, where G_ϕ is:

$$\begin{bmatrix} 0 \\ -z \\ y \end{bmatrix}$$

In [6]:
```
# original Lagrangian
L_origin = CalcLagrangian(q)
L_origin_subs = L_origin.subs(subs_dict)
print("Original Lagrangian:")
display(L_origin_subs)

print("------ Rotation around z-axis ------")
L_th_lin = CalcLagrangian(q_th_lin)
L_th_lin_subs = sym.simplify(L_th_lin.subs(subs_dict))

print("Lagrangian after rotation around z-axis and linearized by small angle approximation:")
display(L_th_lin_subs)
# L_th_lin_eval = L_th_lin_subs.subs(th, 0)
dLdth_th_lin_subs = L_th_lin_subs.diff(th)

print("The derivative of Lagrangian with respect to theta:")
display(dLdth_th_lin_subs)

print("The derivative of Lagrangian with respect to theta, evaluated at theta=0:")
display(dLdth_th_lin_subs.subs(th, 0))

print("Difference between original Lagrangian:")
# display(L_origin_subs - L_th_lin_eval)
display(sym.simplify(L_origin_subs - L_th_lin_subs))
```

Original Lagrangian:

$$-gmz - \frac{k\left(x^2 + y^2 + z^2\right)}{2} + \frac{m\left(\dot{x}^2 + \dot{y}^2 + \dot{z}^2\right)}{2}$$

------ Rotation around z-axis ------
Lagrangian after rotation around z-axis and linearized by small angle approximation:

$$-gmz - \frac{k\left(z^2 + (\theta x + y)^2 + (\theta y - x)^2\right)}{2} + \frac{m\left(\dot{z}^2 + (\dot{x} - \dot{y}\theta)^2 + (\dot{x}\theta + \dot{y})^2\right)}{2}$$

The derivative of Lagrangian with respect to theta:

$$-\frac{k\left(2x\left(\theta x + y\right) + 2y\left(\theta y - x\right)\right)}{2} + \frac{m\left(2\dot{x}\left(\dot{x}\theta + \dot{y}\right) - 2\dot{y}\left(\dot{x} - \dot{y}\theta\right)\right)}{2}$$

The derivative of Lagrangian with respect to theta, evaluated at theta=0:

$$0$$

Difference between original Lagrangian:

$$\frac{\theta^2\left(-\dot{x}^2 m - \dot{y}^2 m + kx^2 + ky^2\right)}{2}$$

In [7]:
```
print("------ Rotation around y-axis ------")
L_psi_lin = CalcLagrangian(q_psi_lin)
L_psi_lin_subs = sym.simplify(L_psi_lin.subs(subs_dict))

print("Lagrangian after rotation around y-axis and linearized by small angle approximation:")
display(L_psi_lin_subs)
# L_psi_lin_eval = L_psi_lin_subs.subs(psi, 0)
dLdpsi_psi_lin_subs = L_psi_lin_subs.diff(psi)

print("The derivative of Lagrangian with respect to psi:")
display(dLdpsi_psi_lin_subs)

print("The derivative of Lagrangian with respect to psi, evaluated at psi=0:")
display(dLdpsi_psi_lin_subs.subs(psi, 0))

print("Difference between original Lagrangian:")
# display(L_origin_subs - L_psi_lin_eval)
display(sym.simplify(L_origin_subs - L_psi_lin_subs))
```

------ Rotation around y-axis ------
Lagrangian after rotation around y-axis and linearized by small angle approximation:

$$gm\left(\psi x - z\right) - \frac{k\left(y^2 + (\psi x - z)^2 + (\psi z + x)^2\right)}{2} + \frac{m\left(\dot{y}^2 + (\dot{x} + \dot{z}\psi)^2 + (\dot{x}\psi - \dot{z})^2\right)}{2}$$

The derivative of Lagrangian with respect to psi:

$$gmx - \frac{k\left(2x\left(\psi x - z\right) + 2z\left(\psi z + x\right)\right)}{2} + \frac{m\left(2\dot{x}\left(\dot{x}\psi - \dot{z}\right) + 2\dot{z}\left(\dot{x} + \dot{z}\psi\right)\right)}{2}$$

The derivative of Lagrangian with respect to psi, evaluated at psi=0:

$$gmx$$

Difference between original Lagrangian:

$$\frac{\psi\left(-\dot{x}^2\psi m - \dot{z}^2\psi m + \psi k x^2 + \psi k z^2 - 2gmx\right)}{2}$$

In [8]:
```
print("------ Rotation around x-axis ------")
L_phi_lin = CalcLagrangian(q_phi_lin)
L_phi_lin_subs = sym.simplify(L_phi_lin.subs(subs_dict))

print("Lagrangian after rotation around x-axis and linearized by small angle approximation:")
display(L_phi_lin_subs)
# L_phi_lin_eval = L_phi_lin_subs.subs(phi, 0)
dLdphi_phi_lin_subs = L_phi_lin_subs.diff(phi)

print("The derivative of Lagrangian with respect to phi:")
display(dLdphi_phi_lin_subs)

print("The derivative of Lagrangian with respect to phi, evaluated at phi=0:")
display(dLdphi_phi_lin_subs.subs(phi, 0))

print("Difference between original Lagrangian:")
# display(L_origin_subs - L_phi_lin_eval)
display(sym.simplify(L_origin_subs - L_phi_lin_subs))
```

```
------ Rotation around x-axis ------
Lagrangian after rotation around x-axis and linearized by small angle approximation:
```

$$-gm\left(\phi y+z\right)-\frac{k\left(x^{2}+\left(\phi y+z\right)^{2}+\left(\phi z-y\right)^{2}\right)}{2}+\frac{m\left(\dot{x}^{2}+\left(\dot{y}-\dot{z}\phi\right)^{2}+\left(\dot{y}\phi+\dot{z}\right)^{2}\right)}{2}$$

```
The derivative of Lagrangian with respect to phi:
```

$$-gmy-\frac{k\left(2y\left(\phi y+z\right)+2z\left(\phi z-y\right)\right)}{2}+\frac{m\left(2\dot{y}\left(\dot{y}\phi+\dot{z}\right)-2\dot{z}\left(\dot{y}-\dot{z}\phi\right)\right)}{2}$$

```
The derivative of Lagrangian with respect to phi, evaluated at phi=0:
```

$$-gmy$$

```
Difference between original Lagrangian:
```

$$\frac{\phi\left(-\dot{y}^{2}\phi m-\dot{z}^{2}\phi m+\phi ky^{2}+\phi kz^{2}+2gmy\right)}{2}$$

Part (3)

In [9]:
```
# conserved quantity for first transformation
conserved_th = sym.Matrix([L_origin.diff(state.diff(t)) for state in q]).T.subs(subs_dict) * G_th.subs(subs_dict)
print("Conserved quantity for first transformation is:")
display(conserved_th[0])
```

```
Conserved quantity for first transformation is:
```

$$-\dot{x}my+\dot{y}mx$$

what does each quantity represent physically?

According to the results above, this quantity is the angular momentum around z axis.

Is there any rationale behind this conservation?

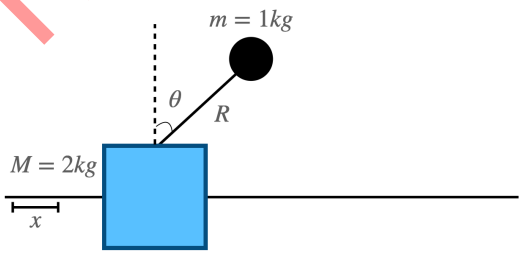
First, in the lecture note there is a similar example with no influence from the gravity. In that case, according the lecture note:

> ... the spring force always points into the origin and thus can never exert a torque about it. There are no other potential sources of torque about this point, and thus the angular momentum about it is constant.

In this case, the spring force is still always pointing to the origin, thus it exerts no torque about it. However, we also have a gravity force aligned with z-axis. Thus, the transformations around y-axis and x-axis are not invariance, but the transformation around the z-axis is a global invariance. Actually, if we think about this problem geometrically, it's not hard to find that the transformation around z-axis is actually a symmetry, which can also explain this conservation.

Problem 3

For the inverted cart-pendulum system in Homework 1 (feel free to make use of the provided solutions), compute the conserved momentum using Nöther's theorem. Plot the momentum for the same simulation parameters and initial conditions. Taking into account the conserved quantities, what is the minimal number of *states* in the cart/pendulum system that can vary? (Hint: In some coordinate systems it may *look* like all the states are varying, but if you choose your coordinates cleverly fewer states will vary.)



Turn in: A copy of code used to calculate the conserved quantity and your answer to the question. You don't need to turn in equations of motion, but you need to include the plot of the conserved quantity evaluated along the system trajectory.

For this problem we can reuse your simulation code for the Cart-Pendulum in Homework 1, and evaluating our conserved quantity from Noether's theorem over the course of the simulation. The conserved quantity is the following:

$$\frac{\partial L}{\partial \dot{q}}G(q) = (m+M)\frac{d}{dt}x(t) - mRcos(\theta(t))\frac{d}{dt}\dot{\theta}(t)$$

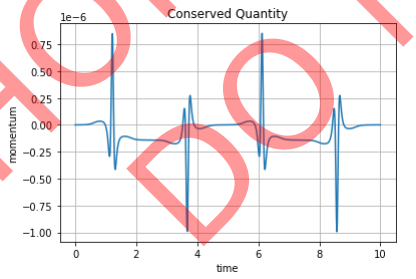
which tells us that locally our x momentum is conserved. Implying that we could use a single state variable (i.e. θ) to fully specify the cart-pendulum dynamics locally.

In [10]:
```
# define constants (as symbols)
t, m, M, R, g = sym.symbols(r't, m, M, R, g')
# define states and time derivatives
x = sym.Function(r'x')(t)
th = sym.Function(r'\theta')(t)
q = sym.Matrix([x, th])

xdot = x.diff(t)
thdot= th.diff(t)
qdot = q.diff(t)

xddot = xdot.diff(t)
thddot = thdot.diff(t)
qddot = qdot.diff(t)
# compute pendulum's velocity in x and y direction
# this is under the "world frame", from where we measure
# the position of the cart as x(t)
pen_xdot = xdot + R*thdot*sym.cos(th)
pen_ydot = R*thdot*sym.sin(th)

# compute Lagrangian
ke = 0.5*M*xdot**2 + 0.5*m*(pen_xdot**2 + pen_ydot**2)
pe = m*g*R*sym.cos(th)
L = ke - pe

# compute the derivatives we needed
L = sym.Matrix([L])
dLdq = L.jacobian(q).T
dLdqdot = L.jacobian(qdot).T
ddLdqdot_dt = dLdqdot.diff(t)

# define Euler-Lagrange equations
el_eqns = sym.Eq(ddLdqdot_dt-dLdq, sym.Matrix([0, 0]))
# solve Euler-Lagrange equations
el_solns = sym.solve(el_eqns, [qddot[0], qddot[1]])

# numerically evaluate
# substitute constants into symbolic solutions
subs_dict = {m:1, M:2, g:9.8, R:1}
xddot_sol = el_solns[xddot].subs(subs_dict)
thddot_sol = el_solns[thddot].subs(subs_dict)
# evaluate solutions as numerical functions
xddot_func = sym.lambdify([x, th, xdot, thdot], xddot_sol)
thddot_func = sym.lambdify([x, th, xdot, thdot], thddot_sol)

def dyn(s):
    return np.array([s[2], s[3], xddot_func(s[0], s[1], s[2], s[3]), thddot_func(s[0], s[1], s[2], s[3])])

s0 = np.array([0, 0.1, 0, 0])
traj = simulate(dyn, s0, [0, 10], 0.01, integrate)
```

In [11]:
```
G = sym.Matrix([1, 0])
p = dLdqdot.T*G
p = sym.simplify(p[0])
print("Conserved quantity: ")
display(p)

p_subs = p.subs(subs_dict)
p_new = sym.lambdify([x, th, xdot, thdot], p_subs)

P = []
for t in traj.T:
    sol1 = p_new(*t)
    P.append(sol1)

# plot it
x_list = np.linspace(0, 10, 1000)
plt.plot(x_list, P)
plt.title('Conserved Quantity')
plt.ylabel('momentum')
plt.xlabel('time')
plt.grid()
plt.show()
```

```
Conserved quantity:
```

$$1.0M\frac{d}{dt}x(t) + m\left(R\cos\left(\theta(t)\right)\frac{d}{dt}\theta(t) + \frac{d}{dt}x(t)\right)$$



Problem 4

Using the same inverted cart pendulum system, add a constraint such that the pendulum follows the path of a parabola with a vertex of $(0, 1)$.

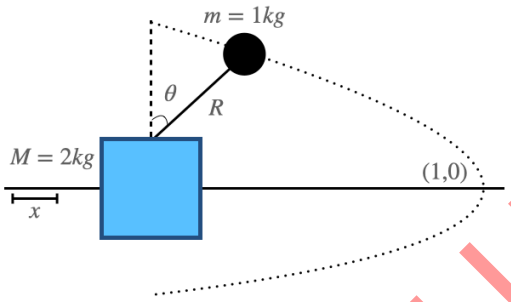
Then, simulate the system using x and θ as the configuration variables for $t \in [0, 15]$ with $dt = 0.01$. The constants are $M = 2, m = 1, R = 1, g = 9.8$. Use the initial conditions $x = 0, \theta = 0, \dot{x} = 0, \dot{\theta} = 0.01$ for your simulation.

You should use the Runge–Kutta integration function provided in previous homework for simulation. Plot the simulated trajectory for x, θ versus time. We have a provided an animation function for testing.

Hint 1: You will need the time derivatives of ϕ to solve the system of equations.

Hint 2: Make sure to be solving for λ at the same time as your equations of motion.

Hint 3: Note that if you make your initial condition velocities faster or dt lower resolution, you may not be able to simulate the system because this is challenging constraint.



Turn in: A copy of code used to simulate the system, you don't need to turn in equations of motion, but you need to include the plot of the simulated trajectories.

In [12]:
```
#1. FIND A SYMBOLIC EXPRESSION FOR THE LAGRANGIAN
# define constants (as symbols)
m, M, R, g, lamb, t = sym.symbols(r'm, M, R, g, \lambda, t')

# define states and time derivatives
x = sym.Function(r'x')(t)
th = sym.Function(r'\theta')(t)
q = sym.Matrix([x, th])

xdot = x.diff(t)
thdot= th.diff(t)
qdot = q.diff(t)

xddot = xdot.diff(t)
thddot = thdot.diff(t)
qddot = qdot.diff(t)

# compute pendulum's position and velocity in x and y direction
# this is under the "world frame", from where we measure
# the position of the cart as x(t)
pen_x = x+R*sym.sin(th)
pen_y = R*sym.cos(th)
pen_xdot = pen_x.diff(t)
pen_ydot = pen_y.diff(t)

# Lagrangian
ke = sym.Rational(1,2)*M*xdot**2 + sym.Rational(1,2)*m*(pen_xdot**2 + pen_ydot**2)
pe = m*g*pen_y
L = ke - pe
print('Lagrangian: ')
display(L.simplify())
```

```
Lagrangian:
```

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

In [13]:
```
## 2. NEW EL EQUATIONS INCLUDING THE CONSTRAINT

# Constraint
phi = pen_x + pen_y**2-1
grad_phi = sym.Matrix([phi]).jacobian(q).T

# EL equations
L = sym.Matrix([L])
dLdq = L.jacobian(q).T
dLdqdot = L.jacobian(qdot).T

ddLdqdot_dt = dLdqdot.diff(t)

EL_lhs = ddLdqdot_dt-dLdq
full_EL_lhs = sym.Matrix([EL_lhs[0].simplify(), EL_lhs[1].simplify(), phi, phi.diff(t).simplify(), phi.diff(t).diff(t).simplify()]) # Give hint for this s
tep. E.g., you need the time derivatives of phi for this to work
full_EL_rhs = sym.Matrix([lamb*grad_phi[0],lamb*grad_phi[1],0,0,0])

# define Euler-Lagrange equations
el_eqns = sym.Eq(full_EL_lhs, full_EL_rhs)
print('\nEuler-Lagrange equations: ')
display(el_eqns.simplify())
```

Euler-Lagrange equations:

$$\begin{bmatrix} \lambda \\ R\lambda\left(-R\sin\left(2\theta(t)\right)+\cos\left(\theta(t)\right)\right) \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} M\frac{d^2}{dt^2}x(t)+m\left(-R\sin\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2+R\cos\left(\theta(t)\right)\frac{d^2}{dt^2}\theta(t)+\frac{d^2}{dt^2}x(t)\right) \\ Rm\left(R\frac{d^2}{dt^2}\theta(t)-g\sin\left(\theta(t)\right)+\cos\left(\theta(t)\right)\frac{d^2}{dt^2}x(t)\right) \\ R^2\cos^2\left(\theta(t)\right)+R\sin\left(\theta(t)\right)+x(t)-1 \\ -R^2\sin\left(2\theta(t)\right)\frac{d}{dt}\theta(t)+R\cos\left(\theta(t)\right)\frac{d}{dt}\theta(t)+\frac{d}{dt}x(t) \\ -R^2\sin\left(2\theta(t)\right)\frac{d^2}{dt^2}\theta(t)-2R^2\cos\left(2\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2-R\sin\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2+R\cos\left(\theta(t)\right)\frac{d^2}{dt^2}\theta(t)+\frac{d^2}{dt^2}x(t) \end{bmatrix}$$

In [14]:
```
#3. SOLVE EL EQUATIONS
# solve Euler-Lagrange equations
el_solns = sym.solve(el_eqns, [qddot[0], qddot[1], lamb])
print('\nSymbolic solution for xddot:')
xdd_sol = el_solns[xddot].simplify()
display(xdd_sol)
print('\nSymbolic solution for thddot:')
thdd_sol = el_solns[thddot].simplify()
display(thdd_sol)
print('\nSymbolic solution for lambda:') # Another place where a hint could be useful. Students don't always know to solve for lambda.
lamb_sol = el_solns[lamb].simplify()
display(lamb_sol)
```

Symbolic solution for xddot:

$$\frac{m\left(4R^3\sin^4\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2-8R^3\sin^2\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2+4R^3\left(\frac{d}{dt}\theta(t)\right)^2-2R^2\sin^3\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2+2Rg\sin\left(\theta(t)\right)\cos\left(\theta(t)\right)+R\left(\frac{d}{dt}\theta(t)\right)^2-g\cos\left(\theta(t)\right)\right)\sin\left(\theta(t)\right)}{\frac{MR^2\left(1-\cos\left(4\theta(t)\right)\right)}{2}-2MR\sin\left(\theta(t)\right)\cos^2\left(\theta(t)\right)-MR\sin\left(2\theta(t)\right)\cos\left(\theta(t)\right)+M\cos^2\left(\theta(t)\right)+\frac{R^2m\left(1-\cos\left(4\theta(t)\right)\right)}{2}-m\cos^2\left(\theta(t)\right)+m}$$

Symbolic solution for thddot:

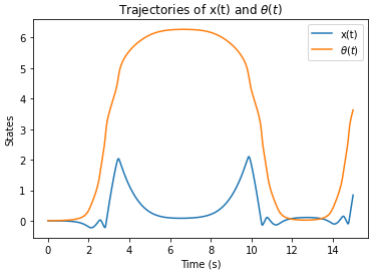
$$\frac{-MR^3\sin\left(4\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2+\frac{MR^2\cos\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2}{2}+\frac{3MR^2\cos\left(3\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2}{2}+\frac{MR\sin\left(2\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2}{2}-R^3m\sin\left(4\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2-\frac{Rm\sin\left(2\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2}{2}+gm\sin\left(\theta(t)\right)}{R\left(\frac{MR^2\left(1-\cos\left(4\theta(t)\right)\right)}{2}-2MR\sin\left(\theta(t)\right)\cos^2\left(\theta(t)\right)-MR\sin\left(2\theta(t)\right)\cos\left(\theta(t)\right)+M\cos^2\left(\theta(t)\right)+\frac{R^2m\left(1-\cos\left(4\theta(t)\right)\right)}{2}-m\cos^2\left(\theta(t)\right)+m\right)}$$

Symbolic solution for lambda:

$$\frac{m\left(2MR^2\cos\left(2\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2+\frac{MRg\cos\left(\theta(t)\right)}{2}-\frac{MRg\cos\left(3\theta(t)\right)}{2}+MR\sin\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2-\frac{Mg\sin\left(2\theta(t)\right)}{2}-\frac{R^2m\left(1-\cos\left(2\theta(t)\right)\right)^2\left(\frac{d}{dt}\theta(t)\right)^2}{2}+\frac{Rgm\cos\left(\theta(t)\right)}{2}-\frac{Rgm\cos\left(3\theta(t)\right)}{2}\right)}{\frac{MR^2\left(1-\cos\left(4\theta(t)\right)\right)}{2}-2MR\sin\left(\theta(t)\right)\cos^2\left(\theta(t)\right)-MR\sin\left(2\theta(t)\right)\cos\left(\theta(t)\right)+M\cos^2\left(\theta(t)\right)+\frac{R^2m\left(1-\cos\left(4\theta(t)\right)\right)}{2}-m\cos^2\left(\theta(t)\right)+m}$$

In [15]:
```
# #4.TURN SYMBOLIC EL EQUATIONS INTO NUMERICAL FUNCTIONS
# Substitute constants
subs_dict = {m:1, g:9.8, R:1, M:2}
xddot_sol = xdd_sol.subs(subs_dict).simplify()
thddot_sol = thdd_sol.subs(subs_dict).simplify()

# Lambdifying
xddot_func = sym.lambdify([x, th, xdot, thdot], xddot_sol)
thddot_func = sym.lambdify([x, th, xdot, thdot], thddot_sol)

# Define dynamics
def cart_pendulum_dyn(s):
    return np.array([s[2], s[3], xddot_func(s[0], s[1], s[2], s[3]), thddot_func(s[0], s[1], s[2], s[3])])
```

In [16]:
```
# #SIMULATE THE SYSTEM
# Simulation
x0 = np.array([0, 0, 0, 0.01]) # Making velocities faster or dt lower resolution will break the simulation, it barely keeps it together as is.
dt = 0.01
tspan = [0, 15]
traj = simulate(cart_pendulum_dyn, x0, tspan, dt, integrate)

# Plot
plt.plot(np.linspace(tspan[0],tspan[1], len(traj[0])), traj[0:2].T)
plt.title(r'Trajectories of x(t) and $\theta(t)$')
plt.xlabel("Time (s)")
plt.ylabel("States")
plt.legend(('x(t)', r'$\theta(t)$'), loc='upper right')
plt.show()
```



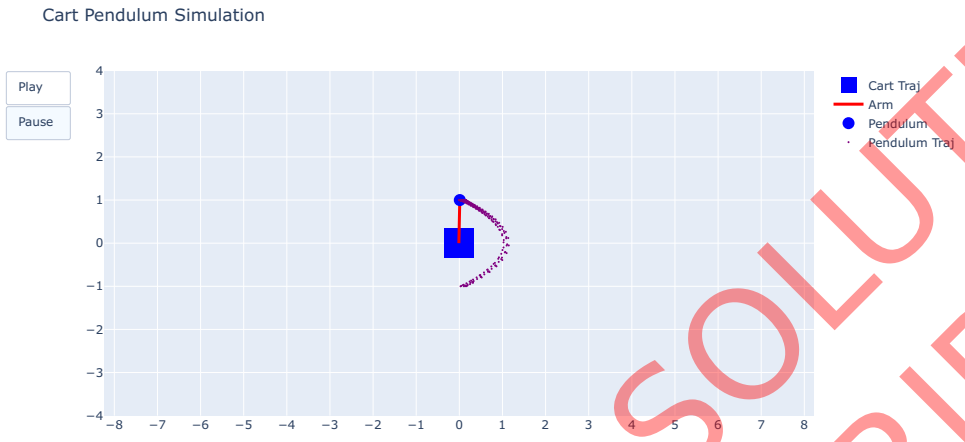
Animation function

In [17]:

```
def animate_cart_pend(traj_array,R=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ================================================
    traj_array:
        trajectory of theta and x, should be a NumPy array with
        shape of (2,N)
    R:
        length of the 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.
    xcart=traj_array[0]
    ycart = 0.0*np.ones(traj_array[0].shape)
    N = len(traj_array[1])

    xx1=xcart+R*np.sin(traj_array[1])
    yy1=R*np.cos(traj_array[1])

     # Need this for specifying length of simulation

    ###################################
    # Using these to specify axis limits.
    xm=-4
    xM= 4
    ym=-4
    yM= 4

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

    ###############################
    # Preparing simulation layout.
    # Title and axis ranges are here.
    layout=dict(xaxis=dict(range=[xm, xM], autorange=False, zeroline=False,dtick=1),
                yaxis=dict(range=[ym, yM], autorange=False, zeroline=False,scaleanchor = "x",dtick=1),
                title='Cart 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=[go.Scatter(
                            x=[xcart[k]],
                            y=[ycart[k]],
                            mode="markers",
                            marker_symbol="square",
                            marker=dict(color="blue", size=30)),
```

```
                    dict(x=[xx1[k],xcart[k]],
                         y=[yy1[k],ycart[k]],
                         mode='lines',
                         line=dict(color='red', width=3)
                         ),
                    go.Scatter(
                         x=[xx1[k]],
                         y=[yy1[k]],
                         mode="markers",
                         marker=dict(color="blue", size=12)),

                    ]) for k in range(N)]

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

In [18]:
```
animate_cart_pend(traj[:,::4], R=1, T=6)
```



In [ ]:
```

```