DavidK_hw4_FA2024

November 1, 2024

1 ME314 Homework 4

###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.

[12]:
```
##########################################################################################
# If you're using Google Colab, uncomment this section by selecting the whole␣
↪section and press
# ctrl+'/' on your and keyboard. Run it before you start programming, this will␣
↪enable the nice
# LaTeX "display()" function for you. If you're using the local Jupyter␣
↪environment, leave it alone
```

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

[11]:
```
from IPython.core.display import HTML, Markdown
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/
↪raw/master/dynhoop2.png' width=528.4' height='500'></table>"))
```

```
<IPython.core.display.HTML object>
```

1.1 Problem 1 (20pts)

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.

[13]:
```
###########################################################
###########################################################
###################### #1 MY SOLUTION ####################
###########################################################
###########################################################
t = sym.symbols('t')
m, R, g, tau = sym.symbols('m R g tau')
theta = sym.Function('theta')(t)
psi = sym.Function('psi')(t)
theta_dot = sym.diff(theta, t)
psi_dot = sym.diff(psi, t)

v_theta = R * theta_dot
v_psi = R * sym.sin(theta) * psi_dot

KE = (1/2) * m * (v_theta**2 + v_psi**2)
V = m * g * R * (1 - sym.cos(theta))
L = KE - V

q = [theta, psi]
qd = [theta_dot, psi_dot]
```

```
L_theta = sym.diff(L, theta) - sym.diff(sym.diff(L, theta_dot), t)
L_theta = sym.simplify(L_theta)
L_psi = sym.diff(L, psi) - sym.diff(sym.diff(L, psi_dot), t) + tau
L_psi = sym.simplify(L_psi)
L_theta_eq = sym.Eq(sym.simplify(L_theta), 0)
L_psi_eq = sym.Eq(sym.simplify(L_psi), 0)
d_theta2 = sym.solve(L_theta_eq, sym.diff(theta, t, 2))[0]
d_psi2 = sym.solve(L_psi_eq, sym.diff(psi, t, 2))[0]

d_theta2, d_psi2
```

```
<IPython.core.display.HTML object>
```

[13]:

$$\left(\frac{\left(R\cos\left(\theta(t)\right)\left(\frac{d}{dt}\psi(t)\right)^2 - g\right)\sin\left(\theta(t)\right)}{R}, \ \frac{-R^2 m \sin\left(2.0\theta(t)\right)\frac{d}{dt}\psi(t)\frac{d}{dt}\theta(t) + \tau}{R^2 m \sin^2\left(\theta(t)\right)} \right)$$

1.2 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.

[14]:

```
##########################################################
##########################################################
###################### #2.1 MY SOLUTION ##################
##########################################################
##########################################################
t = sym.symbols('t')
x = sym.Function('x')(t)
y = sym.Function('y')(t)
z = sym.Function('z')(t)
m, g, k = sym.symbols('m g k')
theta, psi, phi = sym.symbols('theta psi phi')

T = (1/2) * m * (sym.diff(x, t)**2 + sym.diff(y, t)**2 + sym.diff(z, t)**2)
V = (1/2) * k * (x**2 + y**2 + z**2) + m * g * z
L = T - V
display(L)

# define the rotations
R_theta = sym.Matrix([[sym.cos(theta), sym.sin(theta), 0],
                      [-sym.sin(theta), sym.cos(theta), 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)]])

def transformed_lagrangian(R):
    """
    Transform the Lagrangian under a given rotation matrix.

    Parameters:
    -----------
    R: A 3x3 rotation matrix representing the transformation
                  to be applied to the coordinates.

    Returns:
    -----------
    sympy.Expr: The transformed Lagrangian after applying the rotation,
```

```
                 calculated as the difference between the new kinetic energy
                 and potential energy.
    """
    coords = sym.Matrix([x, y, z])
    new_coords = R * coords
    new_dots = sym.Matrix([sym.diff(new_coords[0], t),
                           sym.diff(new_coords[1], t),
                           sym.diff(new_coords[2], t)])
    new_K = (1/2) * m * (new_dots[0]**2 + new_dots[1]**2 + new_dots[2]**2)
    new_V = (1/2) * k * (new_coords[0]**2 + new_coords[1]**2 +␣
 ↪new_coords[2]**2) + m * g * new_coords[2]
    return new_K - new_V


L_theta = transformed_lagrangian(R_theta)
L_psi = transformed_lagrangian(R_psi)
L_phi = transformed_lagrangian(R_phi)

print("Initial L:")
display(L)

print("\nRotate about z-axis (L_theta):")
display(L_theta.simplify())

print("\nRotate about y-axis (L_psi):")
display(L_psi.simplify())

print("\nRotate about x-axis (L_phi):")
display(L_phi.simplify())

is_L_theta_invariant = sym.simplify(L_theta - L) == 0
is_L_psi_invariant = sym.simplify(L_psi - L) == 0
is_L_phi_invariant = sym.simplify(L_phi - L) == 0

print("\nInvariance Check:")
print(f"L_theta is invariant: {is_L_theta_invariant}")
print(f"L_psi is invariant: {is_L_psi_invariant}")
print(f"L_phi is invariant: {is_L_phi_invariant}")
print()
print("After the transformation matrix is applied, rotation about z-axis does␣
 ↪not change the lagrangian.")
print("It does not depend on theta so it is globally invariant. It is locally␣
 ↪invariant on all the points of the circle, so it is globally invariant")
```

<IPython.core.display.HTML object>

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

```
Initial L:

<IPython.core.display.HTML object>
```

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

```
Rotate about z-axis (L_theta):

<IPython.core.display.HTML object>
```

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

```
Rotate about y-axis (L_psi):

<IPython.core.display.HTML object>
```

$$-gm\left(x(t)\sin(\psi) + z(t)\cos(\psi)\right) - 0.5k\left(x^2(t) + y^2(t) + z^2(t)\right) + 0.5m\left(\left(\frac{d}{dt}x(t)\right)^2 + \left(\frac{d}{dt}y(t)\right)^2 + \left(\frac{d}{dt}z(t)\right)^2\right)$$

```
Rotate about x-axis (L_phi):

<IPython.core.display.HTML object>
```

$$gm\left(y(t)\sin(\phi) - z(t)\cos(\phi)\right) - 0.5k\left(x^2(t) + y^2(t) + z^2(t)\right) + 0.5m\left(\left(\frac{d}{dt}x(t)\right)^2 + \left(\frac{d}{dt}y(t)\right)^2 + \left(\frac{d}{dt}z(t)\right)^2\right)$$

```
Invariance Check:
L_theta is invariant: True
L_psi is invariant: False
L_phi is invariant: False

After the transformation matrix is applied, rotation about z-axis does not
change the lagrangian.
It does not depend on theta so it is globally invariant. It is locally invariant
on all the points of the circle, so it is globally invariant
```

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).

[22]:
```
###########################################################
###########################################################
###################### #2.2 MY SOLUTION ##################
###########################################################
###########################################################
print('At small angles, we approximate.')
```

```python
theta, psi, phi, epsilon = sym.symbols(r"theta psi phi epsilon")
sin_theta_approx = sym.sin(theta)
display(sym.Eq(sin_theta_approx, epsilon))
cos_theta_approx = sym.cos(theta)
display(sym.Eq(cos_theta_approx, 1))

print('Substituting the values above into the rotation matrices will linearize␣
 ↪them.')
R_theta = sym.Matrix([[1, -epsilon, 0],
                      [epsilon, 1, 0],
                      [0, 0, 1]])
display(R_theta)
coords = sym.Matrix([x, y, z])
new_coords = R_theta * coords
new_dots = sym.Matrix([sym.diff(new_coords[0], t),
                           sym.diff(new_coords[1], t),
                           sym.diff(new_coords[2], t)])

print('New coordinate system after applying the rotation')
display(new_coords)
L_theta = transformed_lagrangian(R_theta)

print("Initial L:")
display(L)

print("\nRotate about z-axis (L_theta):")
display(L_theta.simplify().expand())

theta_diff = sym.simplify(L_theta - L)

print("L - L_theta:")
display(theta_diff)
```

At small angles, we approximate.

<IPython.core.display.HTML object>

$\sin(\theta) = \epsilon$

<IPython.core.display.HTML object>

$\cos(\theta) = 1$

Substituting the values above into the rotation matrices will linearize them.

<IPython.core.display.HTML object>

$$\begin{bmatrix} 1 & -\epsilon & 0 \\ \epsilon & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
New coordinate system after applying the rotation
```

```
<IPython.core.display.HTML object>
```

$$\begin{bmatrix}-\epsilon y(t) + x(t)\\ \epsilon x(t) + y(t)\\ z(t)\end{bmatrix}$$

```
Initial L:
```

```
<IPython.core.display.HTML object>
```

$$0.5M\left(\frac{d}{dt}x(t)\right)^2 - Rgm\cos\left(\theta(t)\right) + 0.5m\left(R^2\sin^2\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2 + \left(R\cos\left(\theta(t)\right)\frac{d}{dt}\theta(t) + \frac{d}{dt}x(t)\right)^2\right)$$

```
Rotate about z-axis (L_theta):
```

```
<IPython.core.display.HTML object>
```

$$-0.5\epsilon^2kx^2(t) - 0.5\epsilon^2ky^2(t) + 0.5\epsilon^2\left(\frac{d}{dt}x(t)\right)^2 + 0.5\epsilon^2\left(\frac{d}{dt}y(t)\right)^2 - gz(t) - 0.5kx^2(t) - 0.5ky^2(t) - 0.5kz^2(t) + 0.5\left(\frac{d}{dt}x(t)\right)^2 + 0.5\left(\frac{d}{dt}y(t)\right)^2 + 0.5\left(\frac{d}{dt}z(t)\right)^2$$

```
L - L_theta:
```

```
<IPython.core.display.HTML object>
```

$$-0.5M\left(\frac{d}{dt}x(t)\right)^2 + Rgm\cos\left(\theta(t)\right) - gz(t) - 0.5k\left(\left(\epsilon x(t) + y(t)\right)^2 + \left(\epsilon y(t) - x(t)\right)^2 + z^2(t)\right) - 0.5m\left(R^2\left(\frac{d}{dt}\theta(t)\right)^2 + 2R\cos\left(\theta(t)\right)\frac{d}{dt}\theta(t)\frac{d}{dt}x(t) + \left(\frac{d}{dt}x(t)\right)^2\right) + 0.5\left(\epsilon\frac{d}{dt}x(t) + \frac{d}{dt}y(t)\right)^2 + 0.5\left(\epsilon\frac{d}{dt}y(t) - \frac{d}{dt}x(t)\right)^2 + 0.5\left(\frac{d}{dt}z(t)\right)^2$$

As seen in this difference equation, everything is scalled with . The smaller it gets the closer the difference will get to 0.

3. Apply Noether's theorem to determine a conserved quantity. What does this quantity represent physically? Is there any rationale behind its conservation?

[21]:
```
###########################################################
###########################################################
###################### #2.3 MY SOLUTION ##################
###########################################################
###########################################################
G = sym.symbols('G')
q_e = new_coords
q = coords
G = (q_e - q) / epsilon
display(G)
```

```
dL_de = L_theta.diff(epsilon)

print('Verify that the assumptions of the Noethers theorem are right')
print('This shows that dL_de is 0')
print('Now we have q = q_e + e*G and dL_de = 0. So, we can use Noethers␣
↪theorem')
dL_de.subs({epsilon: 0})

print('Using Noethers theorem formula')
conserved_quantity = L.diff(coords.diff(t)).T * G
conserved_quantity.simplify()
```

```
<IPython.core.display.HTML object>
```

$$\begin{bmatrix} -y(t) \\ x(t) \\ 0 \end{bmatrix}$$

```
Verify that the assumptions of the Noethers theorem are right
This shows that dL_de is 0
Now we have q = q_e + e*G and dL_de = 0. So, we can use Noethers theorem
Using Noethers theorem formula
```

```
<IPython.core.display.HTML object>
```

[21]:

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

This represents the system's angular momentum about the origin. The spring force always points to the origin. So, this is intuitive because torque about the origin is not created.

[20]:

```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/atulletaylor/
↪ME314Figures/raw/main/hw4problem3.png' width='600' height='350'></td></tr></
↪table>"))
```

```
<IPython.core.display.HTML object>
```

1.3 Problem 3 (20pts)

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.

[ ]:
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/atulletaylor/
↪ME314Figures/raw/main/hw4p4.png' width='600' height='350'></td></tr></
↪table>"))
```

```
<IPython.core.display.HTML object>
```

[19]:
```
import numpy as np
import matplotlib.pyplot as plt

# 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]])
```

```python
# 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)

# Test numerical solutions
s0 = [0, 0.1, 0, 0]


# Define functions to be used for trajectory plotting
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))
    xtraj[:, 0] = x0
    for i in range(1, N):
        xtraj[:, i] = integrate(f, x, dt)
        x = np.copy(xtraj[:, i])

    return xtraj

# Define extended dynamics for the cart pendulum system
def cart_pendulum_dyn(s):
    """
    System dynamics function (extended)
    Parameters
    ============
    s: NumPy array
    s = [x, th, xdot, thdot] is the extended system
    state vector, including the position and
    the velocity of the particle
    Return
    ============
    sdot: NumPy array
    time derivative of input state vector,
    sdot = [xdot, thdot, xddot, thddot]
    """
    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])])
```

```
# Initial conditions
x0 = np.array([0, 0.1, 0, 0])
# Simulate the trajectory
traj = simulate(cart_pendulum_dyn, x0, [0, 10], 0.1, integrate)
```

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.)

[119]:
```
##########################################################
##########################################################
###################### #3 MY SOLUTION ####################
##########################################################
##########################################################

# X conserves momentum, theta does not
G = sym.Matrix([1, 0])
q = sym.Matrix([x, th])

q_e = q + epsilon * G

L_epsilon_problem3 = L.subs({q[0]: q_e[0], q[1]: q_e[1]})
dL_de = L_epsilon_problem3.diff(epsilon).subs({epsilon: 0})

display(dL_de)
print()
print("Above, we verified that dL_de at epsilon = 0, is 0. Now we can apply␣
↪Noether's theorem")

p = (L[0].diff(qdot)).T * G
print()
print()
print("Conserved Property Below:")
numerical_values = {M: 2, m: 1, R: 1, g: 9.8 }
p_val = p[0].subs(numerical_values)
display(p_val)
print()


p_lmbd = sym.lambdify([x, th, xdot, thdot], p_val)
p_val_list = []
for i in range(traj.shape[1]):
  p_val_list.append(p_lmbd(*traj[:, i]))
```

```
initial_conditions = np.array([0, 0.1, 0, 0])
tspan = [0, 10]
dt = 0.01
tvec = np.linspace(tspan[0], tspan[1], traj.shape[1])
plt.figure(figsize=(10, 6))
plt.plot(tvec, p_val_list, label='$x(t)$', color='b')
plt.xlabel('Time (s)')
plt.ylabel('Momentum (kg*m/s)')
plt.title('Momentum as a function of time')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

```
<IPython.core.display.HTML object>
```

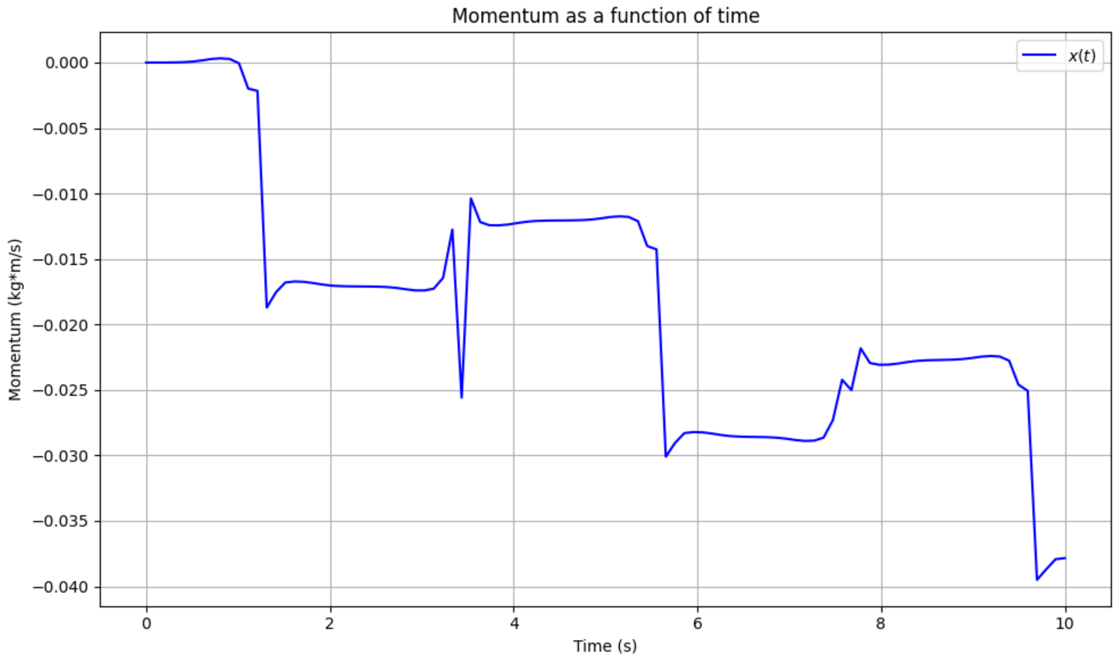
$[0]$

```
Above, we verified that dL_de at epsilon = 0, is 0. Now we can apply Noether's
theorem


Conserved Property Below:
```

```
<IPython.core.display.HTML object>
```

$1.0\cos\left(\theta(t)\right)\frac{d}{dt}\theta(t) + 3.0\frac{d}{dt}x(t)$



1.4 Problem 4 (30 pts)

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

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.

[162]:
```
def animate_cart_pend(traj_array,R=1,T=15):
    """
    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

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

```python
    # 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)
```

[17]:
```
display("Equation")
display(Markdown("$\\frac{y^{2}}{R^{2}} + x = 1$"))
```

'Equation'

$\frac{y^2}{R^2} + x = 1$

[20]:
```
##########################################################
##########################################################
###################### #4 MY SOLUTION ####################
##########################################################
##########################################################

# Define constants (as symbols)
t, m, M, R, g = sym.symbols(r't, m, M, R, g')
```

```python
# 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

y_sys = R * sym.cos(th)
x_sys = R * sym.sin(th) + x
lam = sym.symbols(r'\lambda')
phi_eq = (y_sys ** 2 / R**2) + x_sys - 1
dL_dq_x = sym.diff(L, x)
dL_dq_th = sym.diff(L, th)

dL_dq_dot_x = sym.diff(L, xdot)
dL_dq_dot_th = sym.diff(L, thdot)

dt_dL_dq_dot_x = sym.diff(dL_dq_dot_x, t)
dt_dL_dq_dot_th = sym.diff(dL_dq_dot_th, t)

#eq1
eq1 = sym.Eq(dL_dq_x - dt_dL_dq_dot_x, lam * sym.diff(phi_eq, x)).simplify()
display(eq1)

#eq2
eq2 = sym.Eq(dL_dq_th - dt_dL_dq_dot_th, lam * sym.diff(phi_eq, th)).simplify()
display(eq2)

#eq3
eq3 = sym.Eq(phi_eq.diff(t, 2), 0).simplify()
display(eq3)
```

```python
soln = sym.solve([eq1, eq2, eq3], [xddot, thddot, lam])
numerical_values = {M: 2, m: 1, R: 1, g: 9.8 }


# Run simulate with t = [0, 15], dt = 0.01, and new f_dyn
m = 1
initial_state = [0,0,0,0.01]
tspan = [0, 10]
dt = 0.01

q = sym.Matrix([x.diff(t, 2), y.diff(t, 2), lam])

for sol in soln:
    soln[sol] = soln[sol].subs(numerical_values)

import matplotlib.pyplot as plt

t_values = np.linspace(0, 5, 1000)

x2dot = sym.lambdify([x, th, x.diff(t), th.diff(t)], soln[x.diff(t, 2)].
 ↪simplify())
th2dot = sym.lambdify([x, th, x.diff(t), th.diff(t)], soln[th.diff(t, 2)].
 ↪simplify())

def system_dynamics(s):
    x_val, th_val, xdot_val, thdot_val = s
    xddot_val = x2dot(*s)
    thddot_val = th2dot(*s)
    return np.array([xdot_val,thdot_val, xddot_val, thddot_val])

tspan = [0, 15]
dt = 0.01
initial_conditions = np.array([0, 0, 0, 0.01])
trajectory = simulate(system_dynamics, initial_conditions, tspan, dt, integrate)
tvec = np.linspace(tspan[0], tspan[1], trajectory.shape[1])
plt.figure(figsize=(5, 6))
plt.plot(tvec, trajectory[0], label='$\\theta_1(t)$', color='b')
plt.plot(tvec, trajectory[1], label='$\\theta_1(t)$', color='b')
plt.xlabel('Time (s)')
plt.ylabel('State Variables')
plt.title('Trajectories of $\\theta_1(t)$ and $\\theta_2(t)$')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

```
<IPython.core.display.HTML object>
```

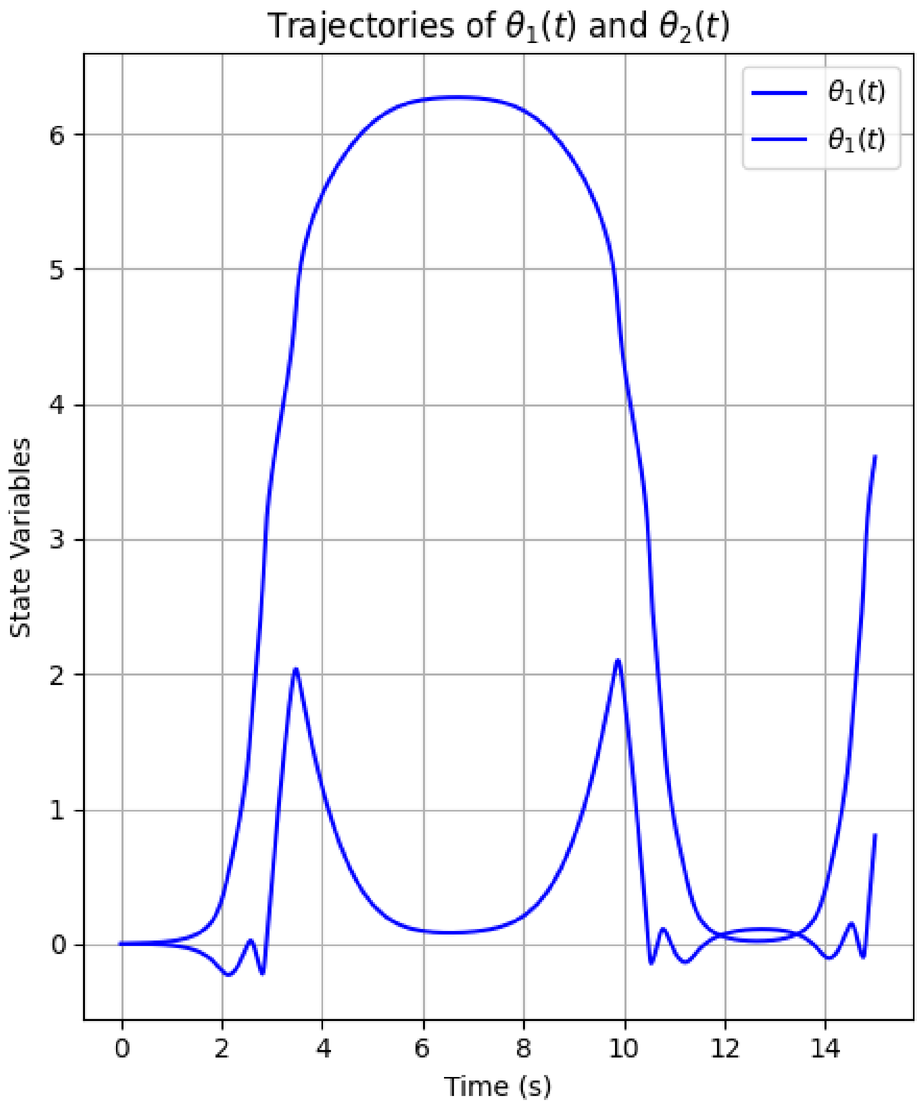
$$\lambda = -1.0M\frac{d^2}{dt^2}x(t) - 1.0m\left(-R\sin(\theta(t))\left(\frac{d}{dt}\theta(t)\right)^2 + R\cos(\theta(t))\frac{d^2}{dt^2}\theta(t) + \frac{d^2}{dt^2}x(t)\right)$$

`<IPython.core.display.HTML object>`

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

`<IPython.core.display.HTML object>`

$$R\sin(\theta(t))\left(\frac{d}{dt}\theta(t)\right)^2 - R\cos(\theta(t))\frac{d^2}{dt^2}\theta(t) + \sin(2\theta(t))\frac{d^2}{dt^2}\theta(t) + 2\cos(2\theta(t))\left(\frac{d}{dt}\theta(t)\right)^2 - \frac{d^2}{dt^2}x(t) = 0$$



[169]:
```
animate_cart_pend(trajectory, R=1, T=15)
```

<IPython.core.display.HTML object>

[ ]:
```
```