DavidK_hw1_FA24

October 8, 2024

1 ME314 Homework 1

###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 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.

[22]:
```python
#IMPORT ALL NECESSARY PACKAGES AT THE TOP OF THE CODE
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Markdown, display
```

[23]:
```python
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)
```

1.1 Problem 1 (15pts)

[24]:
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/
↪raw/master/2mass_spring.png' width=500' height='350'></table>"))
```

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

As shown in the image above, a block of mass m_1 is on one side connected to

---

a wall by a massless spring with a spring constant k_1 and on another side to a block of mass m_2 by a massless spring with a spring constant k_2. Assume that the two springs have lengths of zero when “relaxed”, they are stretched with any positive displacement $\Delta x > 0$, and the magnitude of the force can be computed using Hooke’s law $|F| = k\Delta x$, where k is the spring constant. Furthermore, there is no friction between the blocks and the ground.

Given masses $m_1 = 1kg$ and \$ m_2=2kg\$, spring constants $k_1 = 0.5N/m$ and $k_2 = 0.8N/m$, and positions of the blocks as x_1 and x_2, use Newton’s law $F = ma$ to compute the accelerations of the blocks $a_1 = \ddot{x}_1$ and $a_2 = \ddot{x}_2$. You need to use Pythons’s SymPy package to solve for symbolic solutions, as well as numerically evaluate your solutions for a_1 and a_2 as functions of x_1 and x_2 respectively. Test your numerical functions with $x_1 = 1m$ and $x_2 = 3m$ as function inputs.

Hint 1: You will have two equations based on Newton’s law $F = ma$ for each block. Thus, for each block you need to write down its F in terms of x_1 and x_2 (which can be defined as symbols in SymPy).

*Hint 2: You will need to use SymPy’s **`solve()`** and **`lambdify()`** methods in this problem as seen in Homework 0. This problem is very similar to Problem 5 in Homework 0, except that (1) you need to write down the equations yourself, and (2) you don’t need to solve the equations simultaneously - you can solve them one by one for each block. Feel free to take the example code in Homework 0 as a starting point.*

*Hint 3: You will need to use **`lambdify()`** to numerically evaluate a function with multiple variables. Below is an example.*

[25]:
```
# from sympy.abc import x, y, z or
x,y,z = sym.symbols(r'x,y,z')
expr = sym.sin(x) * y**2 * z + x*z + x/z**2
print('\033[1mExample expression: ')
display(expr)

# method 1: the numerical function will take variables one by one
func = sym.lambdify([x, y, z], expr)
print('\n\033[1mTest at x=1, y=2, z=3: \033[0m{:.2f}'.format(func(1, 2, 3)))
```

```
# method 2: the numerical function will take variables cotained in one list/
↪array
func = sym.lambdify([[x, y, z]], expr)
print('\033[1mTest at x=1, y=2, z=3: \033[0m{:.2f}'.format(func([1, 2, 3])))
```

```
Example expression:

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

$$xz + \frac{x}{z^2} + y^2 z \sin(x)$$

```
Test at x=1, y=2, z=3: 13.21
Test at x=1, y=2, z=3: 13.21
```

Turn in: A copy of the code used to solve for symbolic solutions and evaluate them as numerical functions, output of the code as well as the test results for numerical functions.

[26]:
```
###########################################################
###########################################################
###################### #1 MY SOLUTION ####################
###########################################################
###########################################################



x1,x2,a1,a2,m1,m2,k1,k2 = sym.symbols(r'x1,x2,a1,a2,m1,m2,k1,k2')

# Newtonian forces for the first block
f1_s1 = -k1 * x1 # first spring on the first block
f1_s2 = k2 * (x2 - x1) # second spring on the second block
eq1 = sym.Eq(m1 * a1, f1_s1 + f1_s2)

# Newtonian forces for the second block
f2_s1 = -k2 * (x2 - x1)
eq2 = sym.Eq(m2 * a2, f2_s1)

# only one solution available
x1_2dot = sym.solve(eq1, a1)[0]
x2_2dot = sym.solve(eq2, a2)[0]

# symbolic solution
display(sym.Eq(a1, x1_2dot))
print()
display(sym.Eq(a2, x2_2dot))

numerical_values = {m1: 1, m2: 2, k1: 0.5, k2: 0.8, x1: 1, x2: 3}

# numerical func
```

```
x1_2dot_numerical = sym.lambdify((x1, x2), x1_2dot.subs(numerical_values))
x2_2dot_numerical = sym.lambdify((x1, x2), x2_2dot.subs(numerical_values))

# plug in values for x
accelaration1 = x1_2dot_numerical(1, 3)
accelaration2 = x2_2dot_numerical(1, 3)

print()
print("a1 and a2 have the values below")
accelaration1, accelaration2
```

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

$$a_1 = \frac{-k_1x_1 - k_2x_1 + k_2x_2}{m_1}$$

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

$$a_2 = \frac{k_2(x_1 - x_2)}{m_2}$$

```
a1 and a2 have the values below

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

[26]: $(1.1, \ -0.8)$

1.2 Problem 2 (10pts)

For the same system in Problem 1, compute the Lagrangian of the system using Python's SymPy package with x_1, x_2 as system configuration variables.

Hint 1: For an object with mass m and velocity v, its kinetic energy is $\frac{1}{2}mv^2$.

Hint 2: For a spring stretched with displacement Δx and spring ratio k, its potential energy is $\frac{1}{2}k(\Delta x)^2$. Also, the springs have zero mass.

*Hint 3: Since x_1 and x_2 are actually functions of time t, in order to compute Euler-Lagrange equations you will need to take their derivative with respect to t. Instead of defining position and velocity as two separate symbols, you need to define position as SymPy's **Function** object and the velocity as the derivative of that function with respect to time t. An example is provided below.*

[27]:
```
t = sym.symbols('t')
# define position as function of t
x1 = sym.Function(r'x_1')(t)
# define velocity as derivative of position wrt t
x1dot = x1.diff(t)
# define acceleration as derivative of velocity wrt t
x1ddot = x1dot.diff(t)
```

```
# now write down an expression in terms of position and velocity
expr = x1**2 + x1*x1dot
print('\033[1mExample expression: ')
display(expr)
# you can take the derivative of this expression wrt to time t
expr_dt = expr.diff(t)
print('\n\033[1mDerivative of expression w.r.t. time (t): ')
display(expr_dt)
# or you can take the derivative wrt the functions: x, xdot, xddot
expr_dx = expr.diff(x1)
print('\n\033[1mDerivative of expression w.r.t. x1(t): ')
display(expr_dx)
```

```
Example expression:

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

$$x_1^2(t) + x_1(t)\frac{d}{dt}x_1(t)$$

```
Derivative of expression w.r.t. time (t):

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

$$2x_1(t)\frac{d}{dt}x_1(t) + x_1(t)\frac{d^2}{dt^2}x_1(t) + \left(\frac{d}{dt}x_1(t)\right)^2$$

```
Derivative of expression w.r.t. x1(t):

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

$$2x_1(t) + \frac{d}{dt}x_1(t)$$

Turn in: A copy of the code used to symbolically compute the Lagrangian and corresponding output of the code (i.e. the computed Lagrangian.)

[28]:
```
##########################################################
##########################################################
###################### #2 MY SOLUTION ####################
##########################################################
##########################################################


t, m1, m2, k1, k2 = sym.symbols(r't,m1,m2,k1,k2')

# define position as function of t
x1_t = sym.Function(r'x_1')(t)
x2_t = sym.Function(r'x_2')(t)

# v is just x dot
```

```
v1_t = sym.diff(x1_t, t)
v2_t = sym.diff(x2_t, t)

# kinetic
K1 = (1/2) * m1 * (v1_t**2)
K2 = (1/2) * m2 * (v2_t**2)

# potential
V1 = (1/2) * k1 * (x1_t**2)
V2 = (1/2) * k2 * ((x2_t - x1_t)**2)

# L = kinetic - potential
L = (K1 + K2) - (V1 + V2)
display(L.simplify())
```

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

$$-0.5k_1x_1^2(t) - 0.5k_2\left(x_1(t) - x_2(t)\right)^2 + 0.5m_1\left(\frac{d}{dt}x_1(t)\right)^2 + 0.5m_2\left(\frac{d}{dt}x_2(t)\right)^2$$

1.3 Problem 3 (10pts)

Based on your solution for Problem 2, compute the Euler-Lagrange equations for this system.

Hint 1: In this problem, the system has two configuration variables. Thus, when taking the derivative of the Lagrangian with respect to the system state vector, the derivative is also a vector (sometimes also called the Jacobian of the Lagrangian with respect to system states). Below is an example of several ways to compute the derivative with respect to a vector in SymPy.

```
[29]: print('\033[1mDifferent ways of taking the derivative of the Lagrangian with␣
        ↪respect ot the system state vector (q)')
      t = sym.symbols('t')

      # define two system states as functions
      x1 = sym.Function(r'x_1')(t)
      x2 = sym.Function(r'x_2')(t)
      # wrap system states into one vector (in SymPy would be Matrix)
      q = sym.Matrix([x1, x2])
      # define an expression in terms of x1(t) and x2(t)
      J = sym.sin(x1**2) + sym.cos(x2)**2
      print('\n\033[1mExample expression ')
      display(J)

      # compute derivative wrt a vector, method 1
      # wrap the expression into a SymPy Matrix
      J_mat = sym.Matrix([J])
      # SymPy Matrix has built-in method for Jacobian
      dJdq = J_mat.jacobian(q)
      print('\n\033[1mMethod: built-in Jacobian derivative')
```

```
display(dJdq) # note that dJdq is 1-by-2 vector, not 2-by-1 as q

# compute derivative wrt a vector, method 2: do it one by one
dJdx1 = J.diff(x1)
dJdx2 = J.diff(x2)
dJdq = sym.Matrix([dJdx1, dJdx2]).T # transpose so dimension can match
print('\n\033[1mMethod: one-by-one')
display(dJdq)

# compute derivative wrt a vector, method 3: do it one by one ... but in a␣
↪"fancy" way
dJdq = []
for x in q:
    dJdq.append(J.diff(x))
dJdq = sym.Matrix([dJdq])
print('\n\033[1mMethod: for loop')
display(dJdq)
```

```
Different ways of taking the derivative of the Lagrangian with respect ot

the system state vector (q)

Example expression

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

$\sin\left(x_1^2(t)\right) + \cos^2\left(x_2(t)\right)$

```
Method: built-in Jacobian derivative

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

$\begin{bmatrix} 2x_1(t)\cos\left(x_1^2(t)\right) & -2\sin\left(x_2(t)\right)\cos\left(x_2(t)\right) \end{bmatrix}$

```
Method: one-by-one

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

$\begin{bmatrix} 2x_1(t)\cos\left(x_1^2(t)\right) & -2\sin\left(x_2(t)\right)\cos\left(x_2(t)\right) \end{bmatrix}$

```
Method: for loop

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

$\begin{bmatrix} 2x_1(t)\cos\left(x_1^2(t)\right) & -2\sin\left(x_2(t)\right)\cos\left(x_2(t)\right) \end{bmatrix}$

Turn in: A copy of the code used to symbolically compute the Euler-Lagrange equations and the code outpute (i.e. the resulting equations).

[43]:
```
##########################################################
##########################################################
```

```
###################### #3 MY SOLUTION ####################
##########################################################
##########################################################

# Derivatives for x1
L_simple = L.simplify()
dL_dv1 = sym.diff(L, v1_t)
d_dt_dL_dv1 = sym.diff(dL_dv1, t)
dL_dx1 = sym.diff(L, x1_t)
EL_x1 = sym.Eq(dL_dx1 - d_dt_dL_dv1, 0)

# Derivatives for x2
dL_dv2 = sym.diff(L, v2_t)
d_dt_dL_dv2 = sym.diff(dL_dv2, t)
dL_dx2 = sym.diff(L, x2_t)
EL_x2 = sym.Eq(dL_dx2 - d_dt_dL_dv2, 0)

display(EL_x1.simplify())
display(EL_x2.simplify())
```

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

$$1.0k_1x_1(t) + 1.0k_2\left(x_1(t) - x_2(t)\right) + 1.0m_1\frac{d^2}{dt^2}x_1(t) = 0$$

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

$$1.0k_2\left(x_1(t) - x_2(t)\right) - 1.0m_2\frac{d^2}{dt^2}x_2(t) = 0$$

1.4 Problem 4 (15pts)

Based on your Euler-Lagrange equations from Problem 3, use Python's SymPy package to solve the equations for the accelerations of the two blocks $\ddot{x}_1, \ddot{x}_2$, in terms of position x_1, x_2, and velocity $\dot{x}_1, \dot{x}_2$. Compare your answer to Problem 1 to see if they match with each other and comment if this is not the case.

Hint 1: Since you need to solve a set of multiple equations symbolically in SymPy, it's recommended to wrap both sides of the equations into SymPy's Matrix object. As an example, below is the code solving the following set of equations (feel free to use this code as a starting point).

$$\begin{cases} x^2 + y = 3 \\ x + y = 1 \end{cases}$$

```
[31]: x,y = sym.symbols(r'x,y')
      # define left hand side as a matrix
      lhs = sym.Matrix([x**2+y, x+y])

      # define right hand side as a Matrix
      rhs = sym.Matrix([3, 1])
```

```
# define the equations
eqn = sym.Eq(lhs, rhs)

# solve it for both x and y
q = sym.Matrix([x, y])
soln = sym.solve(eqn, q, dict=True) # this will return the solution
                                     # as a Python dictionary
for sol in soln:
    print('\n\033[1mSolution: ')
    for v in q:
        display(sym.Eq(v, sol[v]))
```

```
Solution:
```

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

$x = -1$

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

$y = 2$

```
Solution:
```

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

$x = 2$

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

$y = -1$

Turn in: A copy of the code you used to solve the Euler-Lagrange equations and the code output (i.e. the resulting symbolic solutions).

[44]:
```
##########################################################
##########################################################
##################### #4 MY SOLUTION ###################
##########################################################
##########################################################

EL_x1_simple = EL_x1.simplify()
EL_x2_simple = EL_x2.simplify()

# solve with respect to x 2 dot
d_dt_dL_dv1_solved = sym.solve(EL_x1_simple, x1_t.diff(t, 2))
display(Markdown(f"Simplified Equation for $a_1$"))
display(sym.Eq(a1, d_dt_dL_dv1_solved[0]))
print()
print()

display(Markdown(f"Simplified Equation for $a_2$"))
d_dt_dL_dv2_solved = sym.solve(EL_x2_simple, x2_t.diff(t, 2))
display(sym.Eq(a2, d_dt_dL_dv2_solved[0]))

numerical_values = {m1: 1, m2: 2, k1: 0.5, k2: 0.8, x1: 1, x2: 3}

# numerical func
x1_2dot_numerical = sym.lambdify((x1, x2), d_dt_dL_dv1_solved[0].
 ↪subs(numerical_values))
x2_2dot_numerical = sym.lambdify((x1, x2), d_dt_dL_dv2_solved[0].
 ↪subs(numerical_values))
accelaration1 = x1_2dot_numerical(1, 3)
accelaration2 = x2_2dot_numerical(1, 3)

print()
display(Markdown("Plugging in"))
display(sym.Eq(a1, accelaration1))
display(sym.Eq(a2, accelaration2))
```

Simplified Equation for a_1

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

$$a_1 = \frac{-k_1 x_1(t) - k_2 x_1(t) + k_2 x_2(t)}{m_1}$$

Simplified Equation for $a_2$

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

$$a_2 = \frac{k_2 \left(x_1(t) - x_2(t)\right)}{m_2}$$

Plugging in

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

$a_1 = 1.1$

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

$a_2 = -0.8$

Explanation: This yields the same result as the Newtonian solution

1.5 Problem 5 (10pts)

[33]:
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://raw.githubusercontent.com/
↪MuchenSun/ME314pngs/master/dyninvpend.png' width=500' height='350'></
↪table>"))
```

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

You are given the unforced inverted cart-pendulum as shown below. The pendulum is in gravity and can swing freely (the cart will not interfere with pendulum's motion). The cart can move freely in the horizontal direction. Take x and θ as the system configuration variables and compute the Lagrangian of the system using Python's SymPy package. Provide symoblic solution of the Lagrangian with m, R, and M as symbols instead of given constants.

Hint 1: Assume that the positive direction for the rotation of the pendulum $\dot{\theta}$ is clockwise. Likewise, the positive direction for the translation of the cart is to the right.

Hint 2: You will need to define m, R, and M as SymPy's symbols.

Hint 3: Note that the pendulum is attached to the cart! In order to compute the kinetic energy of the pendulum, the velocity of the cart needs to be considered. One suggestion is to compute the velocity of the pendulum in a different coordinate system (other than Cartesian with x and y coordinates).

Turn in: A copy of the code used to symbolically compute Lagrangian and the code output (i.e. the computed expression of the Lagrangian).

[34]:
```
###########################################################
###########################################################
###################### #5 MY SOLUTION ####################
###########################################################
###########################################################

# You can start your implementation here :)
# q configuration is theta and x (all the x below means x dot)
# the frame we choose is the pivot point
# kinetic energy of the pendulum bob would be defined
#  1/2m(v^2)
# y_m = Rcos(theta)
# x_m = Rsin(theta) + x
# For kinetic energy we want the magnitude of the velocity
# sqrt(x_m^2 + y_m^2)
# v = (sqrt(x_m^2 + y_m^2))^2 -> x_m^2 + y_m^2
# KE = (1/2)m(R^2cos(theta)^2 + R^2sin(theta)^2 + 2xRsin(theta) + x^2) + (1/
↪2)Mx^2
# PE = mgRcos(theta)


R, m, M, g, t, y_m, x_m = sym.symbols('R m M g t y_m x_m')
x = sym.Function(r'x')(t)
```

```python
theta = sym.Function(r'theta')(t)

pretty_y_m = sym.Eq(y_m, R * sym.cos(theta))
pretty_x_m = sym.Eq(x_m, R * sym.sin(theta) + x)

display(Markdown('**X and Y** have the following configuration'))
display(pretty_x_m)
display(pretty_y_m)

x_m = R * sym.sin(theta) + x
y_m = R * sym.cos(theta)

x_mt = sym.diff(x_m, t)
y_mt = sym.diff(y_m, t)

kinetic_energy_pendulum = (1/2) * m * (x_mt**2 + y_mt**2)
print()
print()
display(Markdown('Pendulum Kinetic Energy'))
display(kinetic_energy_pendulum.simplify())

kinetic_energy_cart = (1/2) * M * (x.diff(t))**2

display(Markdown('Cart Kinetic Energy'))
display(kinetic_energy_cart)

potential_energy_pendulum = m * g * R * sym.cos(theta)

print()
print()
display(Markdown('Pendulum Potential Energy'))
display(potential_energy_pendulum)
display(Markdown('Cart Potential Energy'))
display(0)

Lagrangian = (kinetic_energy_cart + kinetic_energy_pendulum) -␣
  ↪potential_energy_pendulum
print()
print()
display(Markdown('Lagrangian'))
display(Lagrangian.simplify())
```

X and Y have the following configuration

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

$x_m = R \sin(\theta(t)) + x(t)$

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

$y_m = R\cos(\theta(t))$

Pendulum Kinetic Energy

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

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

Cart Kinetic Energy

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

$$0.5M\left(\frac{d}{dt}x(t)\right)^2$$

Pendulum Potential Energy

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

$Rgm\cos(\theta(t))$

Cart Potential Energy

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

0

Lagrangian

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

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

1.6 Problem 6 (15pts)

Based on your solution in Problem 5, compute the Euler-Lagrange equations for this inverted cart-pendulun system using Python's SymPy package.

Turn in: A copy of the code used to symbolically compute Euler-Lagrange equations and the code output (i.e. the computed expression of the Lagrangian).

[45]:
```
############################################################
############################################################
###################### #6 MY SOLUTION ####################
############################################################
```

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

# x
dL_dx_dot = sym.diff(Lagrangian, sym.diff(x, t))
d_dt_dL_dx_dot = sym.diff(dL_dx_dot, t)
dL_dx = sym.diff(L, x)

euler_lagrange_x = dL_dx - d_dt_dL_dx_dot

# theta
dL_dtheta_dot = sym.diff(Lagrangian, sym.diff(theta, t))
d_dt_dL_dtheta_dot = sym.diff(dL_dtheta_dot, t)
dL_dtheta = sym.diff(Lagrangian, theta)

euler_lagrange_theta = dL_dtheta -  d_dt_dL_dtheta_dot

display(sym.Eq(euler_lagrange_x.simplify(), 0))
display(sym.Eq(euler_lagrange_theta.simplify(), 0))
```

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

$$-1.0M\frac{d^2}{dt^2}x(t) - 1.0m\left(-R\sin(\theta(t))\left(\frac{d}{dt}\theta(t)\right)^2 + R\cos(\theta(t))\frac{d^2}{dt^2}\theta(t) + \frac{d^2}{dt^2}x(t)\right) = 0$$

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

$$1.0Rm\left(-R\frac{d^2}{dt^2}\theta(t) + g\sin(\theta(t)) - \cos(\theta(t))\frac{d^2}{dt^2}x(t)\right) = 0$$

1.7 Problem 7 (15pts)

Find symbolic expressions of $\ddot{x}$ and $\ddot{\theta}$ from the Euler-Lagrange equations in Problem 6 using SymPy's **`solve()`** method (NOTE: the expressions should be in terms of x, θ, $\dot{x}$ and $\dot{\theta}$ only). Convert these results to numerical functions using SymPy's **`lambdify()`** method by substituting M, m, R, g symbols with $M = 2, m = 1, R = 1, g = 9.8$ values. Test your numerical functions of $\ddot{x}$ and $\ddot{\theta}$ by evaluating them given $x = 0, \theta = 0.1, \dot{x} = 0, \dot{\theta} = 0$ as function inputs.

Turn in: A copy of the code used to symbolically solve and numerically evaluate the solutions of Euler-Lagrange equations (i.e. $\ddot{x}$ and $\ddot{\theta}$). Include the code output, consisting of symbolic expression of $\ddot{x}$ and $\ddot{\theta}$, as well as your test results for the numerical evaluations.

[46]:
```
##########################################################
##########################################################
###################### #7 MY SOLUTION ####################
##########################################################
##########################################################


# You can start your implementation here :)
```

```
x_solved = sym.solve(euler_lagrange_x, x.diff(t, 2))[0]
theta_solved = sym.solve(euler_lagrange_theta, theta.diff(t, 2))[0]

display(sym.Eq(x.diff(t, 2), x_solved))
print()
display(sym.Eq(theta.diff(t, 2), theta_solved))

numerical_values = {M: 2, m: 1, R: 1, g: 9.8 }


x_solved_numerical = x_solved.subs(numerical_values)
theta_solved_numerical = theta_solved.subs(numerical_values)
print()
print()
display(Markdown("Numerical solutions for x_2dot and theta_2dot"))

display(sym.Eq(x.diff(t, 2), x_solved_numerical))
print()
print()
display(sym.Eq(theta.diff(t, 2), theta_solved_numerical))

# define left hand side as a matrix
lhs = sym.Matrix([x_solved_numerical, theta_solved_numerical])

# define right hand side as a Matrix
rhs = sym.Matrix([x.diff(t, 2), theta.diff(t, 2)])

# define the equations
eqn = sym.Eq(lhs, rhs)

# solve it for both x and y
q = sym.Matrix([x.diff(t, 2), theta.diff(t, 2)])
soln = sym.solve(eqn, q, dict=True) # this will return the solution
                                     # as a Python dictionary

numerical_solutions = {}
symbolic_solutions = {}

for sol in soln:
    print('\n\033[1mSolution: ')
    for v in q:
        display(sym.Eq(v, sol[v]))


        l = sym.lambdify([x, x.diff(t), theta, theta.diff(t)], sol[v])

        display(l(0, 0, 0.1, 0))
        symbolic_solutions[v] = sol[v]
        numerical_solutions[v] = l(0, 0, 0.1, 0)
```

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

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

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

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

Numerical solutions for x_2dot and theta_2dot

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

$$\frac{d^2}{dt^2}x(t) = \frac{\sin\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2}{3} - \frac{\cos\left(\theta(t)\right)\frac{d^2}{dt^2}\theta(t)}{3}$$

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

$$\frac{d^2}{dt^2}\theta(t) = 9.8\sin\left(\theta(t)\right) - \cos\left(\theta(t)\right)\frac{d^2}{dt^2}x(t)$$

Solution:

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

$$\frac{d^2}{dt^2}x(t) = \frac{49.0\sin\left(\theta(t)\right)\cos\left(\theta(t)\right)}{5.0\cos^2\left(\theta(t)\right) - 15.0} - \frac{5.0\sin\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2}{5.0\cos^2\left(\theta(t)\right) - 15.0}$$

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

-0.484326290345888

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

$$\frac{d^2}{dt^2}\theta(t) = \frac{5.0\sin\left(\theta(t)\right)\cos\left(\theta(t)\right)\left(\frac{d}{dt}\theta(t)\right)^2}{5.0\cos^2\left(\theta(t)\right) - 15.0} - \frac{147.0\sin\left(\theta(t)\right)}{5.0\cos^2\left(\theta(t)\right) - 15.0}$$

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

1.46027415938673

1.8 Problem 8 (10pts)

Based on your symbolic and numerical solutions for $\ddot{x}(t)$ and $\ddot{\theta}(t)$, which are now functions of $x(t), \theta(t), \dot{x}(t)$ and $\dot{\theta}(t)$, simulate the system for $t \in [0, 10]$, with initial conditions $x(0) = 0, \theta(0) = 0.1, \dot{x}(0) = 0, \dot{\theta}(0) = 0$ and the numerical integration and simulation functions provided below. Plot the trajectories of $x(t)$ and $\theta(t)$ versus time.

Hint 1: The numerical simulation function below can only simulate systems with first-order dynamics. This means that the function of dynamics (i.e. $f(xt)$), whose input is fed to the numerical integration and simulation functions, needs to return the first-order time derivative. This might be confusing because our solutions $\ddot{x}(t)$ and $\ddot{\theta}(t)$ are second-order time derivative. The trick here is to extend the system state from $[x(t), \theta(t)]$ to $[x(t), \theta(t), \dot{x}(t), \dot{\theta}(t)]$, thus the time derivative of the state vector becomes $[\dot{x}(t), \dot{\theta}(t), \ddot{x}(t), \ddot{\theta}(t)]$. Now, when you write down the system dynamics function, the third and forth elements of input vector $\dot{x}(t)$ and $\dot{\theta}(t)$ can be put into the output vector directly, and we already know the rest two elements of the output vector, from our previous solution of Euler-Lagrange equations. More information can be found in Lecture Note 1 (Background) - Section 1.4 Ordinary Differential Equations. An example is provided below for simulating a particle falling in gravity, where the acceleration $\ddot{x}(t) = -9.8$.

Hint 2: You will need to include the numerical evaluations for $\ddot{x}(t)$ and $\ddot{\theta}(t)$ in system dynamics function. You can either use your previous **`lambdify()`** *results, or hand code the equations from previous symbolic solutions. We recommend using* **`lambdify()`** *for practice because later homeworks will require integration of much more complicated equations.*

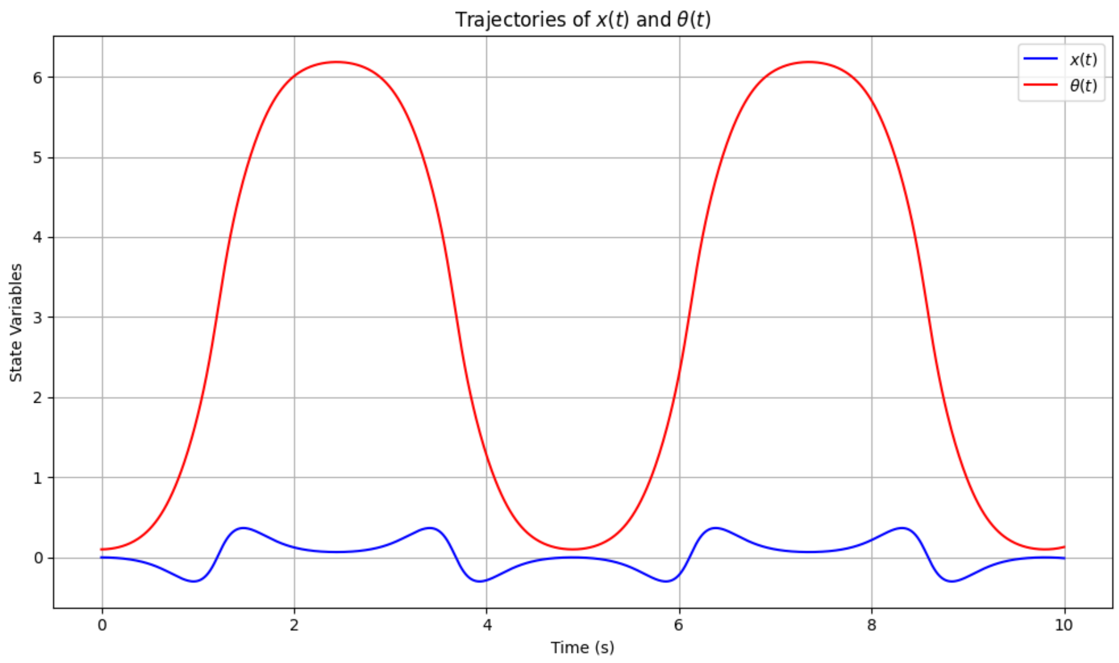
Turn in: A copy of the code used to simulate the system with the plot of the simulated trajectories.

[37]:
```python
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

###############################################
# example: simulate a particle falling in gravity
def xddot(x, xdot):
    """
    Acceleration of the particle in terms of
    position and velocity. Here it's a constant.
    """

    return -9.8

def dyn(s):
    """
    System dynamics function (extended)

    Parameters
    ============
    s: NumPy array
        s = [x, xdot] is the extended system
        state vector, includng the position and
        the velocity of the particle

    Return
    ============
    sdot: NumPy array
        time derivative of input state vector,
        sdot = [xdot, xddot]
    """
    return np.array([s[1], xddot(s[0], s[1])])

# define initial state
s0 = np.array([10, 0]) # at 10m high with zero velocity
# simulat from t=0 to 10, since dt=0.1, the returned trajectory
# will have 10/0.1=100 time steps, each time step contains extended
# system state vector [x(t), xdot(t)]
traj = simulate(dyn, s0, [0, 10], 0.1, integrate)
print('\033[1mShape of traj: \033[0m', traj.shape)
```

```
Shape of traj:  (2, 100)
```

[38]:
```
###########################################################
###########################################################
###################### #8 MY SOLUTION ####################
###########################################################
###########################################################

# You can start your implementation here :)
symbolic_solutions, numerical_solutions

xddot_func = sym.lambdify([x, x.diff(t), theta, theta.diff(t)],␣
 ↪symbolic_solutions[x.diff(t, 2)])
```

```
thetaddot_func = sym.lambdify([x, x.diff(t), theta, theta.diff(t)],␣
 ↪symbolic_solutions[theta.diff(t, 2)])

def system_dynamics(s):
    x_val, theta_val, xdot_val, thetadot_val = s
    xddot_val = xddot_func(x_val, xdot_val, theta_val, thetadot_val)
    thetaddot_val = thetaddot_func(x_val, xdot_val, theta_val, thetadot_val)
    return np.array([xdot_val, thetadot_val, xddot_val, thetaddot_val])

initial_conditions = np.array([0, 0.1, 0, 0])
tspan = [0, 10]
dt = 0.01
trajectory = simulate(system_dynamics, initial_conditions, tspan, dt, integrate)
tvec = np.linspace(tspan[0], tspan[1], trajectory.shape[1])
plt.figure(figsize=(10, 6))
plt.plot(tvec, trajectory[0, :], label='$x(t)$', color='b')
plt.plot(tvec, trajectory[1, :], label='$\\theta(t)$', color='r')
plt.xlabel('Time (s)')
plt.ylabel('State Variables')
plt.title('Trajectories of $x(t)$ and $\\theta(t)$')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



1.9 Problem X (0pts)

1.9.1 Not required for submission!

NOTE: "Problem X" is not required for submission and has zero credit. This is only designed to help you get more familiar with things that are not the primary purposes of this class, but are important in practice, like Python.

You may now have all the code running to compute the dynamics of some "pendulum stuff"… and you will need to compute more later in this class! One suggestion here, for the purpose of handling more sophisticated systems (like a triple-pendulum!), is to wrap up your code for this homework into several functions. This is not required, and it's fine if you don't want to do it (you can always go back here after you're more familiar with Python and course materials). This is only for some of you who just want to have more Python in their life, just as the saying goes, "life is short, use python".

Why is it better to wrap existing code into functions - isn't it a waste of time? Consider a sort of unrelated example: write a program that computes the trace of a matrix, which is the sum of all diagonal elements. If the matrix is already provided, just like we already computed Lagrangian in the homework, we can do something like this:

[39]:
```
mat = [[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]]
trace = mat[0][0] + mat[1][1] + mat[2][2]
```

However, what if we need to compute the trace of a 5-by-5 matrix instead? Then we need to rewrite the code! In our simple example, this isn't that annoying or time consuming. However, for more complicated problems it could be very annoying, especially when we find that we are basically doing the same thing every time we rewrite the code - manually adding all the diagonal elements together as well as very tedious, especially in case if there are multiple places in the code where we perform the same calculation. Thus, why don't we just write a function once, and it will handle matrices with all possible dimensions for us? The challenge is that the function doesn't know what matrix it will be provided ahead of time! One solution is to require matrix dimension as a function input (you might find this very common in lower-level programming languages like C/C++). But in Python, we have better choice - querying the dimension of your input! Here's one possible implementation:

[40]:
```
def compute_trace(mat):
    dim = len(mat)
    trace_val = 0
    for i in range(dim):
        trace_val += mat[i][i]
    return trace_val

mat = [[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]]
trace = compute_trace(mat)
```

Python provides some really nice methods to tell us the dimension of the data! While the default is `len()`, given NumPy or SymPy you can also use `.shape()`. In this way, you can generalize your method through the function, and easily reuse your code later. Besides that, depending on the specific scenarios, some methods can naturally avoid encountering number of parameters, like SymPy's matrix `jacobian()` method. Usually these methods are better choices because their implementations are typically more efficient than using loops (loops are slow).

In this homework, one part of your code you can re-write using a function feature is Problem 3. **Try to write a Python function, which takes in the Lagrangian, a list of variables of the Lagrangian, and returns the Euler-Lagrange equation as a SymPy equation object.** The challenge here is that we don't know how many variables will be provided ahead of time, and the example above might inspire you.

Feel free to use template provided below as a starting point. Note that it's highly recommended to document your function as shown below - letting others understand your code is as important as getting your code to run!

[41]:
```
def euler_equations(L, funcs, t):
    """
    Find the Euler-Lagrangian equations given the Lagrangian equation.

    Parameters:
    ============
    L: SymPy Expression
        L should be a SymPy expression containing necessary system
        configuration variables
    funcs: list of SymPy Functions
        func should included all the system configuration variables
        as functions of time variable "t" in the Lagrangian
    t: SymPy Symbol
        time variable

    Returns:
    ============
    eqns: SymPy Equation
        eqns is the resulting Euler-Lagrangian equations, it should
        be a sinlge SymPy Equation object, each side being a SymPy
        Matrix object
    """
    pass # In Python, "pass" means "do nothing",  many people use it
         # to occupy a space so they don't need to implement a function
         # while getting the program running. Here you just need to
         # replace it with your own implementation.
```

[41]:
```
```