Problem 1

↳ $R(\theta_1) = R_1$ where $R_1^T R_1 = I_{n\times n}$ and $\det(R_1) = 1$

↳ $R(\theta_2) = R_2$ where $R_2^T R_2 = I_{n\times n}$ and $\det(R_2) = 1$

Such that both R_1 and $R_2 \in SO(n)$

For $R_1 R_2 \in SO(n)$, two things must be true:

1) $(R_1 R_2)^T (R_1 R_2) = I_{n\times n}$ — property of transpose

$\Rightarrow R_2^T \underbrace{R_1^T R_1}_{I_{n\times n}} R_2 = I_{n\times n}$ — bc $R_1 \in SO(n)$, $R_1^T R_1 = I_{n\times n}$

$\Rightarrow R_2^T (I_{n\times n}) R_2 = I_{n\times n}$ — multiplying by I gives you og matrix

$\Rightarrow \underbrace{R_2^T R_2}_{I_{n\times n}} = I_{n\times n}$ — bc $R_2 \in SO(n)$ $R_2^T R_2 = I_{n\times n}$

$\Rightarrow I_{n\times n} = I_{n\times n}$ ✓

2) $\det(R_1 R_2) = 1$ — property of determinant

$\Rightarrow \det(R_1)\det(R_2) = 1$

$\Rightarrow 1 \times 1 = 1$ — $\det(R_1) = \det(R_2) = 1$ bc R_1 and $R_2 \in SO(n)$

$\Rightarrow 1 = 1$ ✓

↳ therefore, bc $(R_1 R_2)^T (R_1 R_2) = I_{n\times n}$ and $\det(R_1 R_2) = 1$ the product of R_1 and R_2 is also a rotational matrix.

$$\boxed{R(\theta_1) R(\theta_2) \in SO(n)}$$

Problem 2

$g(x_1, y_1, \theta_1)$ and $g(x_2, y_2, \theta_2) \in SE(2)$

product $g(x_1, y_1, \theta_1) g(x_2, y_2, \theta_2) \in SE(2)$

$SE(2) = \{(R, p) \mid R^T R = I$ and $\det(R) = 1$ and $p \in \mathbb{R}^2\}$

$g(x_1, y_1, \theta_1) \in SE(2)$

↳ g is the homogeneous representation of the rigid body transformation.

let $g(x_1, y_1, \theta_1) = g_1$ and $g(x_2, y_2, \theta_2) = g_2$

↳ these are the variables that the matrix g depends on.

→ the g_1 transformation has both a rotation $R_1 = R(\theta_1)$ and a translation $P_1 = P(x_1, y_1)$.

bc $g_1 \in SE(2)$

↳ $R_1 \in \mathbb{R}^2$; $\boxed{R_1^T R_1 = I_{2\times2}}$; $\boxed{\det(R_1) = 1}$

↳ $\boxed{P_1 \in \mathbb{R}^2}$

$$g_1 = \begin{bmatrix} R_1 & P_1 \\ 0 & 1 \end{bmatrix}$$

where R_1 is a 2×2 rotation matrix
where P_1 is a 2×1 translation vector
where 0 is a 1×2 zero row vector

→ the g_2 transformation has both a rotation $R_2 = R(\theta_2)$ and a translation $P_2 = P(x_2, y_2)$

bc $g_2 \in SE(2)$

↳ $R_2 \in \mathbb{R}^2$; $\boxed{R_2^T R_2 = I_{2\times2}}$; $\boxed{\det(R_2) = 1}$

↳ $\boxed{P_2 \in \mathbb{R}^2}$

$$g_2 = \begin{bmatrix} R_2 & P_2 \\ 0 & 1 \end{bmatrix}$$ where R_2 is a 2×2 rotation matrix
where P_2 is a 2×1 translation vector
where 0 is a 1×2 zero row vector

$$g(x_1, y_1, \theta_1) \times g(x_2, y_2, \theta_2) = g_1 \times g_2$$

$$g_1 \times g_2 = \begin{bmatrix} R_1 & P_1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} R_2 & P_2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_1R_2 & R_1P_2 + P_1 \\ (0+0) & 1 \end{bmatrix}$$

↳ let us note that neither g_1 nor g_2 is a 2×2 matrix, they are actually 3×3 matrices as defined by their elements described above. However, we can treat this multiplication as that of two 2×2 matrices.

let's analyze each element of the g_1g_2 matrix to see if the product satisfies the necessary conditions to be part of SE(2)

- R_1R_2 is a 2×2 matrix R_1 times a 2×2 matrix R_2. It defines the rotation R for the new product transformation g_1g_2, let us call this rotation R_{12}, where $R_{12} = R_1R_2$

  ↳ as proved above, bc $R_1^TR_1 = I_{n\times n}$ and $R_2^TR_2 = I_{n\times n}$, then $(R_1R_2)^T(R_1R_2) = I_{n\times n} \Rightarrow (R_{12})^T(R_{12}) = I_{n\times n}$. Therefore, the product g_1g_2 satisfies the first condition

  $$\boxed{(R_{12})^T(R_{12}) = I_{n\times n} \quad \text{for} \quad g_1g_2 \in SE(2)}$$

  ↳ as proved before, if $\det(R_1) = 1$ and $\det(R_2) = 1$ then $\det(R_1R_2) = 1 \Rightarrow \det(R_{12}) = 1$. Therefore, the product g_1g_2 satisfies the second condition

  $$\boxed{\det(R_{12}) = 1 \quad \text{for } g_1g_2 \in SE(2)}$$

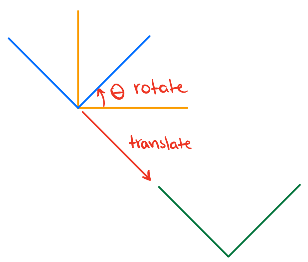
- the element $R_1P_2 + P_1$ defines the translation of g_1g_2 as it is located in the upper right corner of the g_1g_2 new homogeneous representation of the rigid body transformation. Let us call this translation P_{12}.

  ↳ $P_{12} = R_1P_2 + P_1$: $R_1P_2 + P_1$ is a 2×2 matrix R_1 times a 2×1 vector P_2, which gives a 2×1 vector. This is then added to a 2×1 vector P_1 giving a 2×1 vector. A 2×1 vector is in the space $\mathbb{R}^2$. Therefore the product $g_1 g_2$ satisfies the third and final condition

$$\boxed{P_{12} \in \mathbb{R}^2 \quad \text{for} \quad g_1 g_2 \in SE(2)}$$

$$\boxed{\begin{array}{l} \therefore,\ \text{if } g(x_1, y_1, \theta_1) \text{ and } g(x_2, y_2, \theta_2) \in SE(2) \\ \qquad \text{product } g(x_1, y_1, \theta_1)\, g(x_2, y_2, \theta_2) \in SE(2) \end{array}}$$

Problem 3



↳ lets say we have the homogeneous rigid body transformation g. It has a rotation R and a translation P.

↳ the rotation is by an arbitrary value of θ and the translation by arbitrary x and y coordinates.

→ g is a planar transformation, and as mentioned above, defines both the rotation and translation from the orange frame to the green frame. Let g_{PR} be the transformation for translating first and g_{RP} for rotating first.

↳ To rotate the orange frame to the specified angle of rotation of green frame, we can define R

$$\text{for } R \in \mathbb{R}^2 \ ; \ R \text{ is a } 2\times 2$$

↳ We can then define just the rotation as a homogeneous rigid body transformation g_R which is the necessary rotation to get transformation from orange to green frame.

2x2

$$g_R = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix}$$

2x1 vector

scalar

1x2 vector

↳ To translate the orange frame to the x, y position of the green frame, we can define a translation P.

$$\text{for } P \in \mathbb{R}^2 \ ; \ P \text{ is a } 2\times 1$$

↳ we can define just the translation as a homogeneous rigid body transformation g_P which is the necessary translation for the transformation from the orange frame to the green frame.

$$g_P = \begin{bmatrix} I_{2\times 2} & P \\ 0 & 1 \end{bmatrix}$$

P ← 2×1 vector

1 ← scalar

0 ← 1×2 vector

transformation with <u>translation first</u>, then rotation

$$g_{PR} = g_P\, g_R = \begin{bmatrix} I_{2\times 2} & P \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$$

$$\Rightarrow g_{PR} = \begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$$

transformation with <u>rotation first</u>, then translation

$$g_{RP} = g_R\, g_P = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I_{2\times 2} & P \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R & RP \\ 0 & 1 \end{bmatrix}$$

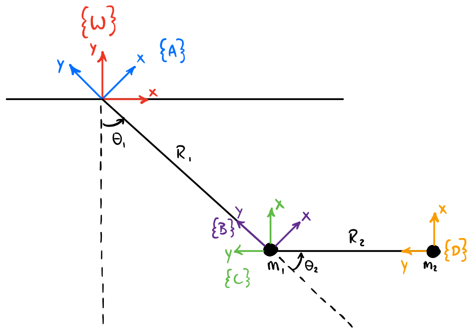
$$\Rightarrow g_{RP} = \begin{bmatrix} R & RP \\ 0 & 1 \end{bmatrix}$$

For the resulting transformation g for both doing translation first and doing rotation first, the rotation, which is the 2×2 matrix in the upper left corner, is R. This is correct as it is the necessary rotation for the transformation from the orange frame to the green frame. However, the translation varies for both g_{RP} (rotation first) and g_{PR} (translation first). For g_{RP} the translation is RP as seen in the upper right corner, this is correct because the original frame is rotated before it is translated. For g_{PR}, the translation is P, meaning the frame will translate but not along the x and y of the new rotated frame, it will move along the xy of the original frame.

↳ The steps above prove that a homogeneous transformation in SE(2) can indeed be separated into a rotation and translation. However, the rotation needs to come first and the translation second. As mentioned before if the translation comes first, the frame will not have the correct transformation.

↳ Since we are performing matrix multiplication if the order of the matrices changes, the homogeneous rigid body transformation will not be the same and the properties of an SE(2) will not be satisfied.

Problem 4



$W \rightarrow$ world frame

$A \rightarrow$ frame rotated θ_1

$B \rightarrow$ frame translated L_1

$C \rightarrow$ frame rotated θ_2

$D \rightarrow$ frame translated L_2

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$g_{WB} = g_{WA}\, g_{AB}$$

$$g_{WA} = \begin{bmatrix} R(\theta_1) & 0 \\ 0 & 1 \end{bmatrix} \; ; \; g_{AB} = \begin{bmatrix} I_{2\times 2} & \begin{bmatrix} 0 \\ -L_1 \end{bmatrix} \\ 0 & 1 \end{bmatrix}$$

Position of m_1 in world frame:

$\bar{r}_{W1} = g_{WB}\, \bar{r}_B$, where $\bar{r}_B$ is position m_1 in B: $\bar{r}_B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ $L_1 \cos$ $L_1 \sin$

$$g_{WD} = g_{WB}\, g_{BC}\, g_{CD}$$

$$g_{BC} = \begin{bmatrix} R(\theta_2) & 0 \\ 0 & 1 \end{bmatrix} \; ; \; g_{CD} = \begin{bmatrix} I_{2\times 2} & \begin{bmatrix} 0 \\ -L_2 \end{bmatrix} \\ 0 & 1 \end{bmatrix}$$

Position of m_2 in world frame:

$\bar{r}_{W2} = g_{WD}\, \bar{r}_D$, where $\bar{r}_D$ is position m_2 in D : $\bar{r}_D = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

$$h_1 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \bar{r}_{W1}$$

$$h_2 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \bar{r}_{W2}$$

$$v_1 = \frac{d}{dt}\left(\bar{r}_{W1}\right)$$

$$v_2 = \frac{d}{dt}\left(\bar{r}_{W2}\right)$$

explains the results.

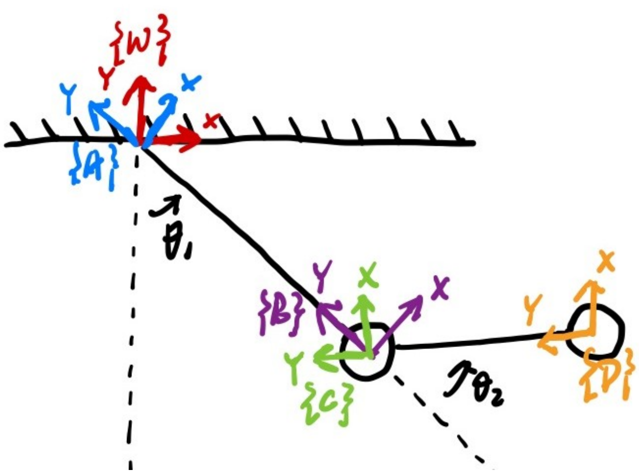
Problem 4 (20pts)

Simulate the same double-pendulum system in previous homework using only homogeneous transformation (and thus avoid using trigonometry). Simulate the system for $t \in [0, 3]$ with $dt = 0.01$. The parameters are $m_1 = m_2 = 1, R_1 = R_2 = 1, g = 9.8$ with initial conditions $\theta_1 = \theta_2 = -\frac{\pi}{3}, \dot{\theta}_1 = \dot{\theta}_2 = 0$. Do not use functions provided in the modern robotics package for manipulating transformation matrices such as RpToTrans(), etc.

> Hint 1: Same as in the lecture, you will need to define the frames by yourself in order to compute the Lagrangian. An example is shown below.

Turn in: Include a copy of your code used to simulate the system, and clearly labeled plot of θ_1 and θ_2 trajectory. Also, attach a figure showing how you defined the frames.

```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/master/doubpend_frames.jpg' width=500' hei
```




```
from os import get_blocking
from sympy import sin, cos

import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Markdown, display

# define time t
t = sym.symbols('t')

# define constants
m1, m2, R1, R2, g = sym.symbols('m1, m2, R1, R2, g')

# define system configuration variables
theta1 = sym.Function('theta_1')(t)
theta2 = sym.Function('theta_2')(t)

# first derivative of configuration
theta1_d = theta1.diff(t)
theta2_d = theta2.diff(t)

# second derivative of configuration
theta1_dd = theta1_d.diff(t)
theta2_dd = theta2_d.diff(t)

#### USING HOMOGENEOUS RIGID BODY TRANSFORMATIONS TO DEFINE THE LAGRANGIAN ####

# define function to return rotation matrix

```

```
def rotation(theta):
 return sym.Matrix([[cos(theta), -sin(theta)], [sin(theta), cos(theta)]])

# define homogenous rigid body transformations

# define the zeros row vector for the lower left corner
zeros_1x2 = sym.zeros(1, 2)

# Define the scalar 1 as a 1x1 matrix
scalar1 = sym.Matrix([1])


### DEFINING G_WB ###

## defining g_wa ##
# define the rotation matrix
R_wa = rotation(theta1)
# define the translation vector
P_wa = sym.zeros(2, 1) # no translation
# Define the 3x3 matrix using BlockMatrix
g_wa = sym.BlockMatrix([[R_wa, P_wa], [zeros_1x2, scalar1]])
# Convert the BlockMatrix to a regular Matrix
g_wa = g_wa.as_explicit()

## defining g_ab ##
# define the rotation matrix
R_ab = sym.eye(2) # no rotation, so just 2x2 identity
# define the translation vector
P_ab = sym.Matrix([0, -R1]) # translation in the negative y-axis
# Define the 3x3 matrix using BlockMatrix
g_ab = sym.BlockMatrix([[R_ab, P_ab], [zeros_1x2, scalar1]])
# Convert the BlockMatrix to a regular Matrix
g_ab = g_ab.as_explicit()

## defining g_wb as the product of g_wa and g_ab
g_wb = g_wa*g_ab   # transformation for frame where mass 1 lies and world frame

# ------------------------------ #

### DEFINING G_WD ###

## defining g_bc ##
# define the rotation matrix
R_bc = rotation(theta2)
# define the translation vector
P_bc = sym.zeros(2, 1) # no translation
# Define the 3x3 matrix using BlockMatrix
g_bc = sym.BlockMatrix([[R_bc, P_bc], [zeros_1x2, scalar1]])
# Convert the BlockMatrix to a regular Matrix
g_bc = g_bc.as_explicit()

## defining g_cd ##
# define the rotation matrix
R_cd = sym.eye(2) # no rotation, so just 2x2 identity
# define the translation vector
P_cd = sym.Matrix([0, -R2]) # translation along the negative y-axis
# Define the 3x3 matrix using BlockMatrix
g_cd = sym.BlockMatrix([[R_cd, P_cd], [zeros_1x2, scalar1]])
# Convert the BlockMatrix to a regular Matrix
g_cd = g_cd.as_explicit()

## defining g_wd as the product of g_wb, g_bc, g_cd
g_wd = g_wb*g_bc*g_cd   # transformation for frame where mass 2 lies and world frame

# ------------------------------- #

# position of m1 and m2 in the world frame
# define rb_bar as the position of mass 1 in frame b (at the origin)
rb_bar = sym.Matrix([0, 0, 1])
# position of mass 1 in world frame
rw1_bar = g_wb*rb_bar

# define rd_bar as the position of mass 2 in frame d (at the origin)
rd_bar = sym.Matrix([0, 0, 1])
# position of mass 2 in world frame
```

```
rw2_bar = g_wd*rd_bar

# define heights of mass 1 and 2
h1 = (sym.Matrix([[0, 1, 0]])*rw1_bar)[0] # the bracket zero is to index the height from a 1x1 matrix
h2 = (sym.Matrix([[0, 1, 0]])*rw2_bar)[0] # the bracket zero is to index the height from a 1x1 matrix

# defining velocities of mass 1 and mass 2 in world frame
v1 = (rw1_bar).diff(t)
v2 = (rw2_bar).diff(t)

# defining Potential Energy
PE = m1*g*h1 + m2*g*h2

# defining Kinetic Energy
v1_square = v1.dot(v1)
v2_square = v2.dot(v2)

KE = 0.5*m1*v1_square + 0.5*m2*v2_square

# Lagrangian KE - PE
L = KE - PE
print("Lagrangian:")
display( sym.Eq( sym.symbols('L'), L.simplify().expand() ) )
print('')
```

```
Lagrangian:
```

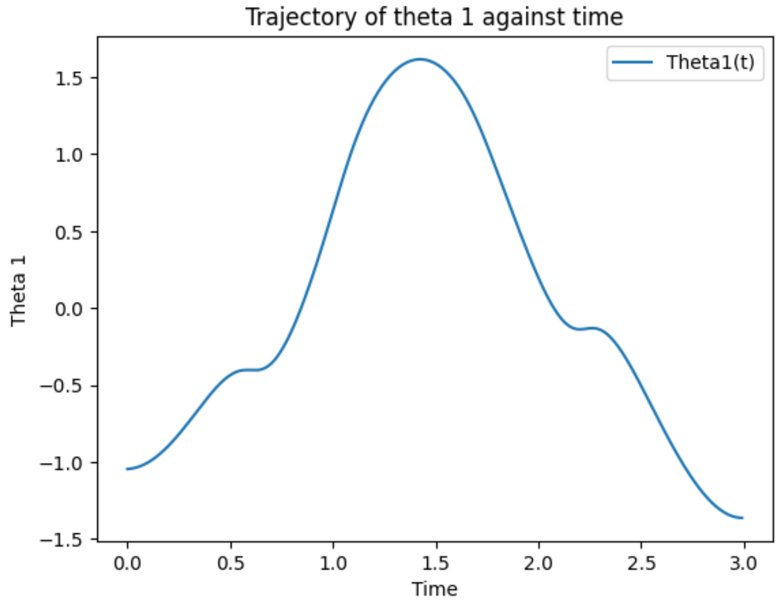
$$L = 0.5R_1^2m_1\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5R_1^2m_2\left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0R_1R_2m_2\cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0R_1R_2m_2\cos\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + R_1gm_1\cos\left(\theta_1(t)\right)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

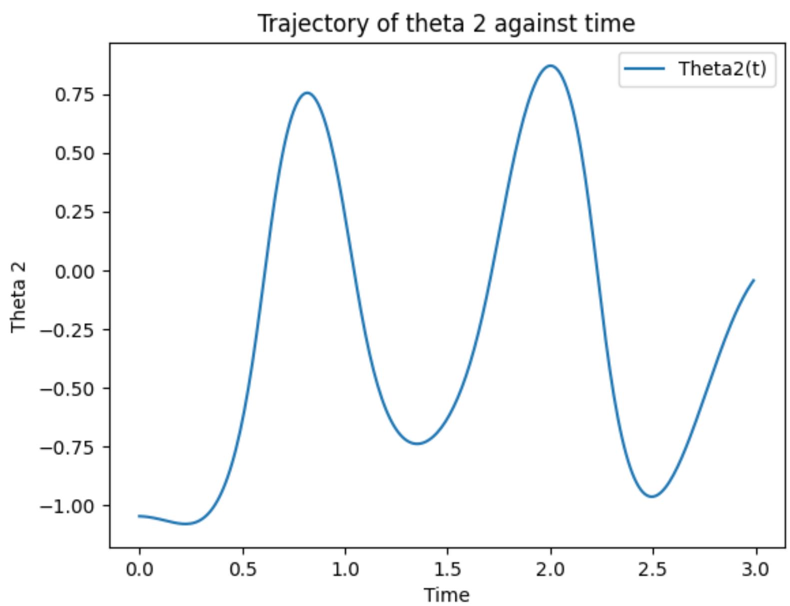
```
#### Solving the E-L Equations ####

# define left hand side of matrix
lhs = sym.Matrix([(L.diff(theta1) - ( L.diff(theta1_d) ).diff(t)).expand().simplify(), ((L.diff(theta2) - ( L.diff(theta2_d)


# define right hand side of matrix
rhs = sym.Matrix([0, 0])

# equation of matrices
eqn = sym.Eq(lhs, rhs)

# define matrix of variables we are solving for
q = sym.Matrix([theta1_dd, theta2_dd])

# solving both equations for q
soln = sym.solve(eqn, q) # this will display a python dictionary

# solved EL
solved1 = soln[theta1_dd].simplify()

solved2 = soln[theta2_dd].simplify()


# Substituing / Evaluating constants
Eq1_subs = solved1.subs({m1:1, m2:1, R1:1, R2:1, g:9.8})
Eq2_subs = solved2.subs({m1:1, m2:1, R1:1, R2:1, g:9.8})


### LAMBDIFY ###
# Lambdify in terms of theta1, theta2, theta1_d, theta2_d (angles and angular velocities) which are the inputs of the equati
l1 = sym.lambdify([theta1, theta2, theta1_d, theta2_d], Eq1_subs)
l2 = sym.lambdify([theta1, theta2, theta1_d, theta2_d], Eq2_subs)



### PLOTTING TRAJECTORY ###
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), theta(t), xdot(t), thetadot(t) from t=0 to tf
    """
    N = int((max(tspan)-0)/dt)
    x = np.copy(x0)
    tvec = np.linspace(0,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

#############################################
def theta1_ddot(theta1, theta2, theta1_d, theta2_d):
  return l1(theta1, theta2, theta1_d, theta2_d)

def theta2_ddot(theta1, theta2, theta1_d, theta2_d):
  return l2(theta1, theta2, theta1_d, theta2_d)


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

    Parameters
    ============
```

```
    s: NumPy array
        s = [theta1, theta2, theta1_dot, theta2_dot] 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 = [theta1_dot, theta2_dot, theta1_ddot, theta2_ddot]
    """
    return np.array([s[2], s[3], theta1_ddot(s[0], s[1], s[2], s[3]), theta2_ddot(s[0], s[1], s[2], s[3])])

# define initial state (theta1 = theta2 = -pi/2 , theta1_d = theta2_d = 0)
s0 = np.array([-(np.pi)/3, -(np.pi)/3, 0, 0])
# simulat from t=0 to 3, since dt=0.01, the returned trajectory
# will have 3/0.01=300 time steps, each time step contains extended
# system state vector [x(t), xdot(t)]
traj = simulate(dyn, s0, [0, 3], 0.01, integrate)
#print('\033[1mShape of traj: \033[0m', traj.shape)

# determening trajectories of configuration variables and their derivatives

theta1_traj = traj[0]
theta2_traj = traj[1]
theta1_d_traj = traj[2]
theta2_d_traj = traj[3]
```

```

# initial state is already defined above and is part of theta 1 and theta 2 trajectories

# time step array
# 3/0.01 will give 300 time steps
timespan = np.arange(0, 3, 0.01)


plt.plot(timespan, theta1_traj, label='Theta1(t)')
plt.title("Trajectory of theta 1 against time")
plt.ylabel('Theta 1')
plt.xlabel('Time')
plt.legend()
plt.show()

plt.plot(timespan, theta2_traj, label='Theta2(t)')
plt.title('Trajectory of theta 2 against time')
plt.ylabel('Theta 2')
plt.legend()
plt.xlabel('Time')
plt.show()
```





Problem 5 (20pts)

Modify the previous animation function for the double-pendulum such that the animation shows the frames you defined in the last problem (it's similar to the `tf` in RViz, if you're familiar with ROS). All the *x axes* should be displayed in green and all the *y axes* should be displayed in red, with axis's length of 0.3 for all. An animation example can be found at https://youtu.be/2H3KvRWQqys. Do not use functions provided in the modern robotics package for manipulating transformation matrices such as RpToTrans(), etc.

> Hint 1: Each axis can be considered as a line connecting the origin and the point $[0.3, 0]$ or $[0, 0.3]$ in that frame. You will need to use the homogeneous transformations to transfer these two axis/points back into the world/fixed frame. Example code showing how to display one frame is provided below.

Turn in: Include a copy of your code used for animation and a video of the animation. The video can be uploaded separately through Canvas, and it should be in ".mp4" format. You can either use screen capture or record the screen directly with your phone.

```
def animate_double_pend(theta_array,L1=1,L2=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ================================================
```

```
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first pendulum
    L2:
        length of the second pendulum
    T:
        length/seconds of animation duration

    Returns: None
    """

    ################################
    # Imports required for animation.
    from plotly.offline import init_notebook_mode, iplot
    from IPython.display import display, HTML
    import plotly.graph_objects as go

    #######################
    # Browser configuration.
    def configure_plotly_browser_state():
        import IPython
        display(IPython.core.display.HTML('''
            <script src="/static/components/requirejs/require.js"></script>
            <script>
              requirejs.config({
                paths: {
                  base: '/static/base',
                  plotly: 'https://cdn.plot.ly/plotly-1.5.1.min.js?noext',
                },
              });
            </script>
            '''))
    configure_plotly_browser_state()
    init_notebook_mode(connected=False)

    ###############################################
    # Getting data from pendulum angle trajectories.
    xx1=L1*np.sin(theta_array[0])
    yy1=-L1*np.cos(theta_array[0])
    xx2=xx1+L2*np.sin(theta_array[0]+theta_array[1])
    yy2=yy1-L2*np.cos(theta_array[0]+theta_array[1])
    N = len(theta_array[0]) # Need this for specifying length of simulation

    ###############################################
    # Define arrays containing data for frame axes
    # In each frame, the x and y axis are always fixed
    x_axis = np.array([0.3, 0.0])
    y_axis = np.array([0.0, 0.3])
    # Use homogeneous tranformation to transfer these two axes/points
    # back to the fixed frame
    frame_a_x_axis = np.zeros((2,N))
    frame_a_y_axis = np.zeros((2,N))
    frame_b_x_axis = np.zeros((2,N))
    frame_b_y_axis = np.zeros((2,N))
    frame_c_x_axis = np.zeros((2,N))
    frame_c_y_axis = np.zeros((2,N))
    frame_d_x_axis = np.zeros((2,N))
    frame_d_y_axis = np.zeros((2,N))
    for i in range(N): # iteration through each time step
        # evaluate homogeneous transformation
        t_wa = np.array([[np.cos(theta_array[0][i]), -np.sin(theta_array[0][i]), 0],
                         [np.sin(theta_array[0][i]),  np.cos(theta_array[0][i]), 0],
                         [                        0,                          0, 1]])
        # transfer the x and y axes in body frame back to fixed frame at
        # the current time step
        frame_a_x_axis[:,i] = t_wa.dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_a_y_axis[:,i] = t_wa.dot([y_axis[0], y_axis[1], 1])[0:2]

        # evaluate homogeneous transformation
        t_ab = np.array([[1, 0, 0], [0, 1, -1], [0, 0, 1]])
        # transfer the x and y axes in body frame back to fixed frame at
        # the current time step
        frame_b_x_axis[:,i] = t_wa.dot(t_ab).dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_b_y_axis[:,i] = t_wa.dot(t_ab).dot([y_axis[0], y_axis[1], 1])[0:2]
```

```
        frame_b_y_axis[:,i] = t_wa.dot(t_ab).dot([y_axis[0], y_axis[1], 1])[0:2]


         # evaluate homogeneous transformation
         # change rotation to use theta2. The rotation from frame b to c depends on theta2
        t_bc = np.array([[np.cos(theta_array[1][i]), -np.sin(theta_array[1][i]), 0],
                         [np.sin(theta_array[1][i]),  np.cos(theta_array[1][i]), 0],
                         [                          0,                          0, 1]])
        # transfer the x and y axes in body frame back to fixed frame at
        # the current time step
        frame_c_x_axis[:,i] = t_wa.dot(t_ab).dot(t_bc).dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_c_y_axis[:,i] = t_wa.dot(t_ab).dot(t_bc).dot([y_axis[0], y_axis[1], 1])[0:2]

        # evaluate homogeneous transformation
        t_cd = np.array([[1, 0, 0], [0, 1, -1], [0, 0, 1]])
        # transfer the x and y axes in body frame back to fixed frame at
        # the current time step
        frame_d_x_axis[:,i] = t_wa.dot(t_ab).dot(t_bc).dot(t_cd).dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_d_y_axis[:,i] = t_wa.dot(t_ab).dot(t_bc).dot(t_cd).dot([y_axis[0], y_axis[1], 1])[0:2]

    ####################################
    # Using these to specify axis limits.
    xm = -3 #np.min(xx1)-0.5
    xM = 3 #np.max(xx1)+0.5
    ym = -3 #np.min(yy1)-2.5
    yM = 3 #np.max(yy1)+1.5

    ###########################
    # Defining data dictionary.
    # Trajectories are here.
    data=[
        # note that except for the trajectory (which you don't need this time),
        # you don't need to define entries other than "name". The items defined
        # in this list will be related to the items defined in the "frames" list
        # later in the same order. Therefore, these entries can be considered as
        # labels for the components in each animation frame
        dict(name='Arm'),
        dict(name='Mass 1'),
        dict(name='Mass 2'),
        dict(name='World Frame X'),
        dict(name='World Frame Y'),
        dict(name='A Frame X Axis'),
        dict(name='A Frame Y Axis'),
        dict(name='B Frame X Axis'),
        dict(name='B Frame Y Axis'),
        dict(name='C Frame X Axis'),
        dict(name='C Frame Y Axis'),
        dict(name='D Frame X Axis'),
        dict(name='D Frame Y Axis'),

        # You don't need to show trajectory this time,
        # but if you want to show the whole trajectory in the animation (like what
        # you did in previous homeworks), you will need to define entries other than
        # "name", such as "x", "y". and "mode".

        # dict(x=xx1, y=yy1,
        #      mode='markers', name='Pendulum 1 Traj',
        #      marker=dict(color="fuchsia", size=2)
        #     ),
        # dict(x=xx2, y=yy2,
        #      mode='markers', name='Pendulum 2 Traj',
        #      marker=dict(color="purple", size=2)
        #     ),
        ]

    ################################
    # Preparing simulation layout.
    # Title and axis ranges are here.
    layout=dict(autosize=False, width=1000, height=1000,
                xaxis=dict(range=[xm, xM], autorange=False, zeroline=False,dtick=1),
                yaxis=dict(range=[ym, yM], autorange=False, zeroline=False,scaleanchor = "x",dtick=1),
                title='Double Pendulum Simulation',
                hovermode='closest',
                updatemenus= [{'type': 'buttons',
                               'buttons': [{'label': 'Play','method': 'animate',
                                            'args': [None, {'frame': {'duration': T, 'redraw': False}}]},
                                           {'args': [[None], {'frame': {'duration': T, 'redraw': False}, 'mode': 'immediate'
```

```
                                        {'args': [[None], {'frame': {'duration': 1, '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=[# first three objects correspond to the arms and two masses,
                       # same order as in the "data" variable defined above (thus
                       # they will be labeled in the same order)
                       dict(x=[0,xx1[k],xx2[k]],
                            y=[0,yy1[k],yy2[k]],
                            mode='lines',
                            line=dict(color='orange', width=3),
                            ),
                       go.Scatter(
                            x=[xx1[k]],
                            y=[yy1[k]],
                            mode="markers",
                            marker=dict(color="blue", size=12)),
                       go.Scatter(
                            x=[xx2[k]],
                            y=[yy2[k]],
                            mode="markers",
                            marker=dict(color="blue", size=12)),
                       # display x and y axes of the fixed frame in each animation frame
                       dict(x=[0,x_axis[0]],
                            y=[0,x_axis[1]],
                            mode='lines',
                            line=dict(color='green', width=3),
                            ),
                       dict(x=[0,y_axis[0]],
                            y=[0,y_axis[1]],
                            mode='lines',
                            line=dict(color='red', width=3),
                            ),
                       # display x and y axes of the {A} frame in each animation frame
                       dict(x=[0, frame_a_x_axis[0][k]],
                            y=[0, frame_a_x_axis[1][k]],
                            mode='lines',
                            line=dict(color='green', width=3),
                            ),
                       dict(x=[0, frame_a_y_axis[0][k]],
                            y=[0, frame_a_y_axis[1][k]],
                            mode='lines',
                            line=dict(color='red', width=3),
                            ),
                       # display x and y axes of the {B} frame in each animation frame
                       dict(x=[xx1[k], frame_b_x_axis[0][k]],
                            y=[yy1[k], frame_b_x_axis[1][k]],
                            mode='lines',
                            line=dict(color='green', width=3),
                            ),
                       dict(x=[xx1[k], frame_b_y_axis[0][k]],
                            y=[yy1[k], frame_b_y_axis[1][k]],
                            mode='lines',
                            line=dict(color='red', width=3),
                            ),
                       # display x and y axes of the {C} frame in each animation frame
                       dict(x=[xx1[k], frame_c_x_axis[0][k]],
                            y=[yy1[k], frame_c_x_axis[1][k]],
                            mode='lines',
                            line=dict(color='green', width=3),
                            ),
                       dict(x=[xx1[k], frame_c_y_axis[0][k]],
                            y=[yy1[k], frame_c_y_axis[1][k]],
                            mode='lines',
                            line=dict(color='red', width=3),
                            ),
                       # display x and y axes of the {D} frame in each animation frame
                       dict(x=[xx2[k], frame_d_x_axis[0][k]],
                            y=[yy2[k], frame_d_x_axis[1][k]],
                            mode='lines',
```

```
                            line=dict(color='green', width=3),
                            ),
                       dict(x=[xx2[k], frame_d_y_axis[0][k]],
                            y=[yy2[k], frame_d_y_axis[1][k]],
                            mode='lines',
                            line=dict(color='red', width=3),
                            )
                      ]) for k in range(N)]

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

animate_double_pend(traj, 1, 1, 10)
```

Double Pendulum Simulation