# ME314 Homework 6

Sean Morton
Collaborators: Noah Yi

## 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.

In [1]:
```
#imports
import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
```

In [2]:
```
#helper functions
def compute_EL(lagrangian, q):
    '''
    Helper function for computing the Euler-Lagrange equations for a given system,
    so I don't have to keep writing it out over and over again.

    Inputs:
    - lagrangian: our Lagrangian function in symbolic (Sympy) form
    - q: our state vector [x1, x2, ...], in symbolic (Sympy) form

    Outputs:
    - eqn: the Euler-Lagrange equations in Sympy form
    '''

    # wrap system states into one vector (in SymPy would be Matrix)
    #q = sym.Matrix([x1, x2])
    qd = q.diff(t)
    qdd = qd.diff(t)

    # compute derivative wrt a vector, method 1
    # wrap the expression into a SymPy Matrix
    L_mat = sym.Matrix([lagrangian])
    dL_dq = L_mat.jacobian(q)
    dL_dqdot = L_mat.jacobian(qd)

    #set up the Euler-Lagrange equations
    LHS = dL_dq - dL_dqdot.diff(t)
    RHS = sym.zeros(1, len(q))
    eqn = sym.Eq(LHS.T, RHS.T)

    return eqn

def solve_EL(eqn, var):
    '''
    Helper function to solve and display the solution for the Euler-Lagrange
    equations.

    Inputs:
    - eqn: Euler-Lagrange equation (type: Sympy Equation())
    - var: state vector (type: Sympy Matrix). typically a form of q-doubledot
        but may have different terms

    Outputs:
    - Prints symbolic solutions
    - Returns symbolic solutions in a dictionary
    '''

    soln = sym.solve(eqn, var, dict = True)
    eqns_solved = []


    for i, sol in enumerate(soln):
```

```python
        for x in list(sol.keys()):
            eqn_solved = sym.Eq(x, sol[x])
            eqns_solved.append(eqn_solved)

    return eqns_solved

def solve_constrained_EL(lamb, phi, q, lhs):
    """Now uses just the LHS of the constrained E-L equations,
    rather than the full equation form"""

    qd = q.diff(t)
    qdd = qd.diff(t)

    phidd = phi.diff(t).diff(t)
    lamb_grad = sym.Matrix([lamb * phi.diff(a) for a in q])
    q_mod = qdd.row_insert(2, sym.Matrix([lamb]))

    #format equations so they're all in one matrix
    expr_matrix = lhs - lamb_grad
    phidd_matrix = sym.Matrix([phidd])
    expr_matrix = expr_matrix.row_insert(2,phidd_matrix)

    print("Equations to be solved (LHS - lambda * grad(phi) = 0):")
    RHS = sym.zeros(len(expr_matrix), 1)
    disp_eq = sym.Eq(expr_matrix, RHS)
    display(disp_eq)
    print("Variables to solve for:")
    display(q_mod)

    #solve E-L equations
    eqns_solved = solve_EL(expr_matrix, q_mod)
    return eqns_solved

def rk4(dxdt, x, t, dt):
    '''
    Applies the Runge-Kutta method, 4th order, to a sample function,
    for a given state q0, for a given step size. Currently only
    configured for a 2-variable dependent system (x,y).
    ==========
    dxdt: a Sympy function that specifies the derivative of the system of interest
    t: the current timestep of the simulation
    x: current value of the state vector
    dt: the amount to increment by for Runge-Kutta
    ======
    returns:
    x_new: value of the state vector at the next timestep
    '''
    k1 = dt * dxdt(t, x)
    k2 = dt * dxdt(t + dt/2.0, x + k1/2.0)
    k3 = dt * dxdt(t + dt/2.0, x + k2/2.0)
    k4 = dt * dxdt(t + dt, x + k3)
    x_new = x + (k1 + 2.0*k2 + 2.0*k3 + k4)/6.0

    return x_new


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))
```

```python
    for i in range(N):
        t = tvec[i]
        xtraj[:,i]=integrate(f,x,t,dt)
        x = np.copy(xtraj[:,i])
    return xtraj
```

In [50]:

```python
#new for this hw
def SO2AndR2ToSE2(R, p):
    #this was something I defined from scratch - I did not look at or consult the MR library
    G = sym.zeros(3)
    G[:2,:2] = R
    G[:2, 2] = sym.Matrix(p)
    G[ 2, 2] = 1
    return G

def SO2AndR2ToSE2_np(R, p):
    #this was something I defined from scratch - I did not look at or consult the MR library
    G = np.zeros([3,3])
    G[:2,:2] = R
    G[:2, 2] = np.array(p).T
    G[ 2, 2] = 1
    return G


#testing
R = sym.Matrix([
    [1,2],
    [3,4]
])
p = sym.Matrix([5,6])
G = SO2AndR2ToSE2(R, p)

R1 = np.matrix([
    [4, 3],
    [2, 1]
])
p1 = [9,7]
G1 = SO2AndR2ToSE2_np(R1, p1)

print("Sympy version:")
print(G)
print("\nNumpy version:")
print(G1)
```

```
Sympy version:
Matrix([[1, 2, 5], [3, 4, 6], [0, 0, 1]])

Numpy version:
[[4. 3. 9.]
 [2. 1. 7.]
 [0. 0. 1.]]
```

## Problem 1 (20pts)

Show that if $R(\theta_1)$ and $R(\theta_2) \in SO(n)$ then the product is also a rotation matrix, that is $R(\theta_1)R(\theta_2) \in SO(n)$.

> Hint 1: You know this is true when $n = 2$ by direct calculation in class, but for $n \neq 2$ you should use the definition of $SO(n)$ to verify it for arbitrary $n$. Do not try to do this by analyzing individual components of the matrix.

**Turn in: A scanned (or photograph from your phone or webcam) copy of your handwritten solution. You can also use *LaTeX*. If you use SymPy, you need to include a copy of your code and the code outputs. Make sure to note why your handwritten solution / code output explains the results.**

In [ ]:

## Problem 2 (20pts)

Show that if $g(x_1, y_1, \theta_1)$ and $g(x_2, y_2, \theta_2) \in SE(2)$ then the product satisfies $g(x_1, y_1, \theta_1)g(x_2, y_2, \theta_2) \in SE(2)$.

**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, you need to include a copy of your code and the code outputs. Make sure to note why your handwritten soultion / code output explains the results.**

In [ ]:

## Problem 3 (20pts)

Show that any homogeneous transformation in SE(2) can be separated into a rotation and a translation. What's the order of the two operations, which comes first? What's different if we flip the order in which we compose the rotation and translation?

> Hint 1: For the rotation and translation operation, we first need to know what's the reference frame for these two operations.

**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, you need to include a copy of your code and the code outputs. Make sure to note why your handwritten soultion / code output explains the results.**

In [ ]:

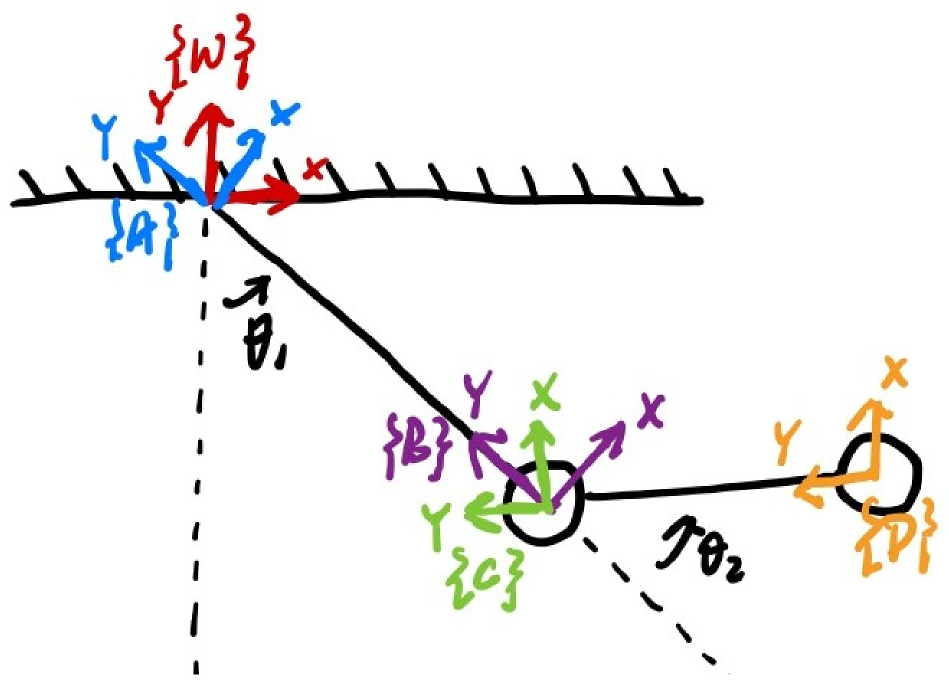
## Problem 4 (20pts)

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 $\theta_1$ and $\theta_2$ trajectory. Also, attach a figure showing how you defined the frames.**

In [4]:
```
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' height='350'></table>"))
```




In [28]:
```
#define variables
m1, m2, L1, L2, g = sym.symbols(r'm_1, m_2, L_1, L_2, g')
t = sym.symbols(r't')

theta1 = sym.Function(r'\theta_1')(t)
theta2 = sym.Function(r'\theta_2')(t)
theta1d = theta1.diff(t)
theta2d = theta2.diff(t)


#define transformation matrices
I = sym.eye(2)
Rwa = sym.Matrix([
    [sym.cos(theta1), -sym.sin(theta1)],
    [sym.sin(theta1),  sym.cos(theta1)]

])

Rbc = sym.Matrix([
    [sym.cos(theta2), -sym.sin(theta2)],
    [sym.sin(theta2),  sym.cos(theta2)]

])

Gwa = SO2AndR2ToSE2(Rwa, [0,   0])
Gab = SO2AndR2ToSE2(I,   [0, -L1])
Gbc = SO2AndR2ToSE2(Rbc, [0,   0])
Gcd = SO2AndR2ToSE2(I,   [0, -L2])


#define kinetic and potential energy
v1 = ( Gwa * Gab * sym.Matrix([0,0,1])  ).diff(t)
v2 = ( Gwa * Gab * Gbc * Gcd * sym.Matrix([0,0,1])  ).diff(t)
y1 = ( Gwa * Gab * sym.Matrix([0,0,1]) ).tolist()[1][0]
y2 = ( Gwa * Gab * Gbc * Gcd * sym.Matrix([0,0,1]) ).tolist()[1][0]

y1 = y1.simplify()
y2 = y2.simplify()

v1 = sym.simplify(v1)
v2 = sym.simplify(v2)


KE = 0.5 * m1 * (v1.T * v1)[0] + 0.5 * m2 * (v2.T * v2)[0]
U = m1 * g * y1 + m2 * g * y2
lagrangian = KE - U
lagrangian = lagrangian.simplify()

# print("Y1, Y2:")
# display(y1)
# display(y2)
```

```
# print("V1, V2:")
# display(v1)
# display(v2)

print("Lagrangian:")
display(lagrangian)
```

```
Lagrangian:
```

$$0.5L_1^2 m_1 \left(\frac{d}{dt}\theta_1(t)\right)^2 + L_1 g m_1 \cos\left(\theta_1(t)\right) + g m_2 \left(L_1 \cos\left(\theta_1(t)\right) + L_2 \cos\left(\theta_1(t) + \theta_2(t)\right)\right) + 0.5 m_2 \left(L_1^2 \left(\frac{d}{dt}\theta_1(t)\right)^2 + 2L_1 L_2 \cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2L_1 L_2 \cos\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + L_2^2\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2L_2^2 \frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + L_2^2\left(\frac{d}{dt}\theta_2(t)\right)^2\right)$$

In [26]:

```
#compute + solve Euler-Lagrange equations

q = sym.Matrix([theta1, theta2])
qd = q.diff(t)
qdd = qd.diff(t)

eqn = compute_EL(lagrangian, q)
eqn = eqn.simplify()

print("Euler-Lagrange equations:")
display(eqn)

solved_eqns = solve_EL(eqn, qdd)
simplified_eqns = []

print("Solved:")
for eq in solved_eqns:
    eq_new = eq.simplify()
    simplified_eqns.append(eq_new)
    display(eq_new)
```

```
Euler-Lagrange equations:
```

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1.0L_1^2 m_1 \frac{d^2}{dt^2}\theta_1(t) - L_1 g m_1 \sin\left(\theta_1(t)\right) - g m_2 \left(L_1 \sin\left(\theta_1(t)\right) + L_2 \sin\left(\theta_1(t) + \theta_2(t)\right)\right) \\ -1.0 m_2 \left(L_1^2 \frac{d^2}{dt^2}\theta_1(t) - 2L_1 L_2 \sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - L_1 L_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 2L_1 L_2 \cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_1(t) + L_1 L_2 \cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_2(t) + L_2^2 \frac{d^2}{dt^2}\theta_1(t) + L_2^2 \frac{d^2}{dt^2}\theta_2(t)\right) \\ -1.0 L_2 m_2 \left(L_1 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + L_1 \cos\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_1(t) + L_2 \frac{d^2}{dt^2}\theta_1(t) + L_2 \frac{d^2}{dt^2}\theta_2(t) + g \sin\left(\theta_1(t) + \theta_2(t)\right)\right) \end{bmatrix}$$

```
Solved:
```

$$\frac{d^2}{dt^2}\theta_1(t) = \frac{0.5L_1 m_2 \sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0L_2 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2.0L_2 m_2 \sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + 1.0L_2 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 - 1.0 g m_1 \sin\left(\theta_1(t)\right) + 0.5 g m_2 \sin\left(\theta_1(t) + 2\theta_2(t)\right) - 0.5 g m_2 \sin\left(\theta_1(t)\right)}{L_1\left(m_1 + m_2 \sin^2\left(\theta_2(t)\right)\right)}$$

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{2\left(-1.0L_1^2 m_1 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0L_1^2 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0L_1 L_2 m_2 \sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0L_1 L_2 m_2 \sin\left(2\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 0.5L_1 L_2 m_2 \sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.5L_1 g m_1 \sin\left(\theta_1(t) - \theta_2(t)\right) - 0.5L_1 g m_1 \sin\left(\theta_1(t) + \theta_2(t)\right) + 0.5L_1 g m_2 \sin\left(\theta_1(t) - \theta_2(t)\right) - 0.5L_1 g m_2 \sin\left(\theta_1(t) + \theta_2(t)\right) - 1.0L_2^2 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 2.0L_2^2 m_2 \sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 1.0L_2^2 m_2 \sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 1.0L_2 g m_1 \sin\left(\theta_1(t)\right) - 0.5L_2 g m_2 \sin\left(\theta_1(t) + 2\theta_2(t)\right) + 0.5L_2 g m_2 \sin\left(\theta_1(t)\right)\right)}{L_1 L_2 \cdot \left(2m_1 - m_2 \cos\left(2\theta_2(t)\right) + m_2\right)}$$

In [29]:

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

consts_dict = {
    m1: 1,
    m2: 1,
    L1: 1,
    L2: 1,
    g:  9.8
}

theta1dd_sy = simplified_eqns[0]
theta2dd_sy = simplified_eqns[1]

theta1dd_sy = theta1dd_sy.subs(consts_dict)
theta2dd_sy = theta2dd_sy.subs(consts_dict)

print("Theta1dd and Theta1dd with constants substituted in:")
display(theta1dd_sy)
display(theta2dd_sy)

q_ext = sym.Matrix([theta1, theta2, theta1d, theta2d])
theta1dd_np = sym.lambdify(q_ext, theta1dd_sy.rhs)
theta2dd_np = sym.lambdify(q_ext, theta2dd_sy.rhs)
```

```
Theta1dd and Theta1dd with constants substituted in:
```

$$\frac{d^2}{dt^2}\theta_1(t) = \frac{4.9\sin\left(\theta_1(t) + 2\theta_2(t)\right) - 14.7\sin\left(\theta_1(t)\right) + 1.0\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 2.0\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) + 1.0\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.5\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{\sin^2\left(\theta_2(t)\right) + 1}$$

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{2\cdot\left(9.8\sin\left(\theta_1(t)-\theta_2(t)\right) - 9.8\sin\left(\theta_1(t)+\theta_2(t)\right) - 4.9\sin\left(\theta_1(t)+2\theta_2(t)\right) + 14.7\sin\left(\theta_1(t)\right) - 3.0\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 2.0\sin\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 1.0\sin\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 - 1.0\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 1.0\sin\left(2\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t) - 0.5\sin\left(2\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2\right)}{3-\cos\left(2\theta_2(t)\right)}$$

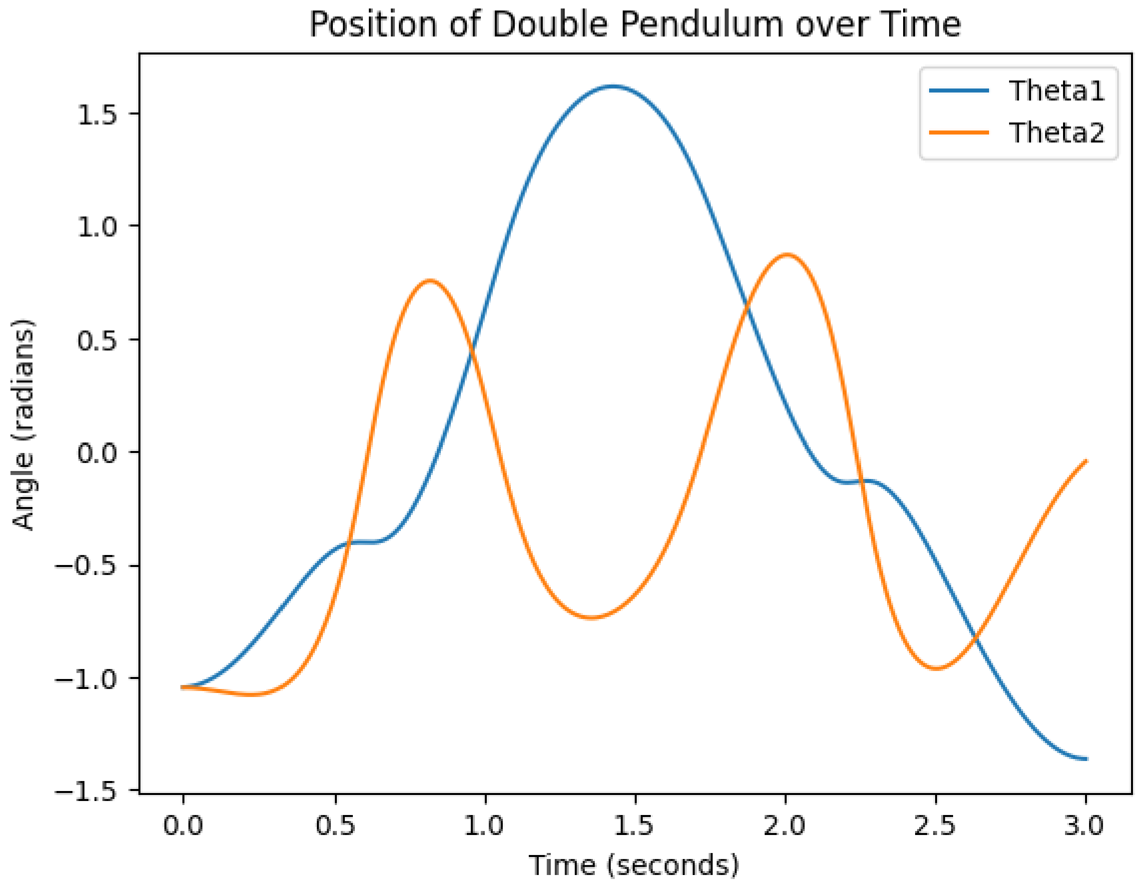
In [31]:
```
#simulate system using initial conditions
th0 = float(-sym.pi/3.0)
s0 = [th0, th0, 0, 0]
t_span = [0, 3]
dt = 0.01

print("Values of theta1dd and theta2dd at t0 with ICs:")
print(str(theta1dd_np(*s0)) + "s^-2")
print(str(round(theta2dd_np(*s0), 2)) + "s^-2")
```

```
Values of theta1dd and theta2dd at t0 with ICs:
7.274613391789283s^-2
-2.42s^-2
```

In [72]:
```
# You can start your implementation here :)
def dxdt(t, s):
    return np.array([s[2], s[3], theta1dd_np(*s), theta2dd_np(*s)])

#use rk4 for numerical integration
q_array = simulate(dxdt, s0, t_span, dt, rk4)
print('shape of trajectory: ', q_array.shape)

#plot
t_array = np.linspace(t_span[0], t_span[1], len(q_array[1]))
theta1_array = q_array[0]
theta2_array = q_array[1]

plt.plot(t_array, theta1_array, label="Theta1")
plt.plot(t_array, theta2_array, label="Theta2")

plt.xlabel("Time (seconds)")
plt.ylabel("Angle (radians)")
plt.title("Position of Double Pendulum over Time")
plt.legend()
```

```
shape of trajectory:  (4, 300)
```

Out[72]: `<matplotlib.legend.Legend at 0x12c54faae90>`



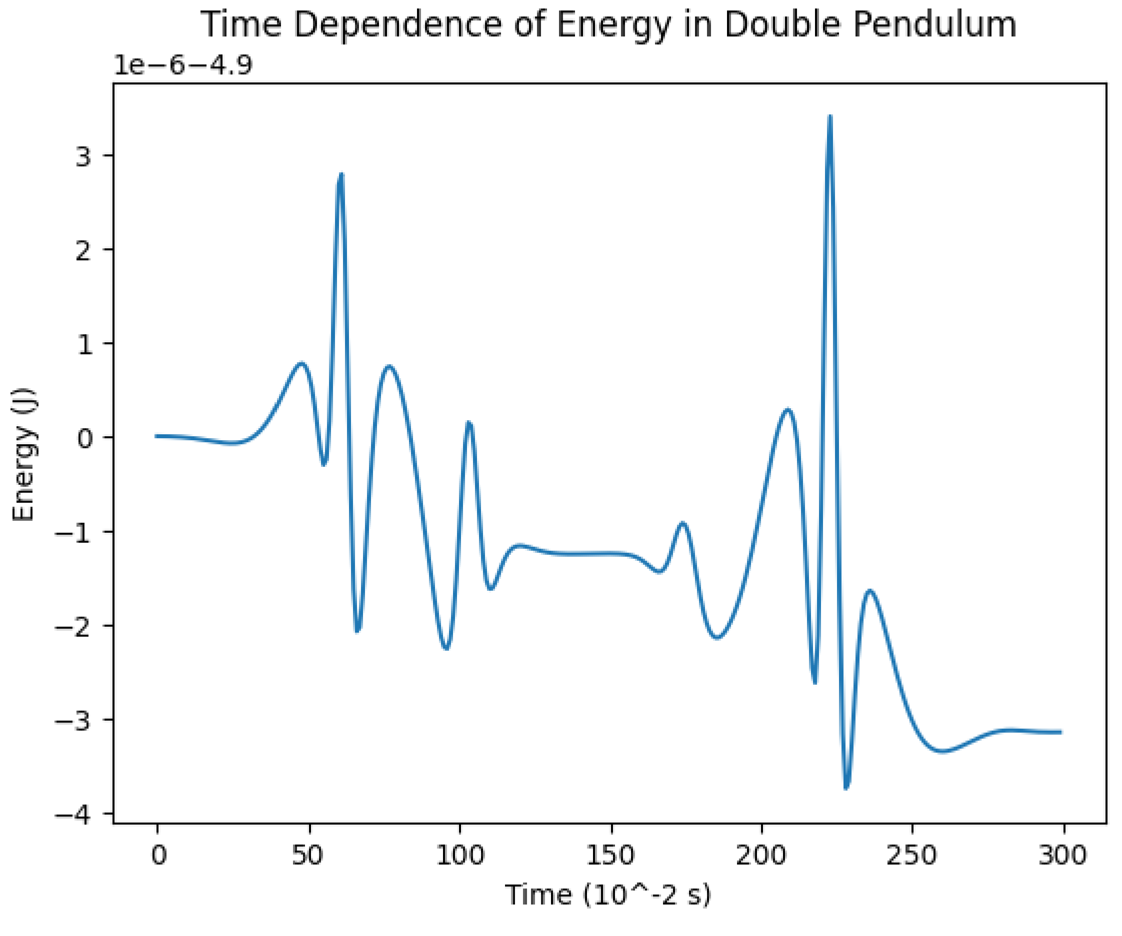

See attached figure of how frames were defined

In [70]:
```
#validation: plot Hamiltonian of system
#positions in q[0] and q[1], velocities in q[2] and q[3]
def KE(s):
    [theta1, theta2, theta1d, theta2d] = s
    [m1, m2, R1, R2] = [1, 1, 1, 1]

    x1d = R1 * np.cos(theta1) * theta1d
    y1d = R1 * np.sin(theta1) * theta1d
    x2d = x1d + R2 * np.cos(theta1 + theta2) * (theta1d + theta2d)
    y2d = y1d + R2 * np.sin(theta1 + theta2) * (theta1d + theta2d)

    return (0.5 * m1 * R1**2 * theta1d**2 ) + (0.5 * m2 * (x2d**2 + y2d**2) )
```

```python
def U(s):
    [theta1, theta2, _, _] = s
    [m1, m2, R1, R2, g] = [1, 1, 1, 1, 9.8]
    return m1 * g * -R1 * np.cos(theta1) + \
        -m2 * g * (
        R1 * np.cos(theta1) +
        R2 * np.cos(theta1 + theta2)
    )

def H(s):
    return KE(s) + U(s)

H_array = [H(s) for s in q_array.T]
```

In [71]:
```python
#plot
plt.figure()
plt.title("Time Dependence of Energy in Double Pendulum")
plt.xlabel("Time (10^-2 s)")
plt.ylabel('Energy (J)')

plt.plot(H_array)
# plt.plot(U_array)
# plt.plot(KE_array)
#plt.legend(["E(t)", "U(t)", "KE(t)"], loc = 'lower left')
```

Out[71]:
```
[<matplotlib.lines.Line2D at 0x12c56f133d0>]
```




## 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.**

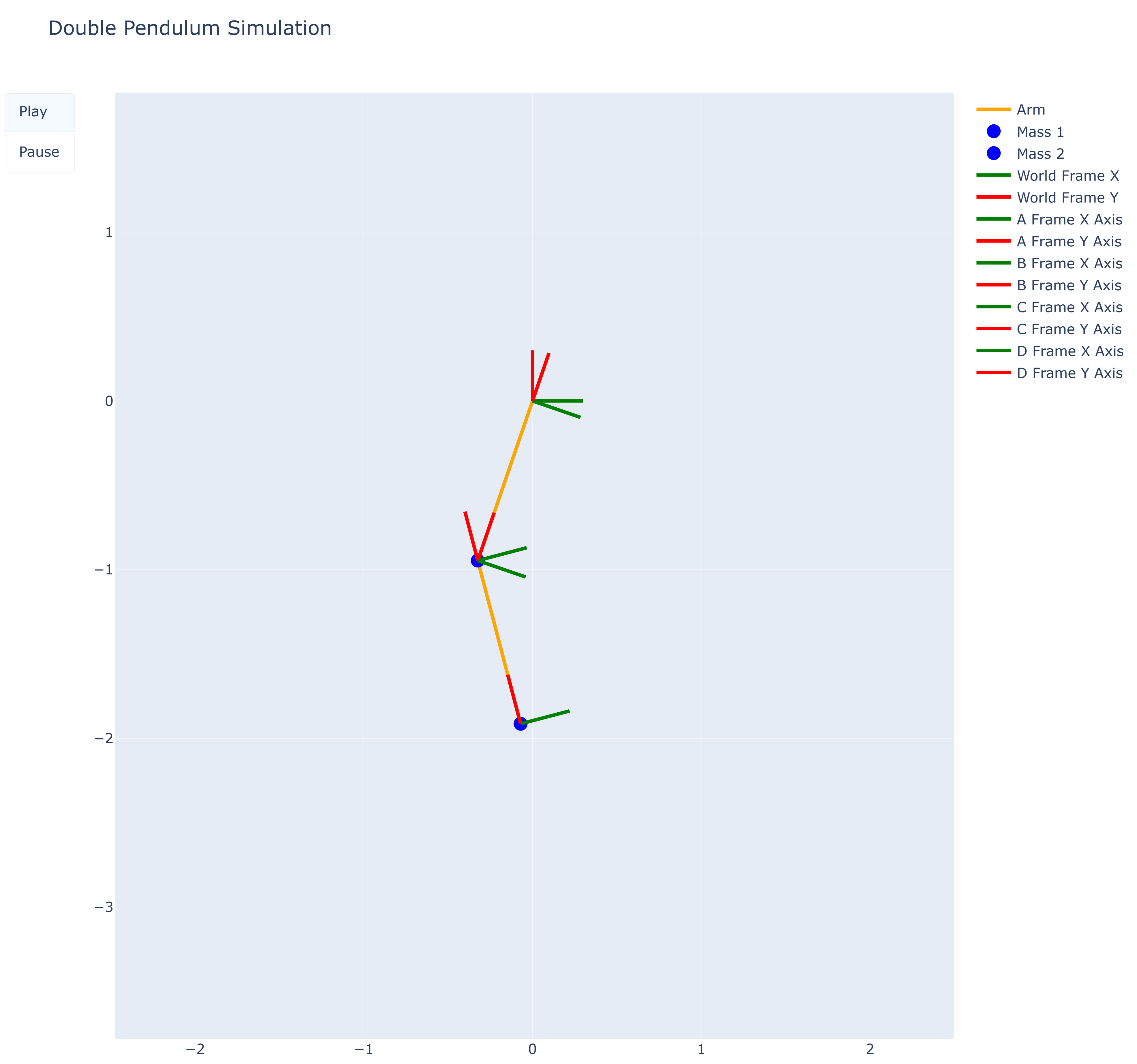
In [63]:
```python
def animate_double_pend(theta_array,L1=1,L2=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    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]

    #Define new frames
    Rwa_i = np.array([[np.cos(theta_array[0][i]), -np.sin(theta_array[0][i])],
                      [np.sin(theta_array[0][i]),  np.cos(theta_array[0][i])]])

    Rbc_i = np.array([[np.cos(theta_array[1][i]), -np.sin(theta_array[1][i])],
                      [np.sin(theta_array[1][i]),  np.cos(theta_array[1][i])]])
    I = np.identity(2)

    Tab = SO2AndR2ToSE2_np(I,     [0, -1])
    Tbc = SO2AndR2ToSE2_np(Rbc_i, [0,  0])
    Tcd = SO2AndR2ToSE2_np(I,     [0, -1])

    frame_b_x_axis[:,i] = t_wa.dot(Tab).dot([x_axis[0], x_axis[1], 1])[0:2]
    frame_b_y_axis[:,i] = t_wa.dot(Tab).dot([y_axis[0], y_axis[1], 1])[0:2]

    frame_c_x_axis[:,i] = t_wa.dot(Tab).dot(Tbc).dot([x_axis[0], x_axis[1], 1])[0:2]
    frame_c_y_axis[:,i] = t_wa.dot(Tab).dot(Tbc).dot([y_axis[0], y_axis[1], 1])[0:2]

    frame_d_x_axis[:,i] = t_wa.dot(Tab).dot(Tbc).dot(Tcd).dot([x_axis[0], x_axis[1], 1])[0:2]
    frame_d_y_axis[:,i] = t_wa.dot(Tab).dot(Tbc).dot(Tcd).dot([y_axis[0], y_axis[1], 1])[0:2]


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

```
##########################
# Defining data dictionary.
# Trajectories are here.
data=[
    # 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',
                                        '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),
                             ),

                         #--------------b 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),
                             ),

                         #--------------c 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),
                             ),

                         #--------------d 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)
```

In [64]:
```
animate_double_pend(q_array, L1=1, L2=1, T=3)
```


Double Pendulum Simulation
Play
Pause
Arm
Mass 1
Mass 2
World Frame X
World Frame Y
A Frame X Axis
A Frame Y Axis
B Frame X Axis
B Frame Y Axis
C Frame X Axis
C Frame Y Axis
D Frame X Axis
D Frame Y Axis

```
In [ ]:
```

1) $R_1(\theta_1)$ and $R_2(\theta_2) \in SO(n)$

Prove $R_1(\theta_1)R_2(\theta_2) \in SO(n)$:

$$\det(R_1) = \det(R_2) = 1$$
$$R_1^T R_1 = R_2^T R_2 = I$$

$$(R_1 R_2)^T R_1 R_2 = R_2^T \overbrace{R_1^T R_1}^{I} R_2$$
$$= R_2^T R_2$$
$$\underline{(R_1 R_2)^T R_1 R_2 = I}$$

$$\det(R_1 R_2) = \det(R_1)\det(R_2)$$
$$= \underline{\underline{1 \cdot 1 = 1}}$$

2) $g_1(x_1, y_1, \theta_1) \in SE(2)$

$$\longrightarrow g_1 = \begin{bmatrix} R_1(\theta_1) & \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \\ 0\ 0 & 1 \end{bmatrix}. \quad R_1(\theta) \in SO(2), \ \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \in \mathbb{R}^{2 \times 1}$$

$g_2(x_2, y_2, \theta_2) \in SE(2)$

$$\longrightarrow g_2 = \begin{bmatrix} R_2(\theta_2) & \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \\ 0\ 0 & 1 \end{bmatrix}. \quad R_2(\theta) \in SO(2), \ \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \in \mathbb{R}^{2 \times 1}$$

continued $\longrightarrow$

2) Show $g_1(x_1, y_1, \theta_1)\, g_2(x_2, y_2, \theta_2) \in SE(2)$.

$$g_1 g_2 = \begin{bmatrix} R_1(\theta_1) & \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \\ 0 \quad 0 & 1 \end{bmatrix} \begin{bmatrix} R_2(\theta) & \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \\ 0 \quad 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} R_1(\theta_1) & P_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_2(\theta_1) & P_2 \\ 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} R_1 R_2 & R_1 P_2 + P_1 \\ 0 & 1 \end{bmatrix}$$

$R_1$ & $R_2 \in SO(2) \longrightarrow R_1 R_2 \in SO(2)$

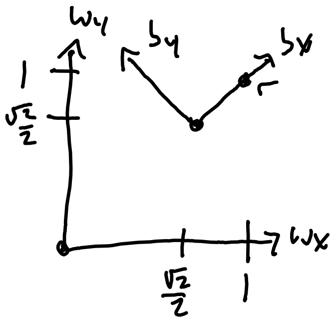
$R_1 P_2$ & $P_1 \in \mathbb{R}^{2\times 1} \longrightarrow R_1 P_2 + P_1 \in \mathbb{R}^{2\times 1}$

Therefore the product $g_1(x_1, y_1, \theta_1)\, g_2(x_2, y_2, \theta_2)$ has form

$$\begin{bmatrix} R \in SO(2) & P \in \mathbb{R}^{2\times 1} \\ 0 & 1 \end{bmatrix},$$

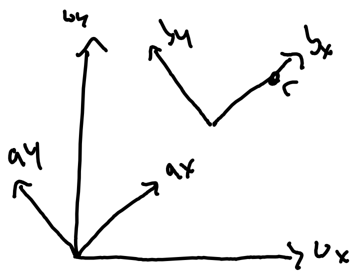
implying $g_1 g_2$ belongs to $SE(2)$.

3. Homogeneous transformation in SE(2):




Take a change of coordinate from w frame to b frame of point r:

use an intermediate frame:




$$g_{wa} = \begin{bmatrix} R_{wa} & 0 \\ 0 & 1 \end{bmatrix}, \quad R_{wa} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$g_{AB} = \begin{bmatrix} I & P_{AB} \\ 0 & 1 \end{bmatrix} \qquad P_{AB} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\bar{r}_w = \begin{bmatrix} r_w \\ 1 \end{bmatrix} = G_A G_B \bar{r}_B$$

"$r_i$" = vector position of r in the "i" frame

"$\bar{r}_i$" = vector with a trailing 1

$$\bar{r}_w = \begin{bmatrix} R_{wa} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & P_{AB} \\ 0 & 1 \end{bmatrix} \bar{r}_B$$

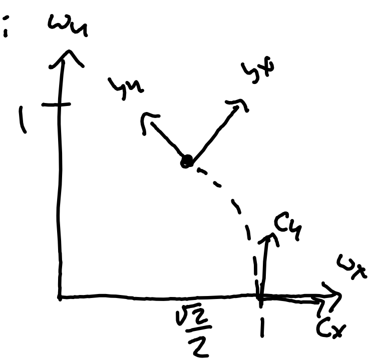
continued ---->

3. $\bar{r}_W = \begin{bmatrix} R_{WA} & R_{WA}P_{AB} \\ 0 & 1 \end{bmatrix} \bar{r}_B$ :

a generalized transformation has been decomposed into a rotation and a translation.

The order of operations we should usually use is rotation $\longrightarrow$ translation, because if we translated first, we'd be revolving our coordinate representation about the origin on a moment arm as well.

If the order were reversed:




$G_{WC} = \begin{bmatrix} I & P_{WC} \\ 0 & 1 \end{bmatrix}, \; P_{WC} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$G_{CB} = \begin{bmatrix} R_{CB} & P_{CB} \\ 0 & 1 \end{bmatrix}, \; P_{CB} = \begin{bmatrix} \frac{\sqrt{2}}{2} - 1 \\ \frac{\sqrt{2}}{2} \end{bmatrix}$

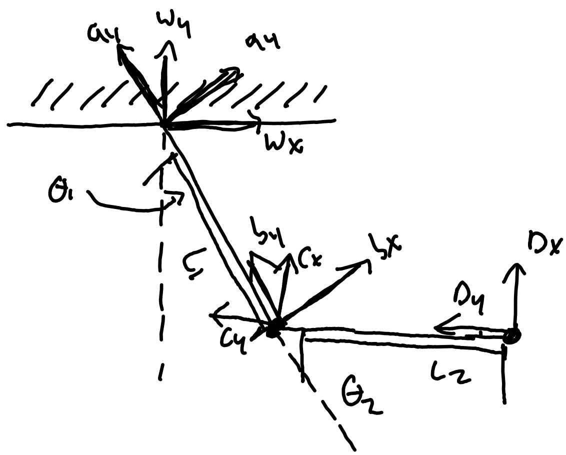
$G_{WC} G_{CB} = \begin{bmatrix} I & P_{WC} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_{CB} & P_{CB} \\ 0 & 1 \end{bmatrix}$

r_w_bar $= \begin{bmatrix} R_{CB} & P_{CB} + P_{WC} \\ 0 & 1 \end{bmatrix}$ r_b_bar

difference:

the rotation in this version has its own associated translation $(R_{CB}, P_{CB})$.

4. Diagram of how frames were defined:




Transformation matrices:

$$G_{WA} = \begin{bmatrix} R_{WA}(\theta_1) & \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ 0 & 1 \end{bmatrix}, \quad R_{WA} = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) \\ \sin(\theta_1) & \cos(\theta_1) \end{bmatrix}$$

$$G_{AB} = \begin{bmatrix} I & \begin{bmatrix} 0 \\ -L_1 \end{bmatrix} \\ 0 & 1 \end{bmatrix}$$

$$G_{BC} = \begin{bmatrix} R_{BC}(\theta) & \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ 0 & 1 \end{bmatrix}, \quad R_{BC} = \begin{bmatrix} \cos(\theta_2) & -\sin(\theta_2) \\ \sin(\theta_2) & \cos(\theta_2) \end{bmatrix}$$

$$G_{CD} = \begin{bmatrix} I & \begin{bmatrix} 0 \\ -L_2 \end{bmatrix} \\ 0 & 1 \end{bmatrix}$$