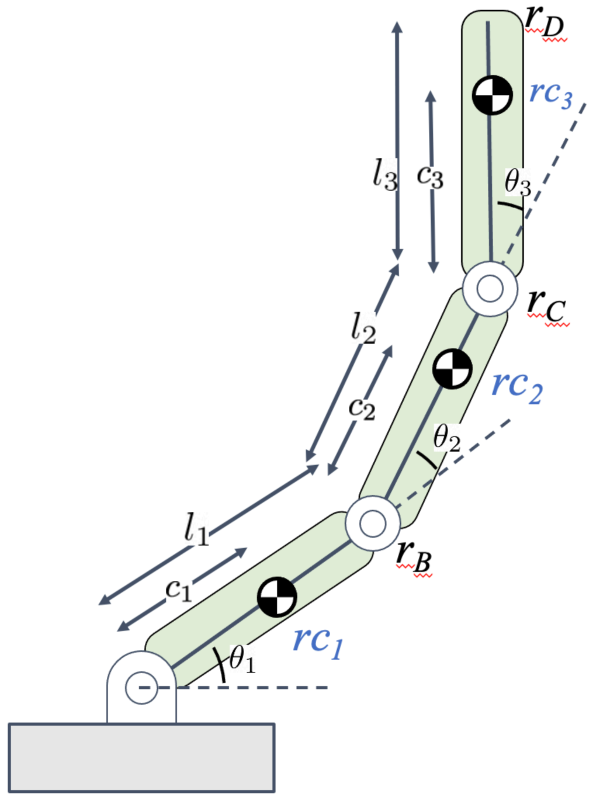
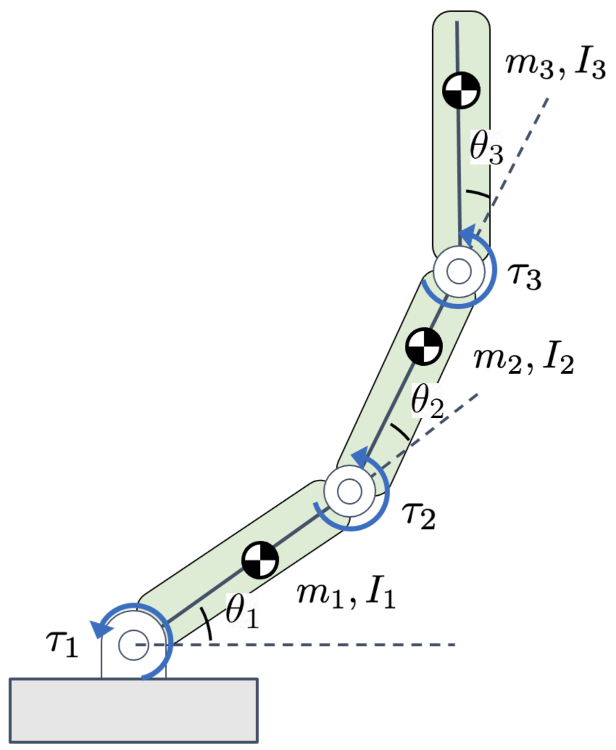
HW 8. Three DoF Planar Arm

The goal of HW 8 is to create a dynamics simulation of a 3 DoF openchain system described in the figures below.





```
!pip3 install sympy matplotlib seaborn ffmpeg-python
```

```
    Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
    Requirement already satisfied: sympy in /usr/local/lib/python3.7/dist-package
    Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-pa
    Requirement already satisfied: seaborn in /usr/local/lib/python3.7/dist-packa
    Requirement already satisfied: ffmpeg-python in /usr/local/lib/python3.7/dist-
    Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.7/dist-
    Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /u
    Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/dist-p
    Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/
    Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-
    Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3
    Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/
    Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pack
    Requirement already satisfied: pandas>=0.23 in /usr/local/lib/python3.7/dist-
    Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.7/dist-pa
    Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-
    Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packag
```

```
import sympy as sym
from sympy import Symbol, simplify, lambdify
from sympy.matrices.expressions import transpose
from sympy import sin, cos, Matrix
import numpy as np
from IPython.display import HTML
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import animation
%matplotlib inline
```

Declare Symbolic Variables

```
# Write your code: Symbolic variable definition. Complete the following variable li
[m1, m2, m3 , c1, c2, c3, l1, l2, l3, I1, I2, I3, g]
= sym.symbols('m1 m2 m3 c1 c2 c3 l1 l2 l3 I1 I2 I3 g')

[th1, th2, th3, dth1, dth2, dth3, ddth1, ddth2 , ddth3]
= sym.symbols('th1 th2 th3 dth1 dth2 dth3 ddth1 ddth2 ddth3')

[tau1, tau2, tau3] = sym.symbols('tau1 tau2 tau3')

# Symbolic variable groupings: No need to change
q  = Matrix([th1,  th2, th3])
dq = Matrix([dth1,  dth2, dth3])
ddq= Matrix([ddth1, ddth2, ddth3])
u  = Matrix([tau1, tau2, tau3]) # control input
z = Matrix([q.transpose(), dq.transpose()]) # state
p   = [l1, l2, l3, c1, c2, c3, m1, m2, m3, I1, I2, I3, g]  # parameters

zp_params = list(z)+list(p)
zup_params = list(z)+list(u)+list(p)

# Parameters: No need to change
m1_p = 1.5
m2_p = 1.
m3_p = 2.0
l1_p = .6
l2_p = .4
l3_p = .5
c1_p = .3
c2_p = .2
c3_p = .4
I1_p = 0.05
I2_p = 0.05
I3_p = 0.09

g_p  = 9.81
params = [l1_p, l2_p , l3_p, c1_p, c2_p, c3_p, m1_p,
          m2_p, m3_p, I1_p, I2_p, I3_p, g_p]


# Handy functions to be used to compute time derivative and generalized forces: No
ddt = lambda x: x.jacobian(q)*dq + x.jacobian(dq)*ddq        # Time derivative
F2Q = lambda F, r : simplify(r.jacobian(q).transpose()*F)    # force contributions
M2Q = lambda M, w : simplify(w.jacobian(dq).transpose()*M)   # moment contributions
```

Q.1. [25 pts] Kinemeatics

Complete the code to compute the positions and velocities of the points of interest. Here, all positions and velocities (angular and linear) should be defined to compute kinetic and potential energies.

```
# Kinemeatics
rO = Matrix([0, 0, 0])    # position of link 1
ehat1 = Matrix([sin(th1), cos(th1), 0])           # Define unit vector along Leg 1
ehat2 = Matrix([sin(th1 + th2), cos(th1 + th2), 0])  # Define unit vector along Leg
ehat3 = Matrix([sin(th1 + th2 + th3), cos(th1 + th2 + th3), 0])  # Define unit vect

ghat  = Matrix([0, -1 , 0])

# Write your code: Define CoM position and link position
rB  = rO + l1 * ehat1    # Position of base of link 2
rC  = rB + l2 * ehat2    # Position of end of link 2
rD  = rC + l3 * ehat3    # Position of end of link 3

rc1 = c1 * ehat1 + rO    # Position of link 1 CoM
rc2 = c2 * ehat2 + rB    # Position of CoM of link 2
rc3 = c3 * ehat3 + rC    # Position of CoM of link 3


# Write your code: Define linear and angular velocity of each link
# Hint: Utilize 'ddt' function

vc1 = ddt(rc1)          # Velocity of link 1 CoM
vc2 = ddt(rc2)          # Velocity of link 2 CoM
vc3 = ddt(rc3)          # Velocity of link 3 CoM

keypoints = Matrix([[rO], [rB], [rC], [rD]]).reshape(4, 3)
keypoints_func = sym.lambdify(zp_params, keypoints)
```

```
# Test Forward Kinematics implementation: No need to change
def test_forward_kinematics(params):

  test_config = [[0, 0, 0, 0, 0, 0],
                 [0, np.pi/4, np.pi/6, 0, 0, 0],
                 [np.pi/8, -0.6, 0, 0, 0, 0]]
  res = []
  soln = np.array([[[ 0., 0., 0.],
           [ 0., 0.6, 0.],
           [ 0., 1., 0.],
           [ 0., 1.5, 0.]],
        [[ 0.,         0.,          0.],
          [ 0.,         0.6,         0.],
          [ 0.28284271, 0.88284271,  0.],
          [ 0.76580563, 1.01225224,  0.]],
        [[0.,          0.,          0.],
          [ 0.22961006, 0.55432772,  0.],
          [ 0.14728232, 0.94576372,  0. ],
          [ 0.04437264, 1.43505872,  0. ]]])

  for i, q in enumerate(test_config):
    xp_params = q + list(params)
    res.append(keypoints_func(*xp_params))
  res_np = np.array(res)

  assert np.allclose(soln, res_np), f'Your Forward kinematics implementation is wrc

  print('Your Forward Kinematics implementation is correct!')


test_forward_kinematics(params)
```

```
    Your Forward Kinematics implementation is correct!
```

Q.2. [20 pts] Kinetic and Potential Energy of Links

Define the kinetic and potential energy of each link

```
# Write your code: Kinetic and Potential Energy of link 1
T1 = simplify(m1 * vc1.dot(vc1) * 1/2 + I1 * pow(dth1, 2) * 1/2)
V1 = m1 * g * rc1.dot(-ghat)

# Write your code: Kinetic and Potential Energy of link 2
T2 = simplify(m2 * vc2.dot(vc3) * 1/2 + I2 * pow(dth1 + dth2, 2) * 1/2)
V2 = m2 * g * rc2.dot(-ghat)

# Write your code: Kinetic and Potential Energy of link 3
T3 = simplify(m3 * vc3.dot(vc3) * 1/2 + I3 * pow(dth1 + dth2 + dth3, 2) * 1/2)
V3 = m3 * g * rc3.dot(-ghat)
```

Q.3. [15 pts] Lagrangian and Total Energy (Hamiltonian)

```
# Write your code: Kinetic, Potential, and total energy of the entire system
KE = T1 + T2 + T3
PE = V1 + V2 + V3
E = KE + PE

# Write your code: Lagrangian
L = Matrix([simplify(KE - PE)])
```

Q.3. [20 pts] Generalized Forces

Define generalized force vectors using a M2Q function.

Hint: The result should be $Q = \begin{pmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{pmatrix}$.

```
# Write your code:
Q_tau1 = M2Q(tau1, Matrix([dth1]))
Q_tau2 = M2Q(tau2, Matrix([dth1 + dth2])) + M2Q(-tau2, Matrix([dth1]))
Q_tau3 = M2Q(tau3, Matrix([dth1 + dth2 + dth3])) + M2Q(-tau3, Matrix([dth1 + dth2])

Q = Q_tau1 + Q_tau2 + Q_tau3

print(Q_tau1)
print(Q_tau2)
print(Q_tau3)

print(Q)
```

```
    Matrix([[tau1], [0], [0]])
    Matrix([[0], [tau2], [0]])
    Matrix([[0], [0], [tau3]])
    Matrix([[tau1], [tau2], [tau3]])
```

Q.4. [10 pts] Equations of Motion

Complete the equations of motion by applying Lagrange formula. Find A (Mass matrix) and b vector explained in the lecture.

```
# Write your code: Find A and b by solving Lagrange's equation
dL_dq = L.jacobian(q).transpose()
dL_dqd = L.jacobian(dq).transpose()
g = ddt(dL_dqd) - dL_dq - Q
A = simplify(g.jacobian(ddq))
b = simplify(A * ddq - g)
```

```
A
```

```
A
```

$$\left[\begin{matrix} 1.0I_1 + 1.0I_2 + 1.0I_3 + 1.0c_1^2m_1 + m_2\left(c_2^2 + 2c_2l_1\cos(th_2) + l_1^2\right) + m_3\left(c_3^2 + 2c_3l_1\cos(th_2 \\ 1.0I_2 + 1.0I_3 + c_2m_2\left(c_2 + l_1\cos(th_2)\right) + m_3\left(c_3^2 + c_3l_1\cos(th_2 + th_3) + 2 \\ 1.0I_3 + c_3m_3\left(c_3 + l_1\cos(th_2 + th_3) + l_2\cos(\end{matrix}\right.$$

Helper Functions For Simulation

```
# Helper Functions: No need to change
A_func = sym.lambdify(zp_params, A)
b_func = sym.lambdify(zup_params, b)
keypoints_func = sym.lambdify(zp_params, keypoints)
kinetic_energy_func = sym.lambdify(zp_params, KE)
potential_energy_func = sym.lambdify(zp_params, PE)
E_func = sym.lambdify(zp_params, E)
```

Q.5. [10 pts] **Dynamics**

Complete the dynamics function returning $\ddot{q}$ by using A and b.

```
def dynamics(x, params):
  u = [0, 0, 0]
  xp_params = list(x) + list(params)
  xup_params = list(x) + u + list(params)
  A_dyn = A_func(*xp_params)
  b_dyn = b_func(*xup_params)

  # Write your code: Computer joint acceleration
  ddq = np.matmul(np.linalg.inv(A_dyn), b_dyn).T

  return ddq
```

Perform Simulation

```
# Simulation: No need to change
dt = 0.001;
tf = 7;
num_step = int(np.floor(tf/dt));
tspan = np.linspace(0, tf, num_step);
x0 = np.array([np.pi/3, -1, 1, 0.0, 0.0, 0.0]).T;
x_out = np.zeros((6,num_step));
x_out[:,0] = x0;
keypoints_hist = []
kinetic_energy_hist = []
potential_energy_hist = []
total_energy_hist = []

for i in range(num_step-1):
    xp_params = list(x_out[:, i]) + list(params)
    keypoints_hist.append(keypoints_func(*xp_params))
    kinetic_energy_hist.append(kinetic_energy_func(*xp_params))
    potential_energy_hist.append(potential_energy_func(*xp_params))
    total_energy_hist.append(E_func(*xp_params))

    ddq = dynamics(x_out[:,i], params);
    x_out[3:,i+1] = x_out[3:,i] + ddq*dt;
    x_out[:3,i+1] = x_out[:3,i] + x_out[3:,i+1]*dt;
```

Visualize Simulation

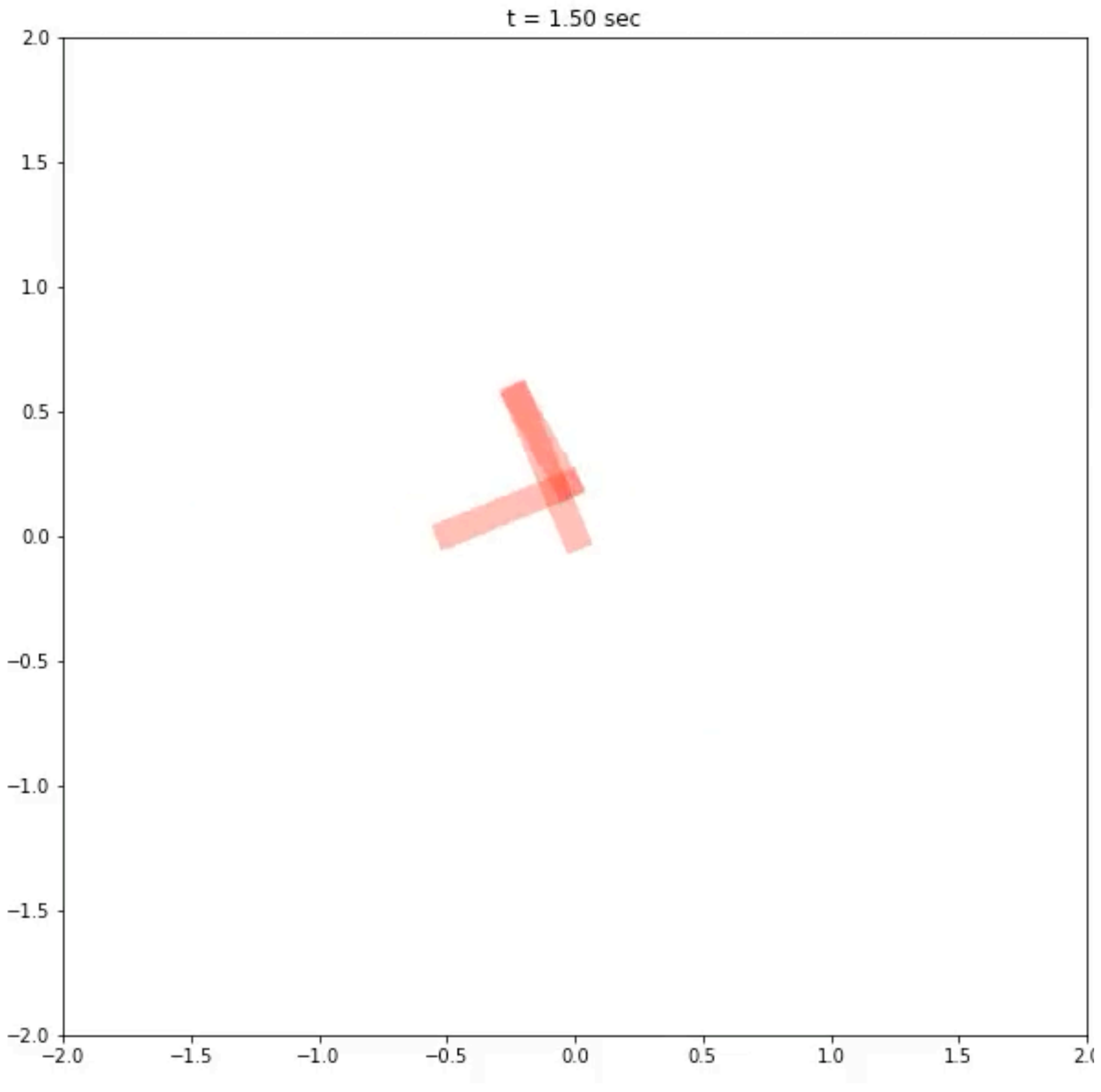
```
# 3 dof arm visualization function: No need to change
def visualize_arm(keypoints_hist, dt = 0.001, num_frames=200):
  fig= plt.figure(figsize=(10,10))
  ax = plt.subplot(1,1,1)
  keypoints = keypoints_hist[0]
  link1, = ax.plot([], [], 'r', lw=15, alpha=0.3)
  link2, = ax.plot([], [], 'r', lw=15, alpha=0.3)
  link3, = ax.plot([], [], 'r', lw=15, alpha=0.3)

  txt_title = ax.set_title('')

  ax.set_xlim(( -2, 2))
  ax.set_ylim((-2, 2))
  txt_title = ax.set_title('')
  interval = len(keypoints_hist)//num_frames
  def drawFrame(k):
    k = interval*k
    keypts = keypoints_hist[k]

    x1 = keypts[0, 0]
    x2 = keypts[1, 0]
    y1 = keypts[0, 1]
    y2 = keypts[1, 1]
    link1.set_data([x1, x2], [y1, y2])
    x1 = keypts[1, 0]
    x2 = keypts[2, 0]
    y1 = keypts[1, 1]
    y2 = keypts[2, 1]
    link2.set_data([x1, x2], [y1, y2])
    x1 = keypts[2, 0]
    x2 = keypts[3, 0]
    y1 = keypts[2, 1]
    y2 = keypts[3, 1]
    link3.set_data([x1, x2], [y1, y2])
    txt_title.set_text(f't = {dt*k:.2f} sec')
    return link1, link2, link3
  anim = animation.FuncAnimation(fig, drawFrame, frames=num_frames, interval=interv
  return anim
```

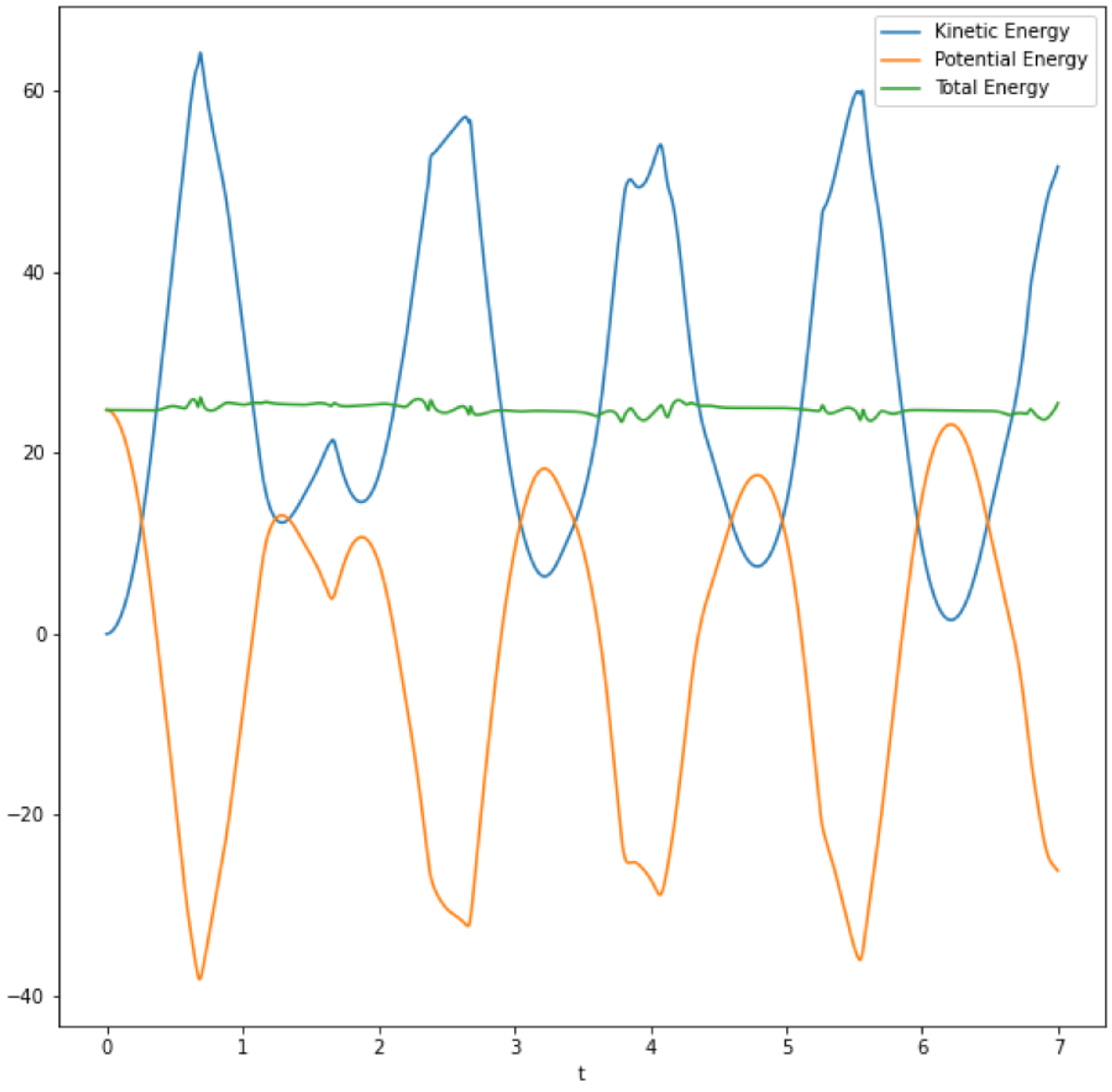
```
anim = visualize_arm(keypoints_hist, dt = dt, num_frames=200)
plt.close()
HTML(anim.to_html5_video())
```



0:01 0:06

```
# Plot Kinetic Energy, Potential Energy and Total Energy: No need to change
```

```
kinetic_energy_hist_np = np.array(kinetic_energy_hist)
potential_energy_hist_np = np.array(potential_energy_hist)
total_energy_hist_np = np.array(total_energy_hist).reshape(-1)
ts = np.arange(0, tf-dt, dt)
fig= plt.figure(figsize=(10,10))
ax = plt.subplot(1,1,1)
ax.plot(ts, kinetic_energy_hist_np, label='Kinetic Energy')
ax.plot(ts, potential_energy_hist_np, label='Potential Energy')
ax.plot(ts, total_energy_hist_np, label='Total Energy')
ax.legend()
ax.set_xlabel('t')
plt.show()
```



(Optional) [10 pts] Explain why the total energy of the arm changes over time.

Hint: You can try different time step other than 0.001 to show how the simulation accuracy changes total energy profile.

Double-click (or enter) to edit

Colab paid products - Cancel contracts here