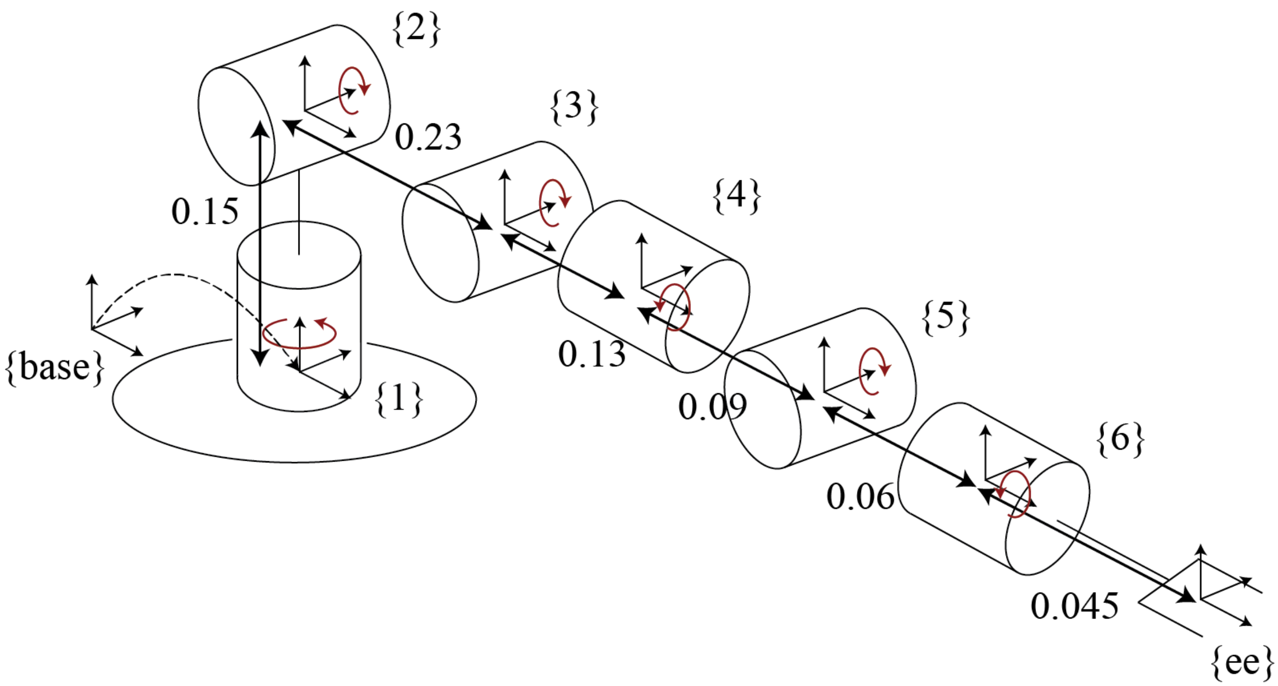
Homework 5: 6-DoF Open-chain Arm Inverse Kinematics



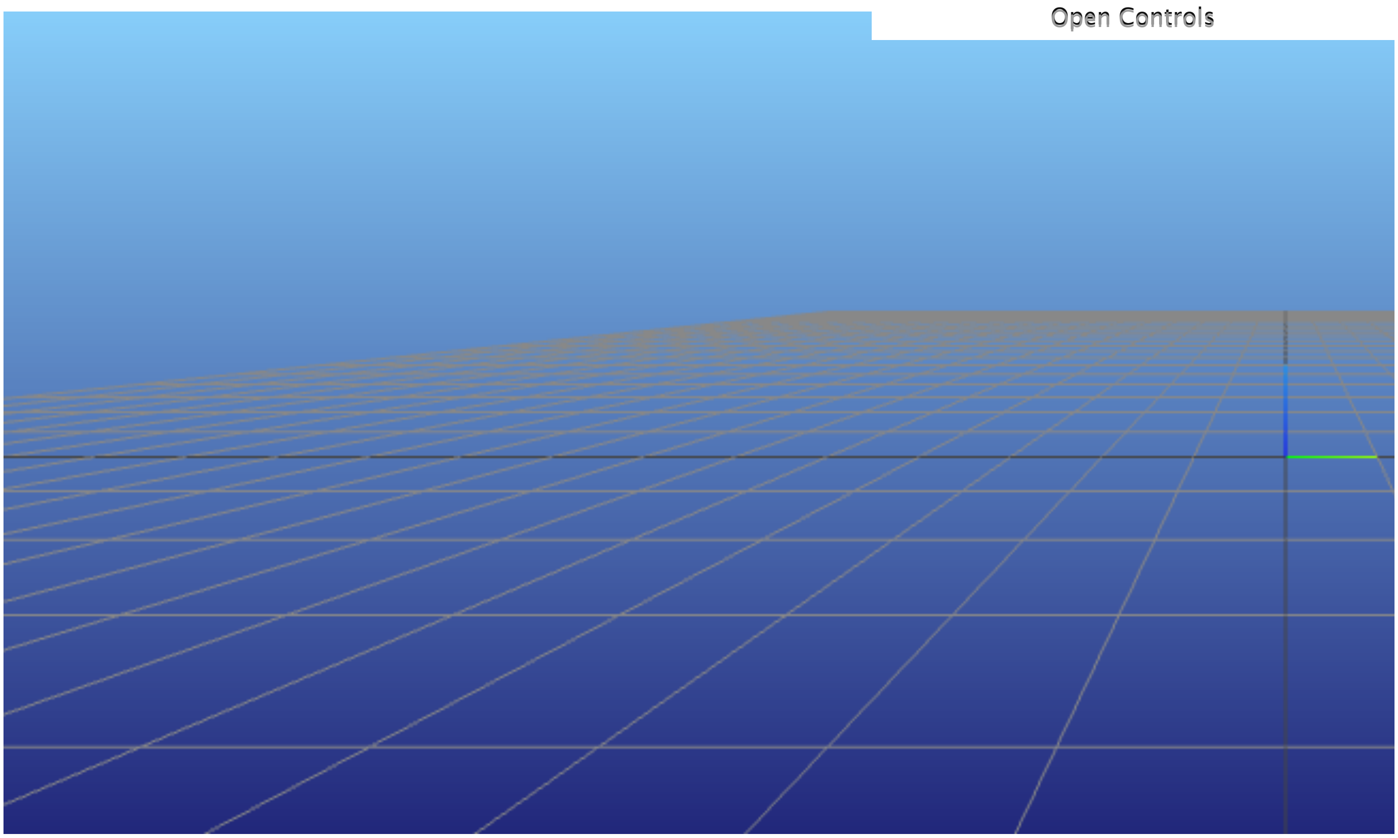
```
#install meshcat visualizer
!git clone https://github.com/RussTedrake/meshcat-python --branch colab --recursive
!pip3 install u-msgpack-python
!git clone https://github.com/DARoSLab/CS403-Intro-Robotics
import sys
sys.path.append('/content/meshcat-python/src')


#Python libraries
from meshcat.jupyter import JupyterVisualizer
import meshcat.geometry as g
import meshcat.transformations as tf
from meshcat.geometry import Box, Mesh, Sphere, MeshLambertMaterial, DaeMeshGeometr
import numpy as np
import os
import time
import matplotlib.pyplot as plt

mesh_path =  '/content/CS403-Intro-Robotics/hws/hw4/meshes/'


def generate_transformation_matrix(xyz=[0,0,0], rpy=[0,0,0]):
    Rx = tf.rotation_matrix(rpy[0], [1, 0, 0])[:3, :3]
    Ry = tf.rotation_matrix(rpy[1], [0, 1, 0])[:3, :3]
    Rz = tf.rotation_matrix(rpy[2], [0, 0, 1])[:3, :3]

    R = np.matmul(Rz, np.matmul(Ry, Rx))
    T = np.zeros((4,4))
    T[:3, :3] = R
    T[0, 3] = xyz[0]
    T[1, 3] = xyz[1]
    T[2, 3] = xyz[2]
    T[3, 3] = 1.0
    return T
```

Visualize robot

```
def visualize_robot():
  vis = JupyterVisualizer()
  vis["arm/base"].set_object(
      g.DaeMeshGeometry.from_file(
          os.path.join(mesh_path, 'arm_base.dae')), MeshLambertMaterial(color=0xA7A
  vis["arm/base/link1"].set_object(
      g.DaeMeshGeometry.from_file(
          os.path.join(mesh_path, 'arm_link1.dae')), MeshLambertMaterial(color=0x42
  )
  vis["arm/base/link1/link2"].set_object(
      g.DaeMeshGeometry.from_file(
          os.path.join(mesh_path, 'arm_link2.dae')), MeshLambertMaterial(color=0x36
  )
  vis["arm/base/link1/link2/link3"].set_object(
      g.DaeMeshGeometry.from_file(
          os.path.join(mesh_path, 'arm_link3.dae')), MeshLambertMaterial(color=0xf2
  )
  vis["arm/base/link1/link2/link3/link4"].set_object(
      g.DaeMeshGeometry.from_file(
          os.path.join(mesh_path, 'arm_link4.dae')), MeshLambertMaterial(color=0x00
  )
  vis["arm/base/link1/link2/link3/link4/link5"].set_object(
      g.DaeMeshGeometry.from_file(
          os.path.join(mesh_path, 'arm_link5.dae')), MeshLambertMaterial(color=0xA7
  )
  vis["arm/base/link1/link2/link3/link4/link5/ee"].set_object(
      g.DaeMeshGeometry.from_file(
          os.path.join(mesh_path, 'arm_ee.dae')), MeshLambertMaterial(color=0xFF42
  )
  return vis
```

```
def ForwardKinematics(viz, q=[0, 0, 0, 0, 0, 0]):
  base_to_joint1 = generate_transformation_matrix(xyz=[0.0, 0.0, 0.0], rpy=[0,0,q[0
  # Fill your code here
  joint1_to_joint2 = generate_transformation_matrix(xyz=[0.0, 0.0, 0.15], rpy=[0,q|
  joint2_to_joint3 = generate_transformation_matrix(xyz=[0.23, 0.0, 0.0], rpy=[0,q|
  joint3_to_joint4 = generate_transformation_matrix(xyz=[0.130, 0.0, 0.0], rpy=[q[3
  joint4_to_joint5 = generate_transformation_matrix(xyz=[0.090, 0.0, 0.0], rpy=[0,c
  joint5_to_joint6 = generate_transformation_matrix(xyz=[0.06, 0.0, 0.0], rpy=[q[5]
  joint6_to_ee = generate_transformation_matrix(xyz=[0.045, 0.0, 0.0], rpy=[0,0,0])

  vis["arm/base/link1"].set_transform(base_to_joint1)
  # Fill your code
  vis["arm/base/link1/link2"].set_transform(joint1_to_joint2)
  vis["arm/base/link1/link2/link3"].set_transform(joint2_to_joint3)
  vis["arm/base/link1/link2/link3/link4"].set_transform(joint3_to_joint4)
  vis["arm/base/link1/link2/link3/link4/link5"].set_transform(joint4_to_joint5)
  vis["arm/base/link1/link2/link3/link4/link5/ee"].set_transform(joint5_to_joint6)

  base_to_ee = base_to_joint1@joint1_to_joint2@joint2_to_joint3@joint3_to_joint4@jc
  return base_to_ee
```

```
vis = visualize_robot()
```




```
q_test = [0, np.pi, 0, 0, 0, 0] # Fill your code: Use your own test configuration
EE = ForwardKinematics(vis, q = q_test)
print(EE)
```

```
[[-1.0000000e+00  0.0000000e+00  1.2246468e-16 -5.5500000e-01]
 [ 0.0000000e+00  1.0000000e+00  0.0000000e+00  0.0000000e+00]
 [-1.2246468e-16  0.0000000e+00 -1.0000000e+00  1.5000000e-01]
 [ 0.0000000e+00  0.0000000e+00  0.0000000e+00  1.0000000e+00]]
```

Q.1 Implement a function that computes both World (explicit) and Body (implicit) Jacobians.

```
# Utility functions
def skew_sym_matrix(a):
  return np.array([[0, -a[2], a[1]],
                  [a[2], 0, -a[0]],
                  [-a[1], a[0], 0]])
def SO3toso3vec(R):
  omega, theta = None, 0
  if np.allclose(R, np.eye(3)): # if R = identity
    pass
  elif np.abs(np.trace(R)+1)<1e-5: # if theta = pi
    theta = np.pi
    r13 = R[0,2]
    r23 = R[1,2]
    r33 = R[2,2]
    omega = np.array([r13,r23, 1+r33])
    omega *= (1./np.sqrt(2*(1+r33)))
  else: # Other normal cases
    # Fill your code to compute so(3)
    theta = np.arccos(0.5*(np.trace(R)-1))
    omega_hat = (1./(2*np.sin(theta)))*(R-R.T)
    omega = np.array([omega_hat[2,1], omega_hat[0, 2], omega_hat[1, 0]])

  so3vec = omega * theta
  return so3vec
```

Q.1 (a) [15pt] Complete the following code computing Jacobian at a given joint position.

```
def make_SE3_from_EE(q):
    T01 = generate_transformation_matrix(xyz=[0.0, 0.0, 0.0], rpy=[0,0,q[0]])
    # Fill your code.
    T12 = generate_transformation_matrix(xyz=[0.0, 0.0, 0.15], rpy=[0,q[1],0])
    T23 = generate_transformation_matrix(xyz=[0.23, 0.0, 0.0], rpy=[0,q[2],0])
    T34 = generate_transformation_matrix(xyz=[0.130, 0.0, 0.0], rpy=[q[3],0,0])
    T45 = generate_transformation_matrix(xyz=[0.090, 0.0, 0.0], rpy=[0,q[4],0])
    T56 = generate_transformation_matrix(xyz=[0.06, 0.0, 0.0], rpy=[q[5],0,0])
    Tee = generate_transformation_matrix(xyz=[0.045, 0.0, 0.0], rpy=[0,0,0])

    T = {};

    # Fill your code to find the T(i_ee), i is equal to joint idx (1 - 6)
```

```
    T[6] = Tee  # T6_ee
    T[5] = T56 @ T[6] # T5_ee
    T[4] = T45 @ T[5] # T4_ee
    T[3] = T34 @ T[4] # T3_ee
    T[2] = T23 @ T[3] # T2_ee
    T[1] = T12 @ T[2] # T1_ee
    T[0] = T01 @ T[1] # T(global_ee)
    return T

def Adj(T):
  # Fill your code to make 6x6 adjoint mapping matrix from SE(3)
    Adj_mtx = np.zeros((6, 6))
    R = T[:3, :3];
    p = T[:3, 3];
    Adj_mtx[0:3, 0:3] = R
    Adj_mtx[3:6, 3:6] = R
    Adj_mtx[3:6, 0:3] = skew_sym_matrix(p) @ R
    return Adj_mtx

def Jacobian(q, coord='world'):

  T = make_SE3_from_EE(q);
  S = {}
  for i in range(6):
    S[i] = np.zeros(6)

  S[0][2] = 1;
  # Fill your code: complete the local se(3)
  # ...
  S[1][1] = 1;
  S[2][1] = 1;
  S[3][0] = 1;
  S[4][1] = 1;
  S[5][0] = 1;

  J = np.zeros((6, 6));
  for i in range(6): # Complete Jacobian calculation
    J[:, i] = Adj(np.linalg.pinv(T[i])) @ S[i]

  T_ee_rot = T[6]
  T_ee_rot[0:3, 3] = np.zeros((1,3))
  if coord=='world':
    # Fill your code to change the Jacobian's frame from EE to global
    J = Adj(T_ee_rot) @ J
```

```
    return J
```

Q.1.(b) [15 pts] Choose the test configuration and compare the result with your expectation. Shortly explain how you select the test configuration and Jacobian.

```
# Debug the Jacobian function using your test configuration:
q_test = np.array([0, np.pi, 0, 0, 0, 0])
J = Jacobian(q_test, 'world')

np.set_printoptions(precision=4)
print(J)
```

```
[[ 3.7103e-17  0.0000e+00  0.0000e+00  1.0000e+00  0.0000e+00  1.0000e+00]
 [ 0.0000e+00  1.0000e+00  1.0000e+00  0.0000e+00  1.0000e+00  0.0000e+00]
 [-1.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00]
 [ 0.0000e+00 -1.5000e-01  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00]
 [-5.1000e-01  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00]
 [ 0.0000e+00 -5.1000e-01 -5.1000e-01  0.0000e+00 -1.5000e-01  0.0000e+00]]
```

```
# Test Jacobian function. Use for debugging
def test_Jacobian():
  test_q = {}
  q0 = np.zeros((6, 1))
  q1 = np.array([np.pi/2, 0, 0, -np.pi/4, np.pi/6, 0.8]).reshape(6, 1)
  q2 = np.array([0, 0, np.pi/2, 0.5, np.pi/6, 0]).reshape(6, 1)

  test_q[0] = q0
  test_q[1] = q1
  test_q[2] = q2
  res = []
  for i in range(3):
    res.append(Jacobian(test_q[i],'world'))

  for i in range(3):
    res.append(Jacobian(test_q[i],'body'))

  soln = np.load('/content/CS403-Intro-Robotics/hws/hw5/jacobian_test.npy')
  if np.allclose(np.array(res), soln):
    print('Your implementation is correct')
  else:
    print('Your implementation is not correct try again')

  print(res[0])
  print(soln[0])


test_Jacobian()
```

Q.2 [10 pts] Report the end effector velocities that correspond to the following joint configurations and velocities.

$$
\begin{aligned}
&(a)\ q = [0, 0, 0, 0, 0, 0] && \dot{q} = [1, 0, 0, 0, 0, 0] \\
&(b)\ q = [0, \frac{\pi}{4}, 0, \pi, 0, 0] && \dot{q} = [1, 0, 1, 0, 0, 0] \\
&(c)\ q = [\frac{\pi}{2}, \frac{\pi}{4}, 0, 0.5, 0, 0] && \dot{q} = [0, 0, 1, 0, 2, 1]
\end{aligned}
$$

```
# Fill your code
q_a = np.array([0, 0, 0, 0, 0, 0]).reshape(6,1)
q_b = np.array([0, np.pi/4, 0, np.pi, 0, 0]).reshape(6,1)
q_c = np.array([np.pi/2, np.pi/4, 0, 0.5, 0, 0]).reshape(6,1)

# Fill your code
q_a_dot = np.array([1, 0, 0, 0, 0, 0]).reshape(6,1)
q_b_dot = np.array([1, 0, 1, 0, 0, 0]).reshape(6,1)
q_c_dot = np.array([0, 0, 1, 0, 2, 1]).reshape(6,1)

# Fill your code
vel_a = Jacobian(q_a,'world') @ q_a_dot
vel_b = Jacobian(q_b,'world') @ q_b_dot
vel_c = Jacobian(q_b,'world') @ q_c_dot

print('(a) vel = \n', vel_a)
print('(b) vel = \n', vel_b)
print('(c) vel = \n', vel_c)
```

```
(a) vel =
 [[0.   ]
 [0.   ]
 [1.   ]
 [0.   ]
 [0.555]
 [0.   ]]
(b) vel =
 [[-1.0147e-17]
 [ 1.0000e+00]
 [ 1.0000e+00]
 [-2.2981e-01]
 [ 3.9244e-01]
 [-2.2981e-01]]
(c) vel =
 [[-2.7552]
 [ 1.3851]
 [-0.0291]
 [-0.1007]
 [-0.3601]
 [-0.3601]]
```

Q.3 Compute the joint configuration that corresponds to the following end-effector pose.

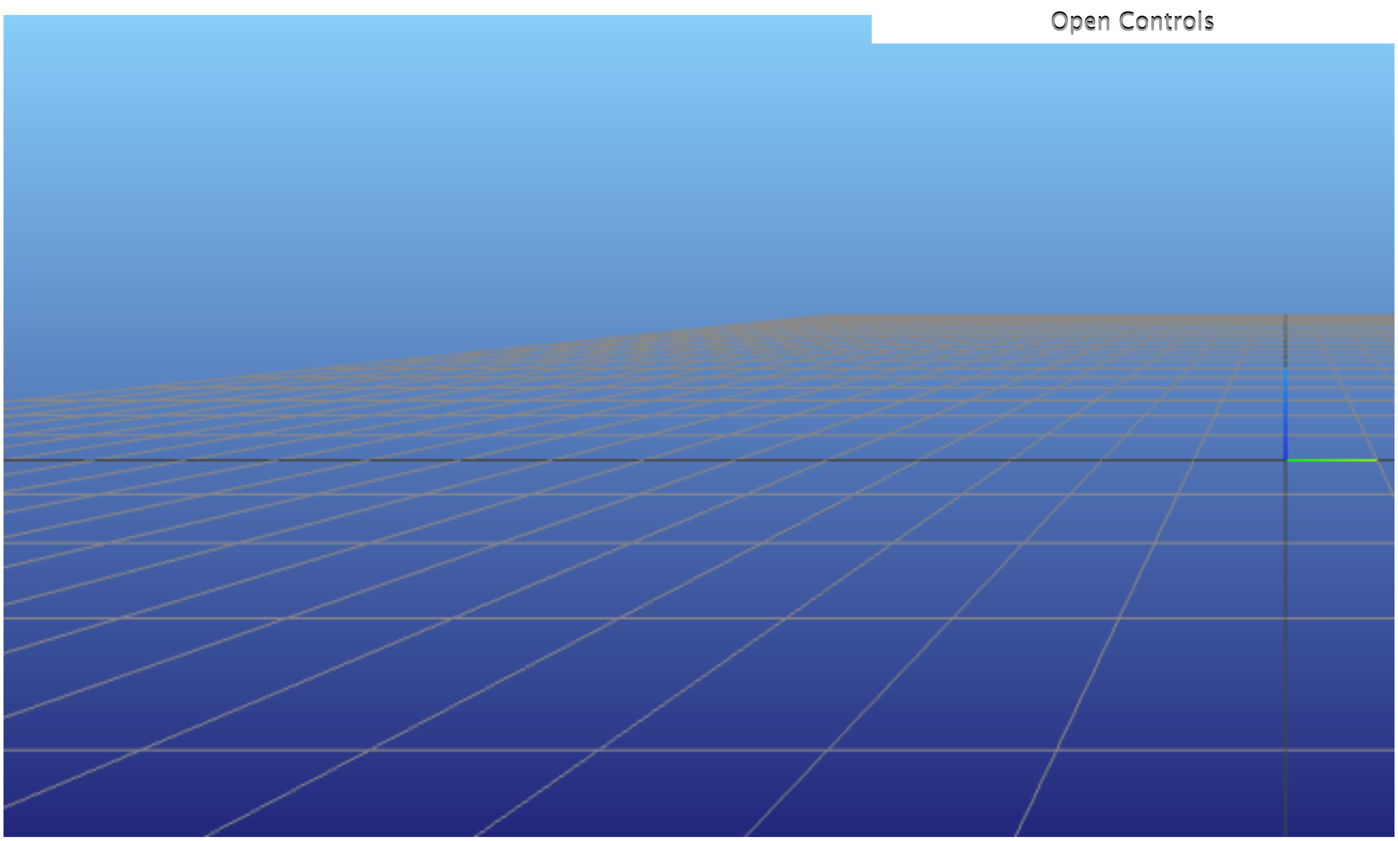
$$P = [0.2, 0.31, 0.2],$$
$$ZYX = [0, 0, \frac{\pi}{2}].$$

Q.3(a) [15 pts] Complete inverse kinematics function using world Jacobian.

```
def newton_raphson_world(T_des, q = np.zeros((6,1)), num_steps=50, lr=0.2):
  q_hist = []
  error_hist = []
  for step in range(num_steps):
      J = Jacobian(q, 'world')

      T = make_SE3_from_EE(q)
      Tee = T[0]

      R_ee  = Tee[:3, :3]
      p_ee = Tee[:3, 3]
      R_des = T_des[:3, :3]
      p_des = T_des[:3,  3]

      err = np.zeros((6,1));
      # Fill your code: Complete error computation
      err[:3, 0] = SO3toso3vec(R_des - np.transpose(R_ee))
      err[3:6, 0] = T_des[0:3, 3] - T[6][0:3, 3]

      # Newton Rapson (no need to change unless you want to implement yourself)
      q = q + lr*np.linalg.pinv(J)@err;
      if np.linalg.norm(err) < 0.0001:
          break
      q_hist.append(q)
      error_hist.append(err)
  print(error_hist)
  return q_hist, error_hist
```

```
# Test inverse kinematics (world Jacobian)

# Fill your code:
# 1. select your initial configuration
# 2. Define the goal SE(3)

q0 = np.array([0,np.pi,0,0,0,0]).reshape(6,1)
T_des = np.array([[-1, 0, 0, -0.555],[0, 1, 0, 0],[0, 0, -1, -0.15],[0, 0, 0, 1]])

N = 50
[q_hist, error_hist] = newton_raphson_world(T_des, q = q0, num_steps=N, lr=0.2)
```

```
# Visualization (no need to change)
vis = visualize_robot()
arrow_w = 0.005
arrow_h = 0.005
arrow_l = 0.1

vis["arm/ee_frame_x"].set_object(Mesh(Box([arrow_l, arrow_w, arrow_w]), MeshLambert
vis["arm/ee_frame_y"].set_object(Mesh(Box([arrow_w, arrow_l, arrow_w]), MeshLambert
vis["arm/ee_frame_z"].set_object(Mesh(Box([arrow_w, arrow_w, arrow_l]), MeshLambert

vis["arm/des_frame_x"].set_object(Mesh(Box([arrow_l, arrow_w, arrow_w]), MeshLamber
vis["arm/des_frame_y"].set_object(Mesh(Box([arrow_w, arrow_l, arrow_w]), MeshLamber
vis["arm/des_frame_z"].set_object(Mesh(Box([arrow_w, arrow_w, arrow_l]), MeshLamber

vis["arm/des_frame_x"].set_transform(T_des)
vis["arm/des_frame_y"].set_transform(T_des)
vis["arm/des_frame_z"].set_transform(T_des)
```



```
# Visualization (no need to change)
for i in range(len(q_hist)):
  Tee = ForwardKinematics(vis, q=q_hist[i])

  vis["arm/ee_frame_x"].set_transform(Tee)
  vis["arm/ee_frame_y"].set_transform(Tee)
  vis["arm/ee_frame_z"].set_transform(Tee)
  time.sleep(0.2)
```

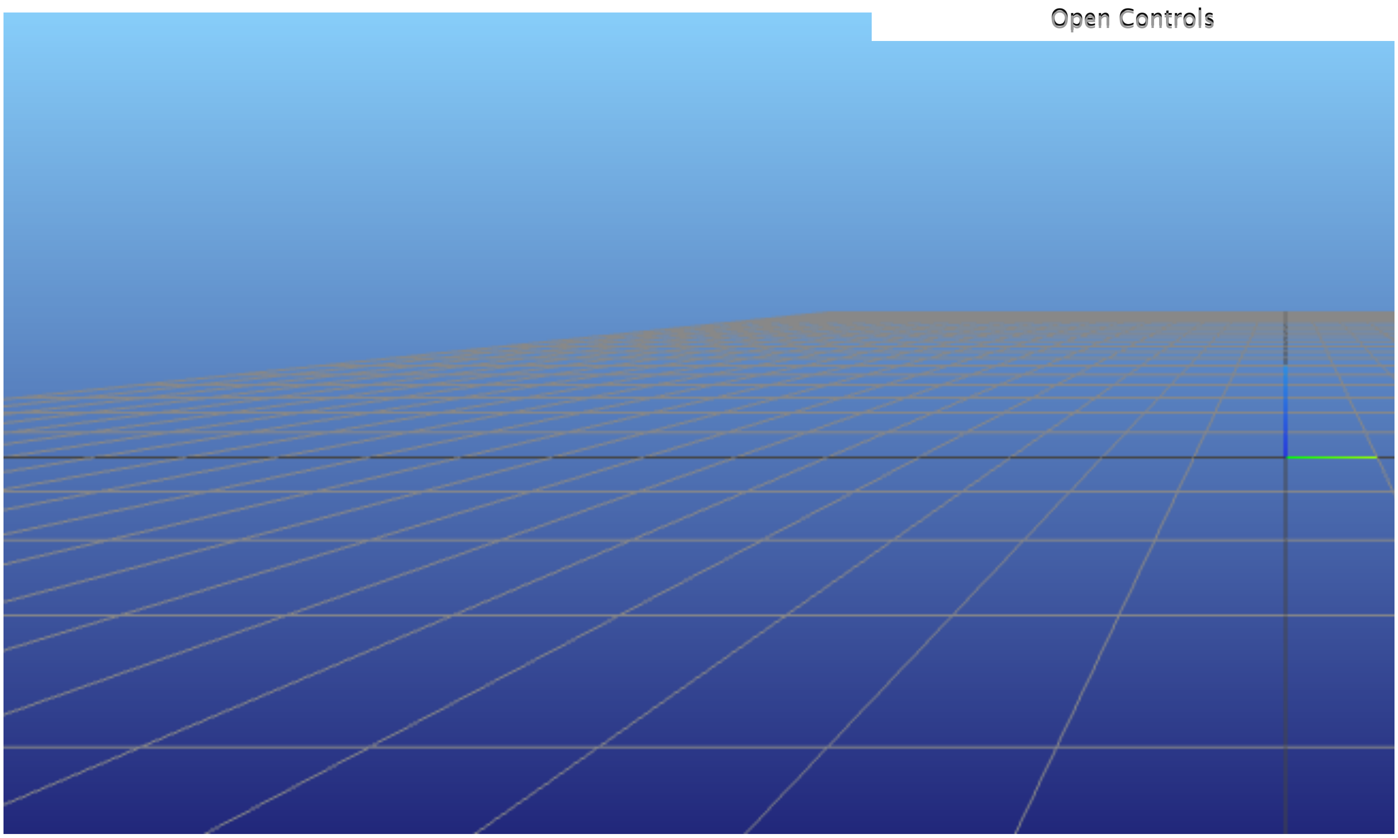
```
# Use this code to check your answer (no need to change)
print('T_ee: \n', Tee, '\n T_des: \n', T_des)
```

```
# Use this code to check your answer (no need to change)

error_hist_np = np.array(error_hist)
plt.plot(error_hist_np[:, 0], label='rx')
plt.plot(error_hist_np[:, 1], label='ry')
plt.plot(error_hist_np[:, 2], label='rz')
plt.plot(error_hist_np[:, 3], label='x')
plt.plot(error_hist_np[:, 4], label='y')
plt.plot(error_hist_np[:, 5], label='z')
plt.xlabel('k')
plt.ylabel('Error (des - act)')
plt.legend()
plt.show()
```

Q.3(b) [15 pts]. Complete inverse kinematics function using body Jacobian (Define the error implicitly).

```
def newton_raphson_body(T_des, q = np.zeros((6,1)), num_steps=50, lr=0.2):
  q_hist = []
  error_hist = []
  for step in range(num_steps):
      J = Jacobian(q, 'body')

      T = make_SE3_from_EE(q)
      Tee = T[0]

      R_ee  = Tee[:3, :3]
      p_ee = Tee[:3, 3]
      R_des = T_des[:3, :3]
      p_des = T_des[:3,  3]

      err = np.zeros((6,1));
      # Fill your code: Complete error computation
      err[:3, 0] = SO3toso3vec(np.transpose(R_ee) @ R_des)
      err[3:6, 0] = np.transpose(R_ee) @ (T_des[:3, 3] - T[6][:3, 3])

      # Newton Rapson (no need to change unless you want to implement yourself)


      q = q + lr*np.linalg.pinv(J)@err;
      if np.linalg.norm(err) < 0.0001:
          break
      q_hist.append(q)
      error_hist.append(err)
  return q_hist, error_hist


# Test inverse kinematics (Body Jacobian)

# Fill your code:
q0 = np.array([0,np.pi,0,0,0,0]).reshape(6,1)
T_des = np.array([[-1, 0, 0, -0.555],[0, 1, 0, 0],[0, 0, -1, -0.15],[0, 0, 0, 1]])

N = 50
[q_hist_imp, error_hist_imp] = newton_raphson_body(T_des, q = q0, num_steps=N, lr=0
```

```
vis = visualize_robot()
arrow_w = 0.005
arrow_h = 0.005
arrow_l = 0.1

vis["arm/ee_frame_x"].set_object(Mesh(Box([arrow_l, arrow_w, arrow_w]), MeshLambert
vis["arm/ee_frame_y"].set_object(Mesh(Box([arrow_w, arrow_l, arrow_w]), MeshLambert
vis["arm/ee_frame_z"].set_object(Mesh(Box([arrow_w, arrow_w, arrow_l]), MeshLambert

vis["arm/des_frame_x"].set_object(Mesh(Box([arrow_l, arrow_w, arrow_w]), MeshLambe
vis["arm/des_frame_y"].set_object(Mesh(Box([arrow_w, arrow_l, arrow_w]), MeshLambe
vis["arm/des_frame_z"].set_object(Mesh(Box([arrow_w, arrow_w, arrow_l]), MeshLambe

vis["arm/des_frame_x"].set_transform(T_des)
vis["arm/des_frame_y"].set_transform(T_des)
vis["arm/des_frame_z"].set_transform(T_des)
```



```
for i in range(len(q_hist)):
  Tee_imp = ForwardKinematics(vis, q=q_hist_imp[i])
  vis["arm/ee_frame_x"].set_transform(Tee_imp)
  vis["arm/ee_frame_y"].set_transform(Tee_imp)
  vis["arm/ee_frame_z"].set_transform(Tee_imp)
  time.sleep(0.2)



print('T_ee: \n', Tee_imp, '\n T_des: \n', T_des)
```

```
T_ee:
 [[ 1.0000e+00  1.5900e-05  9.3912e-06  2.0001e-01]
 [ 9.3903e-06  5.5210e-05 -1.0000e+00  3.0995e-01]
 [-1.5901e-05  1.0000e+00  5.5210e-05  2.0001e-01]
 [ 0.0000e+00  0.0000e+00  0.0000e+00  1.0000e+00]]
 T_des:
 [[ 1.0000e+00  0.0000e+00  0.0000e+00  2.0000e-01]
 [ 0.0000e+00  6.1232e-17 -1.0000e+00  3.1000e-01]
 [ 0.0000e+00  1.0000e+00  6.1232e-17  2.0000e-01]
 [ 0.0000e+00  0.0000e+00  0.0000e+00  1.0000e+00]]
```

```
error_hist_imp_np = np.array(error_hist_imp)
plt.plot(error_hist_imp_np[:, 0], label='rx')
plt.plot(error_hist_imp_np[:, 1], label='ry')
plt.plot(error_hist_imp_np[:, 2], label='rz')
plt.plot(error_hist_imp_np[:, 3], label='x')
plt.plot(error_hist_imp_np[:, 4], label='y')
plt.plot(error_hist_imp_np[:, 5], label='z')
plt.xlabel('k')
plt.ylabel('Error (des - act)')
plt.legend()
plt.show()
```