zs2489 Zhou Shen
yx2725 YaoYao Xu

CAofR HW5

Problem 1: Forward kinematics

(a) We set the coordinate Frame as the figure below shows,

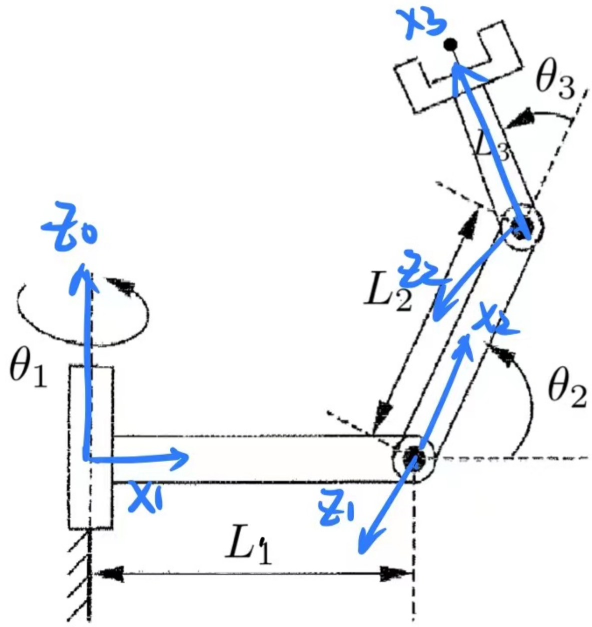

Figure 1: title

(b) The DH table is shown in figure below,

joint	a_i	α_i	d_i	θ_i
1	L_1	90	0	θ_1
2	L_2	0	0	θ_2
3	L_3	0	0	θ_3

Figure 2: DH Table

(c) From Frame 0 to Frame 1, it firstly rotates about Z axis by θ_1, and then rotates about the X axis by 90 degrees. The transformation matrix is shown in below,

$$A_1 = \begin{bmatrix} c_1 & 0 & s_1 & L_1c_1 \\ s_1 & 0 & -c_1 & L_1s_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_2 = \begin{bmatrix} c_2 & -s_2 & 0 & L_2c_2 \\ s_2 & c_2 & 0 & L_2s_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} c_3 & -s_3 & 0 & L_3c_3 \\ s_3 & c_3 & 0 & L_3s_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^0 = A_1A_2A_3 = \begin{bmatrix} c_1c_2c_3 - c_1s_2s_3 & -c_1c_2s_3 - c_1c_3s_2 & s_1 & L_1c_1 + L_2c_1c_2 + L_3c_1c_2c_3 - L_3c_1s_2s_3 \\ c_2c_3s_1 - s_1s_2s_3 & -c_2s_1s_3 - c_3s_1s_2 & -c_1 & L_1s_1 + L_2c_2s_1 - L_3s_1s_2s_3 + L_3c_2c_3s_1 \\ c_2s_3 + c_3s_2 & c_2c_3 - s_2s_3 & 0 & L_2s_2 + L_3c_2s_3 + L_3c_3s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(d) The workspace of RRR arm is set of points that can be reached by its end-effector. The workspace of RR arm with same arm length is a circle. If we add L_1 then the workspace is a torus. Finally, since $L_1 = L2 = L3$, then, $R < r$, so it is a self-intersecting spindle torus. The figure as follows:

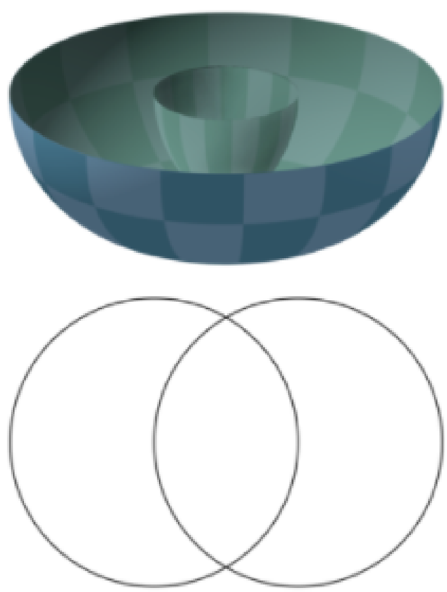

Figure 3: Self-intersecting spindle torus

Problem 2: Analytical Inverse Kinematics

(a) θ_1 is:

$$\begin{aligned}\frac{y_d}{x_d} &= \frac{s_1(L_1 + L_2c_2 + L_3c_2c_3 - L_3s_2s_3)}{c_1(L_1 + L_2c_2 + L_3c_2c_3 - L_3s_2s_3)}\\ &= \frac{s_1}{c_1}\\ &= tan_1\\ \theta_1 &= arctan(\frac{y_d}{x_d})\end{aligned} \tag{1}$$

The second solution is $\theta_1 = arctan(\frac{y_d}{x_d}) + \pi$

(b) After simplification, we get:

$$\begin{aligned}\frac{x_d^2}{c_1^2} + z_d^2 &= L_1^2 + L_2^2 + L_3^2 + 2L_1(\frac{x_d}{c_1} - L_1) + 2L_2L_3c_3\\ \theta_3 &= acos\frac{\frac{x_d^2}{c_1^2} + z_d^2 - 2L_1\frac{x_d}{c_1} + L_1^2 - L_2^2 - L_3^2}{2L_2L_3}\end{aligned} \tag{2}$$

There are two solutions for θ_3, 'elbow up' and 'elbow down'.

(c) Refer to the articulated configuration. We get:

$$\theta_2 = atan2(z_d, \sqrt{x^2 + y^2} - L_1) - atan2(L_3s_3, L_2 + L_3c_3) \tag{3}$$

Problem 3: Velocity Kinematics

(a) J_v could be derived as:

$$J_v = \begin{bmatrix} \frac{\partial X}{\partial q1} & \frac{\partial X}{\partial q2} & \frac{\partial X}{\partial q3}\\ \frac{\partial Y}{\partial q1} & \frac{\partial Y}{\partial q2} & \frac{\partial Y}{\partial q3}\\ \frac{\partial Z}{\partial q1} & \frac{\partial Z}{\partial q2} & \frac{\partial Z}{\partial q3} \end{bmatrix}$$

$$J_v11 = L_3s_1s_2s_3 - L_2c_2s_1 - L_3c_2c_3s_1 - L_1s_1$$

$$J_v12 = L_3s_1s_2s_3 - L_2c_2s_1 - L_3c_2c_3s_1 - L_1s_1$$

$$J_v13 = -L_3c_1c_2s_3 - L_3c_1c_3s_2$$

$$J_v21 = L_1c_1 + L_2c_1c_2 + L_3$$

$$J_v22 = -L_2s_1s_2 + L_3s_1s_{23}$$

$$J_v23 = -L_3s_1s_{23}$$

$$J_v31 = 0$$

$$J_v32 = L_2c_2 + L_3c_{23}$$

$$J_v33 = L_3c_{23}$$

(b) J_ω can be computed as,

$$J_\omega = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & -1 \\ 1 & 0 & 0 \end{bmatrix}$$

(c) The rank of J_ω is 2,

$$\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & -1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix}$$

$$\omega_x = 0$$

$$\omega_y = -\dot{q}_2 - \dot{q}_3$$

$$\omega_z = \dot{q}_1$$

From equations above, we can see that the ω_y are coupled, and ω_z can be independently controlled.

(d) The determinate of J_v is

$$det(J_v) = -L_2L_3s_3(L_1 + L_2c_2 + L_3c_{23})$$

The determinate equals 0 when the RRR manipulator is in a singular configuration, where,

$$sin(q_3) = 0, \quad q_3 = 0 \quad or \quad q_3 = \pi$$

In this case, Link 3 is fully extended or fully retracted.
or

$$L_1 + L_2c_2 + L_3c_{23} = 0$$

In this case, the X position of the end-effector is 0, which means the end effector intersects the z_0 axis of the base frame.

Problem 4: Inverse Velocity Kinematics

(a) ω_x is the single component and $\dot{y}$ with ω_z are the pair components that yield no solution in IK problem. The Jacobian matrix loses rank in these cases.

(b) when ξ_1, which is $[\dot{x}\ \dot{y}\ \dot{z}] = [1\ 1\ 1]$,

$$\dot{q} = J^{-1}\xi_1 = \begin{bmatrix} 0,3848 \\ 0.4998 \\ -2.4311 \end{bmatrix}$$

when ξ_2, which is $[\dot{x}\ \dot{y}\ \dot{\omega}_y] = [1\ 1\ 1]$, the manipulator is over-constrained since the Jacobian matrix have more rows than columns, so we compute the left pseudoinverse.

$$J_L^+ = (J^TJ)^{-1}J^T = \begin{bmatrix} -0.3497 & 0.3944 & 0.2357 \\ 0.1139 & -0.8026 & -0.9022 \end{bmatrix}$$

$$\dot{q} = J_L^+\xi_2 = \begin{bmatrix} 0.2803 \\ -1.5909 \end{bmatrix}$$

In order to have minimum joint velocity norm solution, we set $\dot{q}_1$ to be 0, so

$$\dot{q} = \begin{bmatrix} 0 \\ 0.2803 \\ -1.5909 \end{bmatrix}$$

(c) when ξ_3, which is $[\dot{x}\ \dot{y}] = [1\ 1]$, the manipulator is under-constrained since the Jacobian matrix have fewer rows than columns, so we compute the weighted right pseudoinverse.

$$W = eye(3) \quad J_R^+ = W^{-1}J^T(JW^{-1}J^T)^{-1} = \begin{bmatrix} 0 & 0.3648 \\ -0.3285 & 0 \\ -0.1176 & 0 \end{bmatrix}$$

$$\dot{q} = J_R^+\xi_3 = \begin{bmatrix} 0.3648 \\ -0.3285 \\ -0.1176 \end{bmatrix}$$

$$W = diag(10, 5, 1) \quad \dot{q} = J_R^+\xi_3 = \begin{bmatrix} 0.3648 \\ -0.2259 \\ -0.4044 \end{bmatrix}$$

Compared the two results, $\dot{q}_1$ remains unchanged if a non-identity weighting matrix is used.

(d) We are trying to find $\dot{q}$ as close as possible to $\dot{q}_0$ by minimize $g(\dot{q})$ with $W = eye(3)$,

$$\dot{q} = J_R^+\xi_3 + (I - J_R^+J)\dot{q}_0 = \begin{bmatrix} 0.3648 \\ -0.4304 \\ 0.1669 \end{bmatrix}$$

Problem 5: Manipulability

(a) Take $\dot{q}$ with unit norm, the set of unit joint velocities transforms into a set of work-space velocities characterized by an ellipsoid,

$$\parallel \dot{q} \parallel = \xi^T (JJ^T)^{-1} \xi$$

We computed the eigenvalues and their corresponding eigenvectors in Matlab, which are

$$\lambda = \begin{bmatrix} 0.2328 \\ 7.5131 \\ 8.5957 \end{bmatrix} \quad \sigma_{max} = \sqrt{8.5957} = 2.9318$$

it's eigenvector is $[0.9768\ 0\ \ -0.2141]^T$, the largest possible end effector linear velocity vector lies on the longest principal axis of the manipulability ellipsoid.

(b) We use SVD decomposition in Matlab to find the U, S V

$$S = \begin{bmatrix} 2.9318 & 0 & 0 \\ 0 & 2.7410 & 0 \\ 0 & 0 & 0.4825 \end{bmatrix} \quad V = \begin{bmatrix} 0 & -1 & 0 \\ -0.9530 & 0 & 0.3029 \\ -0.3029 & 0 & -0.9530 \end{bmatrix} \quad U = \begin{bmatrix} 0.9768 & 0 & 0.2141 \\ 0 & -1 & 0 \\ -2.141 & 0 & 0.9768 \end{bmatrix}$$

$$\xi = magnitude(V) eigenvector = \begin{bmatrix} 2.8638 \\ 0 \\ -0.6277 \end{bmatrix} \quad \dot{q} = J^{-1}\xi = \begin{bmatrix} 0 \\ -0.9530 \\ -0.3029 \end{bmatrix}$$

This joint velocity appear in the first column of the orthonormal right singular vectors of J_v.

(c) From the orthonormal left singular vector matrix(U), the first column which corresponds to our largest linear velocity is orthogonal to the third column of U which corresponds to eigenvalue 0.4825, and its eigenvector is $[0.2141\ 0\ 0.9768]^T$.

$$\xi = magnitude(V) eigenvector \quad \dot{q} = J^{-1}\xi = \begin{bmatrix} 0 \\ 0.3029 \\ -0.9530 \end{bmatrix}$$

Joint 2 and 3 are actuated, the second link rotates in counterclockwise direction and the third link rotates in clockwise direction.

Problem 6: Iterative Inverse Kinematics

The following two tables show the results from two methods,

λ	Norm of the Error	Final solution $(\theta_1, \theta_2, \theta_2)$ in radian	Converged Iteration
0.001	0.44	1.096 0.895 0.292	16
0.1	0.44	1.096 0.830 0.406	16
1	0.44	1.096 0.893 0.295	14
10	0.44	1.089 0.810 0.437	88

Figure 4: Results for Newton's Method

Learning rate α	Norm of the Error	Final solution $(\theta_1, \theta_2, \theta_2)$ in radian	Converged Iteration
0.001	0.44	1.088 0.808 0.439	835
0.01	0.44	1.096 0.817 0.406	120
0.1	0.44	1.096 0.830 0.427	18

Figure 5: Results for Gradient Descend Method

Comparing these two methods, it shows that the Newton's Method has slightly larger rate of convergence than Gradient Descend Method, specially when the damping constant is 1, it takes 14 iterations to converge.
The Gradient Descend Method has the smoother trajectory, since it make small changes to the joint configuration in each iteration but the Newton's Method have large jumps in joint configuration.
The Newton's Method is less stable than the Gradient Descend Method because it uses J^{-1}, and J may not have an inverse, so we computed the pseudoinverse and added the damping to solve this problem, which makes the singular values of J^{+} to be close to σ_i^{-1} without blowing up

The tables above show that the Gradient Descend Method is more sensitive to the parameters changes. When we decrease the learning rate, the number of iteration to converge will increase greatly, but when we increase the learning rate, the number of iterations to converge will decrease not that much. And the Newton's Method will not give very different results when we vary the damping constants in the same pattern.

Problem 7: Learning Inverse Kinematics

The following four figures show the model accuracy, model loss, designed end effector trajectory and predicted end effector trajectory.

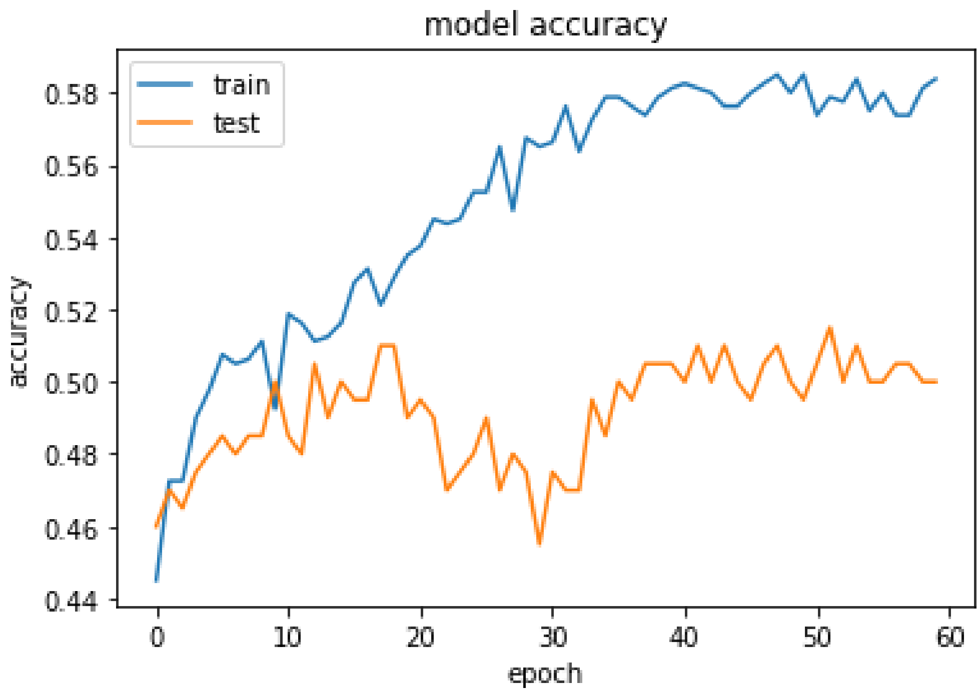

Figure 6: Model Accuracy

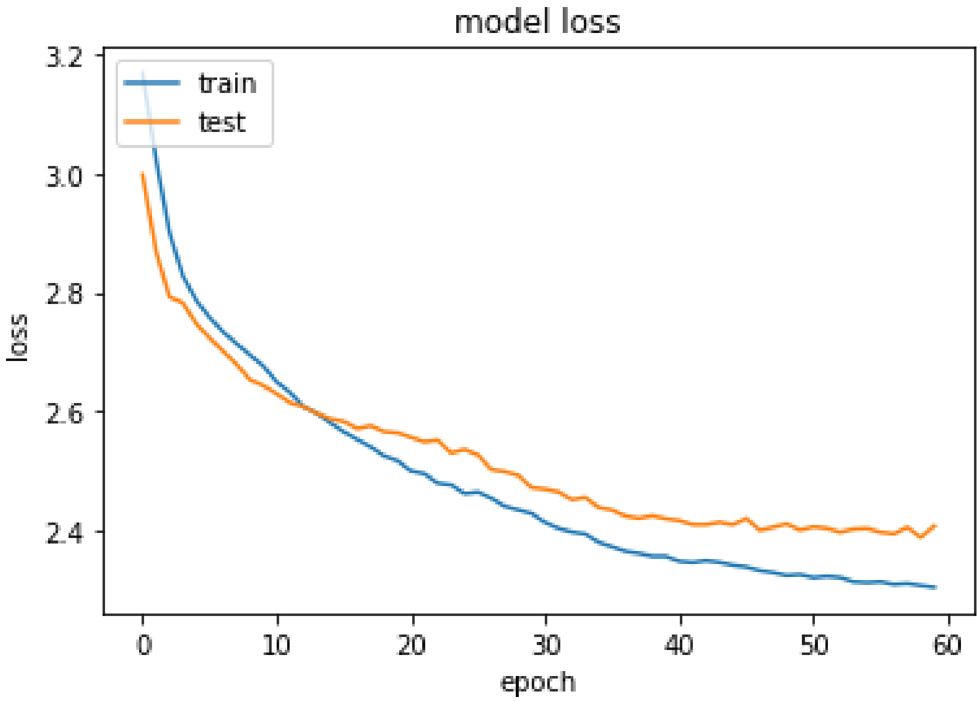

Figure 7: Model Loss

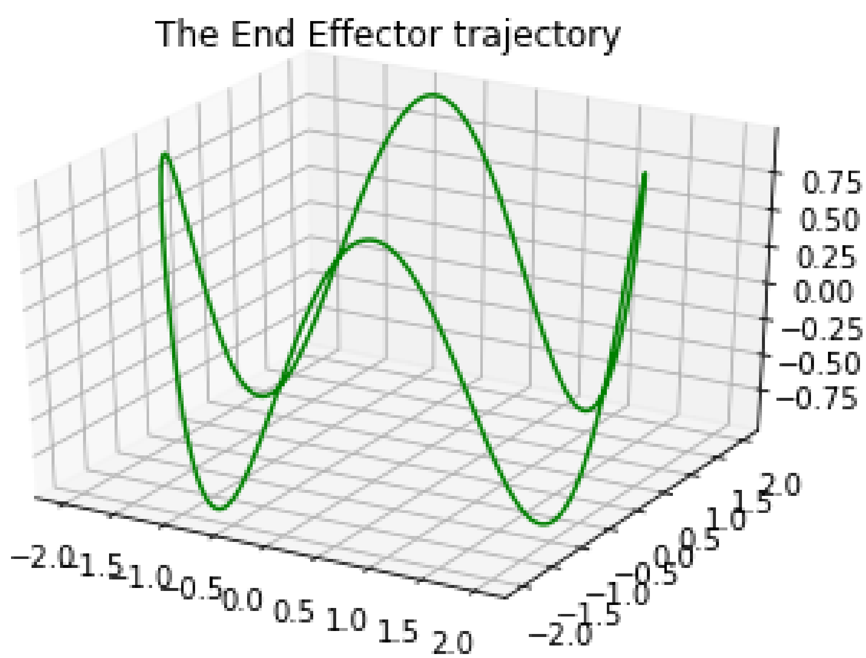

Figure 8: Designed End Effector Trajectory

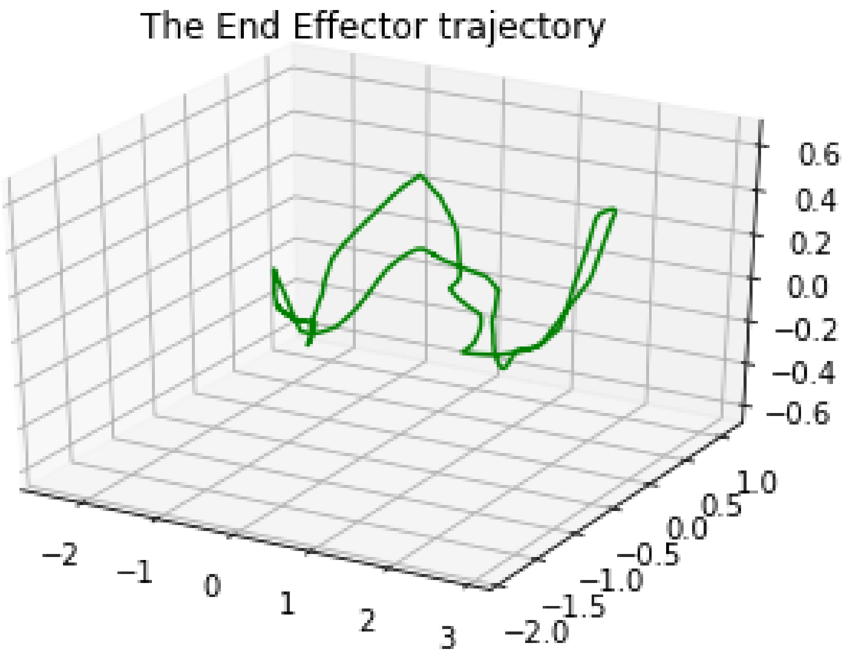

Figure 9: Predicted End Effector Trajectory

Appendix:

Question 6 Code

```
from numpy import *;
import numpy as np;
import math;
T_d=array([[0.078,-0.494,0.866,1],[0.135,-0.855,-0.500,2],[0.988,0.156,0,2],[0,0,0,1]])
Od=T_d[0:3,3]
Rx_d=T_d[0:3,0]
Ry_d=T_d[0:3,1]
Rz_d=T_d[0:3,2]
def error(Rx,Ry,Rz,On):
    delta_O= (Od-On.ravel()).reshape(-1,1)
    delta_theta= 0.5*(np.cross(Rx.ravel(),Rx_d.ravel()) +
        np.cross(Ry.ravel(),Ry_d.ravel()) +
        np.cross(Rz.ravel(),Rz_d.ravel())).reshape(-1,1)
    error=np.vstack((delta_O,delta_theta))
    return error
    def Jacobian(q1,q2,q3, Lambda = 1):
  J11=sin(q1)*sin(q2)*sin(q3) - cos(q2)*sin(q1) - sin(q1) -
      cos(q2)*cos(q3)*sin(q1)
  J12=-cos(q1)*sin(q2)-cos(q1)*cos(q2)*sin(q3)-cos(q1)*cos(q3)*sin(q2)
  J13=-cos(q1)*cos(q2)*sin(q3)-cos(q1)*cos(q3)*sin(q2)
  J21=cos(q1) + cos(q1)*cos(q2) + cos(q1)*cos(q2)*cos(q3) -
      cos(q1)*sin(q2)*sin(q3)
  J22=- sin(q1)*sin(q2) - cos(q2)*sin(q1)*sin(q3) - cos(q3)*sin(q1)*sin(q2)
  J23=- cos(q2)*sin(q1)*sin(q3) - cos(q3)*sin(q1)*sin(q2)
  J31=0
  J32=cos(q2) + cos(q2)*cos(q3) - sin(q2)*sin(q3)
  J33=cos(q2)*cos(q3) - sin(q2)*sin(q3)
  J=[[J11, J12, J13],[J21, J22, J23],[J31, J32, J33],[0,0,0],[0,-1,-1],[1,0,0]]
  J = np.array(J)

  return J
  ###Gradient descent
DELTA_ERROR = 1000
PREV_ERROR = 1000
q1, q2, q3 = 0,0,0
iteration = 0
while DELTA_ERROR > 0.00001:
  iteration += 1
  Rx, Ry, Rz, On = Rotation(q1, q2, q3)
  NEW_ERROR=error(Rx,Ry,Rz,On)
  qk= np.array([q1, q2, q3]) + 0.01*( (Jacobian(q1, q2,
```

```
        q3)).T.dot(NEW_ERROR)).ravel()
  q1=qk[0]
  q2=qk[1]
  q3=qk[2]
  DELTA_ERROR = abs(np.linalg.norm(NEW_ERROR) - PREV_ERROR)
  print(q1,q2,q3)
  PREV_ERROR = np.linalg.norm(NEW_ERROR)
  #break
  print("At the {} iteration, the error is {}".format(iteration,
      np.linalg.norm(NEW_ERROR)))
  print('------------------------------------')
  ## Newton Method
Lambda= 1
DELTA_ERROR = 1000
PREV_ERROR = 1000
q1, q2, q3 = 0,0,0
iteration = 0
while DELTA_ERROR>0.0001:
  iteration += 1
  Rx, Ry, Rz, On = Rotation(q1, q2, q3)
  NEW_ERROR=error(Rx,Ry,Rz,On)
  J = Jacobian(q1, q2, q3)
  dls_pinv = np.linalg.inv(J.T @ J + Lambda**2 * np.eye(3)) @ J.T

  qk= np.array([q1, q2, q3]) +(dls_pinv.dot(NEW_ERROR)).ravel()
  # print(J_inverse(q1, q2, q3).dot(ERROR))
  # print(qk.shape)
  q1=qk[0]
  q2=qk[1]
  q3=qk[2]
  DELTA_ERROR = abs(np.linalg.norm(NEW_ERROR) - PREV_ERROR)
  print(q1,q2,q3)
  PREV_ERROR = np.linalg.norm(NEW_ERROR)
  #break
  print("At the {} iteration, the error is {}".format(iteration,
      np.linalg.norm(NEW_ERROR)))
  print('------------------------------------')
```

Question 7 Code

```
import random
Input = []
Output = []
for _ in range(1000):
  q1 = random.uniform(-np.pi, np.pi)
```

```
  q2 = random.uniform(-np.pi, np.pi)
  q3 = random.uniform(-np.pi, np.pi)
  X=cos(q1)+cos(q1)*cos(q2)+cos(q1)*cos(q2)*cos(q3)-cos(q1)*sin(q2)*sin(q3)
  Y=sin(q1)+cos(q2)*sin(q1)-sin(q1)*sin(q2)*sin(q3)+cos(q2)*cos(q3)*sin(q1)
  Z=sin(q2)+cos(q2)*sin(q3)+cos(q3)*sin(q2)

  Output.append([q1, q2, q3])
  Input.append([X, Y, Z])

Input = np.array(Input)
Output = np.array(Output)
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split

model = tf.keras.Sequential([
    tf.keras.layers.Dense(3, activation="relu"),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(3)
])



model.compile(loss='MSE', optimizer='adam', metrics=['accuracy'])
# Fit the model
history = model.fit(Input, Output, validation_split=0.2, epochs=60,
    batch_size=32, verbose=0)
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
```

```
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
K = 500
traj = np.zeros((K,3))
traj[:,0] = 2*np.cos(np.linspace(0,2*np.pi,num=K))
traj[:,1] = 2*np.sin(np.linspace(0,2*np.pi,num=K))
traj[:,2] = np.sin(np.linspace(0,8*np.pi,num=K))

prediction = model.predict(traj)

fig = plt.figure()
ax = plt.axes(projection ='3d')
z = np.sin(np.linspace(0,8*np.pi,num=K))
x = 2*np.cos(np.linspace(0,2*np.pi,num=K))
y = 2*np.sin(np.linspace(0,2*np.pi,num=K))

# plotting
ax.plot3D(x, y, z, 'green')
ax.set_title('The End Effector trajectory')
plt.show()

fig = plt.figure()
ax = plt.axes(projection ='3d')
z = prediction[:,2]
x = prediction[:,0]
y = prediction[:,1]

# plotting
ax.plot3D(x, y, z, 'green')
ax.set_title('The End Effector trajectory')
plt.show()
```