ME 639

INTRODUCTION TO ROBOTICS

Assignment 5

Submitted by:

Rhitosparsha Baishya
23310039
PhD (Mechanical Engineering)

CONTENTS

Question 1

Source Code: (Link to GitHub)

```
import numpy as np

print("Inverse kinematics of Stanford manipulator :\n")

# Take input for link lengths
l1 = float(input("Enter length of Link 1 (m): "))
l2 = float(input("Enter length of Link 2 (m): "))
l3 = float(input("Enter length of Link 3 (m): "))

# Take input for end-effector coordinates
xc = float(input("Enter x-coordinate of end-effector (m): "))
yc = float(input("Enter y-coordinate of end-effector (m): "))
zc = float(input("Enter z-coordinate of end-effector (m): "))

# Compute inverse kinematics solutions
r = np.sqrt(xc**2 + yc**2)
s = zc - l1
theta1_a = np.arctan2(xc,yc)
theta1_b = theta1_a + np.pi
theta2 = np.arctan2(r,s) + np.pi/2
d3 = np.sqrt(r**2 + s**2)

# Output the computed values
if d3 > l3:
    print("Point lies outside the workspace of the robot!")
else:
    print("\nInverse kinematics solutions :\n")
    print("Angle of link 1 ", np.rad2deg(theta1_a), " deg, ", np.rad2deg(theta1_b), "
deg")
    print("Angle of link 2 ", np.rad2deg(theta2), " deg")
    print("Link extension : ", d3, " m")
```

Output:

```
Inverse kinematics of Stanford manipulator :

Enter length of Link 1 (m): 0.2
Enter length of Link 2 (m): 0.15
Enter length of Link 3 (m): 0.3
Enter x-coordinate of end-effector (m): 0.15
Enter y-coordinate of end-effector (m): 0.1
Enter z-coordinate of end-effector (m): 0.25

Inverse kinematics solutions :

Angle of link 1  56.309932474020215  deg,  236.3099324740202  deg
Angle of link 2  164.498640433063  deg
Link extension :  0.18708286933869708  m
```

Question 2

Source Code: (Link to GitHub)

```
import numpy as np

def calculate_joint_velocities(jacobian, end_effector_linear_velocities,
end_effector_angular_velocities):
    try:
        end_effector_velocities = np.concatenate((end_effector_linear_velocities,
end_effector_angular_velocities))
        jacobian_inv = np.linalg.pinv(jacobian)
        joint_velocities = np.dot(jacobian_inv, end_effector_velocities)
        return joint_velocities[:3], joint_velocities[3:]
    except np.linalg.LinAlgError:
        return None

num_links = int(input("Enter the number of links: "))

jacobian = np.empty((6, num_links))

print("\nEnter the elements of the Jacobian matrix (row by row):")
for i in range(6):
    for j in range(num_links):
        jacobian[i][j] = float(input(f"Jacobian[{i}][{j}]: "))

end_effector_linear_velocities = np.array([float(v) for v in input("\nEnter end-
effector linear velocities (comma-separated) (m/s): ").split(',')])
end_effector_angular_velocities = np.array([float(v) for v in input("Enter end-
effector angular velocities (comma-separated) (rad/s): ").split(',')])

# Calculate joint velocities
joint_linear_velocities, joint_angular_velocities =
calculate_joint_velocities(jacobian, end_effector_linear_velocities,
end_effector_angular_velocities)

if joint_linear_velocities is not None:
    print("\nJoint Linear Velocities:", joint_linear_velocities)
    print("Joint Angular Velocities:", joint_angular_velocities)
else:
    print("\nJacobian is singular. Cannot calculate joint velocities.")
```

Output:

```
Enter the number of links: 2

Enter the elements of the Jacobian matrix (row by row):
Jacobian[0][0]: -1.9869
Jacobian[0][1]: -0.9869
Jacobian[1][0]: 1.9909
Jacobian[1][1]: 0.2588
Jacobian[2][0]: 0
Jacobian[2][1]: 0
Jacobian[3][0]: 0
Jacobian[3][1]: 0
Jacobian[4][0]: 0
Jacobian[4][1]: 0
Jacobian[5][0]: 1
Jacobian[5][1]: 1

Enter end-effector linear velocities (comma-separated) (m/s): 0.2, 0.2, 0
Enter end-effector angular velocities (comma-separated) (rad/s): 0, 0, 0

Joint Linear Velocities: [ 0.08318945 -0.21303629]
Joint Angular Velocities: []
```

Question 5

Source Code: (Link to GitHub)

```
import numpy as np

print("Inverse kinematics of Spherical Wrist manipulator :\n")

# Take input for link lengths
q1 = float(input("Enter angle of Joint 1 (deg): "))
q2 = float(input("Enter angle of Joint 2 (deg): "))
q3 = float(input("Enter angle of Joint 3 (deg): "))

c1 = np.cos(q1)
c23 = np.cos(q2 + q3)
s1 = np.sin(q1)
s23 = np.sin(q2 + q3)

# Define the rotation matrix of the current end-effector orientation
R_current = np.array([[-c1 * c23, -c1 * s23, s1],
                      [s1 * c23, -s1 * s23, -c1],
                      [s23, c23, 0]])

# Take input of desired end-effector orientation
print("\nEnter the elements of the desired end-effector orientation matrix (row by
row):")
R_desired = np.empty((3,3))
for i in range(3):
    for j in range(3):
        R_desired[i][j] = float(input(f"R_desired[{i}][{j}]: "))

# Calculate the transformation matrix that transforms from the current to the desired
orientation
R_d2c = np.dot(R_current.T, R_desired)

# Extract the z-y-z Euler angles from the rotation matrix
phi = np.arctan2(R_d2c[1, 2], R_d2c[0, 2])
theta = np.arccos(R_d2c[2, 2])
psi = np.arctan2(R_d2c[2, 1], -R_d2c[2, 0])

# Print the results
print("\nDesired Rotation Matrix:")
print(R_desired)
print("\nCurrent Rotation Matrix:")
print(R_current)
print("\nCalculated Z-Y-Z Euler Angles:")
print("phi = ", np.degrees(phi), " deg")
```

```
print("theta = ", np.degrees(theta), " deg")
print("psi = ", np.degrees(psi), " deg")
```

Output:

```
Inverse kinematics of Spherical Wrist manipulator :

Enter angle of Joint 1 (deg): 15
Enter angle of Joint 2 (deg): 20
Enter angle of Joint 3 (deg): 30

Enter the elements of the desired end-effector orientation matrix (row by row):
R_desired[0][0]: 0.866
R_desired[0][1]: -0.5
R_desired[0][2]: 0
R_desired[1][0]: 0.5
R_desired[1][1]: 0.866
R_desired[1][2]: 0
R_desired[2][0]: 0
R_desired[2][1]: 0
R_desired[2][2]: 1

Desired Rotation Matrix:
[[ 0.866 -0.5    0.   ]
 [ 0.5    0.866  0.   ]
 [ 0.     0.     1.   ]]

Current Rotation Matrix:
[[ 0.73307303 -0.199323    0.65028784]
 [ 0.62750567  0.17061918  0.75968791]
 [-0.26237485  0.96496603  0.        ]]

Calculated Z-Y-Z Euler Angles:
phi =  105.21102434588396  deg
theta =  90.0  deg
psi =  160.56403508459258  deg
```

Question 6 (a)

Source Code: (Link to GitHub)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Define robot parameters
m1, m2, m3 = 1.0, 1.0, 1.0  # Masses of links
l2, l3 = 1.0, 1.0  # Link lengths
lc2, lc3 = 0.5, 0.5  # Center of mass distances
g = 9.81  # Gravitational acceleration

# Define initial conditions
initial_q = np.array([0.5, -1, 1])  # Initial joint angles
initial_q_dot = np.array([0.0, 0.0, 0.0])  # Initial joint velocities
initial_q_dot_motor = np.array([0.0, 0.0, 0.0])  # Initial motor velocities
initial_q_I_motor = np.array([0.0, 0.0, 0.0])  # Initial motor currents

# Desired joint angles and velocities
q_desired = np.array([2, 3, 1])
q_dot_desired = np.array([0.5, 0.1, 0.3])
tau_motor_desired = np.array([0.01, 0.02, 0.04]) # Small constant motor torques

# Motor dynamics parameters
Jm = np.array([1, 1.5, 0.5])  # Motor inertias
Bm = np.array([2, 3, 2.5])  # Motor damping coefficients
Rm = np.array([1, 1.5, 0.5])  # Motor resistances
Lm = np.array([0.01, 0.015, 0.005])  # Motor inductances
Kb = np.array([0.2, 0.25, 0.1])  # Motor back-emf constants
Kt = np.array([0.1, 0.3, 0.4])  # Motor torque constants

# Initialize state variables
q = initial_q.copy()
q_dot = initial_q_dot.copy()
q_dot_motor = initial_q_dot_motor.copy()
I_motor = initial_q_I_motor.copy()

# Simulation parameters
t_max = 20.0
num_steps = 10000
dt = t_max / num_steps
time_values = np.linspace(0, t_max, num_steps)

# Define the combined inertial and motor dynamics function
def dynamics(state, t):
```

```
    q1, q2, q3, q1_dot, q2_dot, q3_dot, q1_dot_motor, q2_dot_motor, q3_dot_motor, I1,
I2, I3 = state
    q = np.array([q1, q2, q3])
    q_dot = np.array([q1_dot, q2_dot, q3_dot])
    q_dot_motor = np.array([q1_dot_motor, q2_dot_motor, q3_dot_motor])
    q_ddot_motor = np.zeros(3)
    I_motor = np.array([I1, I2, I3])
    I_dot_motor = np.zeros(3)
    tau_motor = np.zeros(3)
    V_back = np.zeros(3)
    V_motor = np.zeros(3)

    C2 = np.cos(q2)
    S2 = np.sin(q2)
    C3 = np.cos(q3)
    S3 = np.sin(q3)
    C23 = np.cos(q2 + q3)
    S23 = np.sin(q2 + q3)

    M = np.array([
        [(1/2 * m1 * lc2**2 + 1/4 * m2 * l2**2 * C2**2 + 1/4 * m3 * l3**2 * C23**2 +
m3 * l2**2 * C2**2 + m2 * l2 * l3 * C2 * C23), 0, 0],
        [0, (1/4 * m2 * l2**2 + m3 * (l2**2 + lc3**2 + l2 * l3 * C3)), 0],
        [0, 0, (1/3 * m3 * l3**2)]
    ])

    C = np.array([
        [0, (-1/4 * m2 * l2**2 * S2 - m3 * l2**2 * S2 - m3 * l2 * l3 * S23 - m3 *
lc2**2 * S23), (-m3 * l2 * l3 * C2 * S23 - m3 * lc3**2 * S23)],
        [0, 0, -m2 * l2 * l3 * S3],
        [0, 0, 0]
    ])

    G = np.array([
        lc2 * m2 * g * C2 + l2 * m3 * g * C2 + lc3 * m2 * g * C23,
        0,
        m3 * g * lc3 * C23
    ])

    # Motor Dynamics
    V_back = Kb * q_dot_motor
    I_motor = tau_motor_desired / (Kt * Rm)
    q_ddot_motor = (tau_motor_desired - Bm * q_dot_motor - Kt * I_motor) / Jm
    q_dot_motor += q_ddot_motor * dt # Euler integration
    V_motor = I_motor * Rm
    I_dot_motor = (V_motor - Rm * I_motor - V_back) / Lm
    tau_motor = V_motor * Kt
```

```
    q_ddot = np.linalg.solve(M, tau_motor - np.dot(C, q_dot) - G)

    return [q_dot[0], q_dot[1], q_dot[2], q_ddot[0], q_ddot[1], q_ddot[2],
q_ddot_motor[0], q_ddot_motor[1], q_ddot_motor[2], I_dot_motor[0], I_dot_motor[1],
I_dot_motor[2]]


# Initial conditions for the ODE solver
state_0 = np.concatenate((q, q_dot, q_dot_motor, I_motor))

# Use odeint to integrate the ODE
solution = odeint(dynamics, state_0, time_values)

# Extract the joint angles from the solution
q1_values = solution[:, 0]
q2_values = solution[:, 1]
q3_values = solution[:, 2]

# Plot the joint angles over time
plt.figure()
plt.suptitle('Plot of Joint Angles (q) vs. Time (t) for 3-DOF Articulated Robot')

plt.subplot(311)
plt.plot(time_values, q1_values, label='q1')
plt.xlabel('t (s)')
plt.ylabel('q1 (rad)')
plt.grid()
plt.legend()

plt.subplot(312)
plt.plot(time_values, q2_values, label='q2')
plt.xlabel('t (s)')
plt.ylabel('q2 (rad)')
plt.grid()
plt.legend()

plt.subplot(313)
plt.plot(time_values, q3_values, label='q3')
plt.xlabel('t (s)')
plt.ylabel('q3 (rad)')
plt.grid()
plt.legend()

plt.show()
```

Result:

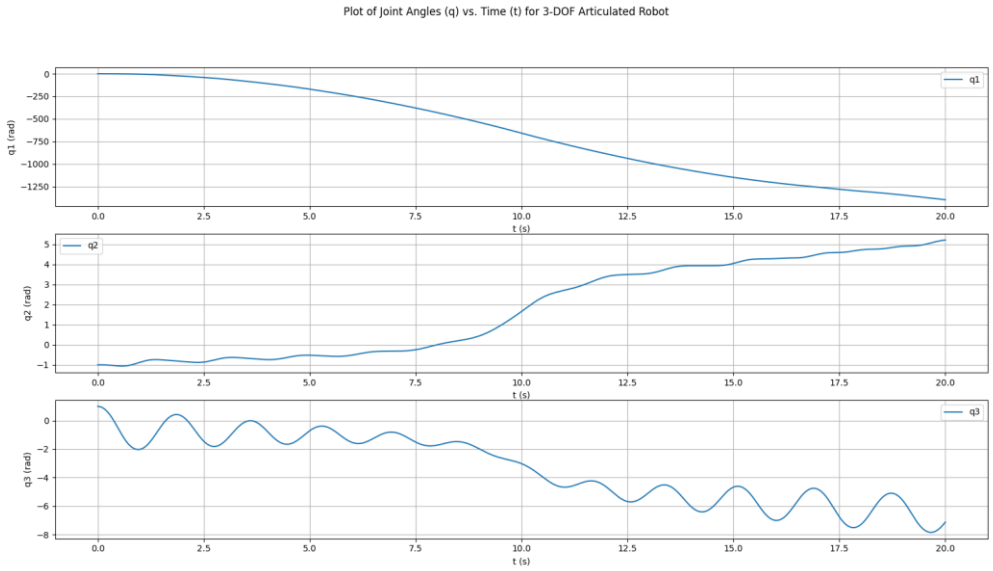

Fig.: q vs. t graph for 3-DOF Articulated Robot

Question 6 (b)

Source Code: (Link to GitHub)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

print("Simulation of 3-DOF Articulated Robot :\n")
print("Select Control Scheme :\n1. Simple PD Control\n2. PD Control with Gravity
Compensation\n3. PD Control with Feed-forward term\n4. Computed Torrque PD Control")
choice = input("\nPress a button from 1 - 4 : ")
print("You chose: ")
if choice == '1':
    print("Simple PD Control")
elif choice == '2':
    print("PD Control with Gravity Compensation")
elif choice == '3':
    print("PD Control with Feed-forward term")
elif choice == '4':
    print("PD Control with Computed Torque Control")
print("\nPlotting results...")

# Define robot parameters
m1, m2, m3 = 1.0, 1.0, 1.0  # Masses of links
l2, l3 = 1.0, 1.0  # Link lengths
lc2, lc3 = 0.5, 0.5  # Center of mass distances
g = 9.81  # Gravitational acceleration

# Define initial conditions
initial_q = np.array([0.5, -1, 1])  # Initial joint angles
initial_q_dot = np.array([0.0, 0.0, 0.0])  # Initial joint velocities
initial_q_dot_motor = np.array([0.0, 0.0, 0.0])  # Initial motor velocities
initial_I_motor = np.array([0.0, 0.0, 0.0])  # Initial motor currents

# Desired joint angles, velocities and accelerations
q_desired = np.array([2, 3, 1])
q_dot_desired = np.array([0.5, 0.1, 0.3])
q_ddot_desired = np.array([0.1, 0.05, 0.15])

# Motor dynamics parameters
Jm = np.array([1, 1.5, 0.5])  # Motor inertias
Bm = np.array([2, 3, 2.5])  # Motor damping coefficients
Rm = np.array([1, 1.5, 0.5])  # Motor resistances
Lm = np.array([0.01, 0.015, 0.005])  # Motor inductances
Kb = np.array([0.2, 0.25, 0.1])  # Motor back-emf constants
Kt = np.array([0.1, 0.3, 0.4])  # Motor torque constants
```

```
# Motor control parameters
Kp = np.array([10.0, 10.0, 10.0])  # Proportional gains
Kd = np.array([2, 2, 2])   # Derivative gain

# Initialize state variables
q = initial_q.copy()
q_dot = initial_q_dot.copy()
q_dot_motor = initial_q_dot_motor.copy()
I_motor = initial_I_motor.copy()

# Simulation parameters
t_max = 20.0
num_steps = 10000
dt = t_max / num_steps
time_values = np.linspace(0, t_max, num_steps)

# Define the combined inertial and motor dynamics function
def dynamics(state, t):
    q1, q2, q3, q1_dot, q2_dot, q3_dot, q1_dot_motor, q2_dot_motor, q3_dot_motor, I1,
I2, I3 = state
    q = np.array([q1, q2, q3])
    q_dot = np.array([q1_dot, q2_dot, q3_dot])
    q_dot_motor = np.array([q1_dot_motor, q2_dot_motor, q3_dot_motor])
    q_ddot_motor = np.zeros(3)
    I_motor = np.array([I1, I2, I3])
    I_dot_motor = np.zeros(3)
    tau_motor_desired = np.zeros(3)
    tau_motor = np.zeros(3)
    V_back = np.zeros(3)
    V_motor = np.zeros(3)

    C2 = np.cos(q2)
    S2 = np.sin(q2)
    C3 = np.cos(q3)
    S3 = np.sin(q3)
    C23 = np.cos(q2 + q3)
    S23 = np.sin(q2 + q3)

    M = np.array([
        [(1/2 * m1 * lc2**2 + 1/4 * m2 * l2**2 * C2**2 + 1/4 * m3 * l3**2 * C23**2 +
m3 * l2**2 * C2**2 + m2 * l2 * l3 * C2 * C23), 0, 0],
        [0, (1/4 * m2 * l2**2 + m3 * (l2**2 + lc3**2 + l2 * l3 * C3)), 0],
        [0, 0, (1/3 * m3 * l3**2)]
    ])

    C = np.array([
        [0, (-1/4 * m2 * l2**2 * S2 - m3 * l2**2 * S2 - m3 * l2 * l3 * S23 - m3 *
lc2**2 * S23), (-m3 * l2 * l3 * C2 * S23 - m3 * lc3**2 * S23)],
```

```
        [0, 0, -m2 * l2 * l3 * S3],
        [0, 0, 0]
    ])

    G = np.array([
        lc2 * m2 * g * C2 + l2 * m3 * g * C2 + lc3 * m2 * g * C23,
        0,
        m3 * g * lc3 * C23
    ])

    # Motor Control
    if choice == '1':
        # Simple PD Control
        tau_motor_desired = Kp * (q_desired - q) + Kd * (q_dot_desired - q_dot)

    elif choice == '2':
        # PD Control with Gravity Compensation
        tau_motor_desired = M.dot(q_ddot_desired) + C.dot(q_dot_desired) + G + Kp *
(q_desired - q) + Kd * (q_dot_desired - q_dot)

    elif choice == '3':
        # PD Control with Feed-forward term
        feedforward_term = M.dot(q_ddot_desired)
        tau_motor_desired = feedforward_term + Kp * (q_desired - q) + Kd *
(q_dot_desired - q_dot)

    elif choice == '4':
        # PD Control with Computed Torque Control
        feedforward_term = M.dot(q_ddot_desired)
        tau_desired = feedforward_term + M.dot(q_ddot_desired) + C.dot(q_dot_desired)
+ G
        tau_motor_desired = tau_desired + Kp * (q_desired - q) + Kd * (q_dot_desired -
q_dot)

    # Motor dynamics
    V_back = Kb * q_dot_motor
    I_motor = tau_motor_desired / (Kt * Rm)
    q_ddot_motor = (tau_motor_desired - Bm * q_dot_motor - Kt * I_motor) / Jm
    q_dot_motor += q_ddot_motor * dt # Euler integration
    V_motor = I_motor * Rm
    I_dot_motor = (V_motor - Rm * I_motor - V_back) / Lm
    tau_motor = V_motor * Kt

    q_ddot = np.linalg.solve(M, (tau_motor - np.dot(C, q_dot)) - G)

    return [q_dot[0], q_dot[1], q_dot[2], q_ddot[0], q_ddot[1], q_ddot[2],
q_ddot_motor[0], q_ddot_motor[1], q_ddot_motor[2], I_dot_motor[0], I_dot_motor[1],
I_dot_motor[2]]
```

```
# Initial conditions for the ODE solver
state_0 = np.concatenate((q, q_dot, q_dot_motor, I_motor))

# Use odeint to integrate the ODE
solution = odeint(dynamics, state_0, time_values)

# Extract the joint angles from the solution
q1_values = solution[:, 0]
q2_values = solution[:, 1]
q3_values = solution[:, 2]

# Plot the joint angles over time
plt.figure()
#plt.suptitle('Plot of Joint Angles (q) vs. Time (t) for 3-DOF Articulated Robot')
if choice == '1':
    plt.suptitle("\nPlot of Joint Angles (q) vs. Time (t) for 3-DOF Articulated Robot
with Simple PD Control")
elif choice == '2':
    plt.suptitle("\nPlot of Joint Angles (q) vs. Time (t) for 3-DOF Articulated Robot
with PD Control with Gravity Compensation")
elif choice == '3':
    plt.suptitle("\nPlot of Joint Angles (q) vs. Time (t) for 3-DOF Articulated Robot
with PD Control with Feed-forward term")
elif choice == '4':
    plt.suptitle("\nPlot of Joint Angles (q) vs. Time (t) for 3-DOF Articulated Robot
with PD Control with Computed Torque Control")

plt.subplot(311)
plt.plot(time_values, q1_values, label='q1')
plt.xlabel('t (s)')
plt.ylabel('q1 (rad)')
plt.grid()
plt.legend()

plt.subplot(312)
plt.plot(time_values, q2_values, label='q2')
plt.xlabel('t (s)')
plt.ylabel('q2 (rad)')
plt.grid()
plt.legend()

plt.subplot(313)
plt.plot(time_values, q3_values, label='q3')
plt.xlabel('t (s)')
plt.ylabel('q3 (rad)')
plt.grid()
plt.legend()
plt.show()
```

Output:

```
Simulation of 3-DOF Articulated Robot :

Select Control Scheme :
1. Simple PD Control
2. PD Control with Gravity Compensation
3. PD Control with Feed-forward term
4. Computed Torrque PD Control

Press a button from 1 - 4 : 1
You chose:
Simple PD Control

Plotting results...
```

Results:

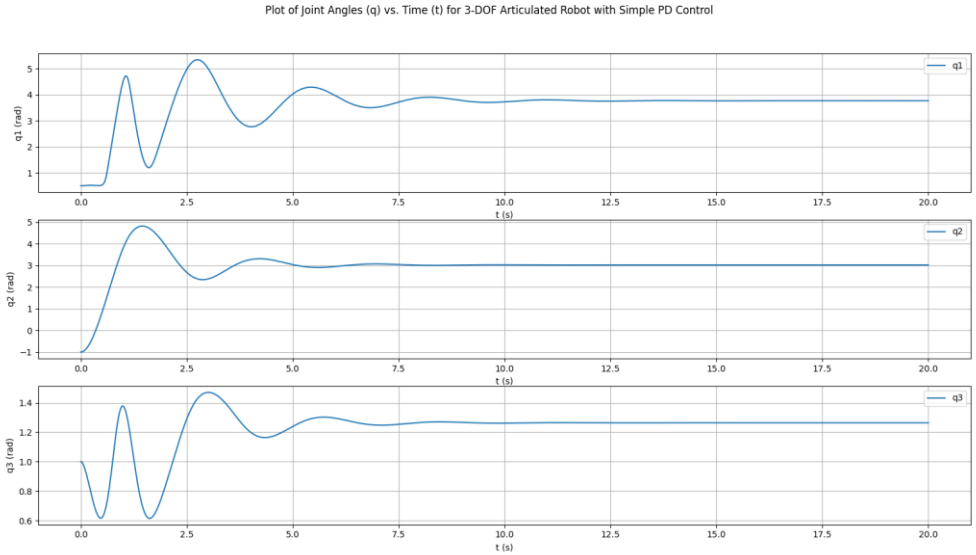

Fig.: q vs. t graph for 3-DOF Articulated Robot with PD Control

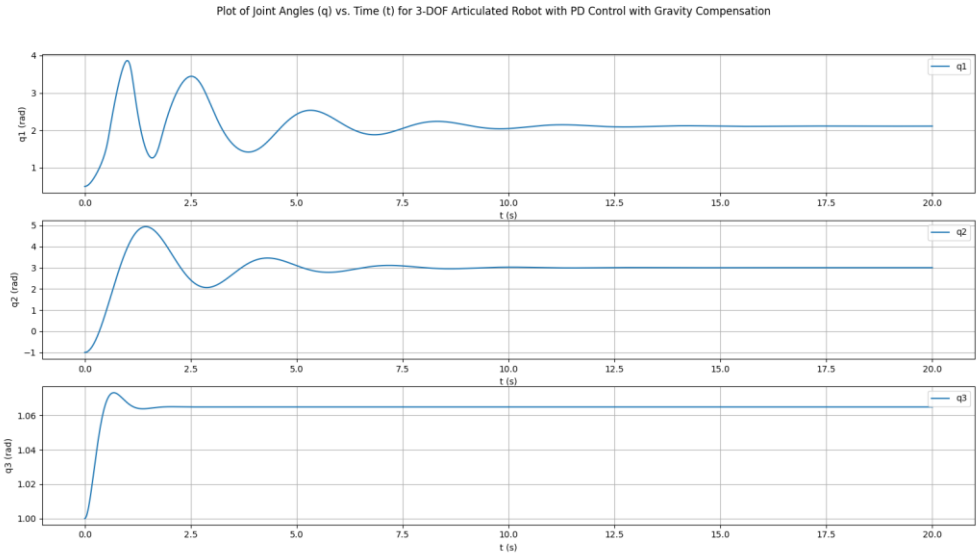

Fig.: q vs. t graph for 3-DOF Articulated Robot with PD Control with Gravity Compensation

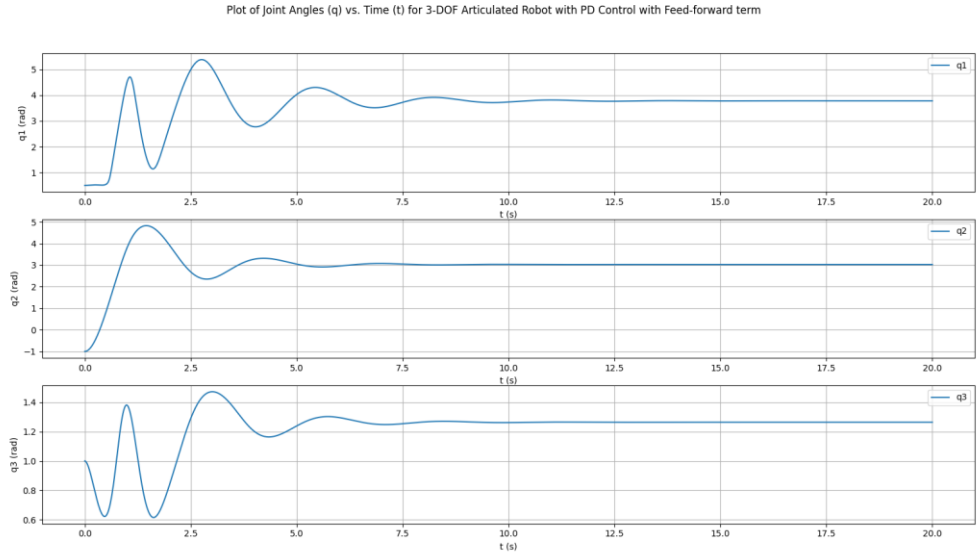

Fig.: q vs. t graph for 3-DOF Articulated Robot with PD Control with Feed-forward term

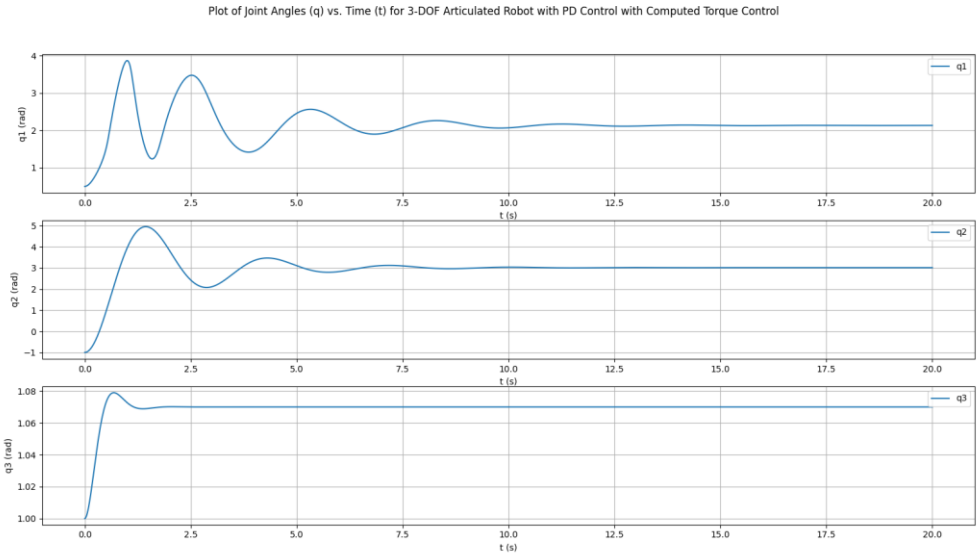

Fig.: q vs. t graph for 3-DOF Articulated Robot with PD Control with Computed Torrques

REFERENCES

1. Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. *Robot Dynamics and Control*. https://www.kramirez.net/Robotica/Tareas/Kinematics.pdf

2. NumPy Documentation: https://numpy.org/doc/stable/

3. SciPy Documentation: https://docs.scipy.org/doc/scipy/index.html

4. Aderajew Ashagrie, Ayodeji Olalekan Salau & Tilahun Weldcherkos | Eldaw Eldukhri (Reviewing editor) (2021) Modeling and control of a 3-DOF articulated robotic manipulator using self-tuning fuzzy sliding mode controller, Cogent Engineering, 8:1, DOI: 10.1080/23311916.2021.1950105

5. C. Agbaraji, E., C. Inyiama, H., & C. Okezie, C. (2017). Dynamic Modeling of a 3-DOF Articulated Robotic Manipulator Based on Independent Joint Scheme. Physical Science International Journal, 15(1), 1–10. https://doi.org/10.9734/PSIJ/2017/33578 Retrieved from: https://www.researchgate.net/publication/318242819_Dynamic_Modeling_of_a_3-DOF_Articulated_Robotic_Manipulator_Based_on_Independent_Joint_Scheme

6. Park, J. (2011). The relationship between controlled joint torque and end-effector force in underactuated robotic systems. Robotica, 29(4), 581-584. doi:10.1017/S0263574710000391

7. The Ultimate Guide to Jacobian Matrices for Robotics: https://automaticaddison.com/the-ultimate-guide-to-jacobian-matrices-for-robotics/

8. Sadegh Lafmejani, Hossein & Zarabadipour, Hassan. (2014). Modeling, Simulation and Position Control of 3DOF Articulated Manipulator. Indonesian Journal of Electrical Engineering and Informatics. 2. 132-140. DOI: 10.11591/ijeei.v2i3.119.

9. ChatGPT: https://chat.openai.com/