ME 639

INTRODUCTION TO ROBOTICS

Assignment 6

Submitted by:

Rhitosparsha Baishya
23310039
PhD (Mechanical Engineering)

CONTENTS

Question 1

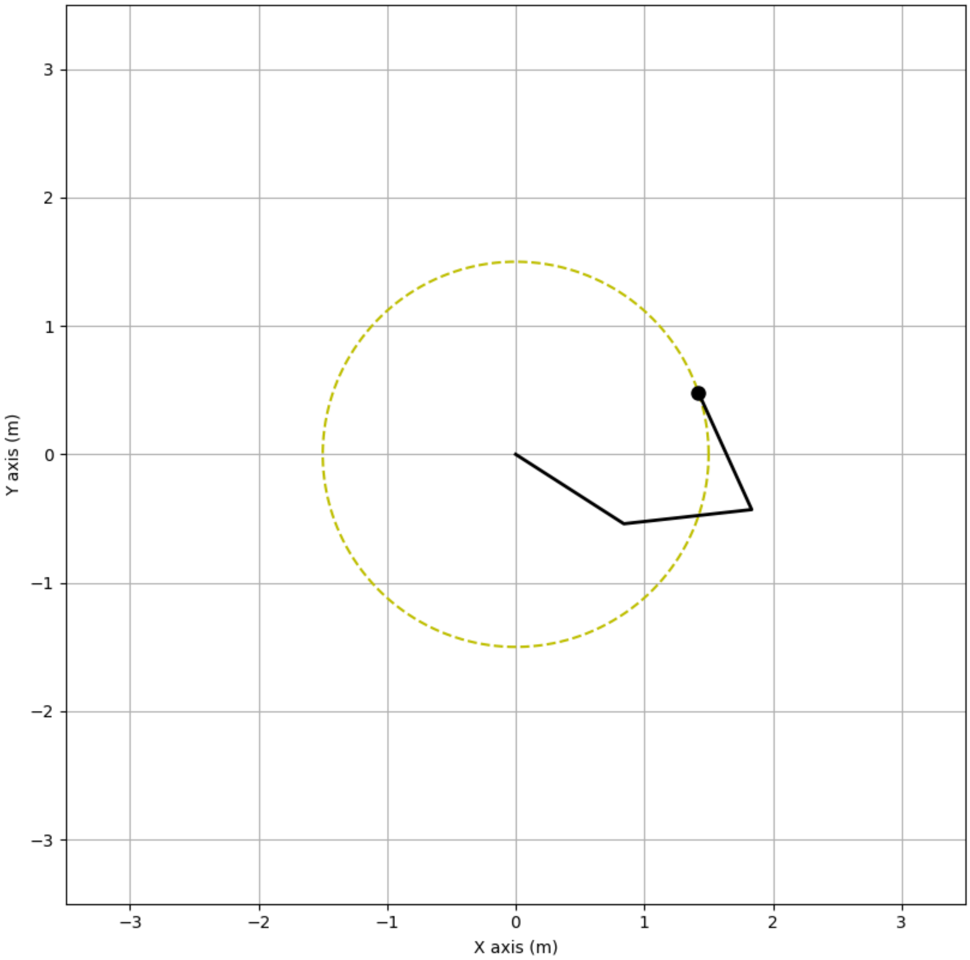
Source Code: (Link to GitHub)

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from scipy.optimize import minimize

l1 = 1
l2 = 1
l3 = 1

def jaco_3(th1, th2, th3):
    J = np.array([[-l1 * np.sin(th1) - l2 * np.sin(th2 + th1)- l3 * np.sin(th3 + th2+
th1),
                   -l2 * np.sin(th2 + th1) - l3 * np.sin(th3+ th2+ th1),
                   -l3 * np.sin(th1+th2+th3)],
                  [l1 * np.cos(th1) + l2* np.cos(th2 + th1) + l3 * np.cos(th3 + th2 +
th1),
                   l2 * np.cos(th2 + th1) + l3 *np.cos(th3 + th2 + th1),
                   l3 * np.cos(th1 + th2 + th3)]])
    return J

def fwd_kin(q):
    x = l1 * np.cos(q[0]) + l2 * np.cos(q[0] + q[1]) + l3 * np.cos(q[0] + q[1] + q[2])
    y = l1 * np.sin(q[0]) + l2 * np.sin(q[0] + q[1]) + l3 * np.sin(q[0] + q[1] + q[2])
    return np.array([x, y])

def animate_3r(q, xt, yt):
    fig, ax = plt.subplots()
    ax.plot(xt, yt, '--y')

    line, = ax.plot([], [], 'k', linewidth=2)
    point, = ax.plot([], [], 'ok', markersize=8)

    def init():
        line.set_data([], [])
        point.set_data([], [])
        return line, point

    def update(frame):
        theta1 = q[0, frame]
        theta2 = q[1, frame]
        theta3 = q[2, frame]

        H01 = np.array([[np.cos(theta1), -np.sin(theta1), 0, l1*np.cos(theta1)],
```

```
                        [np.sin(theta1), np.cos(theta1), 0, l1*np.sin(theta1)],
                        [0, 0, 1, 0],
                        [0, 0, 0, 1]])

        H12 = np.array([[np.cos(theta2), -np.sin(theta2), 0, l2*np.cos(theta2)],
                        [np.sin(theta2), np.cos(theta2), 0, l2*np.sin(theta2)],
                        [0, 0, 1, 0],
                        [0, 0, 0, 1]])

        H23 = np.array([[np.cos(theta3), -np.sin(theta3), 0, l3*np.cos(theta3)],
                        [np.sin(theta3), np.cos(theta3), 0, l3*np.sin(theta3)],
                        [0, 0, 1, 0],
                        [0, 0, 0, 1]])

        H02 = np.dot(H01, H12)
        H03 = np.dot(H02, H23)

        P1 = H01[:2, 3]
        P2 = H02[:2, 3]
        P3 = H03[:2, 3]

        line.set_data([0, P1[0], P2[0], P3[0]], [0, P1[1], P2[1], P3[1]])
        point.set_data(P3[0], P3[1])

        return line, point

    ani = FuncAnimation(fig, update, frames=len(q[0]), init_func=init, blit=True,
interval = 0.1)
    plt.xlim(-3.5, 3.5)
    plt.ylim(-3.5, 3.5)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.grid(True)
    plt.xlabel('X axis (m)')
    plt.ylabel('Y axis (m)')
    plt.show()

# Generate a circular trajectory
dt = 1/1000
t = np.arange(0, 12+dt, dt)
radius = 1.5
theta_circle = np.linspace(0, 2 * np.pi, len(t))
x_circle = radius * np.cos(theta_circle)
y_circle = radius * np.sin(theta_circle)

# Scale the circle trajectory and set the manipulator's starting position
x = x_circle
y = y_circle
dx = np.diff(x) / dt
```

```
dy = np.diff(y) / dt
v = np.array([dx, dy])

q = np.zeros((3, len(t)))

# Since the trajectory is highly dependent on the initial conditions, implement a
minimisation algorithm
# that optimises the joint angles so that the manipulator starts at point (1.5, 0)

# Function to minimize
def objective_function(q, *args):
    target_point = args
    end_effector = fwd_kin(q)
    error = np.linalg.norm(target_point - end_effector)
    return error

# Initial joint angles
initial_angles = np.array([0.5, 0.5, 0.5])

# Target point
target = np.array([1.5, 0])

# BFGS minimization
result = minimize(objective_function, initial_angles, args=target, method='BFGS')

# Extract optimized joint angles
optimized_angles = result.x

print(optimized_angles)

q[:, 0] = np.array(optimized_angles)

for k in range(len(t)-1):
    th1, th2, th3 = q[:, k]
    J = jaco_3(th1, th2, th3)
    q[:, k+1] = q[:, k] + 1 * np.linalg.pinv(J) @ v[:, k] * dt

# Animation
animate_3r(q, x, y)
```

Output:



Question 2 (a)

Source Code: (Link to GitHub)

```
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Robot parameters
link_lengths = [0.25, 0.25, 0.25]
base_offset = 0.25  # Distance from the base to the manifold
end_effector_offset = 0.1  # Distance from the end effector to the top surface of the
manifold

# Coordinates of points A, B, C, D
points = {
    'A': np.array([0.45, 0.075, 0.1]),
    'B': np.array([0.45, -0.075, 0.1]),
    'C': np.array([0.25, -0.075, 0.1]),
    'D': np.array([0.25, 0.075, 0.1]),
}

# Function to check if a point is within the robot workspace
def is_within_workspace(position):
    # Define joint angle ranges
    q1_range = np.linspace(0, 2 * np.pi, 100) # Joint 1 can rotate 360 degrees
    q2_range = np.linspace(0, np.pi, 50) # Joint 2 can rotate 180 degrees
    q3_range = np.linspace(0, 2 * np.pi, 100) # Joint 3 can rotate 360 degrees

    for q1 in q1_range:
        for q2 in q2_range:
            for q3 in q3_range:
                joint_angles = np.array([q1, q2, q3])
                computed_position = forward_kinematics(joint_angles)
                if np.linalg.norm(computed_position - position) < 0.005:
                    return True
    return False

# Forward kinematics function
def forward_kinematics(joint_angles):
    q1, q2, q3 = joint_angles
    x = base_offset + link_lengths[0] * np.cos(q1) + link_lengths[1] * np.cos(q1 + q2)
+ link_lengths[2] * np.cos(q1 + q2 + q3)
    y = link_lengths[0] * np.sin(q1) + link_lengths[1] * np.sin(q1 + q2) +
link_lengths[2] * np.sin(q1 + q2 + q3)
    z = end_effector_offset
```

```
    return np.array([x, y, z])

# Inverse kinematics function
def inverse_kinematics(position):
    x, y, z = position
    l1, l2, l3 = link_lengths
    q1 = np.arctan2(y, x - base_offset)
    r = np.sqrt(x**2 + y**2)
    D = (r**2 + (z - l1)**2 - l2**2 - l3**2) / (2 * l2 * l3)
    q2 = np.arctan2(np.sqrt(1 - D**2), D)
    gamma = np.arctan2(z - l1, r - base_offset)
    beta = np.arctan2(l3 * np.sin(q2), l2 + l3 * np.cos(q2))
    q3 = np.pi/2 - (gamma - beta)
    return np.array([q1, q2, q3])

# Define the optimization objective
def objective_function(joint_angles, target_position):
    current_position = forward_kinematics(joint_angles)
    return np.linalg.norm(current_position - target_position)

# Loop through target points
for point_name, target_position in points.items():
    # Check if the target point is within the workspace
    if not is_within_workspace(target_position):
        print(f"Point {point_name} is outside the robot's workspace.")
        continue  # Skip optimization for points outside the workspace

    # Compute initial joint angles using inverse kinematics
    initial_joint_angles = inverse_kinematics(target_position)

    # Minimize the objective function
    result = minimize(objective_function, initial_joint_angles,
args=(target_position,), method='BFGS')

    # Extract the optimal joint angles
    optimal_joint_angles = result.x

    # Perform forward kinematics using the optimal joint angles
    optimal_position = forward_kinematics(optimal_joint_angles)

    # Output results
    print(f"Point {point_name}:")
    print(f"Target position: {target_position}")
    print(f"Is in Workspace? : {is_within_workspace(target_position)}")
    print(f"Initial joint angles: {initial_joint_angles}")
    print(f"Optimal joint angles: {optimal_joint_angles}")
    print(f"Optimal position: {optimal_position}\n")
```

Output:

```
Point A:
Target position: [0.45  0.075 0.1  ]
Is in Workspace? : True
Initial joint angles: [0.35877067 0.56423116 2.48180694]
Optimal joint angles: [0.06268142 0.78782872 2.83122402]
Optimal position: [0.45000001 0.075      0.1       ]

Point B:
Target position: [ 0.45  -0.075  0.1  ]
Is in Workspace? : True
Initial joint angles: [-0.35877067  0.56423116  2.48180694]
Optimal joint angles: [-0.65485992  0.78782872  2.83122403]
Optimal position: [ 0.45000001 -0.07500001  0.1       ]

Point C:
Target position: [ 0.25  -0.075  0.1  ]
Is in Workspace? : True
Initial joint angles: [-1.57079633  1.84938603  3.99303256]
Optimal joint angles: [-1.9770564   4.17436278 -2.38996958]
Optimal position: [ 0.25  -0.075  0.1  ]

Point D:
Target position: [0.25  0.075 0.1  ]
Is in Workspace? : True
Initial joint angles: [1.57079633 1.84938603 3.99303256]
Optimal joint angles: [ 1.1645351   4.17436228 -2.38996938]
Optimal position: [0.25  0.075 0.1  ]
```

Question 2 (b)

Approach:

- Define the coordinates of the points A, B, C, and D.
- Define the duration in which trajectory is covered as well as the number of points in each segment.
- Define the inverse kinematics function for PUMA robot.
- Apply linear interpolation between the points A, B, C, and D and store the coordinates of all the in-between points.
- Perform inverse kinematics at each point to obtain the respective joint angles.
- Differentiate the joint angle values to obtain joint velocities.
- Again differentiate joint velocity values to obtain joint accelerations.

Source Code: ([Link to GitHub]())

```
import numpy as np
import matplotlib.pyplot as plt

# Robot parameters
base_offset = 0.25
end_effector_offset = 0.1
link_lengths = [0.25, 0.25, 0.25]

# Initial joint angles (you can adjust these based on your robot configuration)
q1 = 0.0
q2 = 0.0
q3 = 0.0

# Define the coordinates of points A, B, C, and D
A = np.array([0.40, 0.06, 0.1])
B = np.array([0.40, 0.01, 0.1])
C = np.array([0.35, 0.01, 0.1])
D = np.array([0.35, 0.06, 0.1])

# Time duration for each segment
time_duration = 2.0  # seconds

# Number of points in each segment
num_points = 100

# Inverse kinematics function
def inverse_kinematics(position):
    x, y, z = position
    l1, l2, l3 = link_lengths
    q1 = np.arctan2(y, x - base_offset)
```

```
    r = np.sqrt(x**2 + y**2)
    D = (r**2 + (z - l1)**2 - l2**2 - l3**2) / (2 * l2 * l3)
    q2 = np.arctan2(np.sqrt(1 - D**2), D)
    gamma = np.arctan2(z - l1, r - base_offset)
    beta = np.arctan2(l3 * np.sin(q2), l2 + l3 * np.cos(q2))
    q3 = np.pi/2 - (gamma - beta)
    return np.array([q1, q2, q3])

# Generate joint angles trajectory
q1_traj = np.zeros(num_points * 4)
q2_traj = np.zeros(num_points * 4)
q3_traj = np.zeros(num_points * 4)

# Generate joint velocities trajectory
q1_dot_traj = np.zeros(num_points * 4)
q2_dot_traj = np.zeros(num_points * 4)
q3_dot_traj = np.zeros(num_points * 4)

# Generate joint accelerations trajectory
q1_ddot_traj = np.zeros(num_points * 4)
q2_ddot_traj = np.zeros(num_points * 4)
q3_ddot_traj = np.zeros(num_points * 4)

# Generate Cartesian trajectory
cartesian_traj = np.zeros((num_points * 4, 3))

# Time array
time_points = np.linspace(0, time_duration, num_points * 4)

# Generate trajectory for each segment
for i in range(4):
    start_index = i * num_points
    end_index = (i + 1) * num_points

    # Generate Cartesian trajectory for the segment
    cartesian_traj[start_index:end_index] = np.linspace([A, B, C, D][i], [A, B, C,
D][(i + 1) % 4], num_points)

    # Convert Cartesian positions to joint angles using inverse kinematics
    for j in range(num_points):
        joint_angles = inverse_kinematics(cartesian_traj[start_index + j])
        q1_traj[start_index + j], q2_traj[start_index + j], q3_traj[start_index + j] =
joint_angles

# Calculate joint velocities using numerical differentiation
q1_dot_traj[:-1] = np.diff(q1_traj) / np.diff(time_points)
q2_dot_traj[:-1] = np.diff(q2_traj) / np.diff(time_points)
q3_dot_traj[:-1] = np.diff(q3_traj) / np.diff(time_points)
```

```python
# Calculate joint accelerations using numerical differentiation
q1_ddot_traj[:-1] = np.diff(q1_dot_traj) / np.diff(time_points)
q2_ddot_traj[:-1] = np.diff(q2_dot_traj) / np.diff(time_points)
q3_ddot_traj[:-1] = np.diff(q3_dot_traj) / np.diff(time_points)

# Display joint angles and Cartesian positions
for i in range(num_points * 4):
    print(f"Time: {i * (time_duration / (num_points * 4)):.3f}s")
    print(f"q1: {q1_traj[i]:.4f} rad - q2: {q2_traj[i]:.4f} rad - q3: {q3_traj[i]:.4f} rad")
    print(f"q1_dot: {q1_dot_traj[i]:.4f} rad/s - q2_dot: {q2_dot_traj[i]:.4f} rad/s - q3_dot: {q3_dot_traj[i]:.4f} rad/s")
    print(f"q1_ddot: {q1_ddot_traj[i]:.4f} rad/s^2 - q2_ddot: {q2_ddot_traj[i]:.4f} rad/s^2 - q3_ddot: {q3_ddot_traj[i]:.4f} rad/s^2")
    print(f"Cartesian Position: {cartesian_traj[i]}")
    print()

# Visualize the trajectory in 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.plot(cartesian_traj[:, 0], cartesian_traj[:, 1], cartesian_traj[:, 2], marker='o')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
plt.show()
```

Output:

```
Time: 0.000s
q1: 0.3805 rad - q2: 1.0601 rad - q3: 2.8715 rad
q1_dot: -0.5797 rad/s - q2_dot: 0.1104 rad/s - q3_dot: 0.1034 rad/s
q1_ddot: -0.2672 rad/s^2 - q2_ddot: -0.1929 rad/s^2 - q3_ddot: -0.1712 rad/s^2
Cartesian Position: [0.4  0.06 0.1 ]

Time: 0.005s
q1: 0.3776 rad - q2: 1.0606 rad - q3: 2.8721 rad
q1_dot: -0.5811 rad/s - q2_dot: 0.1094 rad/s - q3_dot: 0.1025 rad/s
q1_ddot: -0.2662 rad/s^2 - q2_ddot: -0.1927 rad/s^2 - q3_ddot: -0.1713 rad/s^2
Cartesian Position: [0.4        0.05949495 0.1       ]

Time: 0.010s
q1: 0.3747 rad - q2: 1.0612 rad - q3: 2.8726 rad
q1_dot: -0.5824 rad/s - q2_dot: 0.1085 rad/s - q3_dot: 0.1016 rad/s
q1_ddot: -0.2651 rad/s^2 - q2_ddot: -0.1925 rad/s^2 - q3_ddot: -0.1713 rad/s^2
Cartesian Position: [0.4       0.0589899 0.1      ]

Time: 0.015s
q1: 0.3718 rad - q2: 1.0617 rad - q3: 2.8731 rad
q1_dot: -0.5837 rad/s - q2_dot: 0.1075 rad/s - q3_dot: 0.1008 rad/s
q1_ddot: -0.2640 rad/s^2 - q2_ddot: -0.1923 rad/s^2 - q3_ddot: -0.1714 rad/s^2
Cartesian Position: [0.4        0.05848485 0.1       ]

Time: 0.020s
q1: 0.3688 rad - q2: 1.0623 rad - q3: 2.8736 rad
q1_dot: -0.5851 rad/s - q2_dot: 0.1065 rad/s - q3_dot: 0.0999 rad/s
q1_ddot: -0.2629 rad/s^2 - q2_ddot: -0.1922 rad/s^2 - q3_ddot: -0.1715 rad/s^2
Cartesian Position: [0.4       0.0579798 0.1      ]

Time: 0.025s
q1: 0.3659 rad - q2: 1.0628 rad - q3: 2.8741 rad
q1_dot: -0.5864 rad/s - q2_dot: 0.1056 rad/s - q3_dot: 0.0991 rad/s
q1_ddot: -0.2618 rad/s^2 - q2_ddot: -0.1920 rad/s^2 - q3_ddot: -0.1715 rad/s^2
Cartesian Position: [0.4        0.05747475 0.1       ]

Time: 0.030s
q1: 0.3630 rad - q2: 1.0633 rad - q3: 2.8746 rad
q1_dot: -0.5877 rad/s - q2_dot: 0.1046 rad/s - q3_dot: 0.0982 rad/s
q1_ddot: -0.2606 rad/s^2 - q2_ddot: -0.1918 rad/s^2 - q3_ddot: -0.1716 rad/s^2
Cartesian Position: [0.4       0.0569697 0.1      ]
```

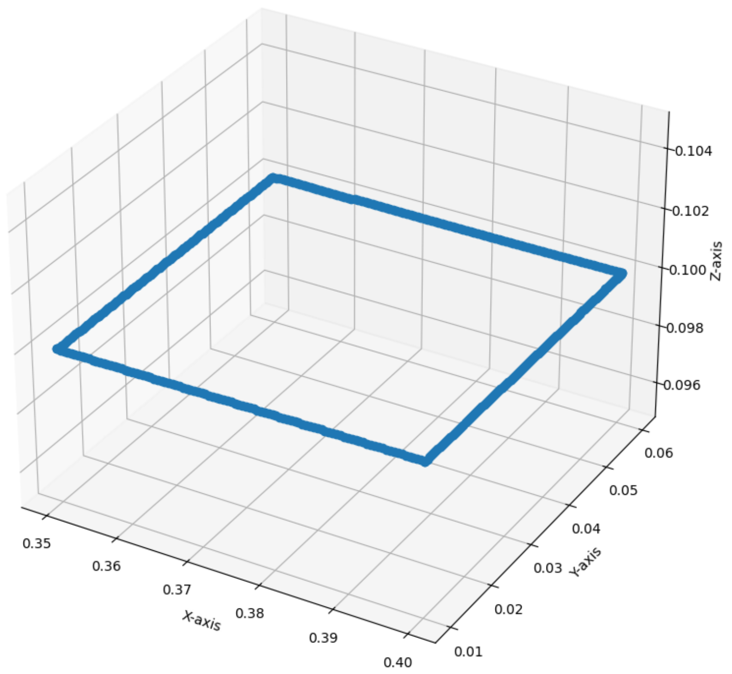


Fig.: Trajectory in Cartesian space

Question 2 (c)

Source Code: ([Link to GitHub]())

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


# ================== DEFINITION OF PARAMETERS ===================

# Robot parameters
m1, m2, m3 = 0.8, 0.8, 0.8  # Masses of links
l1, l2, l3 = 0.25, 0.25, 0.25 # Lengths of links
lc2 = 0.125  # Distance from joint 1 to the center of mass of link 2
lc3 = 0.125  # Distance from joint 2 to the center of mass of link 3
base_offset = 0.25  # Offset from the base to the first joint
g = 9.81  # Acceleration due to gravity

# Time parameters
total_time = 8.0  # Total simulation time
num_steps = 800
dt = total_time / num_steps
time_points = np.linspace(0, total_time, int(num_steps))

# Control gains
kp1, kp2, kp3 = 100, 100, 100  # Proportional gains
Kp = np.array([kp1, kp2, kp3])

# Calculate derivative gains for critical damping
kd1 = 2 * np.sqrt(kp1 * (1/2 * m1 * lc2**2 + 1/4 * m2 * l2**2 + 1/4 * m3 * l3**2 + m3
* l2**2 + m2 * l2 * l3))
kd2 = 2 * np.sqrt(kp2 * (1/4 * m2 * l2**2 + m3 * (l2**2 + lc3**2 + l2 * l3)))
kd3 = 2 * np.sqrt(kp3 * (1/3 * m3 * l3**2))
Kd = np.array([kd1, kd2, kd3])


# ================== TRAJECTORY GENERATION ===================

# Define the coordinates of points A, B, C, and D
A = np.array([0.40, 0.06, 0.1])
B = np.array([0.40, 0.01, 0.1])
C = np.array([0.35, 0.01, 0.1])
D = np.array([0.35, 0.06, 0.1])

# Inverse kinematics function
def inverse_kinematics(position):
    x, y, z = position
```

```python
    q1 = np.arctan2(y, x - base_offset)
    r = np.sqrt(x**2 + y**2)
    D = (r**2 + (z - l1)**2 - l2**2 - l3**2) / (2 * l2 * l3)
    q2 = np.arctan2(np.sqrt(1 - D**2), D)
    gamma = np.arctan2(z - l1, r - base_offset)
    beta = np.arctan2(l3 * np.sin(q2), l2 + l3 * np.cos(q2))
    q3 = np.pi/2 - (gamma - beta)
    return np.array([q1, q2, q3])

# Generate joint angles trajectory
q1_traj = np.zeros(num_steps)
q2_traj = np.zeros(num_steps)
q3_traj = np.zeros(num_steps)

# Generate joint velocities trajectory
q1_dot_traj = np.zeros(num_steps)
q2_dot_traj = np.zeros(num_steps)
q3_dot_traj = np.zeros(num_steps)

# Generate joint accelerations trajectory
q1_ddot_traj = np.zeros(num_steps)
q2_ddot_traj = np.zeros(num_steps)
q3_ddot_traj = np.zeros(num_steps)

# Generate Cartesian trajectory for the entire duration
cartesian_traj = np.zeros((num_steps, 3))

# Generate trajectory for each segment
for i in range(4):
    start_index = i * int(num_steps / 4)
    end_index = (i + 1) * int(num_steps / 4)

    # Generate Cartesian trajectory for the segment
    cartesian_traj[start_index:end_index] = np.linspace([A, B, C, D][i], [A, B, C,
D][(i + 1) % 4], int(num_steps / 4))

    # Convert Cartesian positions to joint angles using inverse kinematics
    for j in range(int(num_steps / 4)):
        joint_angles = inverse_kinematics(cartesian_traj[start_index + j])
        q1_traj[start_index + j], q2_traj[start_index + j], q3_traj[start_index + j] =
joint_angles

# Calculate joint velocities using numerical differentiation
q1_dot_traj[:-1] = np.diff(q1_traj) / np.diff(time_points)
q2_dot_traj[:-1] = np.diff(q2_traj) / np.diff(time_points)
q3_dot_traj[:-1] = np.diff(q3_traj) / np.diff(time_points)
```

```python
# Calculate joint accelerations using numerical differentiation
q1_ddot_traj[:-1] = np.diff(q1_dot_traj) / np.diff(time_points)
q2_ddot_traj[:-1] = np.diff(q2_dot_traj) / np.diff(time_points)
q3_ddot_traj[:-1] = np.diff(q3_dot_traj) / np.diff(time_points)


# =================== SOLVING OF DYNAMICS PROBLEM ===================

# Dynamics function
def dynamics(t, state):
    q1, q2, q3, q1_dot, q2_dot, q3_dot = state

    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
    ])

    index = min(int(t / dt), len(q1_traj) - 1)

    # Feedforward control using desired accelerations
    feedforward_term = np.array([q1_ddot_traj[index], q2_ddot_traj[index],
q3_ddot_traj[index]])

    # Proportional control with tracking error
    error = np.array([q1_traj[index] - q1, q2_traj[index] - q2, q3_traj[index] - q3])
    proportional_term = Kp * error
```

```
    # Derivative control
    error_dot = -np.array([q1_dot - q1_dot_traj[index], q2_dot - q2_dot_traj[index],
q3_dot - q3_dot_traj[index]])
    derivative_term = Kd * error_dot

    # External disturbances (small random values)
    disturbance_scale = 0.0001
    disturbances = disturbance_scale * np.random.randn(3)

    # Calculate control input
    control_input = feedforward_term + proportional_term + derivative_term +
disturbances

    qdd = np.linalg.solve(M, -np.dot(C, [q1_dot, q2_dot, q3_dot]) - G + control_input)
    return [q1_dot, q2_dot, q3_dot, qdd[0], qdd[1], qdd[2]]


# Solve the initial value problem using Runge-Kutta method
sol = solve_ivp(
    fun=dynamics,
    t_span=(0, total_time),
    y0=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0],  # Joint angles and velocities
    method='DOP853',
    t_eval=np.linspace(0, total_time, num_steps),
    args=()
)

# Extract the results
time = sol.t
q = sol.y[:3, :]
qd = sol.y[3:, :]


# =================== PLOTTING OF RESULTS ===================


# Plot for Joint 1
plt.figure()
plt.plot(time, q1_traj, label='Desired Angle')
plt.plot(time, q[0, :], label='Actual Angle')
plt.xlabel('t (s)')
plt.ylabel('q1 (rad)')
plt.grid()
plt.legend()
plt.title('Plot of Joint 1 Angles (q1) vs. Time (t) for PUMA Robot')
plt.show()

# Plot for Joint 2
plt.figure()
```

```
plt.plot(time, q2_traj, label='Desired Angle')
plt.plot(time, q[1, :], label='Actual Angle')
plt.xlabel('t (s)')
plt.ylabel('q2 (rad)')
plt.grid()
plt.legend()
plt.title('Plot of Joint 2 Angles (q2) vs. Time (t) for PUMA Robot')
plt.show()

# Plot for Joint 3
plt.figure()
plt.plot(time, q3_traj, label='Desired Angle')
plt.plot(time, q[2, :], label='Actual Angle')
plt.xlabel('t (s)')
plt.ylabel('q3 (rad)')
plt.grid()
plt.legend()
plt.title('Plot of Joint 3 Angles (q3) vs. Time (t) for PUMA Robot')
plt.show()
```

Results:

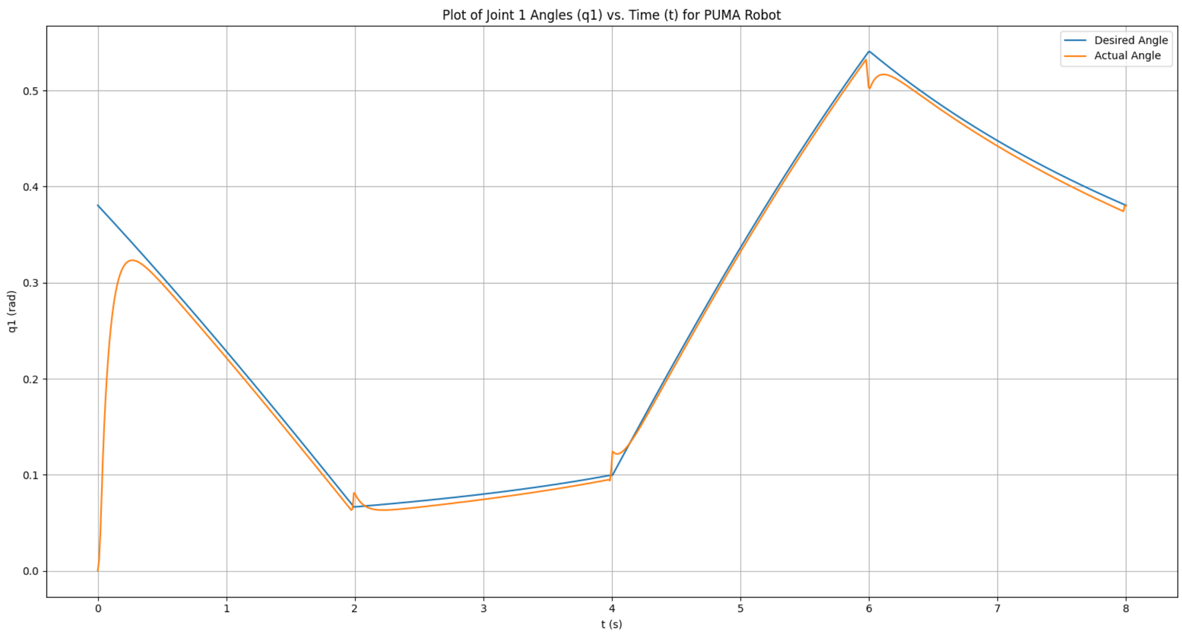


Fig.: Joint Angle vs. Time graph for Joint 1 of robot

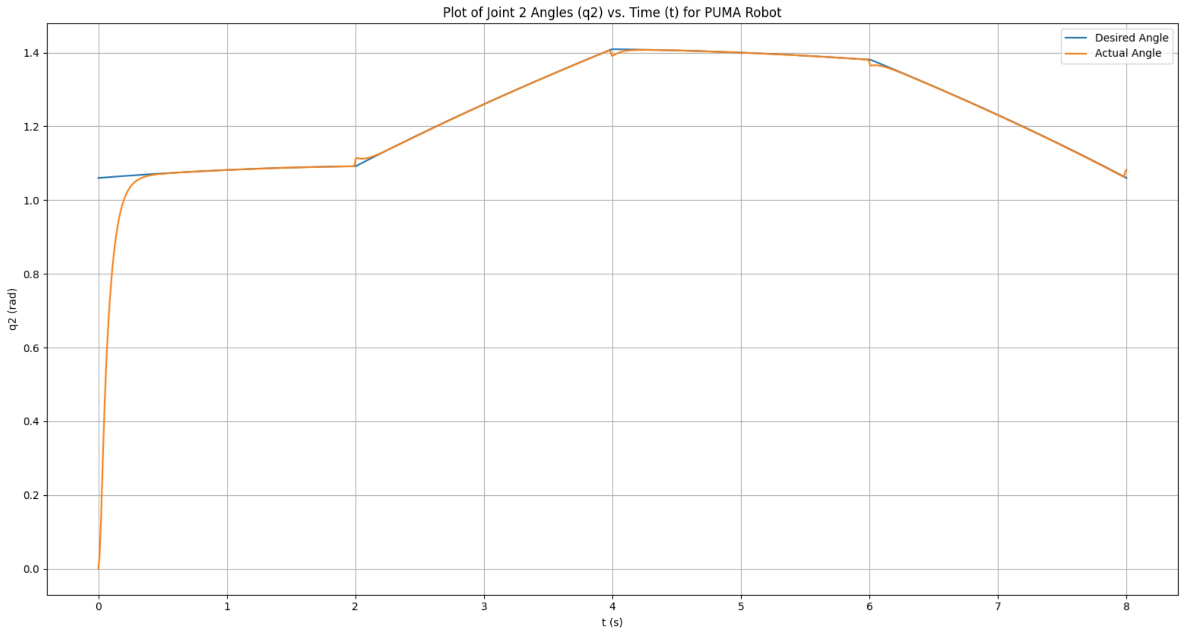


Fig.: Joint Angle vs. Time graph for Joint 2 of robot

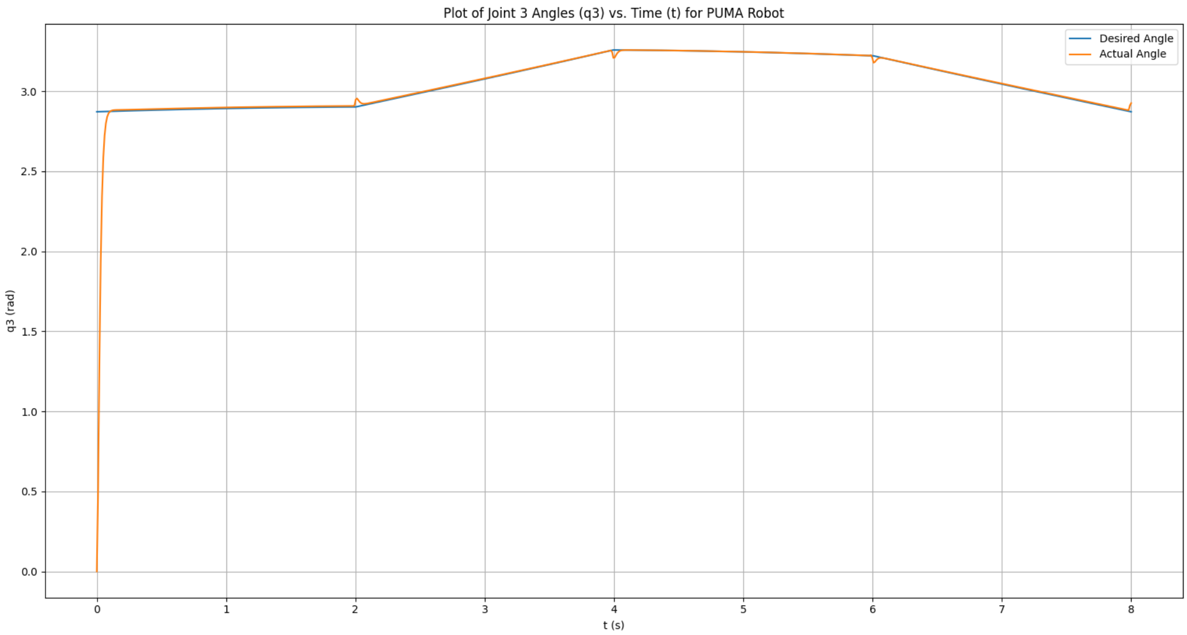


Fig.: Joint Angle vs. Time graph for Joint 3 of robot

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. Y. Yavin (2000). Control of a three-link manipulator with a constraint on the velocity of its end-effector. Computers & Mathematics with Applications, Volume 40, Issues 10–11, Pages 1263-1273, https://doi.org/10.1016/S0898-1221(00)00237-6.

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. Video 8: Inverse Kinematics of 3R Planar Manipulator: https://www.youtube.com/watch?v=AQW0ITpOeBw

10. A Gentle Introduction to the BFGS Optimization Algorithm: https://machinelearningmastery.com/bfgs-optimization-in-python/

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