Hands-on Intervention

Lab #2 - Resolved-rate motion control

Delivered by: Rihab Laroussi

Group Member: Davina Sanghera

Date of Submission:

28/02/2025

Table of Contents

Introduction

The goal of this lab is to implement a kinematic simulation of a planar robotic manipulator. The lab consists of two main exercises: the first exercise focuses on simulating the motion of a planar robot using the Denavit-Hartenberg (DH) formulation, while the second exercise involves implementing the resolved-rate motion control algorithm to manage the robot's end-effector. This report will discuss the methodology, results and the conclusion drawn from the simulation.

Exercise 1: Kinematic Simulation of a Planar Robotic Manipulator

1. Methodology

The kinematic structure of the planar 2-link manipulator was modelled using the DH convention.

The DH parameters (d, θ, a, α) were defined for each joint of the manipulator:

d: Displacement along the Z-axis (set to 0 for a planar manipulator).

θ: Joint angles (variable inputs for simulation).

a: Link lengths (fixed values: a1=0.75m, a2=0.5m).

α: Rotation around the X-axis (set to 0 for a planar manipulator).

The robotic structure, shown in Figure 1, consists of a two-degree of freedom manipulator with two rotational joints.

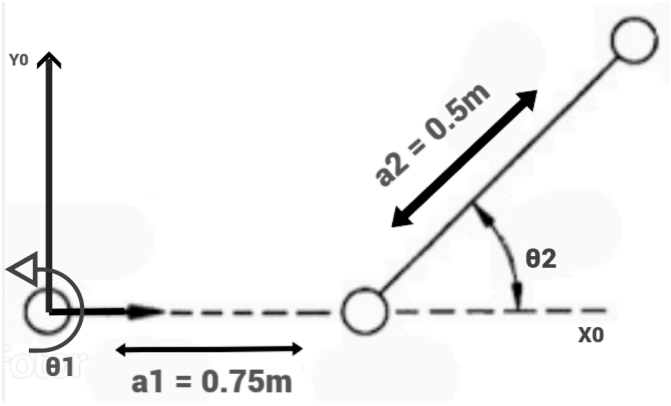

Figure 1: 2D schematic of a two-link robot arm.

A function DH(d, theta, a, alpha) was implemented to compute the homogeneous transformation matrices for each joint based on the DH parameters. The Dh parameter was calculated using the following formula:

$$T_n^{n-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_i) & -\sin(\alpha_i) & 0 \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The forward kinematics of the manipulator were computed using the function kinematics(d, theta, a, alpha) which sequentially multiplies the DH transformation matrices to obtain the pose of the end-effector relative to the base frame. It iterates through each joint, computes the transformation matrix, and accumulates the transformations. The resulting transformation matrices described the position and orientation of each link in the robot's kinematic chain.

2. Code

Exercise1 is simulated in Exercice1.py where it begins by defining the robot's geometry through the DH parameters, the simulation updates the robot's joint positions over time based on predefined joint velocities, and the corresponding transformation matrices are computed to determine the position and orientation of each link. The robot's structure and the end-effector's trajectory are visualized in real time using Matplotlib, while the joint angles are tracked and plotted to analyze their evolution over the simulation duration.

----------Exercice1.py----------

```
# Import necessary libraries
from Common import * # Import our library (includes Numpy)
import matplotlib.pyplot as plt
import matplotlib.animation as anim

# Robot definition (planar 2 link manipulator)
d = np.zeros(2)          # displacement along Z-axis
q = np.array([0.2, 0.5])  # rotation around Z-axis (theta)
a = np.array([0.75, 0.5]) # displacement along X-axis
alpha = np.zeros(2)       # rotation around X-axis

# Simulation params
dt = 0.01 # Sampling time
Tt = 10  # Total simulation time
tt = np.arange(0, Tt, dt) # Simulation time vector
```

```
# Drawing preparation
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2,2))
ax.set_title('Kinematics')
ax.set_xlabel('x[m]')
ax.set_ylabel('y[m]')
ax.set_aspect('equal')
ax.grid()
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'r-', lw=1) # End-effector path

# Memory
PPx = []
PPy = []
q1_pos = []
q2_pos = []
timestamp = []

# Simulation initialization
def init():
    line.set_data([], [])
    path.set_data([], [])
    return line, path # called once

# Simulation loop
def simulate(t):
    global d, q, a, alpha
    global PPx, PPy

    # Update robot
    T = kinematics(d, q, a, alpha) # T is the transformation matrix of the end effector
    dq = np.array([0.6,0.8]) # Joint velocities
    q = q + dt * dq  # Update joint positions

    # Update drawing
    PP = robotPoints2D(T)
    line.set_data(PP[0,:], PP[1,:])
    PPx.append(PP[0,-1])
    PPy.append(PP[1,-1])
    path.set_data(PPx, PPy)

    # Store joint positions
    q1_pos.append(q[0])
    q2_pos.append(q[1])
    timestamp.append(t)
```

```
    return line, path

# Run simulation
animation = anim.FuncAnimation(fig, simulate, tt,
                               interval=10, blit=True, init_func=init, repeat=False)
plt.show()

# Plot joint positions
fig = plt.figure()
ax = fig.add_subplot()
ax.set_title('Joint position')
ax.set_xlabel('Time[s]')
ax.set_ylabel('Angle[rad]')
ax.set_aspect('equal')
ax.grid(True)
plt.plot(timestamp,q1_pos,label='q1')
plt.plot(timestamp,q2_pos,label='q2')
plt.legend()
plt.show()
```

----------Common.py----------

```
import numpy as np # Import Numpy

def DH(d, theta, a, alpha):

    # 1. Build matrices representing elementary transformations (based on input parameters).
    # T1: Translation along Z-axis by d
    Tz = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0],
                   [0, 0, 1, d],
                   [0, 0, 0, 1]])
    # R1: Rotation around Z-axis by theta
    Rz = np.array([[np.cos(theta), -np.sin(theta), 0, 0], [np.sin(theta), np.cos(theta), 0, 0], [0,0, 1, 0],[0,0,
0, 1]])
    # T2: Translation along X-axis by a
    Tx = np.array([[1, 0, 0, a],
                   [0, 1, 0, 0],
                   [0, 0, 1, 0],
                    [0, 0, 0, 1]])
```

```
    # R2: Rotation around X-axis by alpha
      Rx = np.array([[1,0, 0, 0], [0, np.cos(alpha), -np.sin(alpha), 0], [0,np.sin(alpha), np.cos(alpha), 0],[0,0, 0, 1]])
    # 2. Multiply matrices in the correct order (result in T).
    T = Tz @ Rz @ Tx @ Rx

    return T

def kinematics(d, theta, a, alpha):

     T = [np.eye(4)]  # Base transformation m ,next transformstion is end from first link ot the base , second transformation is end from second link to the base

    # For each set of DH parameters:
    for i in range(len(d)):
        # 1. Compute the DH transformation matrix for the current joint.
        # 2. Compute the resulting accumulated transformation from the base frame.
        T_accumulated = T[-1] @ DH(d[i], theta[i], a[i], alpha[i])
        # 3. Append the computed transformation to T.
        T.append(T_accumulated)

    return T
```

3. Results

We created an animation to demonstrate the robot's configuration and the trajectory of its end-effector on a 2D plane, as shown in Figure 2. The orientation of the robot arm's segments is updated at each time interval using transformation matrices computed through the Denavit-Hartenberg (DH) method. The joint velocities were set to 0.6 and 0.9 to drive the motion of the robot.

Additionally, Figure 3 displays the evolution of the joint angles throughout the simulation, demonstrating how the robot's joints change as the simulation progresses.

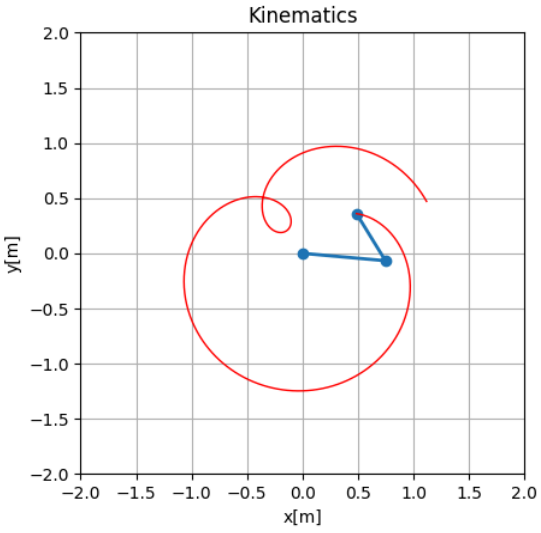

Figure 2: Robot structure in motion

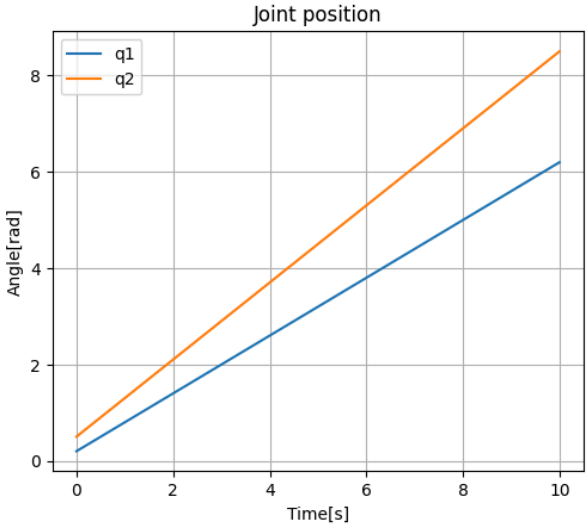

Figure 3: Robot's joints positions over time

Exercise 2: Resolved-Rate Motion Control

1. Methodology

In this exercise, the resolved-rate motion control algorithm is used to compute the joint velocities required to achieve a desired end-effector velocity. In order to implement the algorithm, we used the following functions:

The function jacobian(T, revolute), located in the common.py file, computes the Jacobian matrix for the end-effector of a robot based on the chain of kinematic transformations (T) and joint types (revolute or prismatic). The Jacobian relates joint velocities to the linear and angular velocities of the end-effector. The function follows **Algorithm 1**.

Algorithm 1:

1 Compute: $T_i^{i-1} = DH(d_i, \theta_i, a_i, \alpha_i), \quad i = 1 \ldots n$

2 Compute: $T_i = T_i^0 = \prod_{k=1}^{i} T_k^{k-1}, \quad i = 1 \ldots n$

3 Initialise: $z_0 = [0 \quad 0 \quad 1]^T, O_0 = [0 \quad 0 \quad 0]^T$

4 ***for*** $i \in 1 \ldots n$

5 $$J_i = \begin{bmatrix} \rho_i z_{i-1} \times (O_n - O_{i-1}) + (1 - \rho_i) z_{i-1} \\ \rho_i z_{i-1} \end{bmatrix}$$

6 ***end for***

7 ***return*** J

The DLS solution function , DLS(A, damping), is also included in common.py, this function computes the damped least-squares (DLS) solution to the matrix inverse problem. The DLS method is used to handle singularities and ensure stable control. The solution was calculated using the formula:

$$\zeta = J^T(\mathbf{q})\left(J(\mathbf{q})J^T(\mathbf{q}) + \lambda^2 I\right)^{-1} \dot{x}_E$$

where J is the Jacobian matrix, λ is the damping factor, I is the identity matrix, and xE is the desired end-effector velocity.

In addition to the DLS method, two other solutions were implemented: Pseudoinverse and Transpose methods.

The pseudoinverse was calculated using the formula:

$$\zeta = J^{-1}(\mathbf{q})\dot{x}_E$$

Where it provides a direct solution to the inverse kinematics problem when the Jacobian is non-singular.

The Transpose was calculated using the formula:

$$\zeta = J^T(\mathbf{q})\dot{x}_E$$

Where it takes the transpose of the Jacobian.

2. Code

We first set up the imports and defined the robot's Denavit-Hartenberg parameters, including displacement, rotation, and goal position. Then, we initialized the simulation parameters and memory, set up the figure for visualizing the robot's motion and tracking error norms in `setup_figure()`, and prepared data storage for the end-effector trajectory and error norms.

Within the function simulate(t), the robot's transformation matrices are computed, the geometric Jacobian is derived to relate joint velocities to end-effector motion. Based on the specified control method (Transpose, Pseudoinverse, or Damped Least Squares), the loop calculates the required joint velocity adjustments to minimize the error between the desired and actual end-effector positions.

The end of the code runs each simulation method, visualizes the robot's motion, and saves the respective error norms.

----------Exercice2.py----------

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim
from Common import *

# Robot definition
d = np.zeros(2)  # displacement along Z-axis
q = np.array([0.2, 0.5])  # rotation around Z-axis (theta)
a = np.array([0.75, 0.5])  # displacement along X-axis
alpha = np.zeros(2)  # rotation around X-axis
revolute = [True, True]
sigma_d = np.array([0, 1]).reshape(2, 1)  # Goal Position
K = np.diag([1, 1])  # Gain
control_method = ''
damping = 0.1 # Damping factor
```

```
# Simulation params
dt = 1.0 / 60.0


# Animation setup
def setup_figure():
    fig = plt.figure()
    ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2))
    ax.set_title('Simulation')
    ax.set_aspect('equal')
    ax.grid()
    line, = ax.plot([], [], 'o-', lw=2)  # Robot structure
    path, = ax.plot([], [], 'c-', lw=1)  # End-effector path
    point, = ax.plot([], [], 'rx')  # Target
    return fig, ax, line, path, point

# Memory
PPx = []
PPy = []

# Lists to store the error norms for each method
err_norm_P = []
err_norm_T = []
err_norm_D = []

# Simulation initialization
def init():
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    return line, path, point

# Simulation loop
```

```
def simulate(t):
    global d, q, a, alpha, revolute, sigma_d
    global PPx, PPy
    global err_norm_P, err_norm_T, err_norm_D

    # Update robot
    T = kinematics(d, q, a, alpha)
    J = jacobian(T, revolute)
    J_pos = J[0:2, :]  # Position Jacobian

    # Update control
    sigma = T[-1][:2, 3]            # Position of the end-effector
    err = sigma_d.flatten() - sigma  # Control error

    # Control Solutions
    if control_method == 'Transpose':
        dq = J_pos.T @ (K @ err)  # Transpose method
        err_norm_T.append(np.linalg.norm(err))

    elif control_method == 'Pseudoinverse':
        dq = np.linalg.pinv(J_pos) @ (K @ err)  # Pseudoinverse method
        err_norm_P.append(np.linalg.norm(err))

    elif control_method == 'DLS':
        dq = DLS(J_pos, damping) @ (K @ err)  # Damped Least Squares (DLS) method
        err_norm_D.append(np.linalg.norm(err))

    q += dt * dq  # Update joint angles

    # Update drawing
    P = robotPoints2D(T)
    line.set_data(P[0, :], P[1, :])
    PPx.append(P[0, -1])
    PPy.append(P[1, -1])
```

```
    path.set_data(PPx, PPy)
    point.set_data(sigma_d[0], sigma_d[1])

    return line, path, point


# Run transpose simulation
control_method = 'Transpose'
PPx = []
PPy = []
q = np.array([0.2, 0.5])  # Reset initial joint angles
fig, ax, line, path, point = setup_figure()
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 20, dt),
                               interval=10, blit=True, init_func=init, repeat=False)
plt.pause(20)
plt.close(fig)
np.save("errnorm_transpose.npy", np.array(err_norm_T))
# Run pseudoinverse simulation
control_method = "Pseudoinverse"
PPx = []
PPy = []
q = np.array([0.2, 0.5])  # Reset initial joint angles
fig, ax, line, path, point = setup_figure()
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 20, dt),
                               interval=10, blit=True, init_func=init, repeat=False)
plt.pause(20)
plt.close(fig)
np.save("errnorm_pseudoinverse.npy", np.array(err_norm_P))
# Run DLS simulation
control_method = "DLS"
PPx = []
PPy = []
q = np.array([0.2, 0.5])  # Reset initial joint angles
fig, ax, line, path, point = setup_figure()
```

```
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 20, dt),
                               interval=10, blit=True, init_func=init, repeat=False)
plt.pause(20)
plt.close(fig)
np.save("errnorm_DLS.npy", np.array(err_norm_D))
```

----------error.py----------

In order to visualize the evolution of the control error norm over time, we implemented the following code in the file error.py where we loaded the output of three simulations, all with the same initial conditions and desired end-effector position, using the different solution methods.

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

# Load error norm data
errnorm_transpose = np.load("errnorm_transpose.npy")
errnorm_pseudoinverse = np.load("errnorm_pseudoinverse.npy")
errnorm_DLS = np.load("errnorm_DLS.npy")

# Generate time vector starting from 0 with step size dt
dt = 1.0 / 60.0
transpose_time = np.arange(0, len(errnorm_transpose) * dt, dt)
pseudoinverse_time = np.arange(0, len(errnorm_pseudoinverse) * dt, dt)
DLS_time = np.arange(0, len(errnorm_DLS) * dt, dt)

# Plot error norm curves
plt.figure(figsize=(8, 6))
plt.plot(transpose_time, errnorm_transpose, label="Transpose", linestyle="-")
plt.plot(pseudoinverse_time, errnorm_pseudoinverse, label="Pseudoinverse", linestyle="-")
plt.plot(DLS_time, errnorm_DLS, label="DLS", linestyle="-")

# Figure design
plt.xlabel("Time")
plt.ylabel("Error Norm")
plt.title("Resolved rate motion control")
plt.legend()
plt.grid(True)
plt.show()
```

The Jacobian matrix for the end-effector of the robot is built from the function jacobian(T, revolute). And, the DLS function computes the damped least-squares solution to the matrix inverse problem. Both of these functions are located in the Common.py file.

----------Common.py----------

```
# Inverse kinematics
def jacobian(T, revolute):
    # 1. Initialize J and O.
    J = np.zeros((6, len(T)-1 ))  # Initialize the Jacobian matrix with zeros
    On = np.array([T[-1][:3, 3]]).T  # End-effector's origin (position)
    Z = np.array([[0, 0, 1]]).T  # Z-axis of the base frame
    # 2. For each joint of the robot
    for i in range(len(T)-1):
        #   a. Extract z and o.
        Ri = T[i][:3, :3]   # Extract the rotation matrix and origin from the transformation matrix
        Oi = np.array([T[i][:3, 3]]).T
        Zi = Ri @ Z   # Extract the z-axis from the rotation matrix
        #   b. Check joint type.
        #   c. Modify corresponding column of J.
        if revolute[i]:
            # For revolute joints, use the cross product of z and (O - O_i)
            J[:3, i] = np.cross(Zi.T, (On - Oi).T).T[:, 0]
            J[3:, i] = Zi[:, 0]
        else:
            # For prismatic joints, the linear velocity is along the z-axis, and angular velocity is zero
            J[:3, i] = Zi[:, 0]
    return J


# Damped Least-Squares
def DLS(A, damping):
    I = np.eye(2)
    DLS = np.linalg.inv(A.T @ A + damping**2 * I) @ A.T
return DLS
```

3. Results

The resolved-rate motion control algorithm was successfully implemented, and the robot's end-effector was controlled to follow the desired Cartesian position as shown in the figures below. Figure 4 illustrates the results of the resolved-rate motion control algorithm, presenting three different error control techniques. The first plot corresponds to the transpose method, the second to the pseudoinverse method, and the third to the damped least squares (DLS) method. Figure 5 demonstrates that the DLS approach significantly outperforms the Jacobian transpose method and performs similarly to the pseudoinverse method.

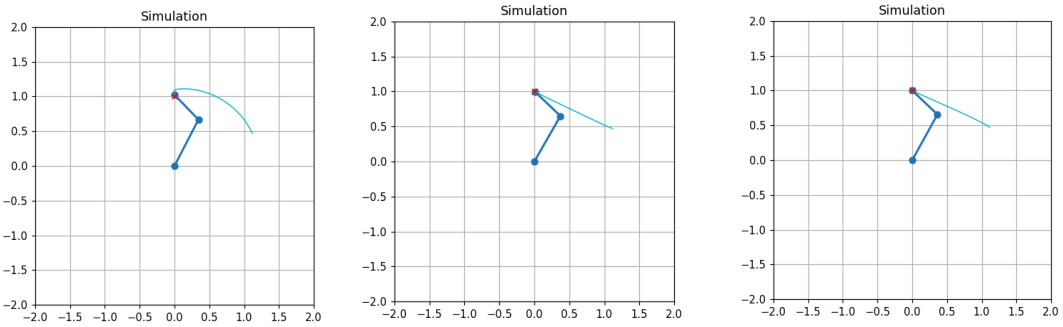

Figure 4: Simulation plots for Transpose, Pseudo-inverse and DLS methods

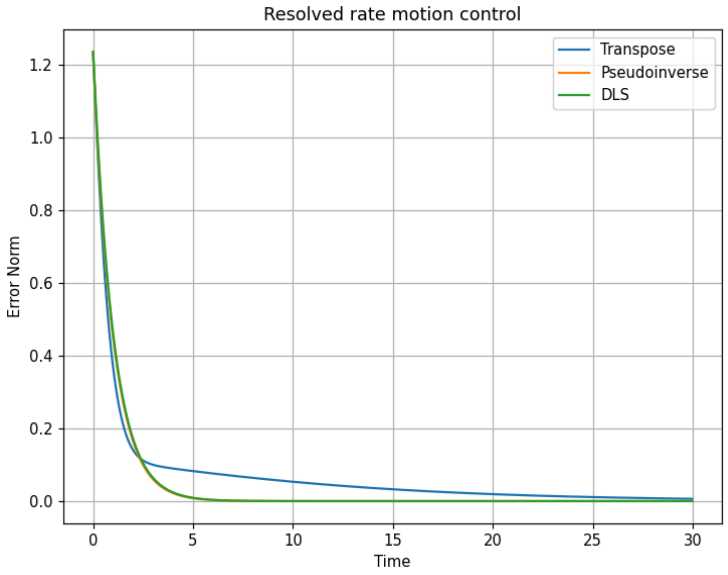

Figure 5: Control error over time

We can deduce that the pseudoinverse method and The DLS method performed well and provided the most accurate control, with the smallest control error norm. While the transpose method had the largest control error norm.

Questions

Q1. What are the advantages and disadvantages of using kinematic control in robotic systems?

Advantages:

- Kinematic control is effective for tasks that require the robot to follow a specific trajectory in space with accuracy [1].

- Kinematics is particularly beneficial for applications such as automotive assembly, where robots can be directed through tasks like simulated welding with the help of intuitive control software [1].

- Kinematic control deals with the movement of the robot without considering the forces and torques, making it simpler to implement [1].

Disadvantages:

- Unlike forward kinematics, inverse kinematics does not have a straightforward solution and requires appropriate techniques to solve the problem for any robot manipulator configuration [1].

- Kinematic control overlooks the forces and torques acting on the robot, which can cause problems in tasks that demand precise force application [1].

- When handling heavy payloads, the lack of dynamic considerations can result in inadequate control performance [1].

Q2. Give examples of control algorithms that may be used in the robot's hardware to follow the desired velocities of the robot's joints, being the output of the resolved-rate motion control algorithm.

The resolved-rate motion control algorithm computes the required joint velocities to achieve a desired end-effector motion. However, to accurately follow these computed velocities, the robot's hardware requires control algorithms that regulate joint movements, compensate for errors, and ensure smooth execution. Various control methods can be used, ranging from simple feedback-based controllers like PID control to more advanced predictive and model-based techniques. The following are some common control algorithms used in robotic hardware to achieve precise velocity tracking:

- *PID Control:* Uses proportional, integral, and derivative processes to minimize the error between desired and actual joint velocities in a closed-loop system [2].
- *Fuzzy Logic Control:* Applies fuzzy logic to manage imprecise information and make decisions based on a set of rules to control the robot's joints [2].
- *Model Predictive Control (MPC):* Utilizes a model of the robot to predict future states and optimize control inputs over a finite time horizon, ensuring the desired velocities are achieved [2].
- *Inverse Dynamics Control:* Calculates the necessary joint torques using the robot's dynamic model to follow the desired velocities [2].
- *Spatial Force Control:* Manages the spatial forces and moments acting on the robot's end-effector to maintain the desired motion [2].
- *Joint Stiffness Control:* Regulates joint stiffness to achieve desired compliance and stability, ensuring accurate joint velocities [2].

Conclusion

In this lab, we were able to implement the kinematic simulation of a planar robotic manipulator using the Denavit-Hartenberg formulation. The resolved-rate motion control algorithm was then applied to control the robot's end-effector. The simulations illustrated the motion of the robot and the performance of different control methods. The results showed that the DLS approach showed more stable outcomes. The lab also highlighted the advantages and disadvantages of using kinematic control in robotic systems and discussed various control algorithms that can be used to follow desired joint velocities.

References

[1] Meegle. (n.d.). Robot Kinematics. Meegle Topics: Robotics. Retrieved October 24, 2023, from https://www.meegle.com/en_us/topics/robotics/robot-kinematics
[2] Tedrake, R. (2024). *Robotic Manipulation: Perception, Planning, and Control*.
Course Notes for MIT 6.421. Retrieved from https://manipulation.csail.mit.edu/index.html