Student Name:
Mir Mohibullah Sazid [sazidarnob@gmail.com]
Group Member [Syma Afsha]

Lab2: Resolved-rate motion control

Introduction

The goal of the lab is to solve the two exercises that are given by implementing a kinematic simulation of a planar robotic manipulator. We need to implement a function to compute the Denavit-Hartenberg transforms and a function to update the position of the robot's links based on the transforms computed by the former. In the second exercise, we implement the resolved-rate motion control algorithm to control the robot we used in Exercise 1. The main tasks of this exercise include the implementation of the recursive computation of the geometrical Jacobian and the control feedback loop.

Methodology

At first, from the common file, D-H parameter is computed using the the following formula.

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

Then the function kinematic is implemented by generating a chain of transformation matrixes, and all the transformations are computed from the base of the robot.

In this exercise 1, I have used a two-degrees of freedom manupulator, which is shown in the figure.

In the given figure, the depicted robotic structure comprises two rotational joints. The framework is established through four separate coordinate systems: the base frame, two associated with each of the robotic joints, and one corresponding to the end-effector. The Denavit-Hartenberg parameters utilized in the algorithm are specified as follows: the lengths of the links along the x-axis are denoted by $a_1 = 0.75$ meters and $a_2 = 0.5$ meters. The offsets of the links along the z-axis are $d_1 = 0$ and $d_2 = 0$, respectively. The angles of twist about the X-axis are as $\alpha_1 = 0$ and $\alpha_2 = 0$

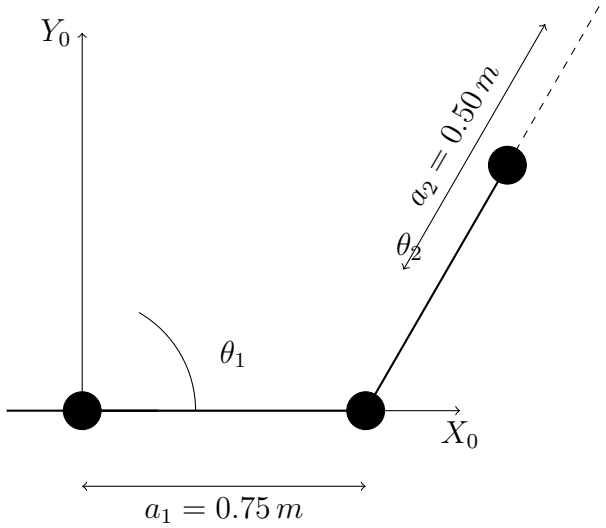


Figure 1: 2D schematic of a two-link robot arm.

For the excercise 2, at first from the common file the Jacobian are calculated which takes two parameter one is whether the robot join is revulote or prismatic and another is the value from kinematic function which gives the chain of transformation matrixcs. The function follows the algorithm 1:

Algorithm 1 Compute Jacobian Matrix

Input: DH parameters $\{d_i, \theta_i, a_i, \alpha_i\} \in \mathbb{R}^n$
Output: Jacobian matrix $J(q) = [J_1, J_2, \ldots, J_n]^T \in \mathbb{R}^{6\times n}$

Compute: $T_i^{i-1} = DH(d_i, \theta_i, a_i, \alpha_i), i = 1 \ldots n$
Compute: $T_i = T_i^0 = \prod_{k=1}^{i} T_k^{k-1}, i = 1 \ldots n$
Initialise: $z_0 = [0\ 0\ 1]^T, O_0 = [0\ 0\ 0]^T$

for $i = 1 \ldots n$ **do**
    $J_i = \begin{bmatrix} \rho_i z_{i-1} \times (O_n - O_{i-1}) \\ \rho_i z_{i-1} \end{bmatrix} + (1 - \rho_i) \begin{bmatrix} z_{i-1} \end{bmatrix}$
end for

return J

In here, I have also implemented DLS solution where the following formula is implemented. The damping factor is set to 0.1.

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

In the exercise 2 file, the formula for Pseudoinverse and Transpose is calculated using the formula shown below:

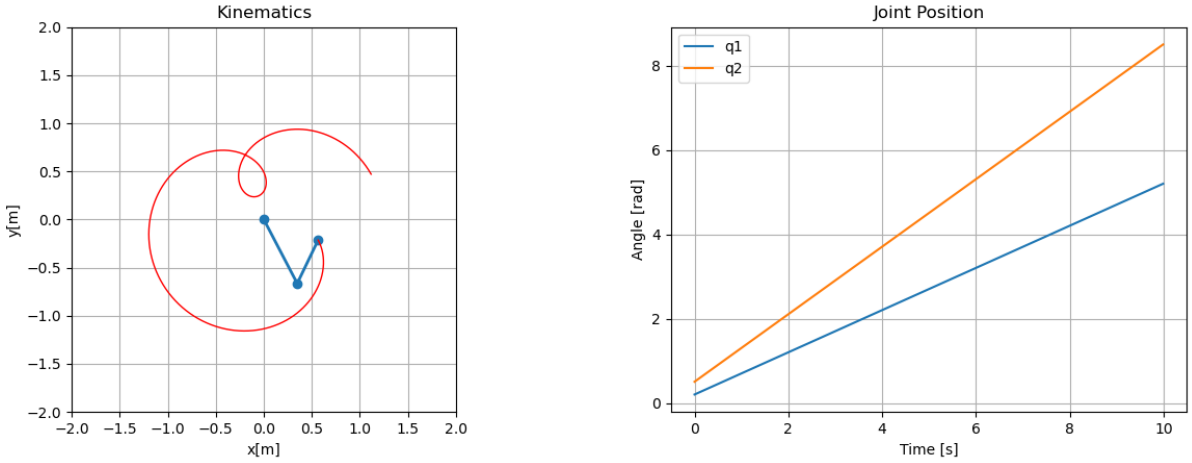
$$\zeta = J^T(q)\dot{x}_E$$

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

Then the controll error of three different solution is ploted to see the differences.

Results

In the first task, the robot arm's segments adjust their orientation after every interval using the transformations derived from the Denavit-Hartenberg method. Subsequently, an animation is created in Python to illustrate both the robot's configuration and the trajectory of the end-effector on a plane. Upon completion of the simulation, an additional graph is generated to show the progression of the joint angles as the simulation proceeded which is shown is figure 2. In this figure the joint velocity change overtime is given as 0.5 and 0.8



(a) Robot structure in motion

(b) Robot's joints positions over time

Figure 2: Robot motion and joints positions

Figure 3 displays the graphical outcome for the applied resolved-rate motion control algorithm, showcasing three distinct error control techniques. The illustration of figure 4 reveals that the damped least squares approach significantly outperforms the Jacobian transpose method, while it bears a close resemblance in performance to the pseudoinverse method.

Question and Answer

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

Advantages of Kinematic Control in Robotics:

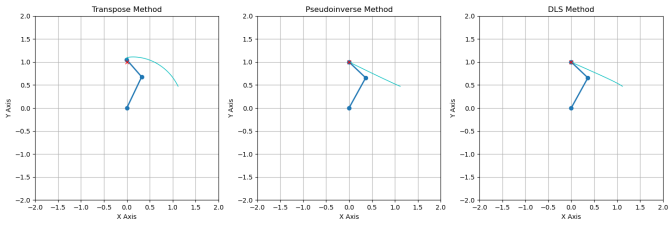


Figure 3: Simulation plots for Transpose, Pseudo-inverse and DLS methods

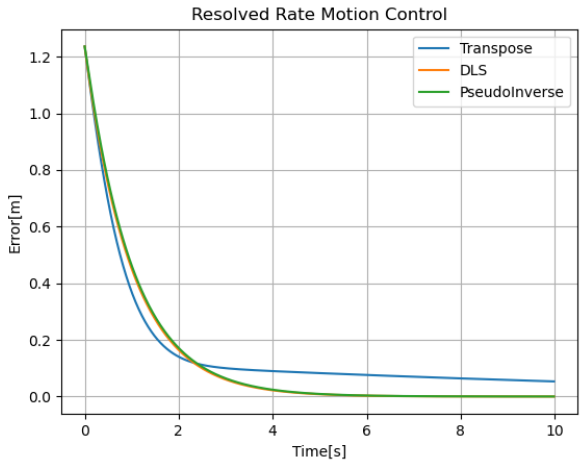


Figure 4: Control error over time

- **Simplified Commanding:** Allows for easier communication of desired movements to the robot by specifying start and end points, without needing to control every degree of freedom.
- **Flexibility in Path Following:** Effective for tasks where the robot needs to follow a specified path in space with precision.
- **Ease of Integration:** Beneficial for applications like automotive assembly where robots can be guided through tasks like simulated welding with informative control software.

Disadvantages of Kinematic Control in Robotics:

- **Complexity of Inverse Kinematics:** Unlike forward kinematics, the inverse kinematics lacks a straightforward or singular solution, necessitating the use of suitable techniques to resolve the problem for any robot manipulator configuration. Therefore, solving inverse kinematics or kinematic control problems poses significant challenges, both mathematically and computationally.
- **Ill-posed Nature of Inverse Kinematics:** The inverse kinematics problem can sometimes be ill-posed due to either the absence of a solution (when the target location is infeasible, such as being outside the reachable workspace) or the existence of multiple solutions (such

as encountering singularities). Additionally, certain kinematic control methods may result in abrupt movements when the end effector attempts to track a position beyond its range.

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.

Control algorithms are essential for enabling robot manipulators to follow the desired velocities of their joints, which are often the output of resolved-rate motion control algorithms. These algorithms ensure smooth, precise, and efficient movements. Below are examples of control algorithms commonly used for this purpose:

1. **Proportional-Integral-Derivative (PID) Control**: Widely used for its simplicity and effectiveness. It calculates an error value as the difference between a desired setpoint and a measured process variable, adjusting the process control inputs to minimize this error.

2. **Feedforward Control**: Uses a model of the system to predict the output for a given input and adjusts the control action accordingly. It is often used alongside PID control to handle predictable parts of the required control action.

3. **State Space Control**: Utilizes a mathematical model of the system to design a controller that brings the system's state (e.g., positions and velocities of the joints) to the desired state, considering interactions between different joints and actuators.

4. **Adaptive Control**: Adjusts its parameters in real-time to cope with changes in the robot's dynamics or the environment, adapting to changes in load, friction, or other factors.

5. **Robust Control**: Ensures that the control system remains effective in the presence of uncertainty and disturbances, maintaining precise joint velocity control despite modeling errors or unpredictable external forces.

6. **Model Predictive Control (MPC)**: Uses a model of the system to predict future states over a horizon and computes control actions by optimizing a cost function, considering constraints like joint limits and avoiding obstacles.

Each of these control algorithms has its strengths and can be chosen based on the specific requirements of the robotic system, such as the required precision, dynamics of the manipulator, and complexity of the tasks it performs. Often, a combination of these methods is used to achieve both high precision and robustness in following the desired trajectories.

Conclusion

The resolved-rate motion controller operates on the principles of Jacobian-based control, utilizing the Jacobian matrix in three primary ways to address the inverse kinematics challenge: through the Jacobian transpose (J^T), its inverse (J^{-1}), and the damped least squares method applied to the Jacobian (DLS(J)). Of these methods, the DLS approach tends to yield more consistent and stable

outcomes. A significant benefit of using the resolved-rate motion controller is its ability to ensure that the robotic arm transitions smoothly between waypoints, avoiding the abrupt movements typically associated with recalculating inverse kinematics at each step. Additionally, various control algorithms can be employed to accurately manage the desired velocities of the robot's joints.

Appendix

Common python code

```
import numpy as np

def DH(d, theta, a, alpha):
    '''
        Function builds elementary Denavit-Hartenberg transformation matrices a

        Arguments:
        d (double): displacement along Z-axis
        theta (double): rotation around Z-axis
        a (double): displacement along X-axis
        alpha (double): rotation around X-axis

        Returns:
        (Numpy array): composition of elementary DH transformations
    '''
    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]])

    Tz = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0],
                   [0, 0, 1, d],
                   [0, 0, 0, 1]])

    Tx = np.array([[1, 0, 0, a],
                   [0, 1, 0, 0],
                   [0, 0, 1, 0],
                   [0, 0, 0, 1]])

    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]])

    T = Rz @ Tz @ Tx @ Rx

    return T

def kinematics(d, theta, a, alpha):
    '''
        Functions builds a list of transformation matrices,
        for a kinematic chain, described by a given set of
        Denavit-Hartenberg parameters. All transformations
        are computed from the base frame.

        Arguments:
        d (list of double): list of displacements along Z-axis
        theta (list of double): list of rotations around Z-axis
        a (list of double): list of displacements along X-axis
        alpha (list of double): list of rotations around X-axis

        Returns:
        (list of Numpy array): list of transformations along the kinematic chai
    '''
    T = [np.eye(4)] # Base transformation

    for i in range(len(d)):
        T_current = DH(d[i], theta[i], a[i], alpha[i])
        T_accumulated = T[-1] @ T_current
        T.append(T_accumulated)

    return T

def jacobian(T, revolute):
    '''
        Function builds a Jacobian for the end-effector of
        a robot, described by a list of kinematic
        transformations and a list of joint types.

        Arguments:
        T (list of Numpy array): list of transformations
        along the kinematic chain of the robot (from the base frame)
        revolute (list of Bool): list of flags specifying if
        the corresponding joint is a revolute joint
```

```
        Returns:
        (Numpy array): end-effector Jacobian
    '''
    n = len(T)-1
    J = np.zeros((6, n))

    O = np.array([T[-1][:3, 3]]).T
    Z = np.array([[0, 0, 1]]).T

    for i in range(n):
        R_i = T[i][:3, :3]
        O_i = np.array([T[i][:3, 3]]).T
        Z_i = R_i @ Z

        if revolute[i]:
            J[:3, i] = np.cross(Z_i.T, (O - O_i).T).T[:, 0]
            J[3:, i] = Z_i[:, 0]
        else:
            J[:3, i] = Z_i[:, 0]

    return J

def DLS(J, damping):
    '''
        Function computes the damped least-squares (DLS)
        solution to the matrix inverse problem.

        Arguments:
        A (Numpy array): matrix to be inverted
        damping (double): damping factor

        Returns:
        (Numpy array): inversion of the input matrix
    '''
    I = np.eye(2)

    damped_J = np.linalg.inv(J.T @ J + damping**2 * I) @ J.T

    return damped_J

def robotPoints2D(T):
    '''
        Function extracts the characteristic points
```

```
        of a kinematic chain on a 2D plane, based
        on the list of transformations that describe it.

        Arguments:
        T (list of Numpy array): list of transformations
        along the kinematic chain of the robot
        (from the base frame)

        Returns:
        (Numpy array): an array of 2D points
    '''
    P = np.zeros((2,len(T)))
    for i in range(len(T)):
        P[:,i] = T[i][0:2,3]
    return P
```

Exercise 1

```
# Import necessary libraries
from lab2_robotics import * # Import our library (includes Numpy)
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import numpy as np

# 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_positions = []
q2_positions = []
# Simulation initialization
def init():
    line.set_data([], [])
    path.set_data([], [])
    return line, path


# Simulation loop
def simulate(t):
    global d, q, a, alpha
    global PPx, PPy, q1_positions, q2_positions


    # Update robot
    T = kinematics(d, q, a, alpha)
    dq = np.array([0.5,0.8]) # Define how joint velocity changes with time!
    q = q + dt * dq
     # Store joint positions
    q1_positions.append(q[0])
    q2_positions.append(q[1])



    # 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)

    return line, path

# Run simulation
```

```
animation = anim.FuncAnimation(fig, simulate, tt,
                               interval=10, blit=True,
                               init_func=init,
                               repeat=False)
plt.show()
# Ensure the lengths of time and position arrays are the same
min_length = min(len(tt), len(q1_positions), len(q2_positions))

tt = tt[:min_length]
q1_positions = q1_positions[:min_length]
q2_positions = q2_positions[:min_length]

# Now create the plots for joint positions over time
plt.figure()
plt.plot(tt, q1_positions, label='q1')  # Plot for q1
plt.plot(tt, q2_positions, label='q2')  # Plot for q2

# Add titles and labels
plt.title('Joint Position')
plt.xlabel('Time [s]')
plt.ylabel('Angle [rad]')
plt.legend()
plt.grid(True)


# Display the plot of joint positions
plt.show()
```

Exercise 2

```
# Import necessary libraries
from lab2_robotics import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim

# Robot definition
d = np.zeros(2)             # displacement along Z-axis
q1 = np.array([0.2, 0.5]) # rotation around Z-axis (theta)[used in Transpose]
q2 = np.array([0.2, 0.5]) # rotation around Z-axis (theta) [used in Pseudoinver
q3 = np.array([0.2, 0.5]) # rotation around Z-axis (theta) [used in DLS]
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([0.0, 1.0])
K = np.diag([1, 1])

# Simulation params
dt = 1.0/60
damping=0.1

# Drawing preparation
fig3, axs = plt.subplots(1, 3, figsize=(18, 5), constrained_layout=True)
titles = ['Transpose Method', 'Pseudoinverse Method', 'DLS Method']

for ax, title in zip(axs, titles):
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.set_aspect('equal')
    ax.grid()
    ax.set_title(title)  # Set the title for each subplot
    ax.set_xlabel('X Axis')  # Set the X-axis label for each subplot
    ax.set_ylabel('Y Axis')  # Set the Y-axis label for each subplot

# Initialize lines, paths, points for each subplot
lines = [ax.plot([], [], 'o-', lw=2)[0] for ax in axs]
paths = [ax.plot([], [], 'c-', lw=1)[0] for ax in axs]
points = [ax.plot([], [], 'rx')[0] for ax in axs]
PPx= [[], [], []]
PPy = [[], [], []]

dq_transpose_history = []
dq_dls_history = []
dq_pseudo_inverse_history = []
time_steps = []

# Simulation initialization function
def init():
    for line, path, point in zip(lines, paths, points):
        line.set_data([], [])
        path.set_data([], [])
        point.set_data([], [])
    return lines + paths + points

# Simulation loop
```

```
def simulate(t):
    global d, q1, q2, q3, a, alpha, revolute, sigma_d, K, dt
    global PPx, PPy, dq_dls_history,
    dq_pseudo_inverse_history, dq_transpose_history,
    time_steps, damping
    qs = [q1, q2, q3]  # List of joint values for each solution

    for i, q in enumerate(qs):
        print(i)
        print(q)

         # Update robot
        T = kinematics(d, q, a, alpha) #getting list of transformations
        J = jacobian(T, revolute)[:2,:] ##calculating jacobian
        and extracting 2x2 matrix

        # Update control
        sigma = T[-1][:2, 3]  # X,Y Position of the end-effector
        err = sigma_d - sigma # Control error

        # Calculate each control solution
        if i == 0:  # Transpose
            dq = J.T @ K @ err

            dq_transpose_history.append(np.linalg.norm(err))
        elif i == 1:  # Pseudoinverse
            dq = np.linalg.pinv(J) @ K @ err
            dq_dls_history.append(np.linalg.norm(err))
        else:  # DLS
            dq = DLS(J, damping) @ K @ err
            dq_pseudo_inverse_history.append(np.linalg.norm(err))


        # Update joint values
        qs[i] += dt * dq

        # Update drawing for each method
        P = robotPoints2D(T)
        lines[i].set_data(P[0, :], P[1, :])
        PPx[i].append(P[0, -1])
        PPy[i].append(P[1, -1])
        paths[i].set_data(PPx[i], PPy[i])
```

```
            points[i].set_data(sigma_d[0], sigma_d[1])

        time_steps.append(t)

        return lines + paths + points

# Create the animation
animation = anim.FuncAnimation(fig3, simulate, frames=np.arange(0, 10, dt),
                               init_func=init, blit=True, repeat=False)

plt.show()


# Plotting Evolution of error values over time
fig4 = plt.figure(4)
plt.plot(time_steps, dq_transpose_history, label='Transpose')
plt.plot(time_steps, dq_dls_history, label='DLS')
plt.plot(time_steps,  dq_pseudo_inverse_history, label='PseudoInverse')

plt.xlabel('Time[s]')
plt.ylabel('Error[m]')
plt.title('Resolved Rate Motion Control')

plt.legend()
plt.grid(True)
plt.show()
```