Students:
Tanakrit Lertcompeesin [u1992761@campus.udg.edu]
Samantha Caballero [u1992677@campus.udg.edu] (Lab partner)

Lab2: Resolved-rate motion control

Introduction

The laboratory aimed to explore resolved-rate motion control as a solution for robot manipulation. It was divided into two main parts. In the first part, we focused on implementing a simulation of a planar robot using the DH (Denavit-Hartenberg) algorithm to derive the transformation matrix of the robot. This simulation provided a virtual representation of the robot's kinematics. In the second part of the lab, we leveraged the simulated planar robot to implement various resolved-rate motion control solutions. These solutions were aimed at controlling the robot arm to reach desired positions.

Methodology

Exercise1

In this section, we conducted the planar robot simulation, visualizing the real-time movement of the robot via plotting. The implementation involved several steps: defining the robot description using DH parameters, calculating the transformation chain based on these parameters, and updating the parameters according to the desired velocity to achieve the desired movement.

DH parameters that used to describe robot's kinematic consist of four components:

- **d**: The distance along the z-axis from the current joint to the intersection of the x-axis.
- **q**: The angle about the z-axis from the current x-axis to the next x-axis.
- **a**: The distance along the x-axis from the current joint to the intersection of the z-axis.
- **alpha**: The angle about the x-axis from the current z-axis to the next z-axis.

The configuration used in this lab is depicted in figure 3, while the robot model corresponding to this configuration is illustrated in figure 2.

```
# 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
plot_limit = 100

# Memory
PPx = []
PPy = []
q1 = []
q2 = []
timestamp = []
```

Figure 1: Code for setting up simulation

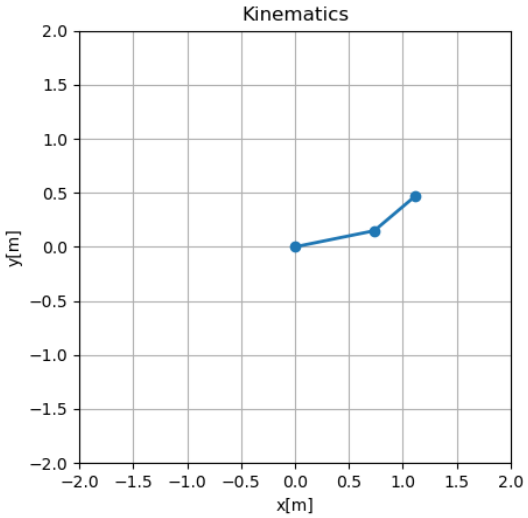


Figure 2: Robot model

To establish the kinematics chain illustrated in figure 2, it's necessary to compute the transformation matrix from the base frame to each joint. This computation was carried out by creating a function named ***kinematics()***, which takes the DH parameters and calculates the accumulated transformation by iteratively accumulating the transformation for each joint. This calculation is performed using equation 1 and implemented within another function called ***DH()***. The output array of the ***kinematics()*** function contains a list of transformations, where each index represents the transformation from base to the respective joint. The implementation for both functions are shown in figure .

$$T_i^{i-1} = \begin{bmatrix} \cos(q_i) & -\sin(q_i)\cos(\alpha_i) & \sin(q_i)\sin(\alpha_i) & a_i\cos(q_i) \\ \sin(q_i) & \cos(q_i)\cos(\alpha_i) & -\cos(q_i)\sin(\alpha_i) & a_i\sin(q_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

```python
def DH(d, theta, a, alpha):
    # 1. Build matrices representing elementary transformations (based on input
       parameters).
    # 2. Multiply matrices in the correct order (result in T).

    T = np.array([[math.cos(theta), -math.sin(theta)*math.cos(alpha),
        math.sin(theta)*math.sin(alpha), a*math.cos(theta)],
                  [math.sin(theta), math.cos(theta)*math.cos(alpha),
                      -math.cos(theta)*math.sin(alpha), a*math.sin(theta)],
                  [0, math.sin(alpha), math.cos(alpha), d],
                  [0, 0, 0, 1]])
    return T

def kinematics(d, theta, a, alpha):
    T = [np.eye(4)] # Base transformation
    # For each set of DH parameters:
    # 1. Compute the DH transformation matrix.
    # 2. Compute the resulting accumulated transformation from the base frame.
    # 3. Append the computed transformation to T.

    for i in range(len(d)):
        dh = DH(d[i],theta[i],a[i],alpha[i]) # Compute the DH transformation from joint
            i-1 to i.
        T.append(T[-1] @ dh) # Compute transformation from base to joint i by
            multiplying the lastest trasformation
    return T
```

Figure 3: Implemention of ***DH()*** and ***kinematics()***

To visualize the movement, we developed the ***simulation()*** function, triggered by ***FuncAnimation*** from Matplotlib for real-time visualization. Within each iteration, we compute the transformation matrix from the current configuration and update the joint positions ($\mathbf{q}$) based on the desired velocity ($\mathbf{dq}$). Afterwards, we plot the trajectory of the end effector by extracting its position from the latest element of the transformation list. The implementation is illustrated in figure 4.

```
# 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

    # Update robot
    T = kinematics(d, q, a, alpha)
    dq = np.array([0,0]) # Define how joint velocity changes with time!
    q = q + dt * dq

    # 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)

    # Update memory for plotting
    q1.append(q[0])
    q2.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)
```

Figure 4: Code for simulation

After the simulation concludes, we summarize the results by plotting the evolution of each joint position. This is achieved through the implementation of the ***plot_summary()*** function, as illustrated in figure 5.

```
def plot_summary():
    # Evolution of joint positions Plotting
    fig_joint = plt.figure()
    ax = fig_joint.add_subplot(111, autoscale_on=False,xlim=(0, 10), ylim=(0,10))
    ax.set_title('joint positions')
    ax.set_xlabel('Time[s]')
    ax.set_ylabel('Angle[rad]')
    ax.grid()
    plt.plot(timestamp,q1,label='q1')
    plt.plot(timestamp,q2,label='q2')
    ax.legend()

    plt.show()
```

Figure 5: Code for plotting joints position

Exercise2

In this section, we employed the simulation developed in a later exercise to implement resolved-rate motion control for the manipulator to reach the desired position. We utilized the robot model defined in Figure 2.

The equation for resolved-rate motion control, shown in equation , calculates the required joint velocities ($\dot{q}$) to achieve the desired end-effector velocities ($\dot{x}$) by multiplying the control matrix (J^{-1}) with the desired end-effector velocities.

$$\dot{q} = J^{-1} \cdot \dot{x} \tag{2}$$

The Jacobian matrix (J) is computed using the function ***jacobian()***. We extract information from the transformation matrix to compute the Jacobian, with each column calculated depending on the joint type (revolute or prismatic). The equation for each Jacobian column is shown in equation 2, and the implementation is depicted in figure 6.

$$J_i = \begin{cases} \begin{bmatrix} \boldsymbol{z}_{i-1} \times (\boldsymbol{p}_n - \boldsymbol{p}_{i-1}) \\ \boldsymbol{z}_{i-1} \end{bmatrix} & \text{for revolute joint} \\ \begin{bmatrix} \boldsymbol{z}_{i-1} \\ \mathbf{0} \end{bmatrix} & \text{for prismatic joint} \end{cases} \tag{3}$$

```python
def jacobian(T, revolute):
    # 1. Initialize J and O.
    # 2. For each joint of the robot
    #   a. Extract z and o.
    #   b. Check joint type.
    #   c. Modify corresponding column of J.
    J = np.zeros((6,len(T)-1)) # number of columns is number of joints
    for i in range(0,len(T)-1): # for each joint
        # extract z and 0 from T
        z = np.array(T[i][0:3,2])
        o = np.array(T[-1][0:3,3] - T[i][0:3,3])

        if revolute[i]:
            J[:,i] = np.concatenate((np.cross(z,o),z)).reshape((1,6)) # a column for
                revolute joint
        else:
            J[:,i] = np.concatenate((np.cross(z,o),np.zeros((3,1)) )).reshape((1,6)) # a
                column for prismatic joint
    return J
```

Figure 6: Implementation of ***jacobian()***

To determine the inversion of the matrix, referred to as the controller matrix, three methods were implemented: transpose, pseudoinverse, and Damped Least Squares (DLS). The function ***controller()*** in figure 7 was developed to return the control matrix based on the selected type. For the transpose method, it involves taking the transpose of the JJ matrix. The pseudoinverse method calculates the inverse of the non-square matrix using the `np.linalg.pinv()` function. For DLS, the equation shown in equation 4 was implemented in the function ***DLS()***.

$$DLS = (J^T J + \lambda^2 I)^{-1} J^T \tag{4}$$

```python
def controller(type, J):
    # return the controller matrix according to the controller type
    if type == "transpose":
        return J.T
    elif type == "inverse":
        return np.linalg.pinv(J)
    elif type == "DLS":
        return DLS(J,0.1)
    else:
        print("Invalid controller type!")
        return J.T
```

Figure 7: Implementation of ***controller()***

During each simulation iteration, the desired joint velocity was adjusted according to the resolved-rate equation. The desired end-effector velocities were computed from the error between the desired position and the current position, multiplied by the controller gain (K). The modification in the simulation step is shown in figure 8.
As illustrated in figure 8, the dimension of each parameter needed to be increased to keep track of the information for each type of controller. Additionally, the norm error of the end-effector position was recorded and plotted after the simulation concluded, as shown in the plotting function in figure 9.

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

    # Update robot state for each controller type
    T = [kinematics(d, q[i], a, alpha) for i in range(len(controller_list))]
    J = [jacobian(T[i], revolute) for i in range(len(controller_list))]
    P = [robotPoints2D(T[i]) for i in range(len(controller_list))]

    # Update control
    sigma = [np.array([P[i][0,-1], P[i][1,-1]]) for i in range(len(controller_list))] #
        Position of the end-effector for each solution
    err =  [sigma_d - sigma[i] for i in range(len(controller_list))] # Control error
        (position error)


    # Control solutions
    dq = [controller(controller_list[i], J[i][:2,:]) @ (K @ err[i]) for i in
        range(len(controller_list)) ] # calculate the velocity command
                            # controller matrix needed slicing to match the
                                end-effector dimensions
    for i in range(len(controller_list)):
        q[i] += dt * dq[i]
        #Error Poltting Elements
        norm_err[i].append(np.linalg.norm(err[i])) # Norm error for each controller

    timestamp.append(t)

    # Update drawing (Only show the first solution [0])
    P = robotPoints2D(T[0])
    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
```

Figure 8: Modification in ***simulation()*** for Exercise2

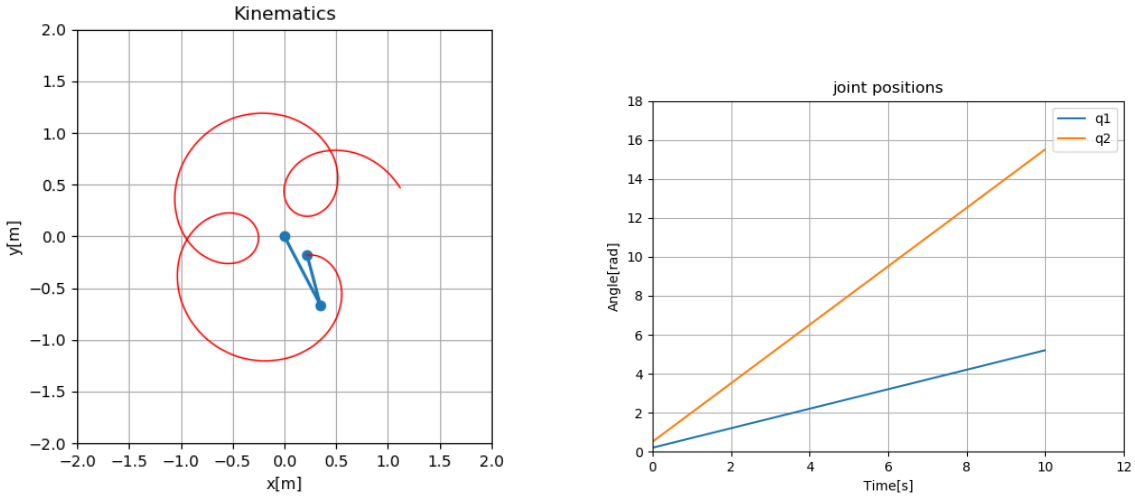
```
def plot_summary():
    # Evolution of joint positions Plotting
    fig_joint = plt.figure()
    ax = fig_joint.add_subplot(111, autoscale_on=False, xlim=(0, 10), ylim=(0,1.1))
    ax.set_title('joint positions')
    ax.set_xlabel('Time[s]')
    ax.set_ylabel('Error[m]')
    ax.grid()
    plt.plot(timestamp,norm_err[0],label='transpose')
    plt.plot(timestamp,norm_err[1],label='psudoinverse')
    plt.plot(timestamp,norm_err[2],label='DLS')

    ax.legend()

    plt.show()
```

Figure 9: Implementation of ***plot_ summary()*** for Exercise2

Results & Discussion

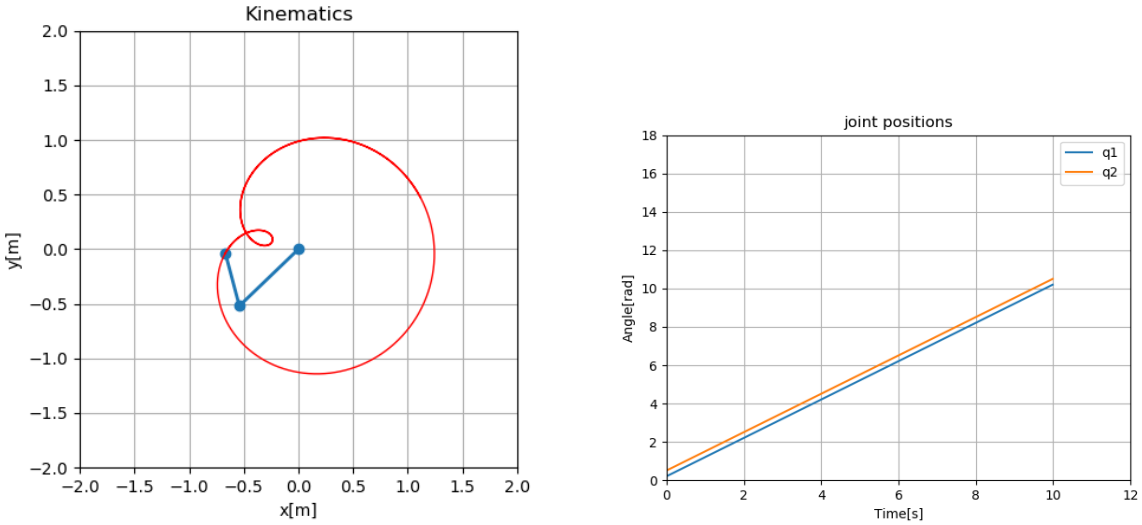
Exercise1

To assess the simulation, we varied the joint velocities to observe different movements of the manipulator. The results are depicted in figure 10, 11 and 12.



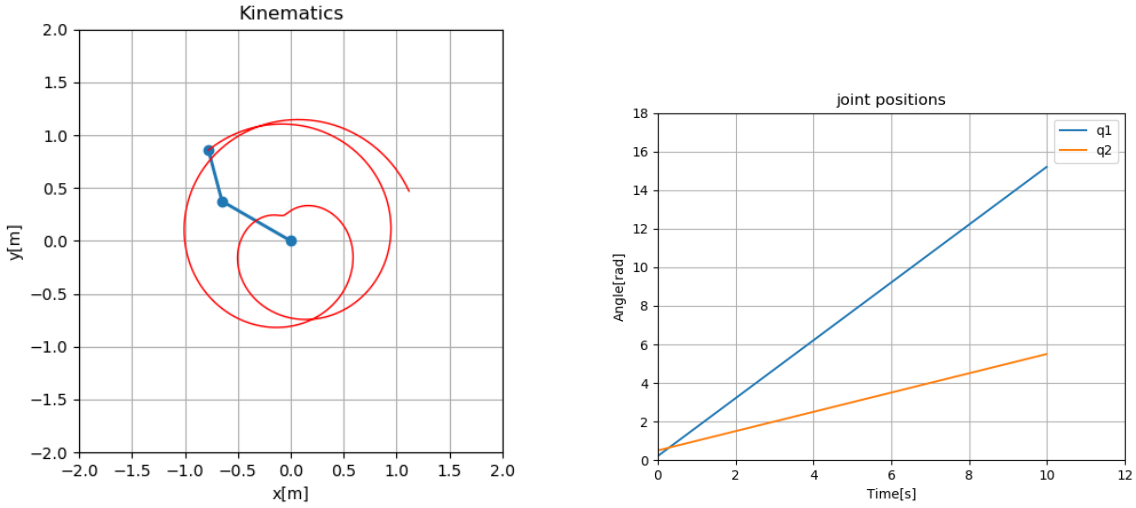
(a) (b)

Figure 10: **dq** = [0.5, 1.5]



(a) (b)

Figure 11: **dq** = [1, 1]



(a) (b)

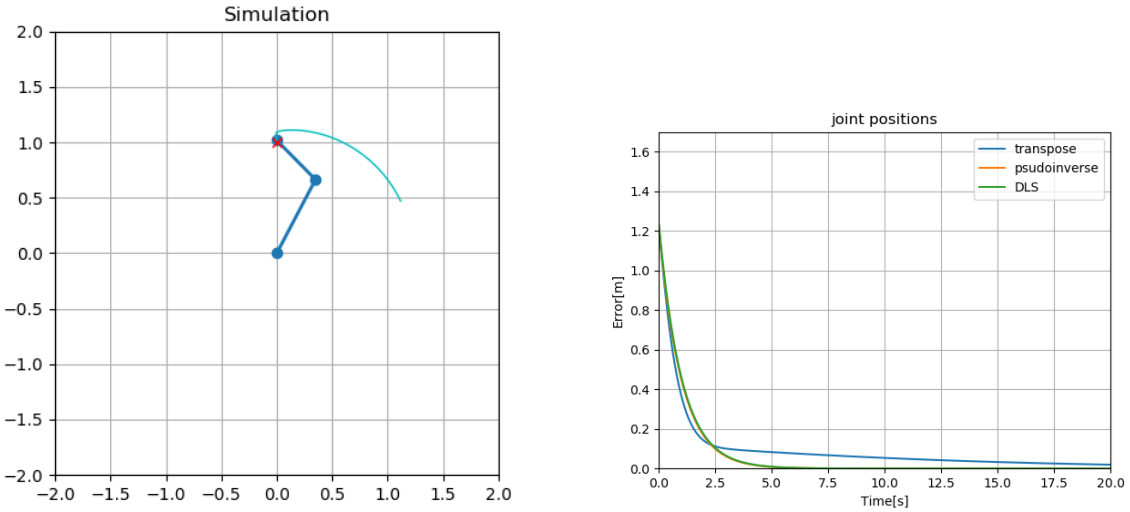
Figure 12: **dq** = [1.5, 0.5]

The joint position plot, depicted in figure 10b, 11b and 12b indicates that each joint operates independently. The rate of angle increase is dependent on the velocity assigned to each joint.

Exercise2

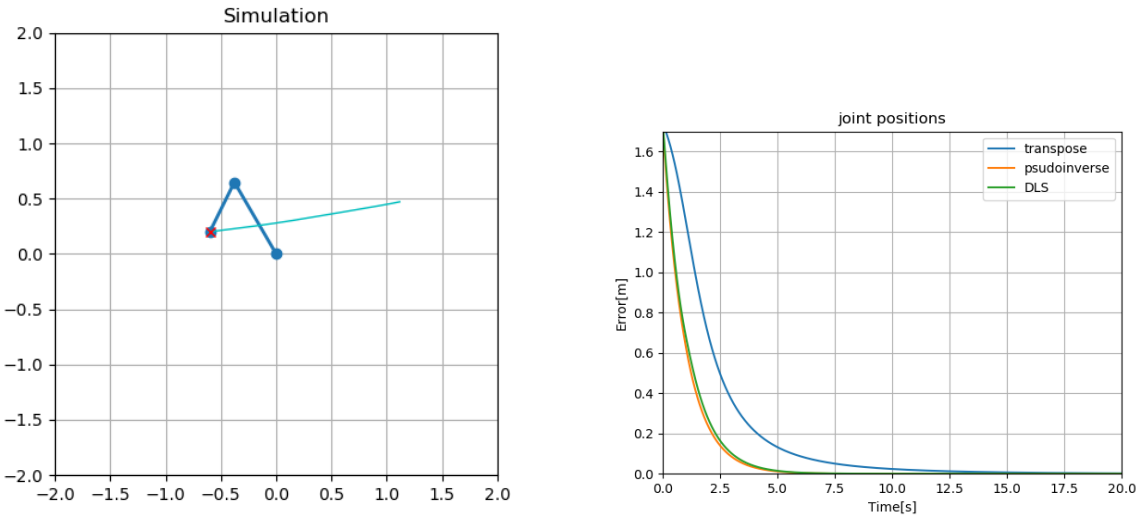
To evaluate and compare each control algorithm, we provided different desired positions and simultaneously computed outputs for all three controllers. The movement paths shown in figure

13a, 14a and 15a are from transpose, DLS and psudoinverse respectively, but the norm error of all controllers is depicted in a single plot.



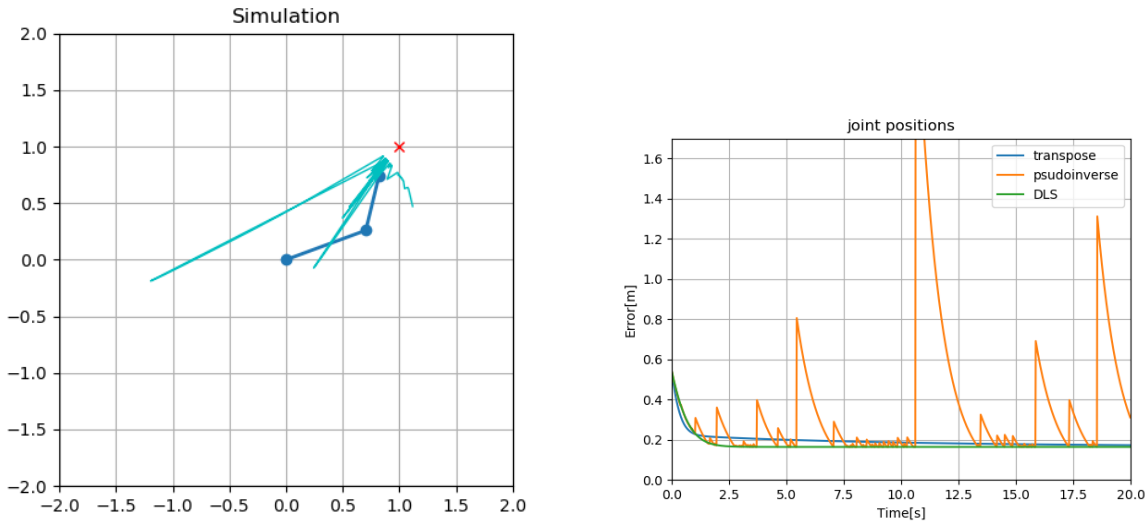
(a) (b)

Figure 13: **sigma_d** = [0, 1] , "transpose"



(a) (b)

Figure 14: **sigma_d** = [-0.6, 0.2], "DLS"



(a) (b)

Figure 15: **sigma_d** = [1, 1], psudoinverse

The norm error plot serves as an indicator of the performance of each control solution. From figure 13 and 14, it is evident that the trend of the Damped Least Squares (DLS) method closely resembles that of the pseudoinverse method which can produce a straight path to the goal, indicating a good estimation of the inverse Jacobian for the DLS method. In contrast, the transpose method, as also witnessed in the curve path in figure 13a, exhibits a different trend. This discrepancy can be attributed to the limitations of the transpose method in providing an accurate estimation.

However, when we set the desired position outside the workspace 15, the pseudoinverse method encounters singularities, leading to wrong computations for. Therefore, the Damped Least Squares (DLS) is the only method provides both a highly accurate estimation and is not suffer to singularities.

Questions

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

Kinematics control manages the position, velocity, or acceleration of a manipulator by using the known relationships between the end-effector and its joints. It's simpler than dynamic and force control, which need accurate system models. This simplicity is kinematic control's main advantage.
However, because of this, kinematic control still needs low-level controllers to adjust actuators for desired values. Also, due to limited system knowledge, some commands may be unachievable. Another issue is that, being a high-level controller, it operates at low frequencies, making the system more prone to oscillations and instability.

"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 output of the resolved-rate motion control algorithm is velocity for each joint. Therefore, the low-level alogrithm need to be able to control the velocity of the harware.

- **PID Controller**: The most commonly used control algorithm for controlling the velocity of a motor. The motor velocity, typically obtained from sensors like encoders, serves as input to the PID controller. By adjusting the PID gains, we can tune the controller's performance to meet desired specifications.

- **Cascade Design Controller**: In the case of a DC motor, a cascade design controller can be implemented. This involves modeling and tuning multiple control loops sequentially. Typically, a PI controller is used in the torque loop, followed by another PI controller in the velocity loop.