Students:
Loc Thanh Pham [u1992429@campus.udg.edu]

Lab2: Resolved-rate motion control

Introduction

This lab sessions is focused on implementing the resolved-rate motion control algorithm and understanding its properties using a simple simulation of a planar manipulator.

Methodology

Exercise 01: Kinematic simulation of a planar robotic manipulator

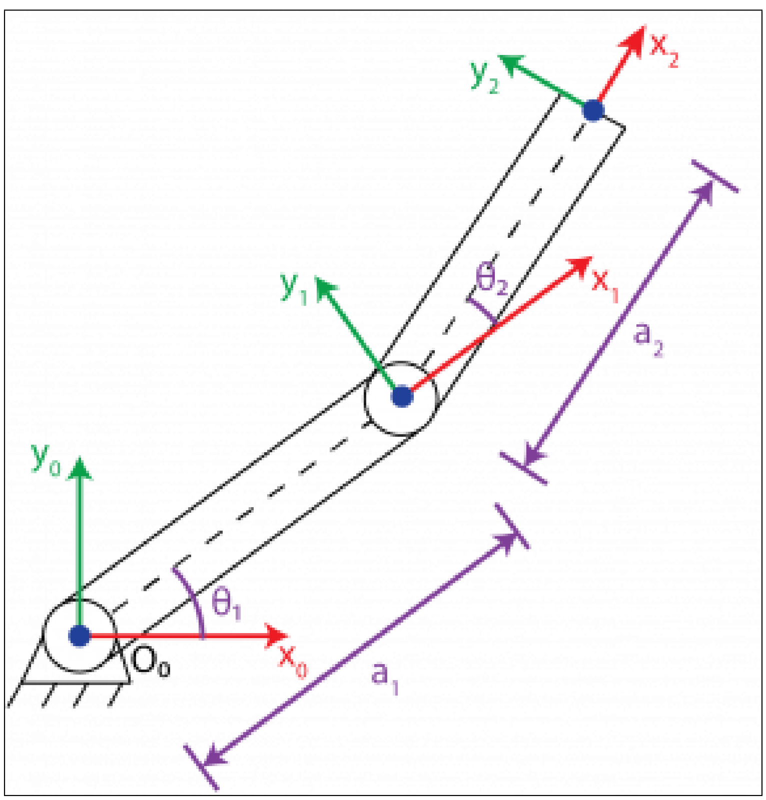

Figure 1: Planar manipulator model, including DH parameters and coordinate systems

Link	**d**	θ	a	α	**Home**
1	0	θ_1	a_1	0	0
2	0	θ_2	a_2	0	0

Table 1: Denavit-Hartenberg table of the planar manipulator

Denavit-Hartenberg formulation which is used to compute transformation matrix between coordinate systems:

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

$$T_n^0(q) = T_1^0 T_2^1 T_3^2 \ldots T_n^{n-1} = \begin{bmatrix} R_n^0(q) & O_n^0(q) \\ 0 & 1 \end{bmatrix} \tag{2}$$

Implementation on python code:

Listing 1: **Denavit-Hartenberg formulation**

```python
def DH(d, theta, a, alpha):
    '''
        Function builds elementary Denavit-Hartenberg transformation matrices
        and returns the transformation matrix resulting from their multiplication.

        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
    '''

    # Calculate trigonometric values
    cos_theta = np.cos(theta)
    sin_theta = np.sin(theta)
    cos_alpha = np.cos(alpha)
    sin_alpha = np.sin(alpha)
```

```python

    # 1. Build matrices representing elementary transformations (based on input parameters).
    T1 = np.array([
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, d],
        [0, 0, 0, 1]
    ])

    T2 = np.array([
        [cos_theta, -sin_theta, 0, 0],
        [sin_theta, cos_theta,  0, 0],
        [0,         0,          1, 0],
        [0,         0,          0, 1]
    ])

    T3 = np.array([
        [1, 0, 0, a],
        [0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])

    T4 = np.array([
        [1, 0,          0,          0],
        [0, cos_alpha, -sin_alpha,  0],
        [0, sin_alpha,  cos_alpha,  0],
        [0, 0,          0,          1]
    ])

    # 2. Multiply matrices in the correct order (result in T).
    T = T1@T2@T3@T4

    return T
```

Listing 2: **Kinematics function**

```python
def kinematics(d, theta, a, alpha):
    '''
        Functions builds a list of transformation matrices, for a kinematic chain,
        descried 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 chain (from the base frame)
    '''
    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.
    N_order = len(d)
    for index in range(N_order):
        T.append(T[index] @ DH(d[index], theta[index], a[index], alpha[
    index]))

    return T
```

Listing 3: **Main function**

```
# Simulation initialization
def init():
    line.set_data([], [])
    path.set_data([], [])
    q1.set_data([], [])
    q2.set_data([], [])
    return line, path, q1, q2

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

    # Update robot
    T = kinematics(d, q, a, alpha)
    dq = np.array([0.1, 0.3]) # Define how joint velocity changes with
    time!
    q[0] += dt * dq[0]
    q[1] += dt * dq[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)

    time_store.append(t)
    q1_store.append(q[0])
    q1.set_data(time_store, q1_store)
    q2_store.append(q[1])
    q2.set_data(time_store, q2_store)
    return line, path, q1, q2

# Run simulation
animation = anim.FuncAnimation(fig1, simulate, tt,
                                interval=10, blit=True, init_func=init,
    repeat=False)
```

Listing 4: **Simulation parameters and Figures setup**

```
# 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 = 0.1 # Sampling time
Tt = 80 # Total simulation time
tt = np.arange(0, Tt, dt) # Simulation time vector

# Drawing preparation the visualisation of the robot structure in motion
fig1 = plt.figure()
ax1 = fig1.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2,2)
    )
line, = ax1.plot([], [], 'o-', lw=2) # Robot structure
path, = ax1.plot([], [], 'r-', lw=1) # End-effector path
ax1.set_title('Kinematics')
ax1.set_xlabel('x[m]')
ax1.set_ylabel('y[m]')
ax1.set_aspect('equal')
ax1.grid()


```

```
# Drawing preparation the evolution of robots joints positions over time
fig2 = plt.figure()
ax2 = fig2.add_subplot(111, autoscale_on=False)
q1, = ax2.plot([], [], 'b-', label='q1') #
q2, = ax2.plot([], [], 'r-', label='q2') #
ax2.set_title('Joint position')
ax2.set_xlabel('Time[s]')
ax2.set_ylabel('Angel[rad]')
ax2.set_xlim(0,Tt)
ax2.set_ylim(0,30)
ax2.legend()
ax2.grid()

# Memory
PPx = []
PPy = []
q1_store = []
q2_store = []
time_store = []
```

Exercise 02: Resolved-rate motion control algorithm

In this exercise, I implemented the resolved-rate motion control algorithm, to control the robot simulated in Exercise 1. It includes implementations of the recursive computation of geometrical Jacobian and the control feedback loop. The base for this exercise is the code of Exercise 1.

Implementation on python code:

Listing 5: **Jacobian function**

```
# Inverse kinematics
def jacobian(T, revolute, a, theta):
    '''
        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
    '''
    # 1. Initialize J and O.
    J = np.zeros((6,len(revolute)))
    z_pre = T[0][0:3,2].reshape(1,3)
    o_pre = T[0][0:3,3].reshape(1,3)

    o_n = T[-1][0:3,3].reshape(1,3)

    # 2. For each joint of the robot
    for index in range(1,len(T)):
        #   a. Extract z and o.
        z = T[index][0:3,2].reshape(1,3)
        o = T[index][0:3,3].reshape(1,3)
        #   b. Check joint type.
        #   c. Modify corresponding column of J.
        J[:,index-1] = np.block([int(revolute[index-1])*np.cross(z_pre,
   o_n - o_pre) + (1 - int(revolute[index-1]))*z_pre, int(revolute[index
   -1])*z_pre])
        #   d. Set z and o for next joint
        z_pre = z
        o_pre = o

    return J
```

Listing 6: **Extract characteristic points of a robot projected on X-Y plane**

```
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
    '''
    # Init P
    P = np.zeros((2,len(T)))
    for i in range(len(T)):
        # Get P from transformation matrix
        P[:,i] = T[i][0:2,3]
```

```
    return P
```

Listing 7: **DLS function**

```
# Damped Least-Squares
def DLS(A, 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
    '''
    return A.T @ np.linalg.inv(A @ A.T + damping**2)
```

Listing 8: **Main function**

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

    # Update robot
    T = kinematics(d, q, a, alpha)
    J = jacobian(T, revolute, a, q)

    # Update control
    P = robotPoints2D(T)
    sigma = np.array([P[0,-1], P[1,-1]])          # Position of the end-effector
    err = sigma_d - sigma                          # Control error

    # Choose controller type
    # 0. Transpose
    # 1. Preudoinverse
    # 2. DLS
    if control_type == 0:
        dq = (J[0:2,0:2].T @ err.T).T
    elif control_type == 1:
        dq = (np.linalg.inv(J[0:2,0:2]) @ err.T).T
    else:
```

```
        dq = (DLS(J[0:2,0:2],0.2) @ err.T).T

    q += dt * dq

    # Store data to plot
    time_store.append(t)
    err_norm = np.linalg.norm(err)
    EE.append(err_norm)
    E.set_data(time_store, EE)
    # Write to .txt file
    f1.write(str(t) + '\n')
    f2.write(str(err_norm) + '\n')

    # Update drawing
    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])
    if t == 10:
        f1.close()
        f2.close()
    return line, path, point, E
```

Exercise 03: Questions and Answers

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

Advantages:

- Simplicity: Kinematic control relies on mathematical kinematic models that describe the relationship between joint motions and end-effector positions. This simplicity makes it easier to implement and understand compared to more complex dynamic control methods.
- Efficiency: Since kinematic control doesn't consider dynamic effects like inertia and friction, it can be computationally more efficient than dynamic control methods. This efficiency is especially beneficial in real-time applications where quick responses are required.
- Accuracy: In situations where dynamic effects are negligible or can be compensated for, kinematic control can provide precise control over the robot's end-effector position and orientation.

- Predictability: Kinematic control provides a predictable behavior of the robot's motion, which simplifies trajectory planning and control design.

Disadvantages:

- Limited to Static Environments: Kinematic control assumes static or slowly changing environments. It doesn't account for dynamic interactions such as collisions, which can lead to inaccuracies and potential safety hazards in dynamic environments.
- Singularities: Kinematic control can encounter singularities, where the robot loses degrees of freedom or exhibits erratic behavior. Handling singularities requires careful design and control strategies.
- Limited Manipulability: Kinematic control might not fully exploit the robot's manipulability, especially in redundant robotic systems where there are more degrees of freedom than necessary to perform a task. Dynamic control methods can better utilize redundancy to optimize performance.
- Difficulty in Handling Uncertainties: Kinematic control assumes perfect knowledge of the robot's geometry and kinematics, as well as the environment. Uncertainties in these parameters can lead to inaccuracies in control, requiring robustness measures or adaptive control strategies.

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.

- Proportional-Integral-Derivative (PID) Control: PID control extends PD control by incorporating an integral term that integrates the error over time. This helps to eliminate steady-state errors and improve the system's response to disturbances. PID control is widely used due to its effectiveness and simplicity.
- Sliding Mode Control (SMC): SMC is a nonlinear control technique that aims to drive the system states onto a predefined sliding surface, where the system dynamics become simpler to control. SMC is robust to uncertainties and disturbances and can provide accurate tracking performance.
- Adaptive Control: Adaptive control algorithms adjust control parameters online based on the system's performance and changes in its dynamics. Adaptive control can improve the robustness of the system by adapting to variations in the system parameters or external disturbances.

Result

Exercise 1:

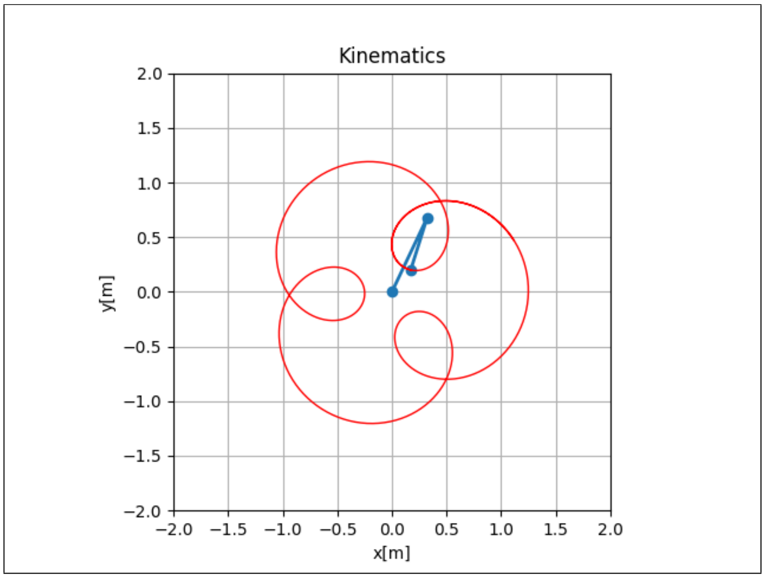

Figure 2: The visualisation of the robot structure in motion

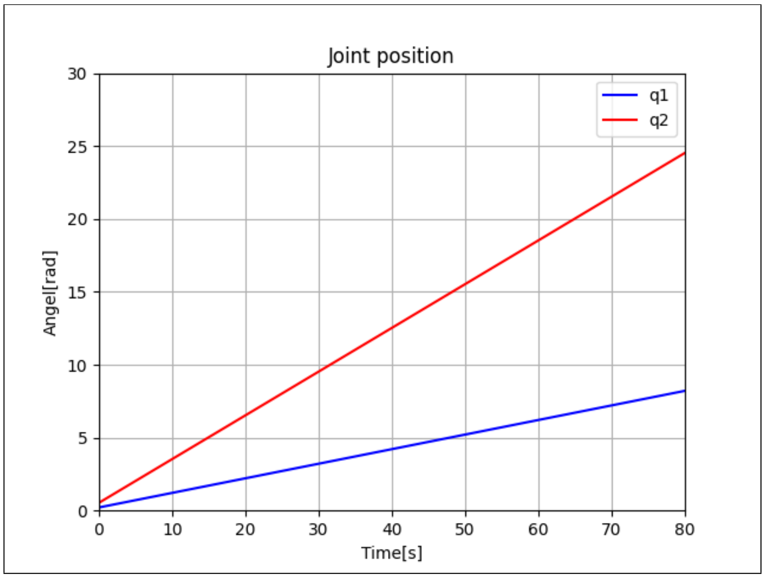

Figure 3: The evolution of robot's joints positions over time

Exercise 2:

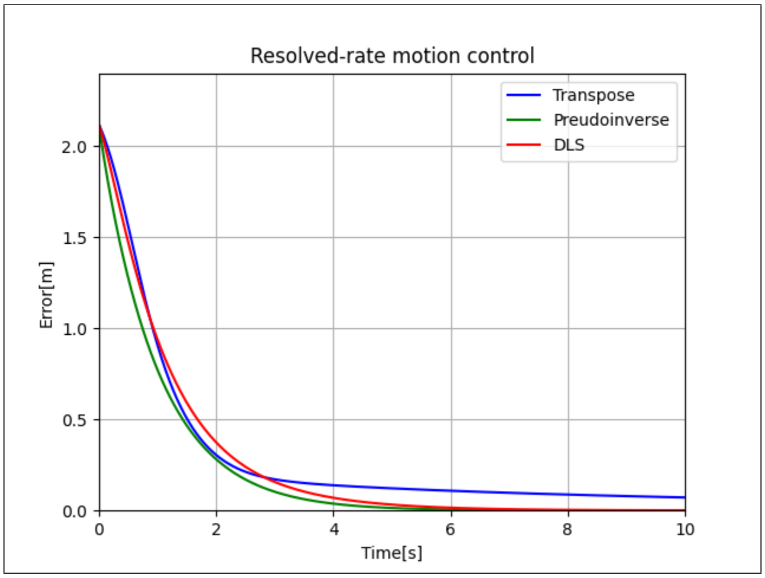

Figure 4: the evolution of the control error norm over time, for all three methods used to solve the control problem