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

Lab3: Task-Priority kinematic control

Introduction

This lab session in focused on understanding the null space concept and the Task-Priority algorithm in its analytical form.

Methodology

Exercise 01: Kinematic simulation and Null space of a three links manipulator

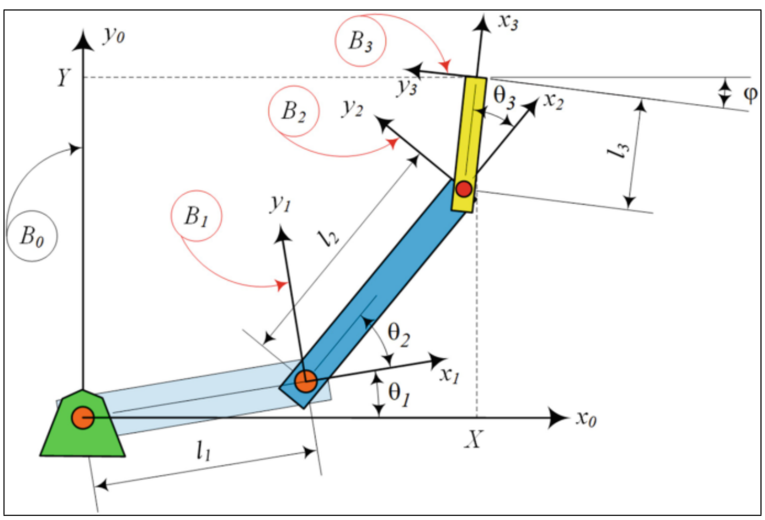

Figure 1: Three links manipulator model, including DH parameters and coordinate systems

Link	**d**	θ	a	α	**Home**
1	0	θ_1	a_1	0	0
2	0	θ_2	a_2	0	0
3	0	θ_3	a_3	0	0

Table 1: Denavit-Hartenberg table of the three links manipulator

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

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

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

Implementation on python code:

Listing 1: **Simulation Loop**

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

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

    # Update control
    P            = robotPoints2D(T)
    # Current position of the end-effector
    sigma        = np.array([P[0,-1], P[1,-1]])
    # Error in position
    err          = sigma_d - sigma
    # Task Jacobian
    Jbar         = J[0:2,0:n_DoF]
    # Null space projector
    P            = np.eye(3,3) - DLS(Jbar, 0.0) @ Jbar
     # Arbitrary joint velocity
    y            = np.array([-5 + 10 * np.sin(0.5 * t),
                             -5 + 10 * np.sin(0.1 * t),
                             -5 + 10 * np.sin(1.0 * t)])
                             # Control signal
    dq           = DLS(Jbar,0.2) @ err + P @ y
    # Simulation update
    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)
    point.set_data(sigma_d[0], sigma_d[1])

    time_store.append(t)
    q1_store.append(normalize_angle(q[0]))
    q1.set_data(time_store, q1_store)
    q2_store.append(normalize_angle(q[1]))
    q2.set_data(time_store, q2_store)
    q3_store.append(normalize_angle(q[2]))
    q3.set_data(time_store, q3_store)

    return line, path, point, q1, q2, q3
```

Listing 2: **Robot definition (3 revolute joint planar manipulator)**

```
# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3)                    # displacement along Z-axis
q = np.array([0.2, 0.5, 0.2])    # rotation around Z-axis (theta)
a = np.array([0.75, 0.5, 0.3])   # displacement along X-axis
alpha = np.zeros(3)                # rotation around X-axis
revolute = [True, True, True]     # flags specifying the type of joints
n_DoF = len(revolute)              # Number of Degree of Freedom
dq_max = np.array([3, 3, 3]) # The maximum joint velocity limit
# Setting desired position of end-effector to the current one
sigma_d = np.array([1.0, 1.0])
```

Exercise 02: Task-Priority control algorithm for a hierarchy of two tasks

In this exercise, I implemented the Task-Priority control algorithm for a hierarchy of two tasks (using the analytic solution), to control the manipulator simulated in Exercise 1. The main tasks of this exercise include definition of task Jacobians and errors and implementation of the control loop.

Implementation on python code:

Listing 3: **Simulation Loop: End-effctor position task at the top of the hierarchy**

```
# Simulation loop
def simulate(t):
    global q, a, d, alpha, revolute, dq_max, sigma1_d, sigma2_d
    global PPx, PPy
```

```
    global time, N_iter, Tt

    # Set new desired end-effector position and joint 1 position at beginning of each 10s
    if t == 0:
        # Set random new desired position
        theta_rand = 2 * math.pi * np.random.rand()
        length_rand = sum(a) * np.random.rand()
        # Position of the end-effector
        sigma1_d = np.array([length_rand * np.cos(theta_rand), length_rand * np.sin(theta_rand)])    # Position of joint 1
        sigma2_d = np.array([0.0])
        # Set number of iteration
        N_iter += 1

    # Update robot
    T       = kinematics(d, q.flatten(), a, alpha)
    J       = jacobian(T, revolute)
    Probot  = robotPoints2D(T)

    # Update control
    # TASK 1: Position of End-Effector
    # Current position of the end-effector
    sigma1      = np.array([Probot[0,-1], Probot[1,-1]])
    # Error in Cartesian position
    err1        = sigma1_d - sigma1
    # Jacobian of the first task
    J1          = J[0:2,0:n_DoF]
    # Null space projector
    P1          = np.eye(3,3) - DLS(J1, 0.0) @ J1

    # TASK 2: Position of Joint 1
    # Current position of joint 1
    sigma2      = np.array([Probot[1,1]])
    # Error in joint position
    err2        = sigma2_d - sigma2
    # Jacobian of the second task
    J2          = jacobian([T[0], T[1]], revolute)[1:2]
    # Augmented Jacobian
    J2bar       = J2 @ P1

    # Combining tasks
    # Velocity for the first task
    dq1         = DLS(J1,0.1) @ err1
    # Velocity for both tasks
```

```
    dq12         = dq1 + DLS(J2bar, 0.2) @ (err2 - J2 @ dq1)
    # Limited velocity
    limited_dq12 = np.where(np.abs(dq12) > dq_max, np.sign(dq12) * dq_max
    , dq12)
    # Simulation update
    q = q + limited_dq12 * dt

    # 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)
    point.set_data(sigma1_d[0], sigma1_d[1])

    time_store.append(t + N_iter * Tt)
    q1_store.append(np.linalg.norm(err1))
    q1.set_data(time_store, q1_store)
    q2_store.append(np.linalg.norm(err2))
    q2.set_data(time_store, q2_store)

    return line, path, point, q1, q2
```

Listing 4: **Robot definition (3 revolute joint planar manipulator)**

```
# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3)                    # displacement along Z-axis
q = np.array([0.2, 0.5, 0.2])      # rotation around Z-axis (theta)
a = np.array([0.75, 0.5, 0.3])     # displacement along X-axis
alpha = np.zeros(3)                # rotation around X-axis
revolute = [True, True, True]      # flags specifying the type of joints
n_DoF = len(revolute)              # Number of Degree of Freedom
dq_max = np.array([3, 3, 3]) # The maximum joint velocity limit
# Setting desired position of end-effector to the current one
# Position of the end-effector
sigma1_d = np.array([-1.0 + 2.0 * np.random.rand(), -1.0 + 2.0 * np..
    random.rand()])
# Position of joint 1
sigma2_d = np.array([0.0])
```

Listing 5: **Simulation Loop: Joint position task at the top of the hierarchy.**

```
# TASK 1: Position of Joint 1
# Current position of joint 1
sigma2      = np.array([Probot[1,1]])
```

```
# Error in joint position
err2         = sigma2_d - sigma2
# Jacobian of the second task
J2           = jacobian([T[0], T[1]], revolute)[1:2]
# Augmented Jacobian
J2bar        = J2
# Null space projector
P2           = np.eye(3,3) - DLS(J2bar, 0.0) @ J2bar

# TASK 2: Position of End-Effector
# Current position of the end-effector
sigma1       = np.array([Probot[0,-1], Probot[1,-1]])
# Error in Cartesian position
err1         = sigma1_d - sigma1
# Jacobian of the first task
J1           = J[0:2,0:n_DoF]
# Augmented Jacobian
J1bar        = J1 @ P2

# Combining tasks
# Velocity for the first task
dq2          = DLS(J2,0.1) @ err2
# Velocity for both tasks
dq21         = dq2 + DLS(J1bar, 0.2) @ (err1 - J1 @ dq2)
limited_dq21 = np.where(np.abs(dq21) > dq_max, np.sign(dq21) * dq_max,
    dq21)

# Simulation update
q = q + limited_dq21 * dt
```

Exercise 03: Questions and Answers

Q1: What are the advantages and disadvantages of redundant robotic systems?

Advantages:

- Flexibility, Enhanced Dexterity: Redundant systems can adapt to various tasks and environments more effectively due to their additional DOFs. They can reach different configurations, achieve more dexterous motions with greater precision and accuracy and avoid obstacles more easily.
- Fault Tolerance: Redundancy provides fault tolerance against failures in individual components. If one joint or actuator fails, the system may still be able to complete the task by

reconfiguring its motion.

- Additional tasks: Thanking to a null space, redundant systems can optimize task performance by considering additional criteria, such as energy consumption, joint velocities, or workspace utilization, leading to more efficient operations.

Disadvantages:

- Increased Complexity: Redundant systems are more complex in terms of design, control, and computation. Managing additional DOFs requires more sophisticated algorithms and control strategies.
- Higher Cost: Redundant robots typically come with higher production and maintenance costs due to the increased number of components and complexity.
- Singularities and Degeneracies: Redundant systems may encounter singularities or degenerate configurations that limit their motion capabilities or result in undesirable behaviors. Managing these issues requires careful planning and control.

Q2: What is the meaning and practical use of a weighting matrix W, that can be introduced in the pseudo inverse/DLS implementation?

Meaning: The weighting matrix W is a diagonal matrix where each diagonal element represents the weight or importance assigned to the corresponding degree of freedom (DOF) in the solution vector. Higher weights indicate less importance, and vice versa.

Practical Use:

- Task Priority: In robotics, different tasks may have different priorities. For example, in a manipulator with redundant DOFs, one task may be the primary task (e.g., reaching a specific position), while others may be secondary tasks (e.g., avoiding obstacles or minimizing joint velocities). By adjusting the weights in W, we can prioritize certain tasks over others.
- Singularity Avoidance: Weighting matrices can be used to avoid singular or near-singular configurations. By assigning lower weights to DOFs that are close to singularities, you can guide the robot away from these problematic configurations.
- Smoothness and Stability: Weighting matrices can promote smoother and more stable motions by penalizing abrupt changes in joint velocities or accelerations. By adjusting the weights, you can control the trade-off between task performance and motion smoothness.

Seudoinverse/DLS implementation: In the pseudo-inverse or DLS method, the weighting matrix W is typically incorporated into the calculation of the weighted pseudo-inverse or damped pseudo-inverse. The weighted is computed as:

Pseudo-inverse:

$$\zeta = J^{\dagger}(\mathfrak{q})\dot{x}_E \quad \rightarrow \quad \zeta = W^{-1}J^T(\mathfrak{q})\left(J(\mathfrak{q})W^{-1}J^T(\mathfrak{q})\right)^{-1}\dot{x}_E \tag{3}$$

Damped pseudo-inverse:

$$\begin{aligned} &\zeta = J^T(\mathfrak{q})\left(J(\mathfrak{q})J^T(\mathfrak{q}) + \lambda^2 I\right)^{-1}\dot{x}_E \\ &\rightarrow \quad \zeta = W^{-1}J^T(\mathfrak{q})\left(J(\mathfrak{q})W^{-1}J^T(\mathfrak{q}) + \lambda^2 I\right)^{-1}\dot{x}_E \end{aligned} \tag{4}$$

Where J is the Jacobian matrix, W is the weighting matrix, λ is the damping factor, and I is the identity matrix.

Result

Exercise 1:

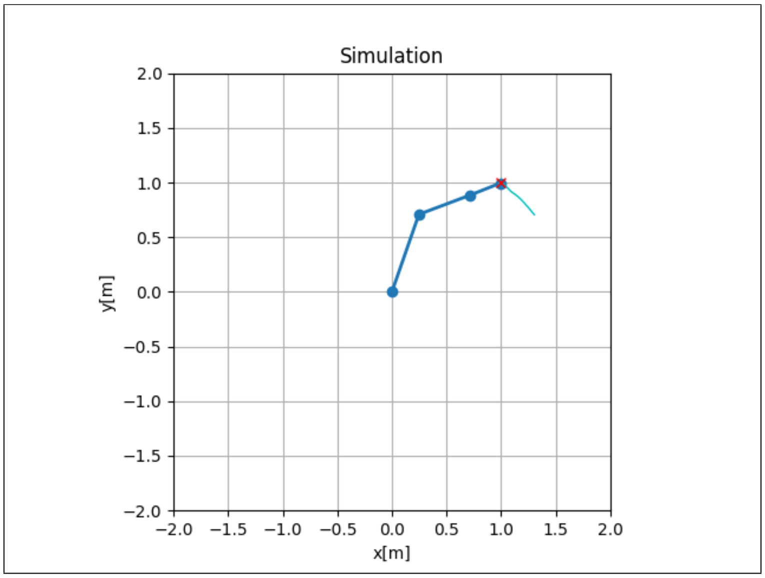

Figure 2: Simulation of the manipulator, including end-effector goal

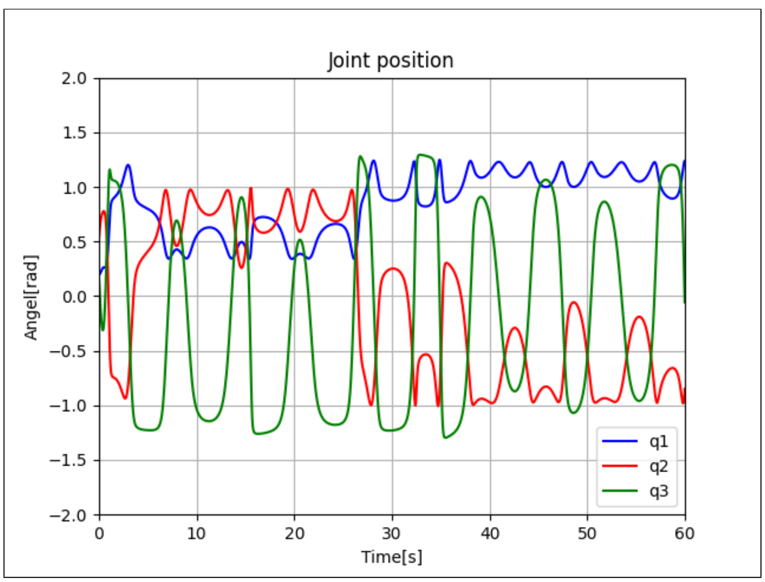

Figure 3: Positions of robot's joints during simulation

Exercise 2:

End-effector position task at the top of the hierarchy

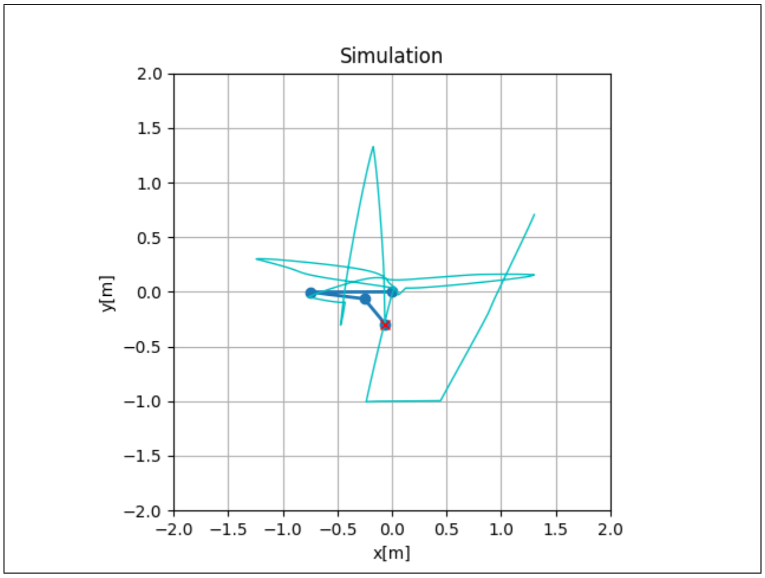

Figure 4: Simulation of the manipulator, including end-effector path

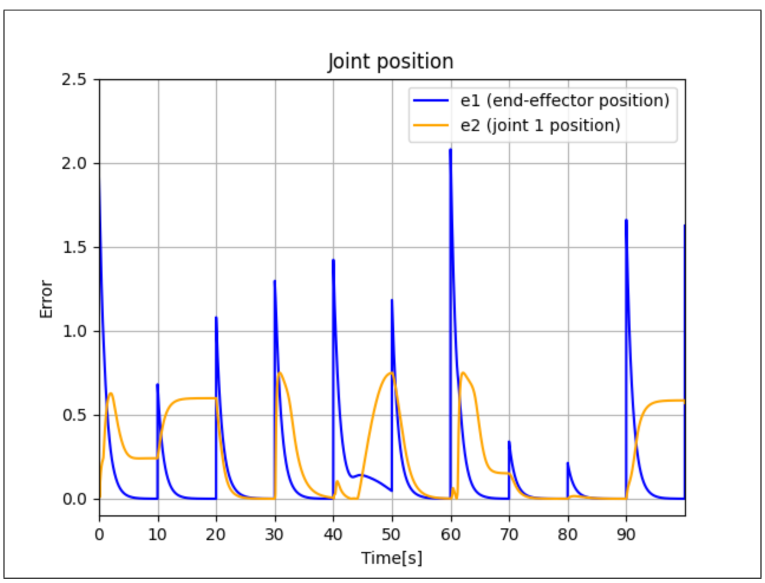

Figure 5: Evolution of the TP control errors

Joint position task at the top of the hierarchy

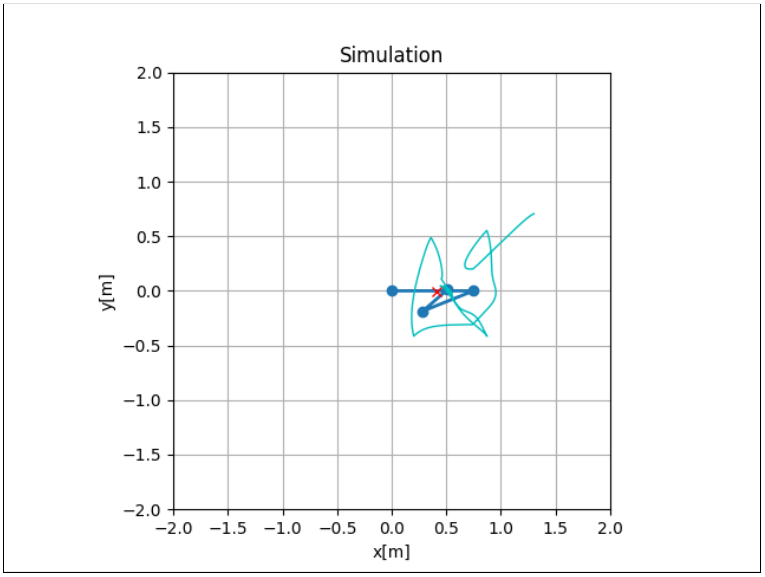

Figure 6: Simulation of the manipulator, including end-effector path

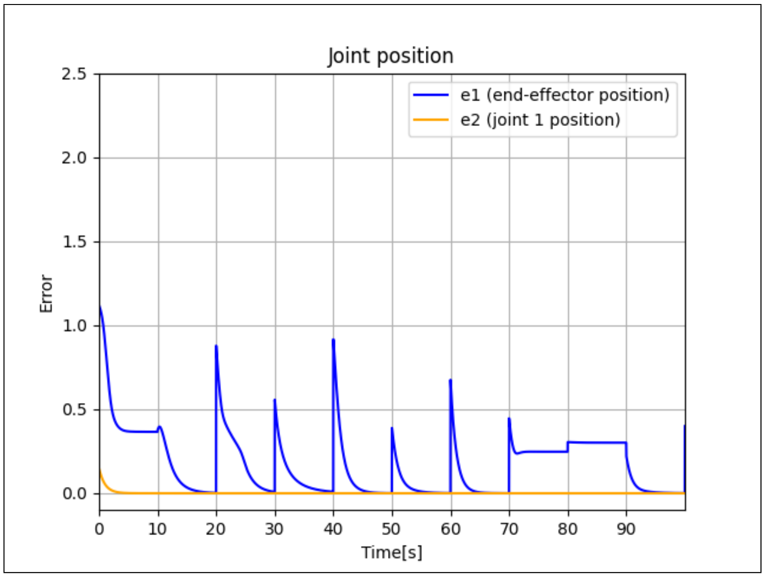

Figure 7: Evolution of the TP control errors

Comment

- Convergence of Tasks: Both tasks (end-effector position and joint position) converge to their desired values over time. The plots of the evolution of the norm of control errors show a decreasing trend until reaching a stable state where both tasks are satisfied.

- Task Prioritization: The algorithm successfully prioritizes the tasks according to the specified hierarchy. In case the end-effector position task is at the top of the hierarchy, it achieved accurately (converged to 0), even if it conflicts with the joint position task. Conversely, if the joint position task is at the top of the hierarchy, it should be achieved regardless of the end-effector position.
- Smoothness of Motion: The smoothness of the robot's motion during task execution indicates effective control and task coordination.