Students:
Tanakrit Lertcompeesin [u1992761@campus.udg.edu]

Lab3: Task-Priority kinematic control (1A)

Introduction

The lab set out to investigate task-priority kinematic control. It had two main sections. Initially, we concentrated on accomplishing a single task: reaching a specific end-effector position at any speed. In the second part, we introduced another task: controlling joint positions. This meant the robot needed to prioritize reaching the position while also meeting joint requirements.

Methodology

Exercise 1

In this part, we employed the planar robot simulation from lab2 to examine the performance of single-task priority kinematic control with arbitrary functions. The DH configuration utilized in this section is depicted in figure 2 of the code, with visualization provided in figure 1.

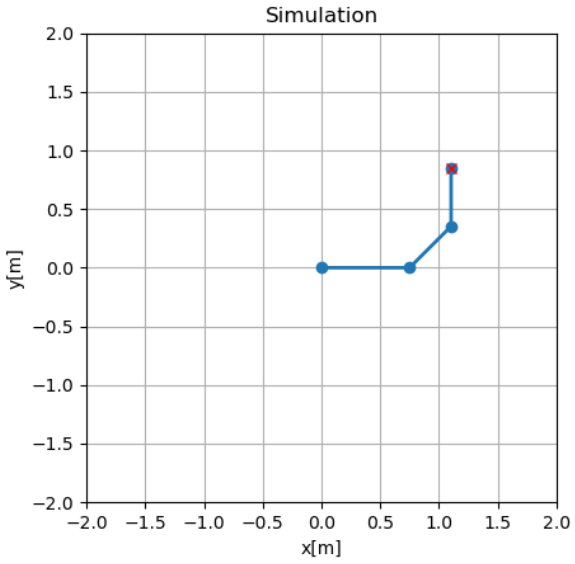


Figure 1: Robot model

```
# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3) # displacement along Z-axis
q = np.array([0, np.pi / 4, np.pi / 4]) # rotation around Z-axis (theta)
alpha = np.zeros(3) # displacement along X-axis
a = np.array([0.75, 0.5, 0.5]) # rotation around X-axis
revolute = [True, True, True] # flags specifying the type of joints
K = np.diag([1, 1])

# Setting desired position of end-effector to the current one
T = kinematics(
    d, q.flatten(), a, alpha
)  # flatten() needed if q defined as column vector !
sigma_d = T[-1][0:2, 3].reshape(2, 1)

# Simulation params
dt = 1.0 / 60.0
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("Simulation")
ax.set_aspect("equal")
ax.set_xlabel("x[m]")
ax.set_ylabel("y[m]")
ax.grid()
line, = ax.plot([], [], "o-", lw=2) # Robot structure
path, = ax.plot([], [], "c-", lw=1) # End-effector path
point, = ax.plot([], [], "rx") # Target
PPx = []
PPy = []
q1 = []
q2 = []
q3 = []
timestamp = []
```

Figure 2: Code for setting up DH configuration and simulation

The equations for the single task are presented as equations 1 and 2. This task is aimed at minimizing the error in the end-effector position. Therefore, the Jacobian used in these equations is the one that maps all joint velocities to the end-effector velocity. P is referred to as the null-space projector, enabling the robot to choose a solution that satisfies both the main task and any subsequent task, specified by an arbitrary vector y in the joint velocity space. These equations

were implemented in the ***simulation()*** function, as depicted in figure 3.

$$\dot{q} = J(q)^{\dagger}\dot{x} + (P)y \tag{1}$$

$$P = (I - J(q)^{\dagger}J(q)) \tag{2}$$

```
def simulate(t):
    global q, a, d, alpha, revolute, sigma_d
    # Update robot
    T = kinematics(d, q.flatten(), a, alpha)
    J = jacobian(T, revolute)

    # Update control
    sigma = T[-1][0:2, -1].reshape(2, 1) # Current position of the end-effector
    err = sigma_d - sigma # Error in position

    Jbar = J[:2, :] # Task Jacobian
    Jinv = np.linalg.pinv(J[:2, :])
    P = np.eye(3) - (Jinv @ Jbar) # Null space projector
    y = np.array([np.cos(t), 1.5 * np.sin(t), -1.75 * np.sin(t)]).reshape(3, 1) #
        Arbitrary joint velocity
    dq = (Jinv @ (K @ err)) + (P @ y) # Control signal
    q = q + dt * dq.reshape(3) # Simulation update

    # 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])

    # Update memory for plotting
    q1.append(q[0])
    q2.append(q[1])
    q3.append(q[2])
    timestamp.append(t)

    return line, path, point
```

Figure 3: Code Exercise 1 Simulation

After the simulation concludes, we summarize the results by plotting the evolution of each joint position. This is achieved through the implementation of ***plot_ summary()*** function, as illustrated in figure 4.

```
def plot_summary():
    # Evolution of joint positions Plotting
    fig_joint = plt.figure()
    ax = fig_joint.add_subplot(111, autoscale_on=False, xlim=(0, 60), ylim=(-2, 3))
    ax.set_title("joint positions")
    ax.set_xlabel("Time[s]")
    ax.set_ylabel("Angle[rad]")
    ax.grid()
    print(q1)
    plt.plot(timestamp, q1, label="q1")
    plt.plot(timestamp, q2, label="q2")
    plt.plot(timestamp, q3, label="q3")

    ax.legend()

    plt.show()
```

Figure 4: Code Exercise 1 Plot summary

Exercise 2

In this section, our focus shifts to incorporating a second priority into the control algorithm, which involves maintaining the joint positions. We continue to utilize the robot configuration from Exercise 1. The equation for the two-task kinematic control is presented in equation 3.

$$\zeta = J_1^{\dagger}\dot{x}_1 + \bar{J}_2^{\dagger}\left(\dot{x}_2 - J_2\left(J_1^{\dagger}\dot{x}_1\right)\right) + \left(I - \bar{J}_2\bar{J}_2^{\dagger}\right)z \tag{3}$$

$$J_2 = [1, 0, 0] \tag{4}$$

As shown in equation 4, J_2. represents the Jacobian for the second task, which maps a selected joint velocity to the end-effector velocity. In this specific case, we aim to maintain the position of the first joint, so we only need to consider the value in the first index. These equations are implemented in figure 5.

```
# Simulation loop
def simulate(t):
    global q, a, d, alpha, revolute, sigma1_d, sigma2_d
    global PPx, PPy, last_time
    # Update robot
    T = kinematics(d, q.flatten(), a, alpha)
    J = jacobian(T, revolute)
    # Update control
    if control_priority == "END-EFFECTOR POSITION":
        # TASK 1
        sigma1 = T[-1][0:2, -1].reshape(2, 1) # Current position of the end-effector
        err1 = sigma1_d - sigma1 # Error in Cartesian position
        J1 = J[:2, :] # Jacobian of the first task
        Jinv = DLS(J1, 0.1)
        P1 = np.eye(3) - (np.linalg.pinv(J1) @ J1) # Null space projectorJinv
        # TASK 2
        sigma2 = q[0] # Current position of joint 1
        err2 = sigma2_d - sigma2 # Error in joint position
        J2 = np.array([1, 0, 0]).reshape(1, 3) # Jacobian of the second task
        J2bar = J2 @ P1 # Augmented Jacobian
        # Combining tasks
        dq1 = (Jinv @ (err1)).reshape(3, 1) # Velocity for the first task
        dq12 = dq1 + (DLS(J2bar, 0.1) @ (err2 - J2 @ dq1)) # Velocity for both tasks
    elif control_priority == "JOINT 1 POSITION":
        # TASK 1
        sigma2 = q[0] # Current position of joint 1
        err2 = sigma2_d - sigma2 # Error in joint position
        J1 = np.array([1, 0, 0]).reshape(1, 3) # Jacobian of the second task
        Jinv = DLS(J1, 0.1) # Augmented Jacobian
        P1 = np.eye(3) - (np.linalg.pinv(J1) @ J1) # Null space projectorJinv
        # TASK 2
        sigma1 = T[-1][0:2, -1].reshape(2, 1) # Current position of the end-effector
        err1 = sigma1_d - sigma1 # Error in Cartesian position
        J2 = J[:2, :] # Jacobian of the first task
        J2bar = J2 @ P1
        # Combining tasks
        dq1 = (Jinv @ (err2)).reshape(3, 1) # Velocity for the first task
        dq12 = dq1 + (DLS(J2bar, 0.1) @ (err1 - J2 @ dq1)) # Velocity for both tasks
    # Scale velocity
    max_vel = np.max(np.abs([dq1, dq12]))
    if max_vel > vel_threshold:
        dq12 = dq12 / max_vel * vel_threshold
    q = q + dq12 * dt # Simulation update
    ...
```

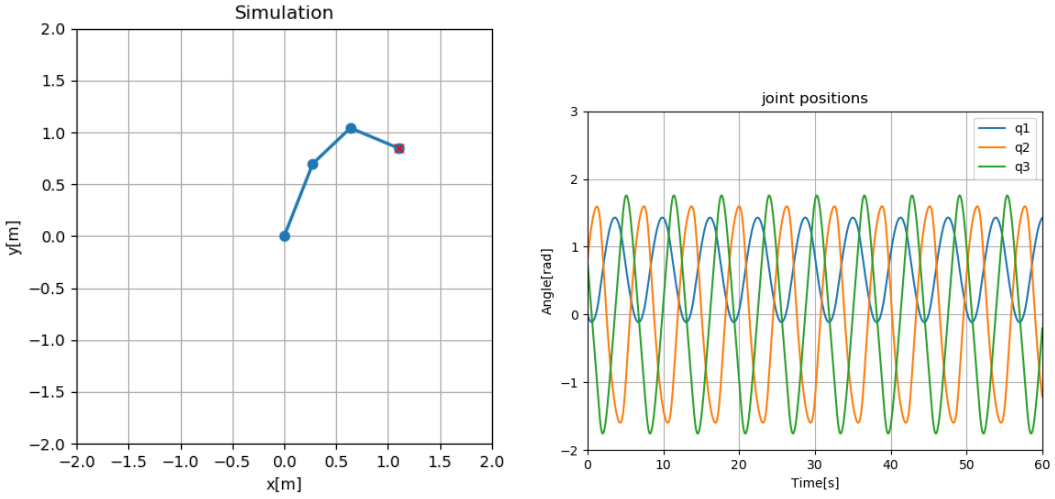
Figure 5: Code Exercise 2 simulation

Several modifications have been made to the simulation loop in this Exercise, in addition to the mentioned equation. Firstly, we now re-initialize the desired position every time the simulation restarts. Secondly, we have provided an option to select which task takes higher priority. Finally, a velocity scaling algorithm has been implemented to prevent excessively high velocities.

Results & Discussion

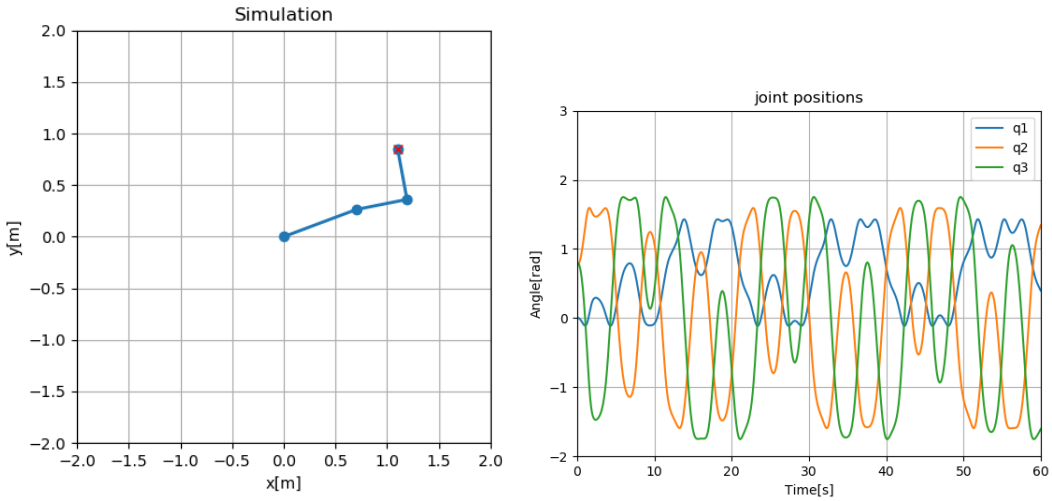
Exercise 1

To validate the algorithm's performance and assess the impact of arbitrary functions, we conducted experiments by setting the desired end-effector position to be the same as the initial position. This allowed us to observe the robot's movement resulting from different arbitrary functions. Figures 6a and 7a depict the movement of the planar robot, while figures 6b and 7b show the corresponding joint positions. Analyzing these plots enables us to see the variations in movement produced by different vector y.



(a) (b)

Figure 6: $y = [2sin(t), 3cos(t), -sin(t)$



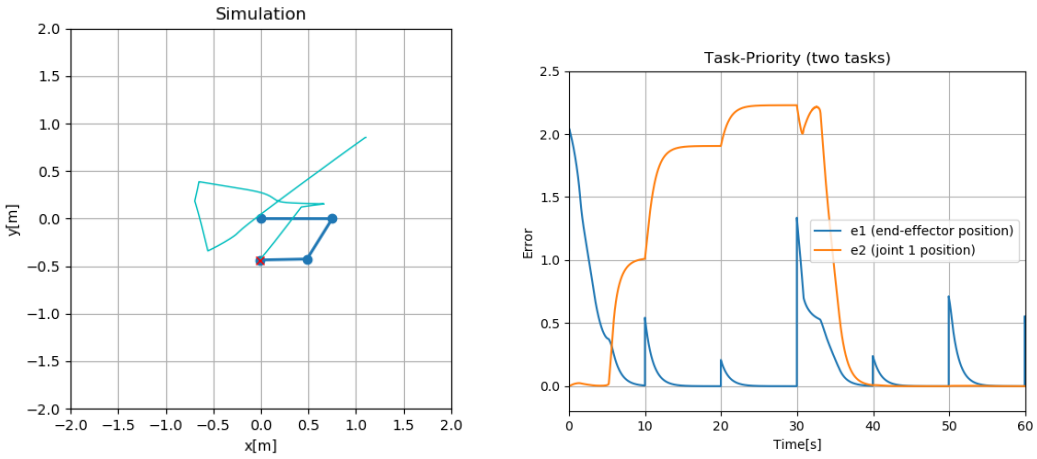
(a) (b)

Figure 7: $y = [cos(t), 1.5sin(t), -1.75cos(t)$

From both results, it's evident that the end-effector can maintain its position while adjusting the joints according to the provided arbitrary velocity. This demonstrates that the task-priority algorithm effectively prioritizes the main task without being compromised by the desired velocity. However, upon observing Figures 6b and 7b, it's noticeable that not all joint positions are able to precisely execute the sine or cosine wave as inputted. This discrepancy is also a result of the algorithm, which may compromise lower priority tasks in order to maintain higher priority ones.

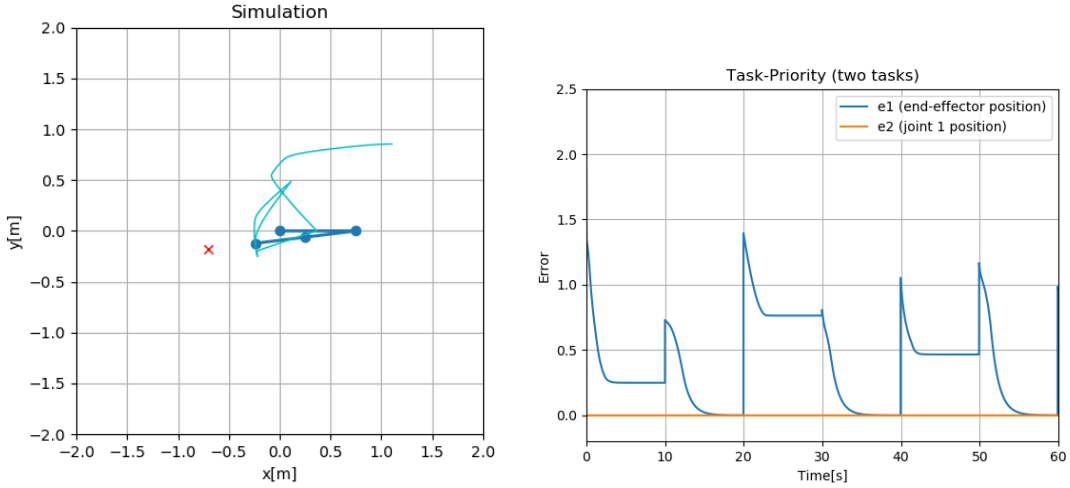
Exercise 2

In this section, we randomly assign a goal position to the manipulator every 10 seconds to observe the contrasting behaviors when prioritizing these goals versus prioritizing joint position maintenance. For the joint maintenance task, we aim to keep the first joint at 0 degrees. Figure 1 displays the outcome when prioritizing the end-effector position as the first task, while Figure 2 illustrates the results when prioritizing the maintenance of joint positions as the first task.



(a) (b)

Figure 8: Prioritizing the end-effector position



(a) (b)

Figure 9: Prioritizing the first joint position

As observed in Figure 8a, the robot moves to the goal position without considering the position of the first joint (e2 cannot reach 0 in first three iterations), resulting in reaching every goal. This is evidenced by the convergence of the e1 error to 0 in every iterations, as depicted in figure 8b. However, even when it reaches the goal, it attempts to adjust the first joint back to 0 while maintaining the end-effector position. This adjustment isn't always successful since the null space solution cannot always achieve the both desired configuration, and the algorithm prevented the lower priority task to violate the main task.
In contrast, as shown in figure 9a, the robot consistently keeps the first link in place, prioritizing joint position maintenance. Consequently, it often fails to reach the desired end-effector position,

which aligns with our expectation since it's not our first priority. This behavior is supported by the error plot in figure 9b, where e2 maintains around 0 while e1 cannot reach 0 in most iterations.

Questions

"What are the advantages and disadvantages of redundant robotic systems?"

The redundancy in robot kinematics allows us to find solutions that meet multiple requirements, such as safety, balance, or preventing calculation failures. However, this means we have multiple solutions to consider, rather than a single exact solution, which can lead to optimization problems. Moreover, the increased complexity of the system results in higher computational loads.

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

The W matrix technique enables us to adjust the level of involvement of each Degree of Freedom (DOF) based on its importance for the task at hand. This provides us with greater control over the resulting movement of the robot, allowing us to control its behavior to meet specific requirements more effectively.