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

Lab 5: Task-Priority kinematic control (2A)

Introduction

In a previous lab, we delved into equality tasks, which are focused on achieving specific manipulator configurations. In this lab, we shift our focus to inequality tasks, which typically require higher priority due to their importance, such as ensuring the safety of robots. The first part of this lab will concentrate on the obstacle avoidance task, while the second part will delve into exploring the joint limit task.

Methodology

Exercise 1

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

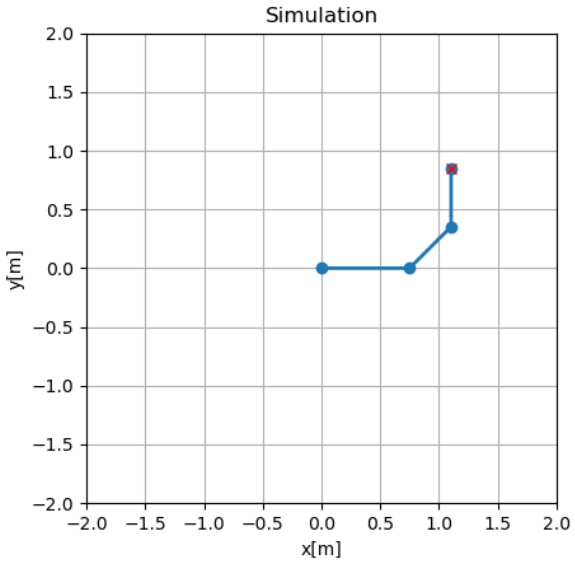


Figure 1: Robot model

```
# Robot model - 3-link manipulator
d = np.zeros(3) # displacement along Z-axis
theta = np.array([0, np.pi / 4, np.pi / 4]).reshape(
    3, 1
)  # 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
robot = Manipulator(d, theta, a, alpha, revolute) # Manipulator object
vel_threshold = 1.5
# Obstacle definition
obstacle_pos = np.array([0.0, 1.0]).reshape(2,1)
obstacle_r = 0.5
obstacle_pos2 = np.array([-0.5, -0.75]).reshape(2,1)
obstacle_r2 = 0.4
obstacle_pos3 = np.array([0.75, -0.5]).reshape(2,1)
obstacle_r3 = 0.275
obstacles_list = [obstacle_pos, obstacle_pos2, obstacle_pos3]
obstacles_r_list = [obstacle_r, obstacle_r2, obstacle_r3]
# Joint limits definition
joint1_min = 0
joint1_max = np.pi/2
ee_min = -0.5
ee_max = 0.5
# Task hierarchy definition
tasks = [
         JointLimits("Joint1-limits",np.zeros(1), 0, np.array([joint1_min ,
            joint1_max])),
         Obstacle2D("Obstacle avoidance", obstacle_pos, np.array([obstacle_r,
            obstacle_r+0.05])),
         Obstacle2D("Obstacle avoidance", obstacle_pos2, np.array([obstacle_r2,
            obstacle_r2+0.05])),
         Obstacle2D("Obstacle avoidance", obstacle_pos3, np.array([obstacle_r3,
            obstacle_r3+0.05])),
         Position2D("End-effector position", np.array([-1, 1]).reshape(2,1))
       ]
```

Figure 2: Code for setting up DH configuration, obstacles and simulation

In this lab, we present an `Obstacle2D` class tailored for evading circular obstacles within the workspace. These obstacles are defined by their center position,radius and threshold radius. The conditions governing the threshold to determine the task activation are detailed in the implementation of this class as depicted in figure 3. The obstacles used in this exercise are defined in the setup section, as shown in Figure 2. Their visualization is presented in figure 4.

```python
class Obstacle2D(Task):
    def __init__(self, name, desired, radius):
        super().__init__(name, desired)
        self.J = None
        self.err = None
        self.r = radius
        self.K = np.eye(len(desired))
        self.FeedForwardVel = np.zeros(desired.shape)

    def update(self, robot):
        self.J = robot.getEEJacobian()[: len(self.sigma_d), :]
        err = robot.getEETransform()[: 2, -1].reshape(self.sigma_d.shape) -
            self.getDesired()
        norm_err = np.linalg.norm(err)
        self.err = self.active *(err / norm_err) # Update task error
        self.err_plot.append(norm_err - self.r[0])

        # update active status
        if self.active == 0 and norm_err <= self.r[0]:
            self.active = 1
        elif self.active == 1 and norm_err >= self.r[1]:
            self.active = 0
```

Figure 3: `Obstacle2D` class

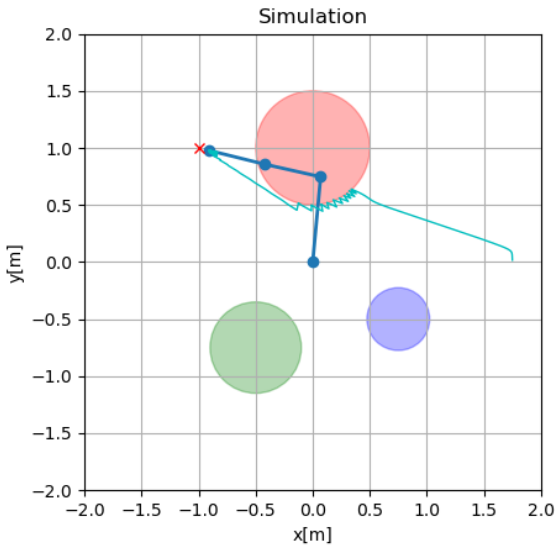


Figure 4: Obstacles visualization

Exercise 2

In this exercise, we introduced another inequality task called joint limit. This task restricts the angle of the specified joint to a certain range defined by 'qmax' and 'qmin'. Two thresholds, gamma and alpha, are used to determine when the task should be activated or deactivated. The conditions related to these thresholds are implemented in the `JointLimits` class, as depicted in figure 5.

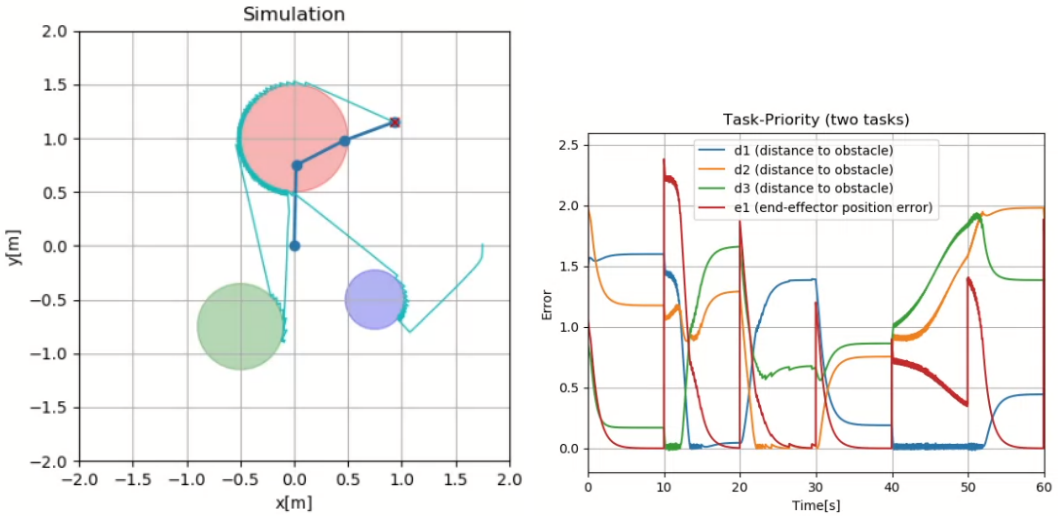
```
class JointLimits(Task):
    def __init__(self, name, desired, joint=1, limit=np.array([0, np.pi/2])):
        super().__init__(name, desired)
        self.joint = joint
        self.J = np.zeros((1, 3))
        self.err = None
        self.K = np.eye(len(desired))
        self.FeedForwardVel = np.zeros(desired.shape)

        self.qmin = limit[0]
        self.qmax = limit[1]
        self.gamma = np.pi/18 # 5 degrees
        self.alpha = np.pi/36 # 2.5 degrees

    def update(self,robot):
        self.J[0,self.joint] = 1
        self.err = np.array([self.active])
        q = robot.getJointPos(self.joint)
        self.err_plot.append(q)

        # update active status
        if self.active == 0 and q >= self.qmax - self.alpha:
            self.active = -1
        elif self.active == 0 and q <= self.qmin + self.alpha:
            self.active = 1
        elif self.active == -1 and q <= self.qmax - self.gamma:
            self.active = 0
        elif self.active == 1 and q >= self.qmin + self.gamma:
            self.active = 0
```

Figure 5: `JointLimits` class

Results & Discussion

Exercise 1

We assessed the algorithm's performance through experiments involving the following steps: defining the obstacles in the environment, as illustrated in figure 4; setting up obstacle avoidance tasks corresponding to these obstacles; and assigning the end-effector position as the lowest priority task, which changes randomly every 10 seconds.



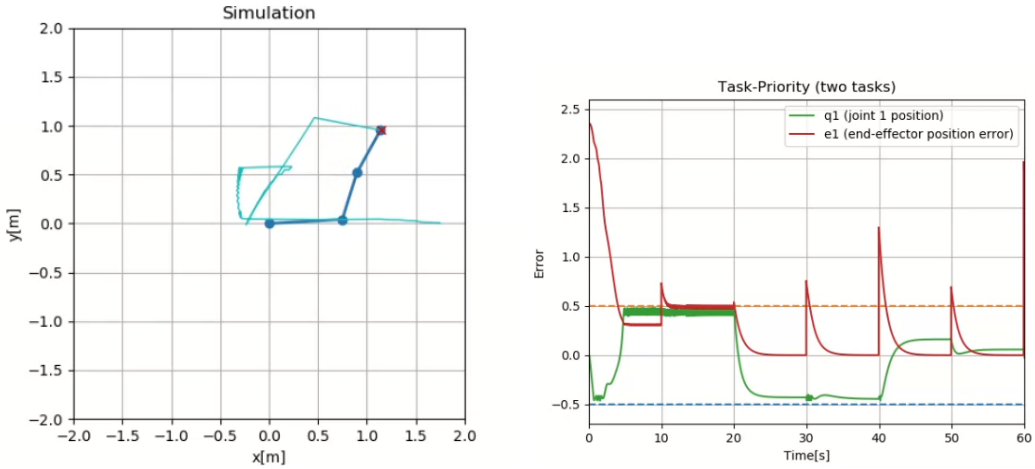
(a) robot structure in motion (b) task error and obstacle distances

Figure 6: Exercise 1

As depicted in Figure 6a, the path of the end effector consistently avoids entering the obstacle area, even when some goals are located inside the obstacles. This demonstrates the algorithm's ability to prioritize obstacle avoidance tasks over end-effector tasks. Figure 6b further confirms that the three obstacle avoidance tasks never violated their constraints, as none of the distances to obstacles fell below 0. However, the end-effector cannot converge to zero in some situations, indicating that it was violated by higher priority tasks.

Exercise 2

In this section, we evaluated the joint limit task by setting the joint limits for joint 1 to a minimum of -0.5 radians and a maximum of 0.5 radians.



(a) robot structure in motion (b) task error and obstacle distances

Figure 7: Exercise 2

The end-effector path illustrated in figure 7a reveals certain configurations on the left side that the robot cannot attain because joint 1 cannot exceed the maximum value defined in the joint limit task. Furthermore, figure 7b demonstrates that for the first two desired positions, joint 1's position was restricted to 0.5 radians, leading to a violation of the 2D position task, as its error could not converge to zero. Additionally, the position of joint 1 consistently remains within the limit boundaries (indicated by the dashed line) throughout the experiment.