ECE 275 Term Project – Robot Pick and Place

David Felipe Alvear Goyes

March 26, 2023

1 Introduction

Different applications in the industry require the use of robotics systems to increase the performance and quality of operations. Manipulators are one of the systems to perform monotonous operations such as, pick and place tasks, assembly, welding, sensing, and others. In this mid-term project, we are developing a solution to a pick-and-place problem using a KUKA robot with 7 degrees of freedom DoF. The objective of the project is to design an LQR torque control algorithm and a pipeline for a manipulator to pick and place cubes on a conveyor system. The implementation was done in a simulation environment using python and PyBullet where the algorithm and the pipeline were tested to solve the assigned task.

2 Problem Formulation

In this project, we aim to design a control algorithm for a robot manipulator to solve a pick-and-place problem, Where colored cubes are randomly placed in a conveyor belt system with a defined constant velocity. The robot should pick up the colored cube which can be red, blue, or green, to then place in a colored tray according to the color of the object. The problem is divided into the following elements to consider some constraints and needs in the solution.

2.1 Robot Manipulator and Constraints

The robot manipulator used in this problem is a KUKA arm with 7 degrees of freedom capable to perform pick and place tasks. The manipulator is capable to grasp and release objects using the end effector and helped with the provided functions 'grasp_object' and 'release_object', the last also return the color of the cube to use for the place pipeline. There exist some constraints that are important to understand in the design of the solution. The manipulator is subject to constraints such as maximum and minimum manipulation area, joint limits, and maximum joint velocity.

2.2 Conveyor System and Task Description

The conveyor system is composed of several rollers moving at a constant speed. The cubes, which are color-coded in red, green, and blue, are placed in a random horizontal position in the rollers to then move impulsed by the constant velocity of the system. The

task of the robot is to pick each one of the cubes while they are moving in the conveyor before falling to the ground and being placed in color-matching trays. Three color-coded trays are placed at a fixed location, where the robot is located between the conveyor and the trays. The manipulator should perform a selective task in base on the color of the cubes, then it should detect the cube's color and match the desired tray. Sorting this selective pick-and-place application adds complexity to the implementation and building of the pipeline.

To facilitate the design and implementation, several useful functions and parameters are provided in the simulation environment to deploy the control algorithm and the pipeline. These functions include 'get_cube_locations', 'get_end_effector_pose', 'get_joint _angles', 'get_joint_velocities', 'grasp_object', and 'release_object'. Parameters such as 'robot_base _location', 'tray_red_location', 'tray_green_location', 'tray_blue_location', and 'tray_width.

3 Solution Design

The pick-and-place task is divided into three scenarios, the first is when the cubes are randomly deployed in the conveyor system, then they start to move along it with a constant velocity, to finally picked up using the manipulator before the cubes reach the end of the conveyor. The second scenario is when the manipulator picked up the cube and now the arm first detects the color code of the object to then follow a predefined trajectory to the matched colored tray, after sequentially following the trajectory the end effector reaches the desired position above the tray to then release the object. The final scenario happens after the arm releases the object and chooses a trajectory to go back to the initial position to be ready to pick up the next cube.

The solution is based on the described three scenarios. Some requirements are needed in the three scenarios such as torque controller design to move the arm to a desired position in the base of the joint angles and joint velocities, and the conversion of the end effector global position to the joint angles desired. Developing these two tools we can select a desired position and orientation in the global frame to then be translated to a joint angle configuration using the inverse kinematics of the KUKA robot.

The solution consists of two main modules in charge of handling the task and computing the control input for the manipulator. In figure 1 we can see an overview of the architecture of the solution, where the **main loop** is responsible for handling the iterations in the code. The **simulation environment** represents the PyBullet simulation environment where the robot, conveyor, and cubes are placed. The **TaskAllocator** block is a python class responsible for receiving the objects that are placed in the conveyor and allocating the task in base of the state of the simulation. The **TaskAllocator** module creates a custom python object **Cubes** which is composed of multiple **Cube** python objects. Each cube has the properties that describe the cube in the simulation as well as the dynamics of the object that will be used to predict future positions of the cube to solve the time delay between the controller and the conveyor. The **TaskAllocator** updates the **Cubes** object in base of the position of each cube in the conveyor belt following the **main loop** iterations. After one of the cubes approach to the workspace of

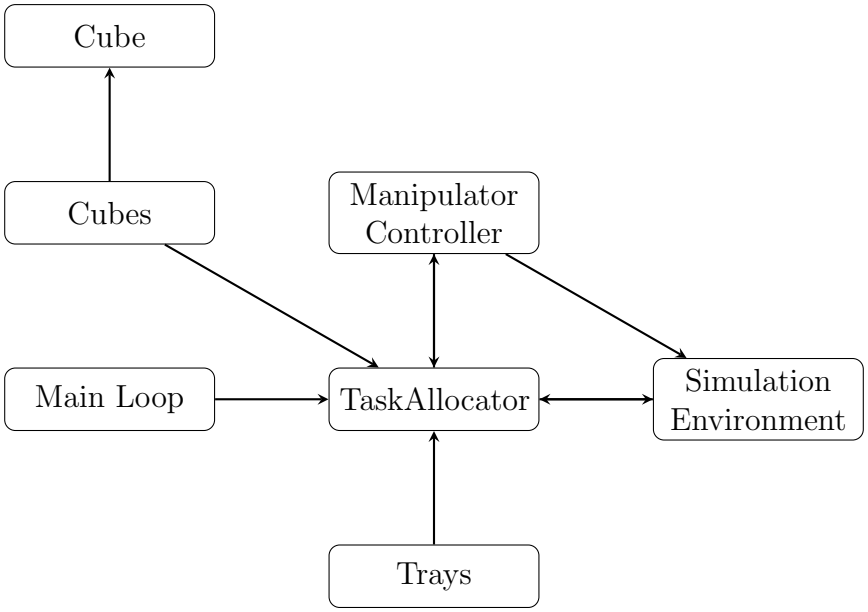

Figure 1: Solution Architecture

the manipulator the **Allocate** function from **TaskAllocator** sends the instruction to the **ManipulatorController** to compute the desired trajectory using inverse kinematics to then design a control input. The use of the **Cube** object gives the possibility to predict the next position of the object and send the robot. Once the object is grasped the **TaskAllocator** sends the manipulator to a home position and proceeds to the second stage.

The **TaskAllocator** class has the trajectories for each one of the trays and also optional trajectories to avoid singularities in the arm. The second scenario starts after the object is grasped with the end effector and the **Cube** object is updated with the color and then eliminated from the **Cubes** object list. With the color, the **Allocator** method gets the pre-defined trajectory to the desired color-coded tray and loads the waypoints needed to reach the destination. The function **Allocator** will keep track of the position and send the desired goal to the **ManipulatorController** to get the torque commands. After passing all the waypoints with a defined threshold error, the manipulator releases the object in the tray and finish the second scenario.

After succeeding in scenario two, the robot starts the third phase which consists in send the robot through waypoints to the home position to be ready to grasp the next **Cube**. The trajectory of scenario three is handled by the main algorithm of scenario two with waypoints from the tray to a predefined home position. Once the robot achieves the home position using the **ManipulatorController** the pipeline starts in scenario one waiting for a **Cube** from the **Cubes** object list.

The python classes **TaskAllocator** and **ManipulatorController** are configured in the main loop before entering the iterations, and the constructor is passed to the main loop to keep track of the **Simulation Environment** on each iteration while designing appropriate torque control inputs. The python class **Cubes** is initialized in the TaskAllocator to be used in the main loop to keep track of the objects placed in the conveyor.

Each of the cubes placed in the simulation is translated to a **Cube** python object and stored in the **Cubes** list in the **TaskAllocator** object.

3.1 Joint Torque Controller

To send the end effector to a desired position in the global frame we need to use the inverse kinematics of the manipulator to compute the desired joint configuration. With this information, we can design a control input to send the robot to a desired position or follow a predefined trajectory using the joint values and joint velocities. The following section is the design of a LQR torque controller for a manipulator.

1. The dynamics for the manipulator is the following:

$$\tau = M(\theta)\ddot{\theta} + V(\theta, \dot{\theta}) + G(\theta) \tag{1}$$

   Where $M(\theta)$ is the mass matrix, $V(\theta, \dot{\theta})$ the centrifugal and Coriolis term, and $G(\theta)$ gravity term. Given that the manipulator is a fully-actuated system, we can apply feedback linearization using equation 2 and obtain an approximate linearized dynamic model in equation 3.

$$\tau = M(\theta)\bar{\tau} + V(\theta, \dot{\theta}) + G(\theta) \tag{2}$$

$$\bar{\tau} = \ddot{\theta} \tag{3}$$

2. Using the feedback linearization we define the state vector and the state equation of the system as follows:

$$\mathbf{x} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}, \quad \dot{\mathbf{x}} = \begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \end{bmatrix}$$

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ I \end{bmatrix} u$$

   Design a LQR controller $u = Kx$, where $K = (K_1, K_2)$. Then, $\bar{\tau} = K_1\theta + K_2\dot{\theta}$. Using this and equation 1 and 2 result equation 5 that will be applied to the manipulator.

$$\tau = M(\theta)\bar{\tau} + V(\theta, \dot{\theta}) + G(\theta) \tag{4}$$

$$\tau = M(\theta)(K_1\theta + K_2\dot{\theta}) + V(\theta, \dot{\theta}) + G(\theta) \tag{5}$$

3. Given a reference trajectory θ_{ref}, $t > 0$. Design a controller for trajectory tracking. Defining the new state and state equation for reference trajectory as follows:

$$\mathbf{x}_{ref} = \begin{bmatrix} \theta \\ \dot{\theta_{ref}} \end{bmatrix}, \quad \dot{\mathbf{x}}_{ref} = \begin{bmatrix} \dot{\theta_{ref}} \\ \ddot{\theta_{ref}} \end{bmatrix}$$

$$\dot{\mathbf{x}}_{ref} = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix} \mathbf{x}_{ref} + \begin{bmatrix} 0 \\ I \end{bmatrix} u_{ref}$$

Define now $\bar{x} = x - x_{ref}$ and $\bar{u} = u - u_{ref}$. Then we can design a controller for this system using LQR. We can compute the K matrix using the **control** library in **python** and send the manipulator to a trajectory, and apply the control input $\bar{u} = K\bar{x} = K_1(\theta - \theta_{ref}) + K_1(\dot{\theta} - \dot{\theta}_{ref})$.

4 Implementation

The designed algorithm follows the architecture solution shown in figure 1. We are using PyBullet as our Simulation environment where the cubes, conveyor, trays, and robot lies. The main loop is in charge of handling the iterations of the simulation and providing access to the simulation environment through some useful functions to get the state variables of the manipulator, the conveyor state, and the cube state in the simulation. The classes **TaskAllocator** and **ManipulatorController** are created before entering the main loop to configure the constructor of the class **Cubes** to handle cubes entering in the simulation, on the other hand, the ManipulatorController compute the control gains for being used in the torque controller computation. In the main loop, the TaskAllocator has access to the simulation environment getting access to the state variables of the cubes and the robot, as well as, sending the torque commands that have been computed using the manipulator_controller inside the TaskAllocator class. The following subsections explain the components of the implementation.

4.1 Class Descriptions

Description of the implementation of the code based on the designed classes and the key functions. For further details, the complete code of each python class is added in the appendices at the end of the document (Section 7).

1. **Cube**: This class represents an individual cube and its properties, such as position, color, grasping state, predicted position on the conveyor, and the trajectory points.

    - 'predict_position()': This method predicts the position of the cube based on its current position and the cube speed calculated using the trajectory.

2. **Cubes**: This class represents a list of **Cube** objects and handles updates of positions, creation, and removal. The class tracks the cubes in base of the euclidean distance between iterations and updates the trajectory of each Cube object.

    - 'remove_out_of_workspace()': This method removes cubes that are no longer within the manipulator's workspace.
    - 'update_positions()': This method updates the positions of the cubes in the list based on the information provided by the simulation environment. If the cube is not in yet in the conveyor it will create the object Cube.
    - 'create_cube()': This method creates a cube and adds it to the cubes list when the update method doesn't find any match with the current cubes.

3. **ManipulatorController**: This class implements the Linear Quadratic Regulator (LQR) control algorithm and computes torque control for a robotic manipulator. It also includes functions for joint angle control and inverse kinematics to compute the joint angles from the desired end effector position.

    - 'compute_control_gain()': This method compute the control gain matrices K_1 and K_2 for the LQR control algorithm.
    - 'compute_torque_control()': This method computes the torque control for the robotic manipulator given joint angles and joint velocities.
    - 'joint_angle_control()': This method calculates the desired joint angles given end-effector pose and orientation.

4. **TaskAllocator**: This class is responsible for managing the overall pick-and-place task. It orchestrates the interactions between the **ManipulatorController** and the **Cubes** to ensure that the objects are picked up and placed on the corresponding trays. The constructor has the definition for the home or rest position of the manipulator and the two possible trajectories for each color-coded tray. The **ManipulatorController** instance is passed in the initialization to be used in the orchestration.

    - 'allocate()': This method is the main control loop of the TaskAllocator. It determines which cube to pick up and delegates the pick-and-place action to the manipulator controller. This method handles the three scenarios to pick up the object, transport it to the color-matched tray, and return to the home position while updating the Cubes list with the new positions.
    - 'send_to_home()': This method calculates the torques required to move the manipulator to the home position to wait for the next cube to pick up.
    - 'send_to_pose()': This method calculates the torques required to move the manipulator to a target pose.

Algorithm 1 describes a task allocation process for a robotic manipulator. its goal is to grasp objects and place them onto designated trays. First, the algorithm selects a target cube and calculates the torques required to move the manipulator toward it. When an object is grasped, the manipulator follows a series of waypoints to place it onto a tray. The procedure considers conditions such as object distances, manipulator workspace limits, and singularities checking configuration, to ensure an efficient manipulation.

Algorithm 1 Allocate Function for task orchestration

```
procedure ALLOCATE(objects, joint_angles, joint_velocities, end_effector_pose)
    cubes.remove_out_of_workspace()
    cubes.update_positions(objects)
    if flag_obj_grasped is False then
        if length(objects) > 0 then
            cube ← cubes.cubes[0]
            obj_pose ← cube.predict_position()
            obj_pose_last ← cube.last_position
            torques, q_ref ← send_to_pose(joint_angles, joint_velocities, obj_pose)
            distance ← calculate_distance(obj_pose_last, end_effector_pose[0])
            if distance < grasp_threshold then
                obj_grasped ← grasp_object(controller.pb_client, controller.robot_id)
                if obj_grasped ≠ -1 and flag_obj_grasped is False then
                    flag_obj_grasped ← True
                    grasped_obj ← obj_grasped
                    grasped_waypoints ← copy(paths[obj_grasped[1]])
                    cubes.cubes.remove(cube)
                    return send_to_home(joint_angles, joint_velocities)
                end if
            else if distance > distance_manipulator then
                return send_to_home(joint_angles, joint_velocities)
            end if
        else
            torques, q_ref ← send_to_home(joint_angles, joint_velocities)
        end if
    else
        if length(grasped_waypoints) < 1 and flag_release then
            flag_obj_grasped ← False
            flag_release ← False
            grasped_obj ← -1
            grasped_waypoints ← None
            return send_to_home(joint_angles, joint_velocities)
        end if
        waypoint ← grasped_waypoints[0]
        distance ← calculate_distance(waypoint, end_effector_pose[0])
        torques, q_ref ← send_to_pose(joint_angles, joint_velocities, waypoint)
        if distance < pose_threshold then
            if waypoint[:2] == trays[grasped_obj[1]][:2] then
                relase_object(controller.pb_client)
                flag_release ← True
            end if
            delete grasped_waypoints[0]
        end if
        flag ← checking_singuarity(end_effector_pose[0])
        if flag then
            grasped_waypoints ← copy(optpaths[grasped_obj[1]])
        end if
    end if
    return torques, q_ref
end procedure
```

5 Results and Discussion

5.1 Simulation Results

The simulation starts with the robot manipulator in a default configuration and the conveyor without any cube. The designed solution sends the robot from the default initial configuration to a home position inside the workspace between the conveyor and the arm. Inverse kinematics was used to compute the desired Joint configuration for the KUKA robot given an end effector desired position. Once we have the joint values the designed LQR controller is capable to use the desired joints with the current joint positions and velocities to compute the torque commands at the current timestep. The designed LQR controller achieved high accuracy in performing a smooth trajectory from the initial position to a final destination. However, the torque commands resulted in slow-motion behavior compared with the velocity of the conveyor belt. Given that we need to pick up objects moving at the conveyor's speed, the LQR parameters were tuned to increase the velocity of the end effector in the simulation. Additionally, the designed Cube class was capable to keep track of the trajectory of the cube while moving in the conveyor, and with this information, the task allocator class is capable to send the end effector to a predicted position, with this solution the effect of the delay was reduced and increased the efficiency of the manipulator to grasp the cubes in the conveyor.

The TaskAllocator class orchestrates the application receiving the position of the objects from the main loop and calling the Cubes instance to update the Cube objects. The Allocate function get always the old element in the conveyor and estimate the position some milliseconds ahead, then the function call the instance ManipulatorController to calculate the torque commands aiming to send the end effector to the desired position. This describes the first scenario when the robot approaches from the home position to grasp the object in the predicted position. Once the object is grasped we evidenced in the simulation that the robot reaches the home position and loads the waypoints trajectory to start the second scenario. The Allocate function loads the waypoints in base of the color received from the Cube, to then call the controller to compute the torque commands and send the end effector to each waypoint until reaches the destination and releases the object. Sometimes the manipulator does not achieve the goal to place the end effector in the tray position showing singularities, where the arm gets stuck. To solve the problem of singularities the Allocate function checks if the arm is stuck and loads an optional waypoint trajectory. Finally, in the last scene of the simulation, the robot releases the Cube and returns to the home position to be ready to pick up another Cube.

5.2 Discussion

The pick-and-place task was successfully executed by the robot in the majority of the simulations. The joint torque controller and the task orchestration demonstrated their effectiveness in solving the problem. However, in a few cases, the robot failed to reach the trays because of singularities. This problem exists because we are using a waypoint predefined trajectory which adds limitation in not having a smooth trajectory where we can compute the desired velocity and use it to compute the torque commands. The waypoints generated trajectory assumes zero desired joint velocities at each waypoint, which could generate high values in the resulted torque commands. This limitation could

be addressed by improving the trajectory generation for the scenarios of pick up and place.

The overall performance of the algorithm was satisfactory in terms of speed, accuracy, and stability. The arm behavior is smooth and controlled. However, adding a trajectory planner will solve the issue with the singularities and increase the efficiency of the application.

During the implementation, we faced challenges related to parameter tuning and optimization in the LQR controller, intending to increase speed without reducing accuracy and smoothness in the manipulator's behavior.

6 Conclusion

In this project, we designed and implemented a control algorithm for a robotic manipulator to perform a pick-and-place task with color-coded cubes on a conveyor system. The simulation results demonstrated satisfactory performance in terms of accuracy, efficiency, and smoothness. However, we still have limitations due to singularities in the arm because of the waypoint-generated trajectories. The proposed solution compounds a task orchestrator to handle the cubes on the conveyor and uses the control algorithm to manage the task until each cube reaches the desired location. Future work could involve designing a collision-free trajectory planner to improve the robot's stability and accurately pick up cubes from the conveyor system and place them in the color-coded trays.

7 Appendices

7.1 Cube python class

```
class Cube:
    """Represents a cube object in the simulation with tracking of
    ↪ position, trajectory and velocity.
    """
    def __init__(self, position, idx) -> None:
        """Initializes the Cube object.

        Args:
            position (tuple): The initial position of the cube.
            idx (int): The index of the cube.
        """
        self.last_position = position
        self.trajectory = []
        self.iter_time = 1/240
        self.idx = idx
        self.velocity = [0,0,0]
        self.time_prediction = 0.20 # seconds  0.38
        self.grasped = False
```

```
def update_position(self, position):
    """Updates the position of the cube and appends it to the
    ↪ trajectory.

    Args:
        position (tuple): The new position of the cube.
    """
    self.trajectory.append(position)
    self.last_position = position

def calculate_velocity(self):
    """Calculates the average velocity of the cube based on its
    ↪ trajectory."""
    # given the last positions of the cube calculate the velocity
    if len(self.trajectory) > 1:
        dist = []
        for i in range(len(self.trajectory) - 1):
            current = self.trajectory[i]
            last = self.trajectory[i+1]
            distance = self.calculate_distance(current, last)
            dist.append(distance)
        self.velocity = np.array(dist).mean(axis=0) / self.iter_time
        # print(self.velocity)

def predict_position(self):
    """Predicts the future position of the cube based on its current
    ↪ position and velocity.

    Returns:
        tuple: The predicted position of the cube.
    """
    # predict the next position of the object
    self.calculate_velocity()
    x, y, z = self.last_position
    vx, vy, vz = self.velocity
    t = self.time_prediction
    return (x - vx*t, y - vy*t, z + vz*t)

def calculate_distance(self, pos1, pos2):
    """Calculates the distance between two positions.

    Args:
        pos1 (tuple): The first position.
        pos2 (tuple): The second position.

    Returns:
        list: A list with the distances in each axis [x, y, z].
```

```
        """
        # calculate the distance between two objects
        x1, y1, z1 = pos1
        x2, y2, z2 = pos2
        return [x1 - x2, y1 - y2, z1 - z2]
        # return math.sqrt((x1 - x2)**2 + (y1 - y2)**2 + (z1 - z2)**2)
```

7.2 Cubes python class

```
class Cubes:
    """Manages multiple Cube objects in the simulation, handling updates,
    ↪  creation, and removal.
    """
    def __init__(self, iter_time, same_cube_threshold=0.1) -> None:
        """Initializes the Cubes object.

        Args:
            iter_time (float): The iteration time for the simulation.
            same_cube_threshold (float, optional): The distance threshold
↪   to consider two cubes as the same. Defaults to 0.1.
        """
        self.iter_time = iter_time
        self.cubes = []
        self.same_cube_threshold = same_cube_threshold
        self.idx = 0
        self.belt_end_position_x = -0.3
        self.belt_end_position_y_robotside = -0.25
        self.belt_end_position_y_beltside = -0.8

    def update_positions(self, positions):
        """Updates the positions of the cubes in the simulation or
        ↪  creates new ones if necessary.

        Args:
            positions (list): A list of tuples representing the positions
↪   of the cubes.
        """
        # check if the cube is already created
        # if not create the object
        # self.remove_out_of_workspace()
        for pos in positions:
            updated_flag = False
            for cube in self.cubes:
                # get the last position of the cube
                pos_cube = cube.last_position
                distance = self.calculate_distance(pos, pos_cube)
                # print(distance)
```

```
                if distance < self.same_cube_threshold:
                    # distance is less than threshold
                    # correspond to same cube
                    cube.update_position(pos)
                    updated_flag = True
                    break
            if updated_flag == False:
                # check if the cube is not in the belt
                self.create_cube(pos, self.idx)
                # print(f"created cube from total {len(self.cubes)}")
                self.idx += 1

    def remove_out_of_workspace(self):
        """Removes cubes that are outside of the belt's workspace."""
        init_val = len(self.cubes)
        for idx, cube in enumerate(self.cubes):
                # get the last position of the cube
                position = cube.last_position
                if position[0] < self.belt_end_position_x or position[1]
                ↪  > self.belt_end_position_y_robotside or position[1] <
                ↪  self.belt_end_position_y_beltside:
                    # self.cubes.pop(idx)

                    self.cubes.remove(cube)
        # print(f"init_val: {init_val}, final: {len(self.cubes)}")

    def create_cube(self, position, idx):
        """Creates a new Cube object with the given position and index.

        Args:
            position (tuple): The initial position of the cube.
            idx (int): The index of the cube.
        """
        # create the new cube object
        new_cube = Cube(position, idx)
        # update the current position of the object
        new_cube.update_position(position)
        # add to the cube list
        self.cubes.append(new_cube)

    def calculate_distance(self, pos1, pos2):
        """Calculates the Euclidean distance between two positions.

        Args:
            pos1 (tuple): The first position.
            pos2 (tuple): The second position.

        Returns:
```

```
            float: The Euclidean distance between the two positions.
        """
        # calculate the distance between two objects
        x1, y1, z1 = pos1
        x2, y2, z2 = pos2

        return math.sqrt((x1 - x2)**2 + (y1 - y2)**2 + (z1 - z2)**2)
```

7.3 ManipulatorController python class

```
class ManipulatorController:
    """
    This class implements the Linear Quadratic Regulator (LQR) control
↪   algorithm and computes torque control
    for a robotic manipulator. It also includes functions for joint angle
↪   control and inverse kinematics.
    """
    def __init__(self, pb_client, robot_id, arm_joint_indices):
        """
        Constructor for the ManipulatorController class.

        Args:
            pb_client: PyBullet client for the physics simulation.
            robot_id: The unique identifier of the robot in the
↪   simulation.
            arm_joint_indices: A list of joint indices for the robot's
↪   arm.
        """
        self.arm_joint_indices = arm_joint_indices
        self.pb_client = pb_client
        self.robot_id = robot_id

        ## Compute K_1 and K_2 for LQR control using control python
        ↪  package
        K_1 = np.zeros((7,7))
        K_2 = np.zeros((7,7))
        # Define state equation variables
        A1 = np.concatenate((np.zeros((7,7)), np.eye(7)), axis=1)
        A2 = np.concatenate((np.zeros((7,7)), np.zeros((7,7))), axis=1)

        self.A = np.concatenate((A1, A2), axis=0)
        self.B = np.matrix(np.concatenate((np.zeros((7,7)), np.eye(7))))
        self.Q = np.identity(14)

        for i in range(7):
            self.Q[i, i] = 80 #800
            self.Q[i + 7, i + 7] = 1 # 1
```

```
        # Modify the Q matrix diagonal values for end-effector joints
        for i in range(5, 7):  # Adjust the range if needed
            self.Q[i, i] = 3500  # Increase the value for joint angle
            ↪  error 2500
            self.Q[i + 7, i + 7] = 50  # Increase the value for joint
            ↪  velocity error 50

        self.R = 0.02*np.identity(7) # 0.1
        # compute lqr with control package
        print(f'Manipulator Controller: Comupte LQR control gain')
        self.compute_control_gain(self.A, self.B, self.Q, self.R)

    def compute_control_gain(self, A, B, Q, R):
        """
        Compute the control gain matrices K_1 and K_2 for the LQR control
↪   algorithm.

        Args:
            A: State-transition matrix.
            B: Control input matrix.
            Q: State cost matrix.
            R: Control input cost matrix.
        """
        control_gain, _, _ = control.lqr(A,B,Q,R)
        self.K_1 = -control_gain[:,:7] # get the K1
        self.K_2 = -control_gain[:,7:] # get the K2
        # print(f'K1: {self.K_1}')
        # print(f'K2: {self.K_1}')

    def compute_torque_control(self, joint_angles, joint_velocities,
    ↪  q_ref, q_dot_ref):
        """
        Compute the torque control for the robotic manipulator given
↪   joint angles and joint velocities.

        Args:
            joint_angles: A NumPy array of joint angles.
            joint_velocities: A NumPy array of joint velocities.
            q_ref: A NumPy array of reference joint angles.
            q_dot_ref: A NumPy array of reference joint velocities.

        Returns:
            tau: A NumPy array of computed torques.
        """
        theta_ddot = self.K_1.dot(joint_angles - q_ref) +
        ↪  self.K_2.dot(joint_velocities - q_dot_ref)
```

```
        tau = iterative_ne_algorithm(len(self.arm_joint_indices),
        ↪  joint_angles, joint_velocities, theta_ddot)
        return tau

    def joint_angle_control(self, ee_pose, ee_orientation):
        """
        Calculate the desired joint angles given end-effector pose and
↪   orientation.

        Args:
            ee_pose: A list containing the end-effector position [x, y,
↪   z].
            ee_orientation: A list containing the end-effector
↪   orientation as Euler angles [roll, pitch, yaw].

        Returns:
            joint_angle_desired: A NumPy array of desired joint angles.
        """
        joint_angle_desired = inverse_kinematics(self.pb_client,
        ↪  self.robot_id, ee_pose, ee_orientation)
        return np.array(joint_angle_desired)
```

7.4 TaskAllocator python class

```
class TaskAllocator:
    """
    This class is responsible for allocating tasks to the robotic
↪   manipulator, such as picking up objects
    and placing them in specified trays. The class contains various
↪   utility functions for calculating distances,
    manipulating waypoints, and avoiding singularities.
    """
    def __init__(self, controller:ManipulatorController, trays,
    ↪  tray_width, h=1/240) -> None:
        """Constructor TaskAllocator.

        Args:
            controller (ManipulatorController): An instance of the
↪   ManipulatorController class.
            trays (list): A list containing the positions of trays.
            tray_width (float): The width of the trays.
            h (float): The time interval for the trajectory.
        """
        self.controller = controller # use the manipulator controller to
        ↪  compute the torques
        self.trays = trays # positions of the trays default(red, green,
        ↪  blue)
```

```
self.tray_width = tray_width

self.home_pose = [[0,-.5,1.1], [np.pi,0,0]]
self.flag_obj_grasped = False
self.flag_release = False
self.grasped_waypoints = None
self.grasped_obj = -1

self.object_clearance = 0.10 # 0.12
self.distance_manipulator = 0.8 # 1.1
self.grasp_threshold = 0.38 # self.object_clearance
self.pose_threshold = 0.20
self.tray_clearance = 0.35
self.orientation_threshold = 0.5
self.time_tolerance = 150

# task assigned
self.previous_ee_position = None
self.count_singul = 0

# Smooth trajectory section
# Define the total time T and the time interval h
self.T = 10 # time estimated to complete the trajectory
self.h = 0.5

# cubes
self.cubes = Cubes(h)

# paths to each tray
self.trays[0][2] += self.tray_clearance
self.trays[1][2] += self.tray_clearance
self.trays[2][2] += self.tray_clearance
self.trays[0][0] -= 0.1 # update for red in x
self.trays[0][1] += 0.1 # update for red in x
# self.trays[2][0] += self.tray_clearance # update for blue in x
self.path_tray_1 = [self.home_pose[0], [-0.5,-0.25,0.94],
↪  [-0.7,0.3,0.92], self.trays[0], [-0.7,0.31,0.93],
↪  [-0.51,-0.26,0.91]]
self.path_tray_2 = [self.home_pose[0], [-0.5,-0.25,1.2],
↪  [-0.7,0.3,1.2], self.trays[1], [-0.7,0.3,0.9],
↪  [-0.5,-0.25,0.9]]
self.path_tray_3 = [self.home_pose[0], [0.4,-0.25,1.1],
↪  [0.5,0.3,1.1], [0.75,0.5,0.9], self.trays[2], [0.75,0.5,0.9],
↪  [0.5,0.3,1.1], [0.4,-0.25,1.1]]
self.paths = [self.path_tray_1, self.path_tray_2,
↪  self.path_tray_3]

# optional paths for the trajectories
```

```
        self.optpath_tray_1 = [self.home_pose[0], [0.4,-0.25,1.1],
        ↪  [0.5,0.3,1.1], [0.75,0.5,0.9], self.trays[1], self.trays[0],
        ↪  self.trays[1], [0.75,0.5,0.9],  [0.5,0.3,1.1],
        ↪  [0.4,-0.25,1.1]]
        self.optpath_tray_2 = [self.home_pose[0], [0.4,-0.25,1.1],
        ↪  [0.5,0.3,1.1], [0.75,0.5,0.9], self.trays[1], [0.75,0.5,0.9],
        ↪  [0.5,0.3,1.1], [0.4,-0.25,1.1]]
        self.optpath_tray_3 = [self.home_pose[0], [-0.5,-0.25,1.1],
        ↪  [-0.7,0.3,1.1], [0.25,0.6,0.9], self.trays[2], [0,0.4,1.1],
        ↪  [-0.7,0.3,1.1], [-0.5,-0.25,1.1]]
        self.optpaths = [self.optpath_tray_1, self.optpath_tray_2,
        ↪  self.optpath_tray_3]

    def allocate(self, objects, joint_angles, joint_velocities,
    ↪  end_effector_pose):
        """
        Determines the appropriate torque and joint angle commands for
↪   the manipulator
        based on the current state and environment.

        Args:
            objects (list): A list of objects with their positions.
            joint_angles (numpy.ndarray): The current joint angles of the
↪   manipulator.
            joint_velocities (numpy.ndarray): The current joint
↪   velocities of the manipulator.
            end_effector_pose (list): The current pose of the end
↪   effector.

        Returns:
            torques (numpy.ndarray): The computed torque commands for the
↪   manipulator.
            q_ref (numpy.ndarray): The computed joint angle commands for
↪   the manipulator.
        """
        # always allocate first object
        # print(end_effector_pose)
        self.cubes.remove_out_of_workspace()
        self.cubes.update_positions(objects)
        if self.flag_obj_grasped == False:

            if len(objects) > 0:
                # obj_pose_1 = objects[0]
                cube = self.cubes.cubes[0]
                obj_pose = cube.predict_position()
                obj_pose_last = cube.last_position
                # print(f'original pose: {obj_pose_1}, predicted:
                ↪  {obj_pose}')
```

```
            torques, q_ref = self.send_to_pose(joint_angles,
            ↪  joint_velocities, obj_pose)
            distance = self.calculate_distance(obj_pose_last,
            ↪  end_effector_pose[0])
            # error_orientation =
            ↪  self.calculate_orientation_error(self.home_pose[1],
            ↪  end_effector_pose[1])

            if distance < self.grasp_threshold:# and
            ↪  error_orientation < self.orientation_threshold:
                # print("Near to object. Performing action.")
                obj_grasped = grasp_object(self.controller.pb_client,
                ↪  self.controller.robot_id)
                if obj_grasped != -1 and self.flag_obj_grasped ==
                ↪  False:
                    self.flag_obj_grasped = True
                    self.grasped_obj = obj_grasped
                    self.grasped_waypoints =
                    ↪  copy(self.paths[obj_grasped[1]])
                    print(f"1. Picked up: {obj_grasped}")
                    # Delete cube from the pool
                    self.cubes.cubes.remove(cube)
                    return self.send_to_home(joint_angles,
                    ↪  joint_velocities)


            elif distance > self.distance_manipulator:# or
            ↪  error_orientation > self.orientation_threshold:
                return self.send_to_home(joint_angles,
                ↪  joint_velocities)
        else:
            torques, q_ref = self.send_to_home(joint_angles,
            ↪  joint_velocities)

    else:
        if len(self.grasped_waypoints) < 1 and self.flag_release:
            print("3. Return to next \n")
            self.flag_obj_grasped = False
            self.flag_release = False
            self.grasped_obj = -1
            self.grasped_waypoints = None
            return self.send_to_home(joint_angles, joint_velocities)



        # calculate distance to waypoint
        waypoint = self.grasped_waypoints[0]
```

```
            distance = self.calculate_distance(waypoint,
            ↪  end_effector_pose[0])
            torques, q_ref = self.send_to_pose(joint_angles,
            ↪  joint_velocities, waypoint)
            # check if the end effector is near to the waypoint and pass
            ↪  the next one
            # print(f"distance {distance}")
            if distance < self.pose_threshold:
                # del self.grasped_waypoints[0] # eliminate the achieved
                ↪  waypoint
                if waypoint[:2] == self.trays[self.grasped_obj[1]][:2]: #
                ↪  check if the waypoint is a tray
                    # release the object and go for the next
                    print("2. release object")
                    relase_object(self.controller.pb_client)
                    self.flag_release = True

                del self.grasped_waypoints[0] # eliminate the achieved
                ↪  waypoint

            # check if the arm get stuck
            flag = self.checking_singuarity(end_effector_pose[0])
            if flag:
                self.grasped_waypoints =
                ↪  copy(self.optpaths[self.grasped_obj[1]])



        return torques, q_ref

    def send_to_home(self, joint_angles, joint_velocities):
        """
        Computes the torque and joint angle commands to send the
↪   manipulator to its home pose.

        Args:
            joint_angles (numpy.ndarray): The current joint angles of the
↪   manipulator.
            joint_velocities (numpy.ndarray): The current joint
↪   velocities of the manipulator.

        Returns:
            torques (numpy.ndarray): The computed torque commands for the
↪   manipulator.
            q_ref (numpy.ndarray): The computed joint angle commands for
↪   the manipulator.
        """
        # calculate the joint angles desired
```

```
        q_ref = self.controller.joint_angle_control(self.home_pose[0],
        ↪  self.home_pose[1])
        q_dot_ref = np.zeros_like(q_ref)
        # send to calculate the torque control
        torques = self.controller.compute_torque_control(joint_angles,
        ↪  joint_velocities, q_ref, q_dot_ref)

        # return torques and joint_angles
        return torques, q_ref

    def send_to_pose(self, joint_angles, joint_velocities, pose):
        """
        Computes the torque and joint angle commands to send the
↪   manipulator to a target pose.

        Args:
            joint_angles (numpy.ndarray): The current joint angles of the
↪   manipulator.
            joint_velocities (numpy.ndarray): The current joint
↪   velocities of the manipulator.
        pose (list): The target pose for the manipulator (position and
↪   orientation).

        Returns:
            torques (numpy.ndarray): The computed torque commands for the
↪   manipulator.
            q_ref (numpy.ndarray): The computed joint angle commands for
↪   the manipulator.
        """
        # calculate the joint angles desired
        pose = [pose[0], pose[1], pose[2]+self.object_clearance]
        q_ref = self.controller.joint_angle_control(pose,
        ↪  self.home_pose[1]) # always maintain the pose of end effector
        q_dot_ref = np.zeros_like(q_ref)
        # send to calculate the torque control
        torques = self.controller.compute_torque_control(joint_angles,
        ↪  joint_velocities, q_ref, q_dot_ref)

        # return torques and joint_angles
        return torques, q_ref

    def calculate_distance(self, obj_pose, ee_pose):
        """
        Calculate the Euclidean distance between the end effector and the
↪   object pose.

        Parameters:
        obj_pose (tuple): (x, y, z) coordinates of the object pose.
```

```
        ee_pose (tuple): (x, y, z) coordinates of the end effector pose.

        Returns:
        float: Euclidean distance between the object and end effector
↪   poses.
        """
        """Calculate the distance between the end effector and the
        ↪  pose"""
        x1, y1, z1 = obj_pose
        x2, y2, z2 = ee_pose

        distance = math.sqrt((x1 - x2)**2 + (y1 - y2)**2 + (z1 - z2)**2)

        return distance

    def build_path(self, grasp_obj):
        """
        Create the waypoints to carry the object to the trays.

        Parameters:
        grasp_obj (tuple): A tuple containing object information.

        Returns:
        list: A list of waypoints to guide the manipulator from its home
↪   position to the target tray.
        """
        """Create the waypoints to carry the object to the trays"""
        way1 = self.home_pose[0]
        tray = self.trays[grasp_obj[1]]
        # way2 = [self.trays[1][0], self.trays[1][1],
        ↪  self.trays[1][2]+self.tray_clearance]
        way2 = [tray[0], tray[1], tray[2]+self.tray_clearance*1]
        # way3 = [tray[0], tray[1], tray[2]+self.tray_clearance]
        return [way1, way2]

    def calculate_orientation_error(self, desired, quaternion):
        """
        Calculate the orientation error between the desired and current
↪   end effector orientations.

        Parameters:
        desired (tuple): Desired (x, y, z) Euler angles.
        quaternion (tuple): Current end effector orientation in
↪   quaternion form.

        Returns:
        float: Euclidean distance representing the orientation error.
        """
```

```
        r = Rotation.from_quat(quaternion)
        ee_rot = r.as_euler('xyz', degrees=False)
        # calculate the error in orientation
        # print(f'desired: {desired}, current: {ee_rot}')

        return self.calculate_distance(desired, ee_rot)

    def checking_singuarity(self, ee_position):
        """
        Check if the manipulator is in a singular configuration or if it
↪   is stuck.

        Parameters:
        ee_position (tuple): (x, y, z) coordinates of the end effector
↪   position.

        Returns:
        bool: True if the manipulator is in a singular configuration or
↪   stuck, False otherwise.
        """
        flag = False
        if type(self.previous_ee_position) == np.ndarray:
            distance = self.calculate_distance(self.previous_ee_position,
             ↪  ee_position)
            self.previous_ee_position = ee_position
            # print(distance)
            if distance < 0.001:
                self.count_singul += 1
        else:
            self.previous_ee_position = ee_position

        # check if the arm is stuck
        if self.count_singul > self.time_tolerance:
            flag = True
            self.count_singul = 0
            print("Singularity")

        return flag
```