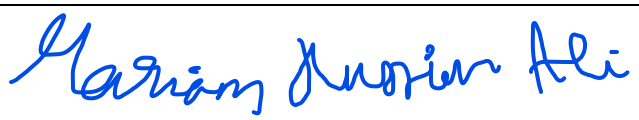
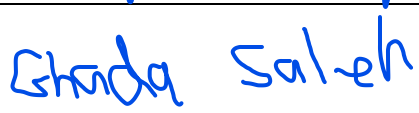
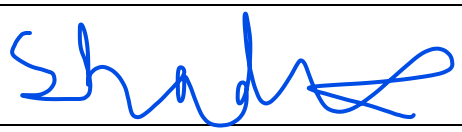
Faculty of Engineering
MCG 5353 – Robotics
Spring/Summer 2023
Dr. Amirhossein Monjazeb, P.Eng

Design and Development of a Robotic Sorting System for A Production Line

#	Group member's full last and first name	UOIT student ID number	Signature
1	Mariam Hussien	300389400	
2	Ibrahim Elshenhapy	300389386	
3	Ahmed Asem	300389894	
4	Ghada Salah	300389364	
5	Shady Osama	300389887	

Contents

Introduction

Open Manipulator Arms and Turtlebot are two distinct yet complementary robotic platforms that have gained significant attention in the field of robotics. These platforms are designed to perform various tasks in different environments, ranging from research and education to industrial applications.

In this project we used two open manipulator arms to perform master-slave connection between them, with one arm designated as the master and the other as the slave. The master arm's movements are tracked and duplicated by the slave arm in real-time. This setup enables the user to control the master arm while the slave arm automatically mirrors its motions. in addition to recording and performing the recorded trajectory when needed.

Objective

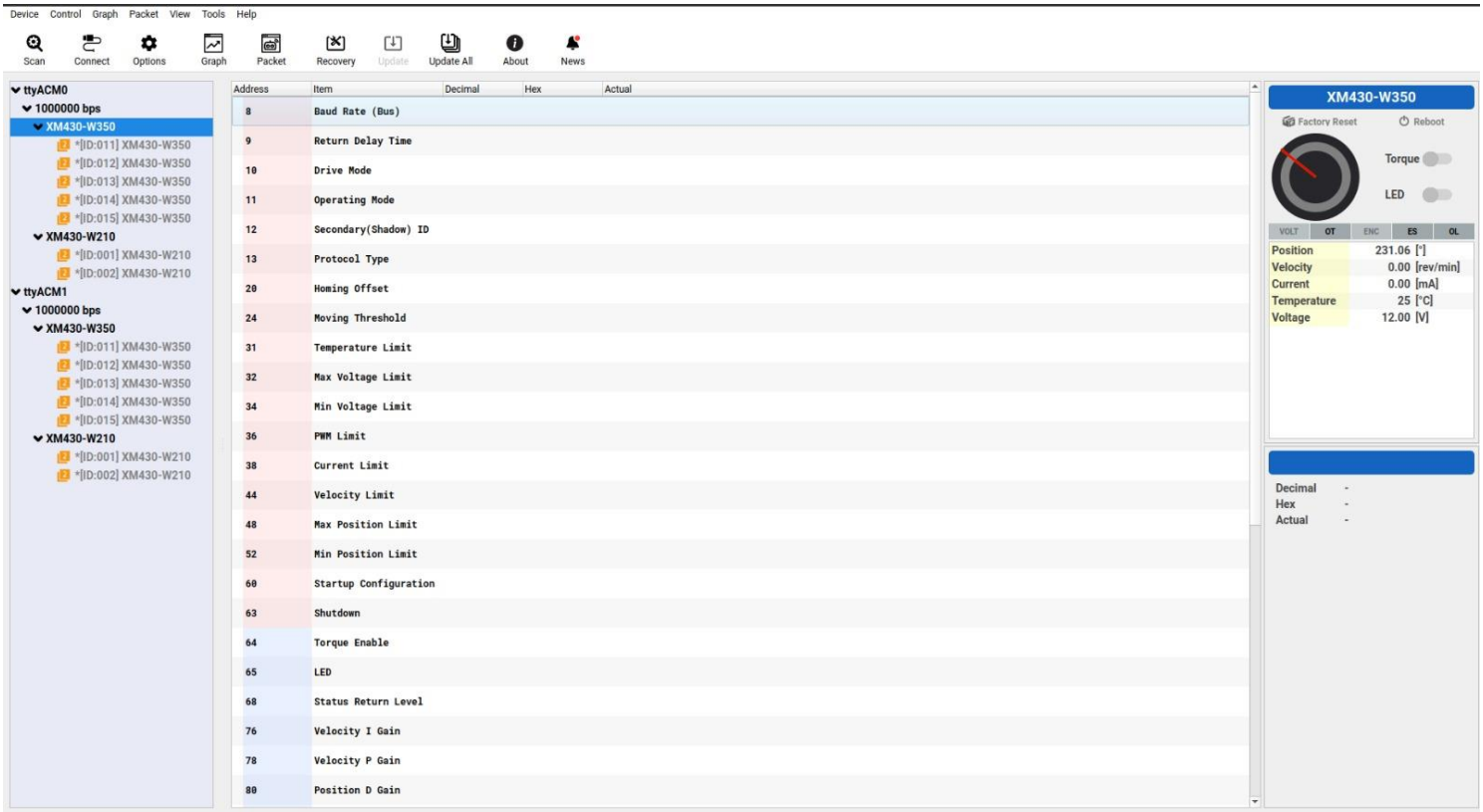
- Apply gravity compensation: The master robot should be configured to operate in gravity compensation mode, ensuring that it remains stable and unaffected by gravity forces.
- Slave mimics movement of the master
- Trajectory recording: the master trajectory is recorded, saved in a file and then played and performed by the slave.

Requirements

1. Apply gravity compensation: The master robot should be configured to operate in gravity compensation mode, ensuring that it remains stable and unaffected by gravity forces.
2. Slave mimics movement of the master
3. Trajectory recording: the master trajectory is recorded, saved in a file and then played and performed by the slave.

1- Configuration:

Before attempting to apply gravity on the master or mimic the master to the slave, we first have to do some configurations, we configure the arms using dynamixel software:



For the master arm we have to change the operation mode for the master arm to current control or current-based position control

2- Master Robot (Gravity Compensation Mode):

Gravity compensation for the OpenManipulator IS a technique used to counteract the effects of gravitational forces on the manipulator arm. Gravity can exert significant forces on the individual joints of a robotic arm, leading to unintended movements or inaccuracies in the arm's position. Gravity compensation aims to minimize these effects, allowing the manipulator arm to maintain its desired position and trajectory even when facing gravitational forces.

To apply gravity compensation in our master arm, we first have to get the plugin for gravity compensation, we open the terminal and write the following commands.

$ cd ~/catkin_ws/src/

$ git clone https://github.com/ROBOTIS-GIT/open_manipulator_controls.git

$ cd ~/catkin_ws && catkin_make

Then, we have to load it to our master arm, that is going to be done in a launch file that is used to launch the controllers of the master, gravity plugin and the slave controllers also.

This is how the master group in the launch file should look like:

```
<group ns="master">
    <include file="$(find open_manipulator_hw)/launch/open_manipulator_control.launch">
        <arg name="usb_port" value="/dev/ttyACM1"/>
        <arg name="interface" value="effort"/>
    </include>

    <include file="$(find open_manipulator_hw)/launch/controller_utils.launch"/>

    <rosparam file="$(find open_manipulator_controllers)/config/gravity_compensation_controller.yaml" command="load"/>
    <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false" output="screen" args="gravity_compensation_controller"/>
</group>
```

Figure 1 Robot links in a time instant.

In this file we assign the mater to be on USB_port of ACM1. This value changes depending on the hardware connection of the robot. Also, the controller is loaded using yaml file loading, then it is spawned in the master node. The controller gets the master to be free to move but locks the outside movement of the slave. It only allows the slave to move using orders from the controller.

```xml
<?xml version="1.0" ?>
<launch>
    <group ns="master">
        <include file="$(find open_manipulator_hw)/launch/open_manipulator_control.launch">
            <arg name="usb_port" value="/dev/ttyACM1"/>
            <arg name="interface" value="effort"/>
        </include>

        <include file="$(find open_manipulator_hw)/launch/controller_utils.launch"/>

        <rosparam file="$(find open_manipulator_controllers)/config/gravity_compensation_controller.yaml" command="load"/>
        <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false" output="screen" args="gravity_compensation_controller"/>
    </group>

    <group ns="slave">
        <include file="$(find open_manipulator_hw)/launch/controller_utils.launch"/>
        <rosparam file="$(find open_manipulator_controllers)/config/joint_trajectory_controller.yaml" command="load"/>
        <node name="arm_controller_spawner" pkg="controller_manager" type="controller_manager" respawn="false" output="screen" args="spawn arm_controller"/>
        <node name="gripper_controller_spawner" pkg="controller_manager" type="controller_manager" respawn="false" output="screen" args="spawn gripper_controller"/>
        <include file="$(find open_manipulator_moveit_config)/launch/move_group.launch"/>
    </group>
</launch>
```

3- Slave Robot (Mimicking Master):

The slave robot should replicate the movements of the master robot in real-time. Therefore we developed a control mechanism that receives the joint states or commands from the master robot and applies them to the slave robot with a python file.

In order to do the mimicking function we started exploring the topics which appear in the rqt after running the two arms.

The rqt command shows the following topics including "/master/joint_states" for us to get the position and required info from the master, we also find "/slave/arm_controller/command" which we use to publish on the slave.

Topic	Type	Bandwidth	Hz	Value
/calibrated	std_msgs/Bool			not monitored
/master/joint_states	sensor_msgs/JointState			not monitored
/rosout	rosgraph_msgs/Log			not monitored
/rosout_agg	rosgraph_msgs/Log			not monitored
/slave/arm_controller/follow_joint_trajectory/cancel	actionlib_msgs/GoalID			not monitored
/slave/arm_controller/follow_joint_trajectory/feedback	control_msgs/FollowJointTrajectoryActionFeedback			not monitored
/slave/arm_controller/follow_joint_trajectory/goal	control_msgs/FollowJointTrajectoryActionGoal			not monitored
/slave/arm_controller/follow_joint_trajectory/result	control_msgs/FollowJointTrajectoryActionResult			not monitored
/slave/arm_controller/follow_joint_trajectory/status	actionlib_msgs/GoalStatusArray			not monitored
/slave/arm_controller/state	control_msgs/JointTrajectoryControllerState			not monitored
/slave/execute_trajectory/feedback	moveit_msgs/ExecuteTrajectoryActionFeedback			not monitored
/slave/execute_trajectory/result	moveit_msgs/ExecuteTrajectoryActionResult			not monitored
/slave/execute_trajectory/status	actionlib_msgs/GoalStatusArray			not monitored
/slave/gripper_controller/follow_joint_trajectory/cancel	actionlib_msgs/GoalID			not monitored
/slave/gripper_controller/follow_joint_trajectory/feedback	control_msgs/FollowJointTrajectoryActionFeedback			not monitored
/slave/gripper_controller/follow_joint_trajectory/goal	control_msgs/FollowJointTrajectoryActionGoal			not monitored
/slave/gripper_controller/follow_joint_trajectory/result	control_msgs/FollowJointTrajectoryActionResult			not monitored
/slave/gripper_controller/follow_joint_trajectory/status	actionlib_msgs/GoalStatusArray			not monitored
/slave/gripper_controller/state	control_msgs/JointTrajectoryControllerState			not monitored
/slave/joint_states	sensor_msgs/JointState			not monitored
/slave/move_group/display_contacts	visualization_msgs/MarkerArray			not monitored
/slave/move_group/display_planned_path	moveit_msgs/DisplayTrajectory			not monitored
/slave/move_group/feedback	moveit_msgs/MoveGroupActionFeedback			not monitored
/slave/move_group/monitored_planning_scene	moveit_msgs/PlanningScene			not monitored
/slave/move_group/ompl/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/slave/move_group/ompl/parameter_updates	dynamic_reconfigure/Config			not monitored
/slave/move_group/plan_execution/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/slave/move_group/plan_execution/parameter_updates	dynamic_reconfigure/Config			not monitored
/slave/move_group/planning_scene_monitor/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/slave/move_group/planning_scene_monitor/parameter_updates	dynamic_reconfigure/Config			not monitored
/slave/move_group/result	moveit_msgs/MoveGroupActionResult			not monitored
/slave/move_group/sense_for_plan/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/slave/move_group/sense_for_plan/parameter_updates	dynamic_reconfigure/Config			not monitored
/slave/move_group/status	actionlib_msgs/GoalStatusArray			not monitored
/slave/move_group/trajectory_execution/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/slave/move_group/trajectory_execution/parameter_updates	dynamic_reconfigure/Config			not monitored
/slave/pickup/feedback	moveit_msgs/PickupActionFeedback			not monitored
/slave/pickup/result	moveit_msgs/PickupActionResult			not monitored
/slave/pickup/status	actionlib_msgs/GoalStatusArray			not monitored
/slave/place/feedback	moveit_msgs/PlaceActionFeedback			not monitored
/slave/place/result	moveit_msgs/PlaceActionResult			not monitored
/slave/place/status	actionlib_msgs/GoalStatusArray			not monitored
/tf	tf2_msgs/TFMessage			not monitored
/tf_static	tf2_msgs/TFMessage			not monitored

We have to subscribe on the joint states topic for the master to be able to get its positions and then publish the message to the slave arm, so we initialize our publisher and subscriber. In the publisher we use Join trajectory while in the subscriber we use joint states. Also we add the queue size to be 10 to make the process synchronous and prevent any congestion.

```
# Initialize ROS publishers and subscribers
self.joint_pub = rospy.Publisher('/slave/arm_controller/command', JointTrajectory, queue_size=10)
self.joint_cmd_sub = rospy.Subscriber('/master/joint_states', JointState, self.joint_states_callback)
```

We use the callback function used in the subscriber to execute the functions we want during runtime:

This call function is executed whenever the Topic is published to. The function has the message as an argument. It checks on the current robot mode. If It is in the mimicking mode it updates the slave position variable and call the update function. On the other hand, if it is in the recording mode it appends every step of the movement to the trajectory variable.

```
def joint_states_callback(self, msg):
    if self.is_mimicking:
        self.joint_position.position = msg.position [1:]
        self.update_slave_mimic()

    if self.is_recording:
        self.mimic_trajectory.append(msg)
```

Then we wrote the function that does the mimicking. In every call, the function updates the new positions in a point variable. Then, it updates the slave trajectory points with the new point variable that holds the updated positions. After that, it published the new point to the slave robot to move to and follow the master.

```
def update_slave_mimic(self):
    self.slave_joint_trajectory_points.positions = self.joint_position.position
    self.slave_joint_trajectory_points.time_from_start = rospy.Duration(0.1)
    self.slave_joint_trajectory.points = [self.slave_joint_trajectory_points]

    self.joint_pub.publish(self.slave_joint_trajectory)
```

•Ensure that the slave robot accurately mimics the master robot's position and orientation

4- Trajectory Recording:

Recording Trajectory:

In Trajectory recording we used ROS Bag file to save our trajectory. Ros bags stores the messages as an indexable object in the ROS Bag to retrieve it again later after the execution of the program stops. In our project we added a record trajectory function in the class where it gets the file name as a parameter. Then we add an error handling condition to edit the file extension to be .bag file. Then if the robot is in the recording mode it writes in the file and adds each trajectory message point one by one until the recording stops.

```
def record_trajectory(self,filename):
    if not filename.endswith('.bag'):
        filename += '.bag'

    if self.is_recording:
        self.record_bag = rosbag.Bag(filename, 'w')

        for point in self.mimic_trajectory:
            self.record_bag.write('/master/joint_states', point)

        self.record_bag.close()
        print('Recorded trajectory to {}'.format(filename))
```

Playback Trajectory:

In the playback trajectory we get the file name and the class arguments. Then we add an error handling condition to check that the ROS Bag name is used correctly with the .bag extension. Then we read the ROS Bag file and separating the message and topic in different variables. After that we check if the topic in the class in the correct topic (master/joint_states) we get the position one by one and update the class joint_position variable to get the new position. Then we call the update slave mimic function to publish the new position to the slave robot. Hence, the slave robot follows the trajectory of the recorded master robot.

```
def playback_trajectory(self,filename):
    if not filename.endswith('.bag'):
        filename += '.bag'

    playback_bag = rosbag.Bag(filename, 'r')

    for topic, msg, t in playback_bag.read_messages():
        if topic == '/master/joint_states':

            self.joint_position.position = msg.position [1:]
            self.update_slave_mimic()
            rospy.sleep(0.01)

    playback_bag.close()
```

5- Running the file:

In the run function, we assign the publishing rate to be 10. Then while the program is running we gets inputs from te user to raise flags and activates that activates the function. For each assigned keyboard key, we raise a corresponding flag that activates a different function.

```
def run(self):
    rate = rospy.Rate(10)
    while not rospy.is_shutdown():
        option = input("Enter option: ")

        if option == '1':
            self.is_mimicking = True
            print("Mimicking Master Movements...")

        elif option == '2':
            self.is_mimicking = False
            print("Stop The Mimic ...")

        elif option == '3':
            self.mimic_trajectory = []
            self.is_recording = True
            print("Recording Mimic Trajectory... Press 4 or 5 to stop recording and save the file")

        elif option == '4':
            filename = input("Enter the filename to record: ")
            self.record_trajectory(filename)
            print("Stopped Recording Mimic Trajectory.")
            self.is_recording = False
```

```
        elif option == '5':
            self.is_mimicking = False
            if self.is_recording:
                filename = input("Enter the filename to record at: ")
                self.record_trajectory(filename)
                print("Stopped Recording Mimic Trajectory.")
                self.is_recording = False

            print("Playing Recorded Mimic Trajectory...")
            filename = input("Enter the filename to play: ")
            self.playback_trajectory(filename)
            print("Finished")

        else:
            print("Invalid option. Please enter a valid option.")

        rate.sleep()
```

On the terminal:

This the terminal results of the function that appears to the user to choose from.

```
Mimic Controller
Options:
1: Start Mimicking Master Movements
2: Stop Mimicking Master Movements
3: Start Recording Mimic Trajectory
4: Stop Recording Mimic Trajectory
5: Play Recorded Mimic Trajectory
Enter option:
```

Future Work

In future work we plan to add the gripper functionality too. Also , we would add a trajectory order using the terminal where the user gives order to the master or the slave to move to a specified point by publishing to their topics. Further more, we would add some functionality to the program to read all the previously saved ROS bags to choose from and an option of running the last saved one. Also, we would add a third mode where it combines the following with the recording. Furthermore, we would add a functionality to play the ROS bag file using the master robot as well.

Conclusion

In conclusion, the two open manipulator robots are connected at the controller. The robots have gravity compensation where they are held at the desired position and don't fall when they stop moving. They have different modes chosen by the user to control them. In the mimicking mode, the slave follows the master movement. In the recoding mode, the master movements are recorded in a bag file to be played later by the slave. Different files are saved to save different trajectory. Then in the play recorded mode, the slave follows the chosen recorded trajectory. The user might stop any mode and start another having a full control of the robots.

References

1. OpenMANIPULATOR-X: Robotis e-manual (https://emanual.robotis.com/)