ECE4310/CIE6042 Programming for Robotics

Project Report 02

Project 02: Warehouse Robot

Name: Yang Jiao

Student ID: 118010121

Date: 25 May 2022

School of Science and Engineering

The Chinese University of Hong Kong, Shenzhen

2021-2022 Term 2

Contents

I. Introduction

Project 2 is aimed at to integrate the ROS programming concepts and packages learned from previous labs (especially Lab 6) and to build a mobile warehouse robot accordingly. The warehouse robot, which is implemented as a manipulator, is required to perform pick-and-place task, coordinated by an external camera. The sorting system is shown as in Fig. 1. The manipulator is a six-axis robot arm and has a gripper actuator that can be mounted on the end of the arm. The drive system consists of servo motors, with one motor controlling one axis. Sensing systems, which is camera-based vision sensing system, is installed on the motor of the gripper. The detection area of the camera is shown as the rectangular area in Fig. 1. Cubes with color blue, red, and green are randomly placed inside of the detection area. The manipulator is required to sort the cubes to the trash bins with corresponding colors.

MoveIt! with RViz interface was utilized in this project to plan and control the movement of the robot arm. The color of the cubes is distinguished by the H parameter (hue) in HSV. With eye-in-hand calibration, the manipulator (including the robot arm and the gripper as the end effector) can work together with the monocular camera. In this report, Section II introduce the planning algorithm of the sorting system. Section III states a problem (about camera calibration) encountered during the project implementation and the corresponding solution. Section IV presents the demonstration and Section V concludes.

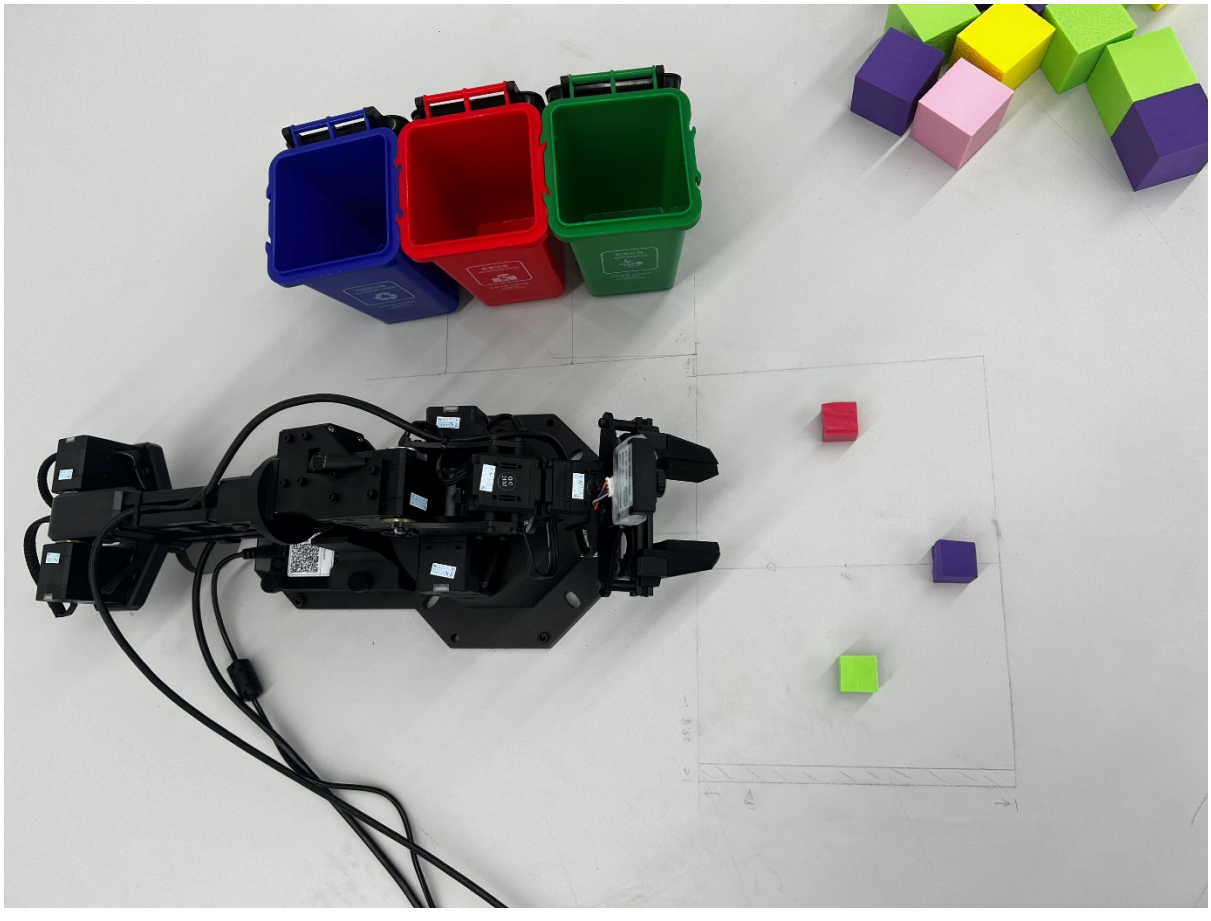


Fig. 1: The intelligent sorting system

II. Methodology

A. Preparation

Before realizing the sorting system, color calibration and robot arm eye-in-hand calibration were carried first. The color calibration allows the program to distinguish different colors by separating intervals of the hue values. During the color calibration procedure, we first executed the following commands in separated terminals:

```
$ roslaunch interbotix_moveit interbotix_moveit.launch robot_name:=wx250s
use_actual:=true dof:=6      # Launch MoveIt! RViz interface
$ roslaunch object_color_detector usb_cam.launch      # Launch the USB camera
$ rqt_image_view              # View the received image from the camera
```

The image containing a cube with the target color in ***rqt_image_view*** was saved and placed under the path ***object_color_detector/tool***. Another terminal in this path was opened and the following command was entered by replacing the file name with the saved file name:

$. /para_test <name_of_my_image>.png

Another window was opened and the hue value can be adjusted to a proper range, such that only the cube's color was recognized (Fig. 2). The color calibration procedure was repeated for all of the three colors, red, green, and blue, and the obtained values of *hmin* and *hmax* were filled in the file ***object_color_detector/config/vision_config.yaml***.

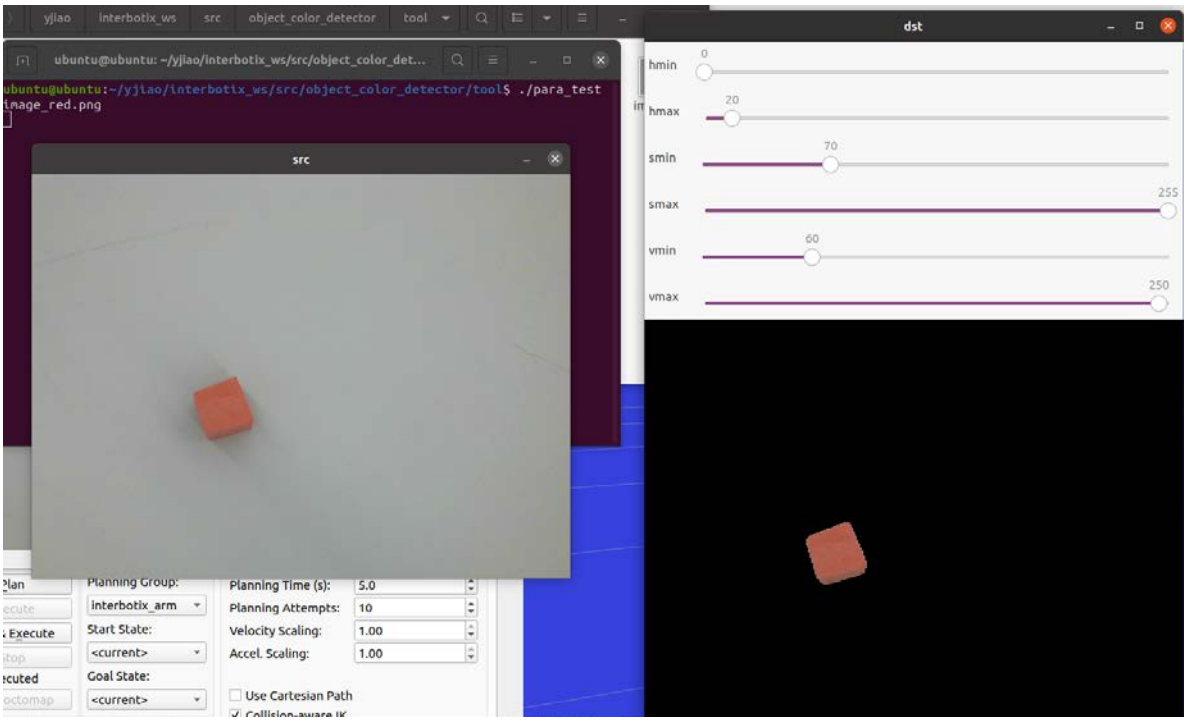


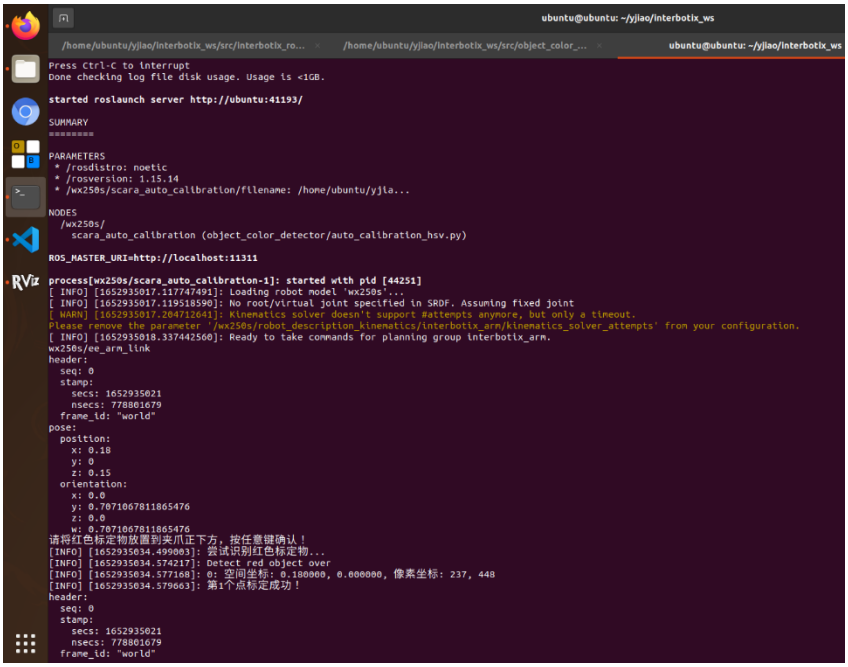
Fig. 2: Color detection of a red cube

The robot arm eye-in-hand calibration calculates a linear transformation from the pixel position of the detected cube in the camera image to the 2-dimensional location (only contains the x and y information) in the base frame of the robot arm. A few points were selected from the sorting filed, and the transformation equation was obtained from linear regression. To run the camera

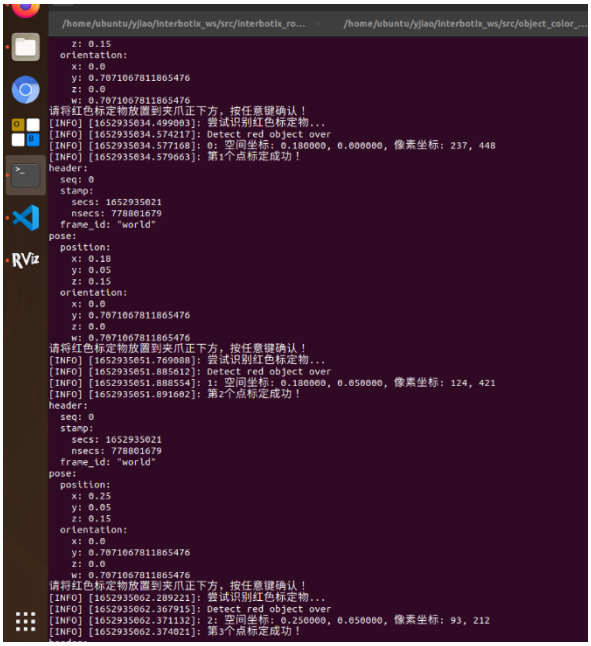
calibration program, the following commands were executed:

```
$ roslaunch interbotix_moveit interbotix_moveit.launch robot_name:=wx250s
use_actual:=true dof:=6      # Launch MoveIt! RViz interface
# Launch the colored cube detection program, where the calibrated hue ranges were adopted.
$ roslaunch object_color_detector object_detect_hsv.launch
# Launch the calibration program
$ roslaunch object_color_detector camera_calibration_hsv.launch __ns:=wx250s
```

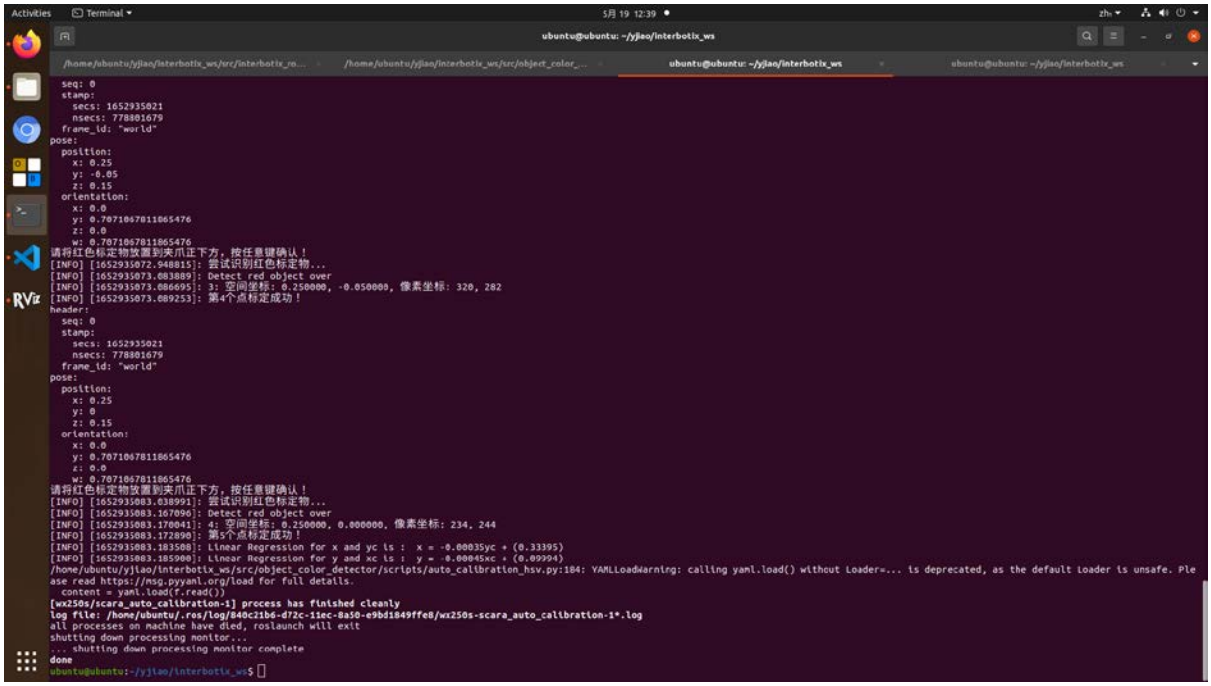
The terminal output was shown in Fig. 3 and poses of the manipulator during the calibration was presented in Fig. 4.



(a)

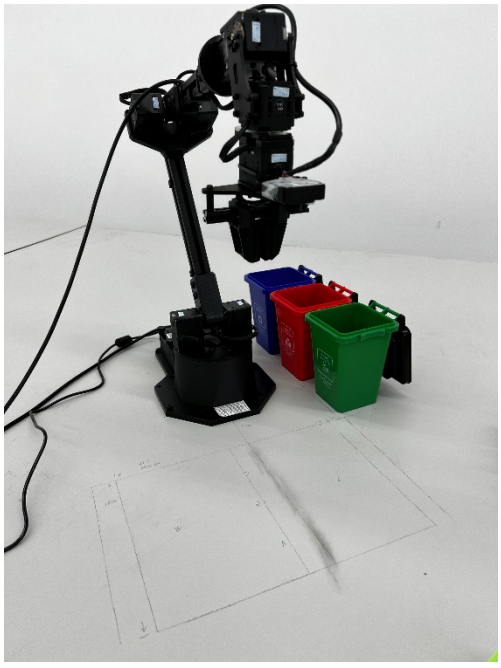


(b)

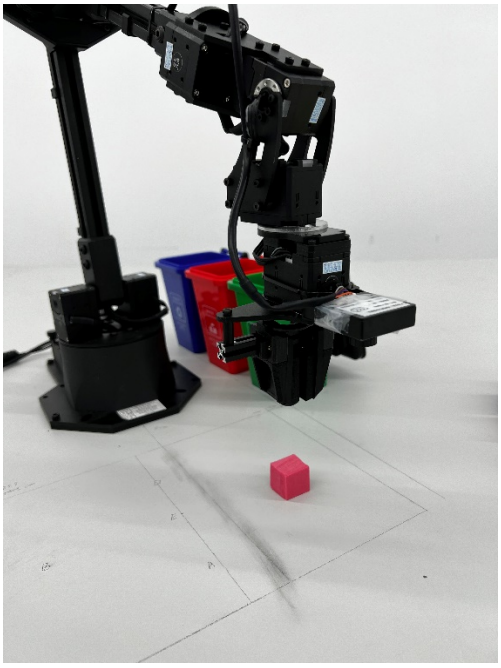


(c)

Fig. 3: The terminal output during the camera position calibration



(a) Pose for image capturing



(b) Pose for a calibration point

Fig. 4: The calibration poses of the robot arm and the gripper

B. Planning Algorithm

With a certain goal pose assigned to the end effector, MoveIt! can solve the inverse kinematics, plan and execute to reach the goal pose. One can also use cartesian path planning method to let the end effector moving along a preset path. In a nut shell, MoveIt! implements the low-level planning and execution of the manipulator, and our goal is to design a robust high-level

planning algorithm to achieve the intelligent sorting system.

Fig. 5 shows the flow chart of the high-level planning algorithm. Step 1 and step 6 only execute once when the sorting system is launched and when the sorting task finished respectively. The program will shut down if the camera does not detect any cubes on the sorting filed for a certain time interval $\mathcal{D}$ (Here we set $\mathcal{D} = 10$s). The initialization includes:

(1) Define arm and gripper in MoveIt;
(2) Define the camera as an attached object to the robot arm and the trash bins as obstacles in the workspace;
(3) Load camera calibration coefficients.

The program shutdown includes:

(1) Move the gripper to the *Home* pose;
(2) Move the robot arm to the *Sleep* pose;
(3) Shutdown the MoveIt commander.

Step 2 and Step 3 can be executed repeatedly if the grasping failed. The program regards a grasping as failed when the program detects the same number of cubes of the same color from the captured image after the grasping executed. Step 4 will be executed if the grasping is successful, and then goes back to the capture pose immediately (step 5).

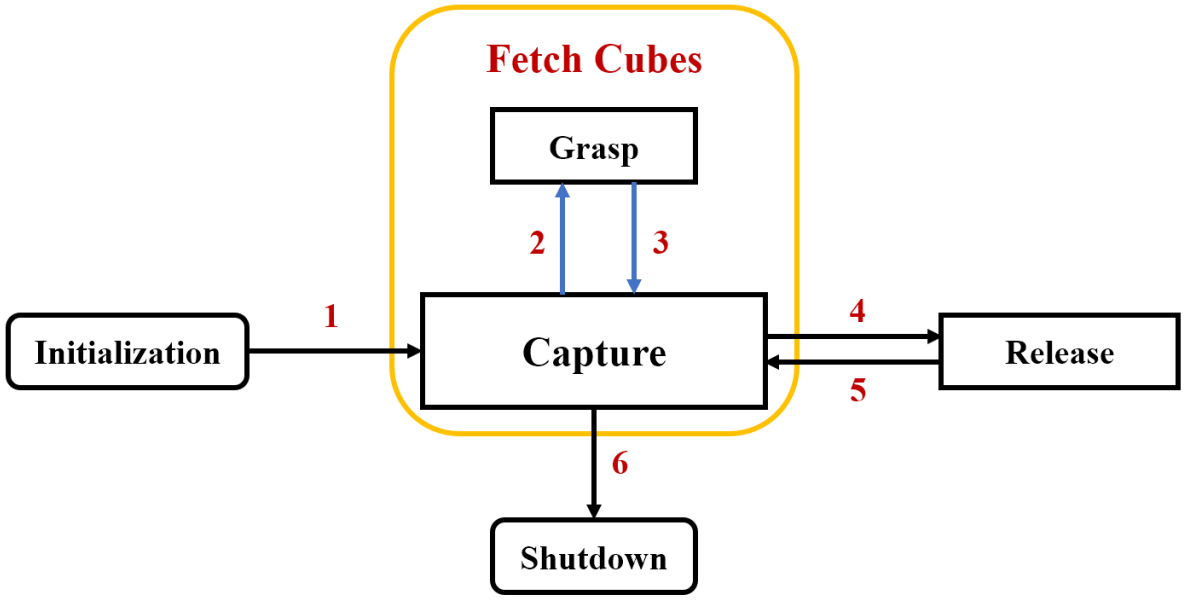


Fig. 5: The flow chart of the high-level planning for the warehouse robot

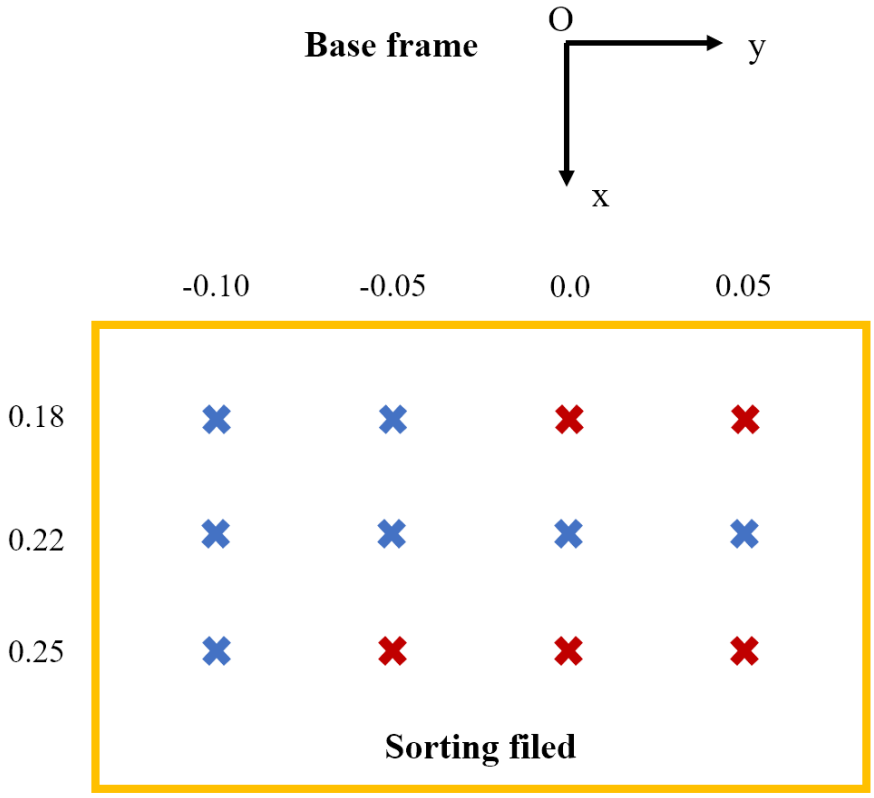
III. Trouble Shooting and Improvement

During the task execution, it was found that the gripper failed to move to a proper grasping position because of the calibration error. The camera is not located right above the x-axis in the base frame, which makes the sorting filed is shifted to the left side for a certain distance. As shown in Fig. 6 (b), the default calibration points cannot cover the left part of the sorting field. Therefore, more calibration points were added (Fig. 6 (a)) and the camera position calibration was executed again to obtain more accurate regression parameters. Moreover, the calibration

point height was also modified (Fig. 6 (a) line 30). This makes the calibration pose of the gripper closer to the table and thus decrease the human error as the camera calibration procedure requires placing the testing cube under the gripper manually.

```
29  # 定义标定位姿点
30  calibration_points_z = 0.11
31  calibration_points = []
32  calibration_points.append(Point(0.18, 0.05,  calibration_points_z))
33  calibration_points.append(Point(0.18, 0,     calibration_points_z))
34  calibration_points.append(Point(0.18, -0.05,  calibration_points_z))
35  calibration_points.append(Point(0.18, -0.1,  calibration_points_z))
36  calibration_points.append(Point(0.22, 0.05,     calibration_points_z))
37  calibration_points.append(Point(0.22, 0,     calibration_points_z))
38  calibration_points.append(Point(0.22, -0.05,     calibration_points_z))
39  calibration_points.append(Point(0.22, -0.1,     calibration_points_z))
40  calibration_points.append(Point(0.25, -0.1, calibration_points_z))
41  calibration_points.append(Point(0.25, -0.05, calibration_points_z))
42  calibration_points.append(Point(0.25, 0,     calibration_points_z))
43  calibration_points.append(Point(0.25, 0.05,  calibration_points_z))
```

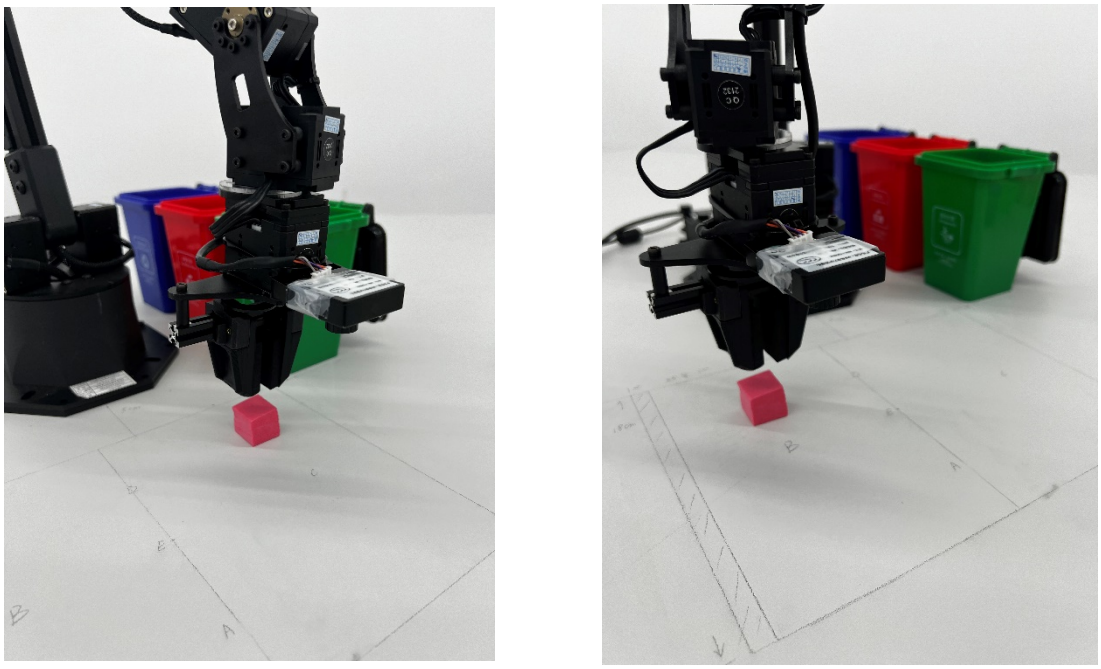
(a) Modification in the code



(b) Diagrammatic drawing of the adjustment on the calibration points. The red x-shaped points are the default calibration points. The blue x-shaped points are the added calibration points

Fig. 6: Adjustment on the camera calibration points

Fig. 7 also shows that the sorting filed is not symmetric about the x-axis of the base frame. Thus, the added calibration points are demanded.



(a) Gripper at x = 0.18m, y = 0.05m (b) Gripper at x = 0.18m, y = -0.10m

Fig. 7: Two clips of the camera position calibration procedure after the adjustments

IV. The Sorting Experiment

Fig. 1 shows the environment setup for the intelligent sorting system and Section IIA already introduces the camera calibration procedure. In this section, some clips from the demonstration video are provided (Fig. 8) to show the effectiveness of the warehouse robot system. Notice that the execution flow for the grasping and releasing of all cubes is the same, which Fig. 8 omits some repeated clips.

(a) First capture (b) Grasp the red cube (c) Move back to the capture pose

(d) Deposit the red cube

(e) Finished depositing

(f) Move back to the capture pose

(g) Grasp the green cube

(h) Check if grasp success at the capture pose

(i) Deposit the green cube

(j) Grasp the blue (replaced by purple) cube

(k) Deposit the blue cube

Fig. 8: The sorting procedure

The following commands are required to launch the sorting system:

```
$ roslaunch interbotix_demos moveit_sorting.launch
$ rosrun interbotix_demos sorting.py __ns :=wx250s
```

V. Conclusion

In Project 2, we have successfully implemented the warehouse robot, which can sort the colored cubes to their corresponding bins. To achieve this objective, two calibrations, color calibration and camera position calibration were first proceeded. Then a high-level sorting planning algorithm was implemented to coordinate the camera detection information with the robot arm and gripper movement. Finally, the hardware sorting experiment was realized and the manipulator can work properly.