ECE4310: Programming for Robotics, Project 2 Technical Report

Project 2: Warehouse Robot

Zixing JIANG
School of Science and Engineering
The Chinese University of Hong Kong, Shenzhen
119010130@link.cuhk.edu.cn

***Abstract*—This text is a technical report of ECE4310 Project 2. In this project, students are asked to implement an intelligent sorting program on a manipulator to perform automatic block-sorting. The manipulator is visually guided by a camera mounted co-axially with the end-effector. The algorithm is deployed on an Interbotix 6-DoF robotic manipulator. Test results shown that the algorithm can perform accurate 3-block sorting within 30s. The project handout and material are available at this link, and a video demonstration is available at this link.**

***Index Terms*—Warehouse Robot, Manipulator, Sorting, ROS**

I. Introduction

In this project, students are asked to program a 6-DoF robotic manipulator with a camera mounted co-axially with the end-effector to perform an auto-sorting task. An illustration of the task set-up is shown in Fig. 1. There are three different colored blocks (red, green, and blue) placed in front of the manipulator. And there are three different colored bins (red, green, and blue) placed on the right of the manipulator. The manipulator is required to grasp the blocks and place them in the bins with same color.

This report is organized as follows. Section II introduces robotic perception involved in this project, including *object detection* and *eye-in-hand calibration*. Section III explains how the sorting program is implemented, including *overall sorting logic*, *actuation*, and *obstacle avoidance*. Section IV addresses some efforts the author made to improve sorting accuracy and speed. Section V provides a demonstration of a sorting task. Section VI leaves some instructions on reproduce this project and lists some supplementary information.

II. Perception

A. Object Detection

Object detection is implemented based on HSV-color detection. Objects are highlighted from the image feed by HSV-masks. In this project, the range of S-value and V-value is fixed ($S \in [70, 255]$, $V \in [60, 250]$), the range of H-value varies based on objects' colors. The value of H is measured with the "*para_test*" tool provided according to the instructions

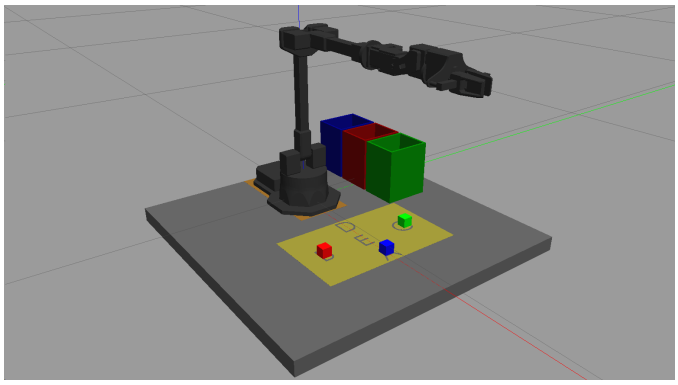


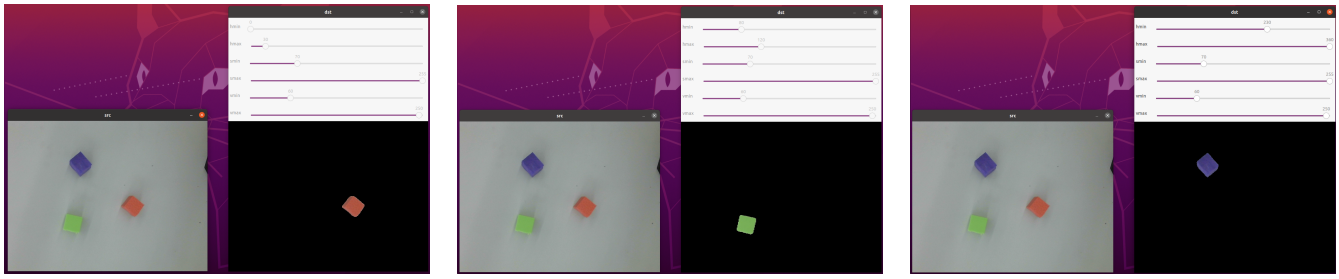
Fig. 1. A illustration of the intelligent sorting task. The 6-DoF robotic manipulator is required to recognize different colored blocks in front of it, grasp and place them in the corresponding colored bins. The entire process should be performed fully automatically.

in project handout section 1.3 "Visual recognition parameter adjustment". The measured H-value range is:

$$Red : 0 \sim 30$$
$$Green : 80 \sim 120$$
$$Blue : 230 \sim 360$$

The measuring process is illustrated in Fig. 2. These parameters are filled into the configuration file "*src_object_color_detector/config/vision_config.yaml*". When the parameters are set, object detection can be done by requesting the ROS service "*/object_detect*". This service will return a data structure named "*DetectObjectSrv*", which contains four attributes: *result*, *redObjList*, *greenObjList*, *blueObjList*, and *blackObjList*. The last four are lists that store the poses of the corresponding color objects in the camera frame. An example is presented in Listing 1.

```
result: 0
redObjList: # list of detected red objects
  -
    position:
      x: 254.0
      y: 365.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
```



(a) Red: $h_{max} = 30$, $h_{min} = 0$ (b) Green: $h_{max} = 120$, $h_{min} = 80$ (c) Blue: $h_{max} = 360$, $h_{min} = 230$

Fig. 2. Measure HSV parameters

```
        z: 0.0
        w: 0.0
greenObjList: [] # list of detected green objects
blueObjList: [] # list of detected blue objects
blackObjList: [] # list of detected black objects
```

Listing 1. An illustration of the returned data structure. In this example, the algorithm detects one red object.

B. Eye-in-hand Calibration

The calibration process is performed according to project handout, section 1.4 "*Robotic arm eye-in-hand calibration*". The purpose of eye-in-hand calibration is to find a mapping between coordinate of pixel in camera frame and coordinate of manipulator end-effector in world frame. In this project, we only consider the 2D XoY calibration, and the mapping is found by linear regression. The regression problem is formulated as follows. Suppose the coordinates of target object in world frame are x_w and y_w, the coordinate of target object in camera frame are x_c and y_c. By linear mapping, we have

$$\begin{bmatrix} x_w \\ y_w \end{bmatrix} = \mathbf{k}^\top \begin{bmatrix} y_c \\ x_c \end{bmatrix} + \mathbf{b}$$

The reason why the x and y axis in these two frames are upside-down is that their coordinates are upside-down. In this project, these coefficients are solved by taking 5 samples. The results are[1]

$$\mathbf{k} = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} -0.00035 \\ -0.00041 \end{bmatrix}$$
$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 0.30600 \\ 0.09525 \end{bmatrix}$$

These four coefficients are encapsulated as class attributes in the sorting program.

III. Sorting

A. Sorting Logic

The logic of sorting is described in Fig. 3. In each iteration, the manipulator first go to the calibration pose and request object detection. Once the pose of the target object in image frame in acquired, the manipulator will compute its pose in world frame according to eye-in-hand calibration results. Then, the end-effector of the manipulator will go to this pose, grasp the object, and then place it in the corresponding bin. This detect-grasp-place loop is performed in each iteration until the sorting task is completed. How this logic is designed will be introduced in Section IV. Optimization.

[1]The serial number of the calibrated manipulator is NXWI2022010507

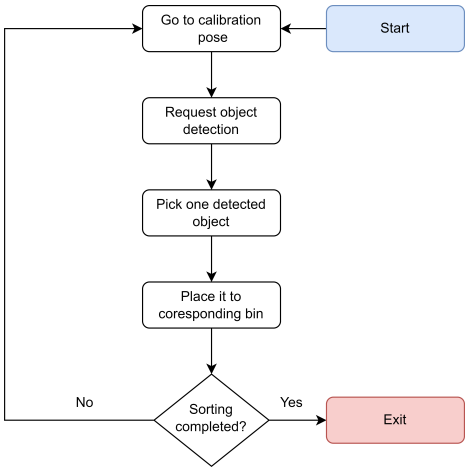


Fig. 3. Flow chart of the sorting logic. The manipulator performs detect-grasp-place loops until sorting completion.

B. Actuation

Both pose-only kinematics and Cartesian path planing are deployed. The following lists some important poses than the actuation methods to achieve them. For more code-level information like called API, please refer to the co-submitted source code.

1) Calibration pose: The calibration pose is reached by simple pose-only kinematics. The coordinates of this pose are:

$$position.x = 0.15$$
$$position.y = 0$$
$$position.z = 0.3$$
$$orientation.x = 0$$
$$orientation.y = \frac{\sqrt{2}}{2}$$
$$orientation.z = 0$$
$$orientation.w = \frac{\sqrt{2}}{2}$$

2) grasping pose: Since grasping required high accuracy, the simple pose-only kinematics may not apply because it may cause undesired path and collision. Therefore, Cartesian path planing is deployed for reaching grasping pose. Waypoints for path planing are calibration pose (start) and detected object pose in world frame (end). The vertical displacement is set as -55 cm.

3) Placing pose: Same with calibration pose, the placing poses is reached by pose-only kinematics. The poses of these three bins are given by manipulator 6-axis representation.

$$\text{Red bin} = \begin{bmatrix} 1.713456630706787 \\ -0.3911651074886322 \\ -0.3942330777645111 \\ -0.13038836419582367 \\ -1.6030099391937256 \\ 1.7103886604309082 \end{bmatrix}$$

$$\text{Blue bin} = \begin{bmatrix} 2.1460392475128174 \\ -0.23623304069042206 \\ -0.32827189564704895 \\ -0.11044661700725555 \\ -1.5830682516098022 \\ 2.144505262374878 \end{bmatrix}$$

$$\text{Green bin} = \begin{bmatrix} 1.26553416252136231 \\ -0.36048549413681032 \\ -0.34361171722412113 \\ -0.12578642368316658 \\ -1.58767020702362066 \\ 1.25633025169372569 \end{bmatrix}$$

These 6-axis values are measured as follows. First, drag the manipulator to desired pose by Rviz visual control, then read the 6-axis data by reading ROS topic "*/wx250s/joint_states*".

4) Gripper operation: To achieve fast and accurate gripper operation including open and close, this project bypassed MoveIt! API. Instead, it publishes Float64-type message to ROS topic "/ws250s/gripper/command" directly. For close operation, the sent message.data is -0.015. For open operation, the sent message.data is -0.02.

C. Obstacle Avoidance

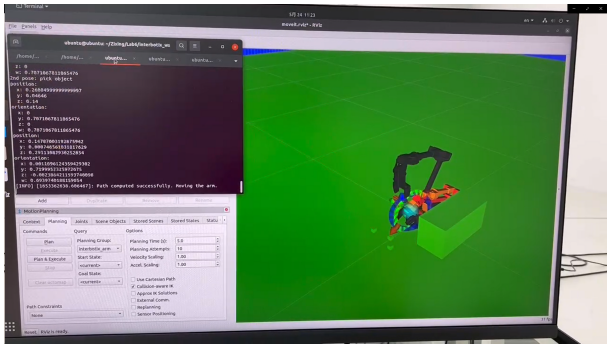
To avoid collision, obstacle avoidance is implemented. The table, bins, and blocks are all set as obstacles, as shown in Fig. 4. When grasping blocks, the corresponding block-obstacle will be deleted, in case of planning failure.

IV. Optimization

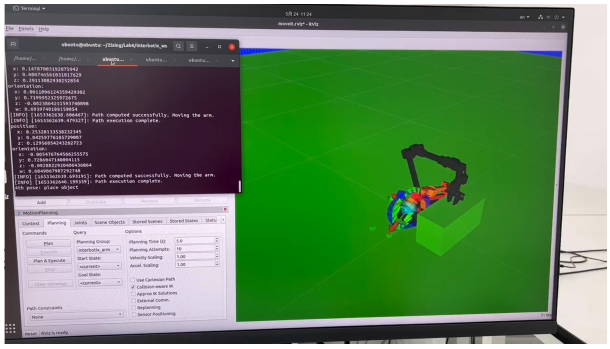
This section introduce the effect this project made to improve the sorting performance, including sorting accuracy and sorting speed.

A. Sorting Accuracy

Collision may occurs, resulting in displacement of the target object and inaccurate grasping. This project requests the object detection service before each grasping to update the target location to improve the accuracy.



(a) Table, bins, and blocks set as obstacles (3 block-obstacles on the table)



(b) Delete target block-obstacle when grasping (2 block-obstacles on the table)

Fig. 4. Obstacle Avoidance

B. Sorting Speed

In this project, some tricks are utilized to improve the sorting speed without reducing the sorting accuracy.

1) Don't update obstacle location: Refer to section IV.A, intuitively speaking, after update object location, obstacle location should be updated as well. The obstacle update costs time. Tests revealed that the displacement of target is normally not large. Hence, to reduce time cost, this process is neglected.

2) Don't attach obstacle to end-effector: Intuitively speaking, when the manipulator grasp object successfully, an obstacle should be attached to the end-effector for obstacle avoidance consideration. Given obstacle update in Rviz costs time, this project neglects this process. As compensation, the height of the end-effector is lowered during gripping to minimize the effect of the gripped object on the shape of the manipulator.

V. Demonstration

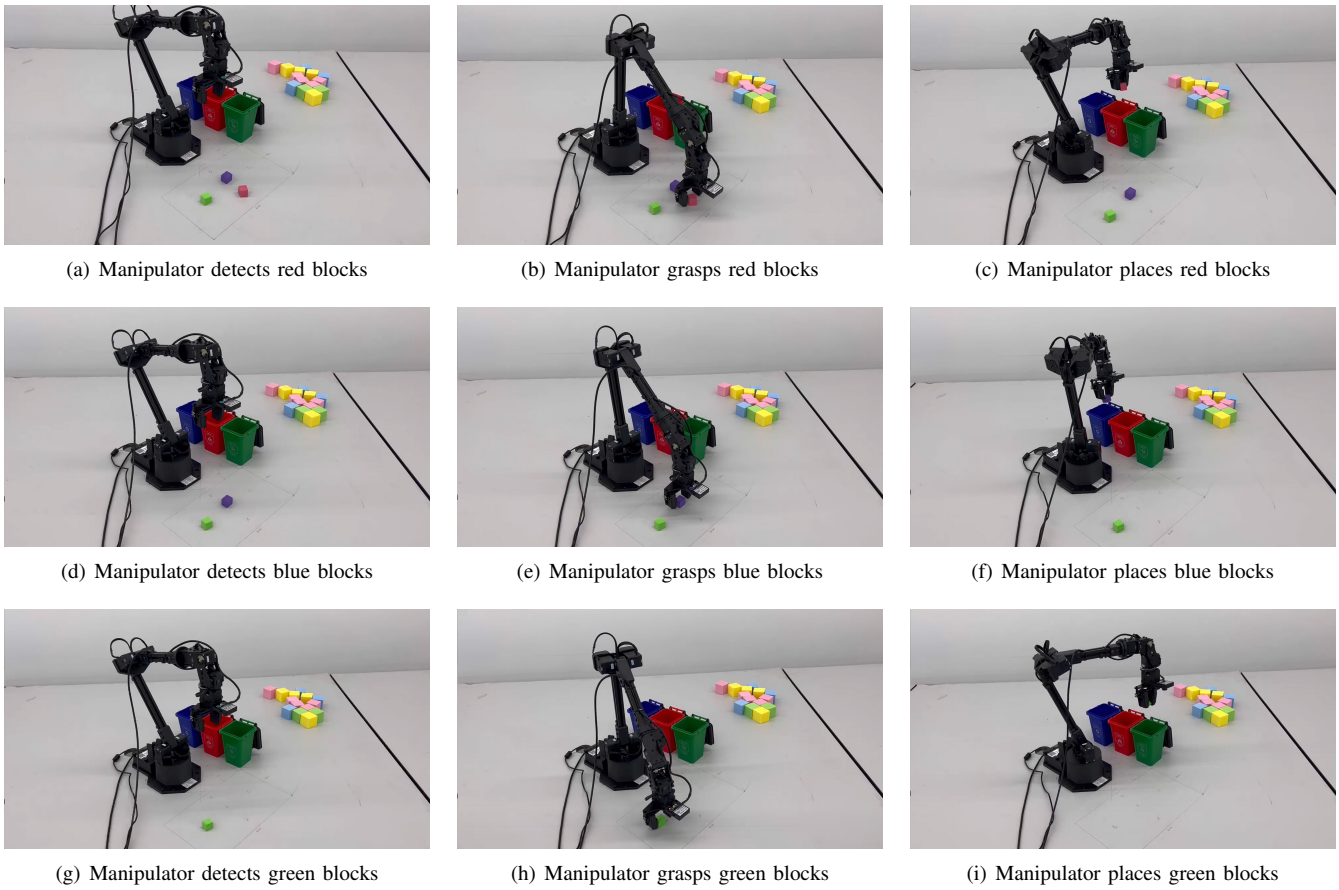
Sequential view of an intelligent sorting demonstration is shown in Fig. 5. For more information, please check the video demo.

VI. Appendix

A. How to run my code

1) Go to and build the workspace.

```
cd interbotix_ws && catkin_make
```



(a) Manipulator detects red blocks (b) Manipulator grasps red blocks (c) Manipulator places red blocks

(d) Manipulator detects blue blocks (e) Manipulator grasps blue blocks (f) Manipulator places blue blocks

(g) Manipulator detects green blocks (h) Manipulator grasps green blocks (i) Manipulator places green blocks

Fig. 5. Sequential view of an intelligent sorting demonstration.

2) Start the robotic manipulator with provided bash script.

```
. real/environment.bash
```

3) Start the camera with provided bash script.

```
. real/camera.bash
```

4) Start the sorting program with provided bash script.

```
. real/sort.bash
```

B. Where is the core source file

The core source file of this project locates in *interbotix_ws/src/interbotix_demos/src/sort.py*.

C. ROS node involved

ROS nodes involved in this project is listed in Listing 2.

```
/intelligent_sort # the node to perform the auto-sorting task
/object_detector # the node providing the object detection service
/rosout # the ROS master
/wx250s/arm_node # the robotic arm
/wx250s/move_group # the moveit move group
/wx250s/robot_state_publisher # robot arm state publisher
/wx250s/rviz_ubuntu_6880_5747233684899934003 # rviz parameter server
```

Listing 2. ROS nodes involved in this project.