# ROB 550 Arm Lab Report - Section 2 Group 9

Ziqi Han, Siyuan Yin, Hongjiao Qiang
{ziqihan, yinsy, hqiang}@umich.edu

***Abstract*—This report delves deeply into computer vision and robotic arm motion control within the ROB 550 Armlab project. By utilizing a LiDAR camera, we have established a potent vision system that seamlessly integrates with the Interbotix ReactorX-200 5-DOF robotic arm. We have implemented automatic determination of camera intrinsic and extrinsic parameters and workspace rectification. Employing various OpenCV algorithms in conjunction with clustering techniques, we achieved real-time precise identification of object types, sizes, colors, and poses. In terms of robotic arm motion control, our forward kinematics component adopts the product of exponentials method, ensuring precise control over robot movements. Moreover, by deconstructing the joint variable set, we adeptly address the inverse kinematics problem, guaranteeing accurate positioning and manipulation of the gripper. Our path planning module produces a comprehensive list of the robotic arm's joint angles, optimizing its efficiency during block pick-up and placement tasks. The efficacy of our algorithms is validated through six distinct events.**

**Key Words: block detection, color detection, kinematics, inverse kinematics**

## I. Introduction

In the ROB 550 Armlab, we developed an autonomous system allowing a 5-DOF Interbotix ReactorX-200 robotic arm to adeptly arrange blocks of varying sizes, colors, and positions into predetermined configurations. Our system leverages analytical inverse kinematics to ascertain the optimal waypoints for the intended end-effector positioning. To enhance this, a heuristic motion planning strategy was conceptualized to yield feasible waypoints. An overhead Intel RealSense LiDAR Camera L515 provides the capability to identify and map blocks within the operational space. Through the application of homogeneous transformations, pixel and depth coordinates are translated into real-world spatial metrics. To bolster reliability and precision, the extrinsic matrix is meticulously calibrated using four strategically placed AprilTags with known spatial references.

The project, executed collaboratively, saw our team tackle intricate challenges both in computer vision and robotic control domains. In the realm of computer vision, extensive camera calibration was performed using multiple methodologies, culminating in a finely-tuned workspace reconstructed using the calibrated parameters. A sophisticated block detection system was constructed, drawing on an array of advanced computer vision models and algorithms. On the robotic control front, we meticulously addressed both Forward and Inverse Kinematics, ensuring the robot's precise maneuvering towards the designated target. Subsequently, a comprehensive path planning solution was established, incorporating various states for diverse manipulation tasks within our meticulously designed state machine.

Hardware utilized:

- Interbotix ReactorX-200 5-DOF Robot Arm
- Intel RealSense LiDAR Camera L515
- ASUS ROG Laptop equipped with an Intel i7-10750H CPU

## II. Methodology

### A. *Computer Vision*

*1) **Camera Calibration**:* Camera calibration is a fundamental prerequisite for all the subsequent processes, as it involves transformation from camera to workspace coordinates.

- **Intrinsic Calibration**:

First we found the intrinsic matrix for the Realsense camera by using the ROS camera calibration package and provided checkerboards while running the Realsense node. We calibrated for 4 times and took the average of obtained intrinsic matrix.

- **Rough Extrinsic Calibration**:

The naive approach to obtain the extrinsic matrix is using physical measurements of the lab apparatus. We measured the (X,Y,Z) distances respectively between the camera's position and the origin. We used the realsense-view program to find the readout of the camera's accelerometer, from which we got an approximation of the orientation of the camera. We then constructed the extrinsic matrix based on the following assumtions: 1) X axis of the world frame are parallel to U axis of the sensor; 2) YZ plane of the camera frame is parallel to the YZ plane of the world; 3) Camera frame is at the front surface of the sensor.

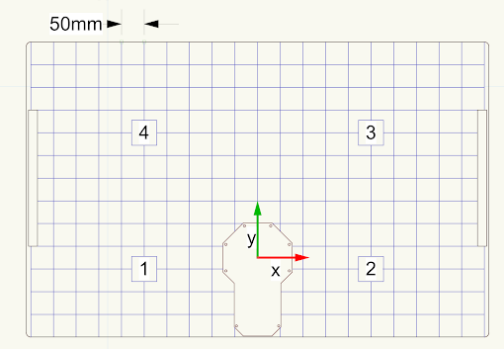


Fig. 1: The Board and origin of the robot frame

- **Apriltags Camera Calibration**:

For better accuracy, we used Apriltags as fiducials for camera calibration. The tag family we used is TagStandard41h12.

To ensure the accuracy of our subsequent computation, we first drew the Apriltags on our video frame by implementing the *drawTagsInRGBImage()* function in *src/camera.py*. We got the ROS message which include Apriltags' detection information from the topic *apriltag_msgs/msg/AprilTagDetectionArray*. Then we iterated through detected Apriltags and used cv2 functions such as *cv2.circle()*, *cv2.line()*, and *cv2.putText()* to draw centers, sides, and IDs of the Apriltags on the RGB video frame.

With both the coordinates of the Apriltags in the world frame and their coordinates in the image frame by reading *apriltag_msgs/msg*, we could obtain *rvec* and $\mathbf{T}$ by calling the function *cv2.solvePnP()*. For the other input parameters, we used the average intrinsic matrix and distortion matrix we got with manual calibration. We then used the *cv2.Rodrigues()*[1] function to get $\mathbf{R}$ with *rvec* as the input. With both $\mathbf{R}$ and $\mathbf{T}$, we were then able to construct our extrinsic matrix $\mathbf{H}$. The equations we used to turn [u,v,d] coordinats in the image frames to [x,y,z] coordiniates in the world frame are the following, where $\mathbf{K}$ is the intrinsic matrix, and $\mathbf{H}$ is the extrinsic matrix:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = Z_c(u,v)\mathbf{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} = \mathbf{H}^{-1} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{2}$$

*2) **Workspace reconstruction**:* We employ an inversion process of the pinhole model to transform pixel coordinates into a reconstructed workspace that incorporates bounding boxes. The extrinsic parameters are autonomously calibrated, leveraging Apriltags that are observable within the workspace (as depicted in Figure 4) and identified by IDs 1-4. Additionally, the depth calibration is performed only once, with subsequent reuse of the homography matrix. This is made feasible by the homography matrix's validity when confronted with planar rotations about its origin, provided there is no translation involved.

- **Pixel Coordinate to World Coordinate**:

Utilizing both the intrinsic and extrinsic matrices, we have the capability to project a pixel coordinate onto an object point within the global/world coordinate system through the inversion of the complete camera model.

The process commences with the projection of pixel coordinates into the camera frame, followed by the removal of scaling effects through the inclusion of depth information. Subsequently, we convert these coordinates into homogeneous form and perform matrix multiplication with the inverse extrinsic matrix.

- **Add boundary for Workspace**:

The image frames undergo masking via the incorporation of two distinct bounding boxes. The larger bounding box serves as a delineation of the workspace boundary, while the smaller bounding box functions to obscure the presence of the robot arm. It is worth noting that the distance between each of the corners of these bounding boxes and the origin point of the robot arm does not exceed 550 millimeters, a value representative of the arm's maximum reach distance.

Furthermore, the workspace is bisected by a horizontal line, demarcating the positive and negative half planes. This division facilitates the rapid adjustment of detection ranges, a necessity during competitive scenarios.

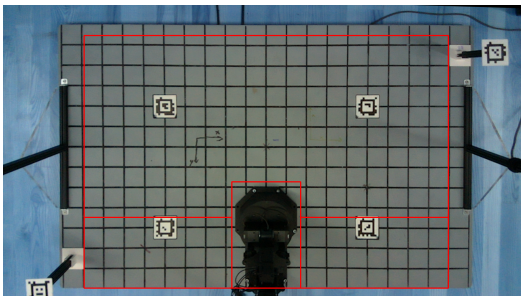

Fig. 2: Workspace boundary

*3) **Block detection**:* Our implemented block detector exhibits robustness in the detection of 2D and 3D coordinates, colors, orientations of both large and small square-shaped blocks, as well as accurately discerning the number of stacked blocks. In addition to these capabilities, it extends its proficiency to the detection of

various other geometric shapes, such as arches, cylinders, triangles, and rectangles. Furthermore, it demonstrates accurate recognition of semi-circles when they are not positioned too closely to the edge of work space.

In the subsequent sections, we will delineate the procedural steps of our algorithm, which primarily encompass the following key components: initial object detection, Contour optimization and color recognition via clustering, and object classification.

- **Initial object detection**

1. Height-based filtering

Initially, we implemented a height-based filtering procedure. This process involved the application of the cv2.inRange function to exclude objects exceeding a height of 150 millimeters. This operation effectively eliminated regions within the workspace associated with the presence of the robotic arm, thereby enhancing the clarity of the scene representation by removing such interference.

2. HSV-based filtering

To eliminate grid patterns and two-dimensional codes within the workspace, we performed HSV-based filtering on the RGB image. This process entailed converting the RGB image into the HSV color space and subsequently applying a filter based on specific HSV value ranges, namely [0, 43, 46] to [180, 255, 255]. This filtering operation effectively removed regions in the image corresponding to black, white, and gray colors. Additionally, we employ median filtering to attenuate noise artifacts.

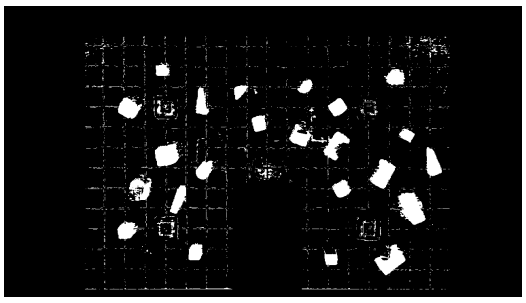

Fig. 3: Binary image after HSV filtering

It can be observed that the majority of the gray-white background color in Figure3 has been filtered out, leaving behind small noise points during the contour formation process. These residual noise points can be filtered out by setting a threshold for contour size.

3. Find initial contours

We employ the cv2.findContours function to identify all contours within the binarized HSV image. Subsequently, we calculate the moments of these contours using cv2.moments(), and filter out contours with a first moment of area (M00) smaller than 200 or larger than 7000. Such contours are deemed inconsistent with the characteristics of blocks, either being excessively large or small.

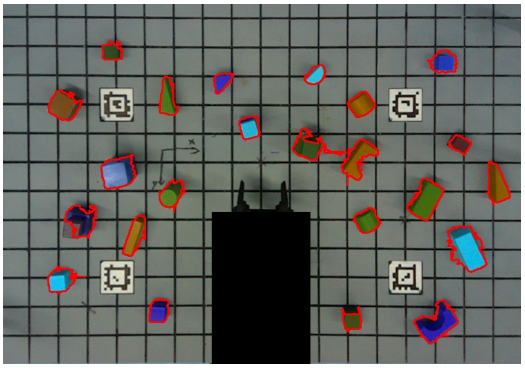

Fig. 4: Raw contours

From Figure4, it is evident that although we have captured the outlines of all objects without any omissions, these outlines are not precise, as they do not closely align with the edges of the top surfaces of the objects. This inaccuracy in contour alignment results in an imprecise calculation of the objects' center points, subsequently leading to failures in the mechanical arm's grasping attempts.

Based on the obtained initial contours, we compute their centroid positions. We compute the centroids of these contours and extract depth values from eight points surrounding the centroid. Taking the average of these depth values provides a relatively accurate depth measurement, enabling us to discern whether the objects are stacked. Due to the inherent limitations of depth image in achieving pixel-level precision, the contours obtained require further refinement.

- **Contour optimization and color recognition**

We apply a clustering algorithm to process RGB images within subregions near these centroids. Initially, we obtain the LAB color space representation for these subregions. Following this, we merge RGB, LAB, and depth map information, forming seven-dimensional feature vectors for each pixel coordinate. Utilizing a higher number of dimensions for feature vectors enhances the accuracy and granularity of clustering analysis. As HSV color space filtering has been applied during preprocessing, HSV features are omitted at this stage.

$$\mathbf{v}^T = \begin{bmatrix} R & G & B & L & A & B & \mathrm{Zc} \end{bmatrix} \tag{3}$$

We employ the K-means clustering algorithm [2] to cluster these multi-dimensional feature vectors, partitioning the image into four clusters. For cases involving stacked blocks, where a broader spectrum of colors is expected, we divide the image into a greater number of clusters, given their propensity for exhibiting a greater

color diversity. Each cluster's pixel count is tabulated, and the cluster with the highest frequency is regarded as the top surface of the object.

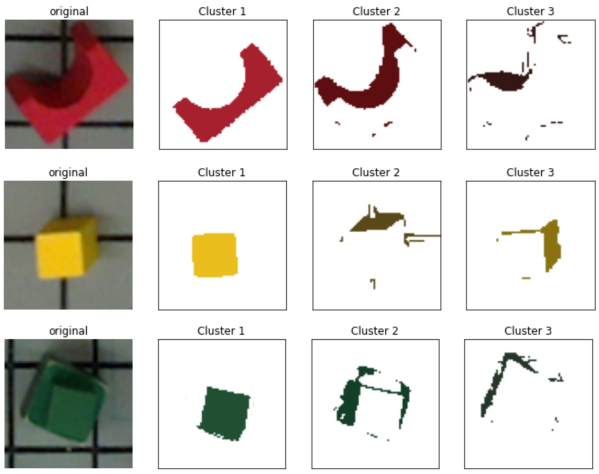


Fig. 5: Example of clustering

From Figure5, it is evident that by employing clustering and selecting the most prominent cluster, we are able to filter out the lateral sides of the object, pinpointing the object's top surface with precision. Even in the presence of errors in the depth map, through clustering, we achieve an effect analogous to using an RGB image for auxiliary localization of the object.

Additionally, based on the mean values of the most frequent cluster, we extract the first six dimensions (excluding depth information) and calculate the Minimum Vector Distance to the average feature vectors predefined for six distinct rainbow colors. The object's color is determined as the one with the shortest distance.

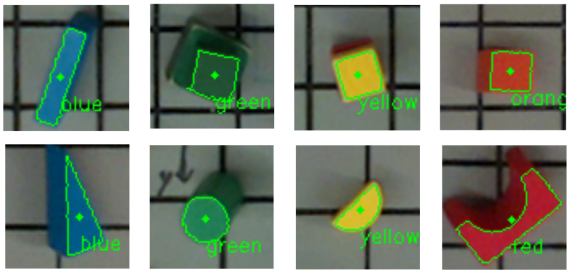


Fig. 6: Example of precise contours and color detection

At this point, we have obtained precise contours of the object's top surface. This approach exhibits remarkable accuracy in color recognition, as evidenced by the absence of color recognition errors during experimentation.

- **Object classification**

For Task 3 and Task 4, it is imperative to identify objects with alternative geometries, including cylinders, triangles, arches, upright cuboids (arches in a vertical orientation), semicircles, and upright semicircles. In the case of these objects, we proceed with further recognition based on the optimized contours obtained.

1. Semi-circles

Given that semicircles exhibit dimensions in length and width that closely resemble those of small blocks, distinguishing them solely from RGB contours is challenging. Consequently, our approach prioritizes semicircle detection based on the depth map. Initially, we isolate subregions of all contours within the depth map, slicing them around their respective centroids. These slices are made at intervals of 5 units in height, totaling two slices. Due to the characteristic shape of semicircles, featuring higher height in the center and lower heights on the sides, the pixel counts on each depth slice vary significantly. The closer one moves to the top of the semicircle, the fewer pixels are observed on the depth slice. We calculate the ratio of pixel counts between adjacent depth slices and classify a region as a semicircle if the ratio is above the threshold of 2.5. This method demonstrates effective recognition, particularly for semicircles not positioned at the four corner locations of the workspace.

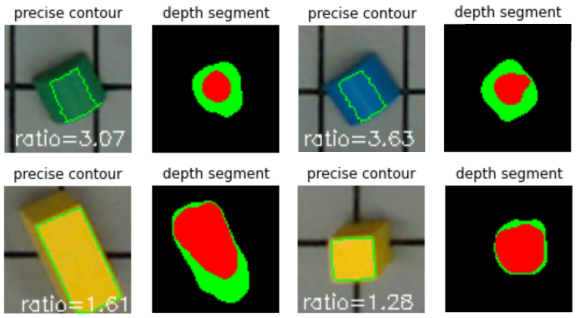


Fig. 7: Example of depth segmentation for detecting semi-circles

2. Cylinders

Subsequently, we proceed to identify cylinders. Employing the Hough circle detection algorithm, specifically the cv2.HoughCircles function, allows us to distinguish cylinders with precision from other objects.

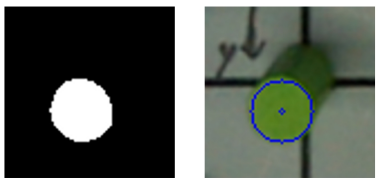

Fig. 8: Example of Hough algorithm for detecting cylinders

3. Triangles, cuboids, arches, upright semicircles

For objects other than cylinders and semicircles, we employ the cv2.arcLength and cv2.approxPolyDP functions to calculate the precise contour's perimeter and its approximated contour. If the approximated contour has four edges, we further assess it using cv2.minAreaRect to determine the aspect ratio between the longer and shorter sides of the minimum bounding rectangle. If this ratio exceeds 1.3, it is identified as a cuboid; otherwise, it is classified as a square object.

If the approximated contour consists of three edges, it is recognized as either an upright semicircle or a triangle. For contours with five edges, they are categorized as arches. Moreover, cylinders typically have more than seven edges, but we have employed alternative methods for their recognition. Furthermore, distinguishing upright semicircles from triangles accurately is challenging using traditional computer vision techniques. However, for the tasks in this assessment, the grasping mechanism of the gripper is designed to grasp objects by their longer edges when removing obstacles of these shapes. Therefore, it is sufficient to determine that an object is not a square.

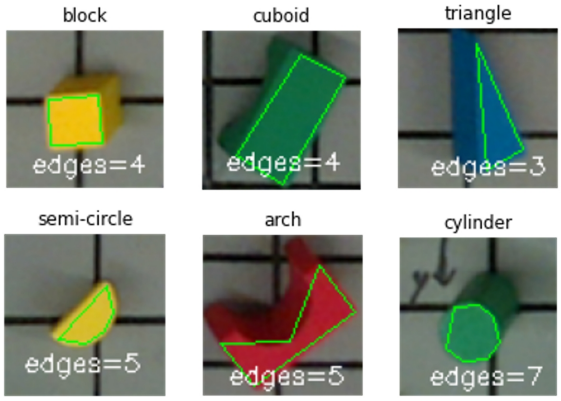


Fig. 9: Example of edges approximation for detecting various blocks

Upon completion of detection, we transmit the attributes of the objects, including their type, color, centroid coordinates, angle between the long edge and the positive x-axis, and whether they are part of a stacked configuration, to the control unit of the robotic arm. This facilitates the execution of various tasks.

The pseudocode for the overall block detection process is as shown in Algorithm 1.

**Algorithm 1** Block Detector Algorithm

**procedure** BLOCK DETECTOR(I) ▷ I: Image
  Init params
  $I_f \leftarrow$ HEIGHTFILTER(I)
  $I_h \leftarrow$ HSVFILTER($I_f$)
  $C_{Raw} \leftarrow$ FINDCONTOURS($I_h$) ▷ C: Contours
  **for** each $c$ in $C_{Raw}$ **do**
    $S_{RGB} \leftarrow$ SUBREGION($c$)
    $S_{HSV} \leftarrow$ RGB2HSV($S_{RGB}$) ▷ S: Subregion
    $v \leftarrow$ VECTOR($S_{RGB}$, $S_{HSV}$, $depth$)
    $K \leftarrow$ KMEANS($v$)
    $S_{Opt} \leftarrow$ MAX($K$) ▷ K: num clusters
    $C_{Opt} \leftarrow$ FINDCONTOURS($S_{Opt}$)
    $color \leftarrow$ MINNORM($v$,$v_{ref}$)
    **if** DEPTHSEG($C_{Opt}$) **then**
      $type \leftarrow$ semi-circle
    **else if** HOUGHCIRCLES($C_{Opt}$) **then**
      $type \leftarrow$ cylinder
    **else if** APPROXPOLYDP($C_{Opt}$) == 3 **then**
      $type \leftarrow$ triangle
    **else if** APPROXPOLYDP($C_{Opt}$) >5 **then**
      **if** $side$ >35 **then**
        $type \leftarrow$ arch
      **else if** $side$ <35 **then**
        $type \leftarrow$ up-semi-circle
    **else if** APPROXPOLYDP($c$) == 4 **then**
      $aspectRatio \leftarrow$ ASPECTRATIO($C_{Opt}$)
      **if** $aspectRatio$ >1.5 **then**
        $type \leftarrow$ cuboid
      **else if** $side$ >30 **then**
        $type \leftarrow$ big block
      **else if** $side$ <30 **then**
        $type \leftarrow$ small block
    $block \leftarrow color, type, pos, ori$
    $blocks \leftarrow block$
  return $blocks$

## B. Robot Control

*1) **Forward Kinematic Solution**:* The objective of forward kinematics (FK) is to determine the position of the joints in the global frame, given the position of each of the joints in the robot frame. We implemented the Denavit-Hartengberg (DH) table first.

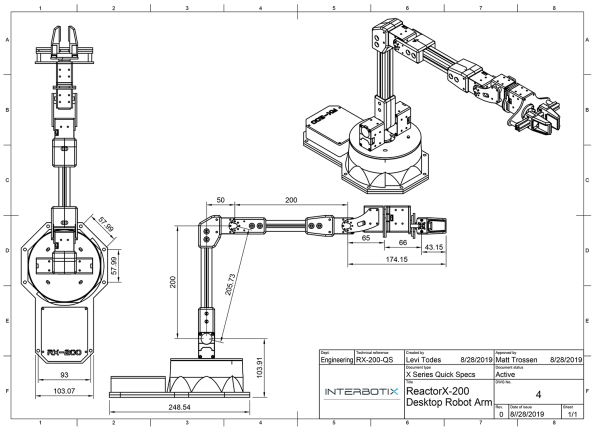


Fig. 10: Technical drawings of the RX200 arm

*2) **Inverse Kinematic Solution**:* All the critical parameters could be found from the technical drawings of the RX200 arm. We also implement the Product of Exponentials (PoX) method and compare its result with DH method to ensure correct computation. The same result indicates the correction of our methods

Since RX200 arm has five degrees of freedom, the solution of its inverse kinematics is a little bit different from conventional 6-DOF manipulator due to the lack of the 3-DOF wrist. The first step of inverse kinematics is to give a single orientation angle $\psi$, defined as the angle from the horizontal plane. The first joint angle could be computed from the projection of the end effector on the horizontal plane, and the second and third joint angles could be computed using trigonometry, and we can obtainnig the fourth joint angle by subtracting the sum of the second and the third joint from $\phi$. The last joint could be chosen arbitrarily based on the orientation of the blocks we want to pick. In our code, we set the last joint angle equal to 0 if there is a tilt in the last link, while it will conform with the block orientation if $\phi$ is or approximately equal to 90 degrees.

*3) **Path Planning**:*

- **Teach and Repeat**:

Teach and repeat is a very common way of programming robotic arms to do repetitive tasks. In this task, we implemented a logic to take a set of discrete waypoints in joint space and commands the arm to follow the path connecting them all. The first step is to "teach" the robot the pattern that it must complete. We added a button in the GUI to record the current joint angles of the robot arm joints. Pressing this button will save the joint angles as an array and add that into a list variable. For the second step "repeat", we created another button in the GUI, which will trigger a replay of the recorded waypoints by traversing the list and calling the *set_position()* function.

- **Click to Grab/Drop:**

There are two prerequisites for our Click to Grab/Drop function to run: 1) Block detection is complete; 2) There are valid blocks detected in workspace. With the above two requirements satisfied, if a block in workspace is clicked, we will get the coordinates of the clicked position and pass it into our verification function *inArea()* to validate there is a block close to the clicked position. If so, the block's world coordinate and orientation will be obtained from the *self.camera.blocks* class.

To execute the grab and place movement, we only used one function called *grab_or_put_down_a_block()*, since the only difference between them is whether to open or close the gripper. We pass in one variable called *isgrab* to switch between the two options. After the block's coordinate is found, we call the *grab_or_put_down_a_block()* function with *isgrab* variable set to True. Then the *kinematics.IK_geometric* function will be called to validate the location is reachable by the arm and compute joint angles to reach that position. If that's confirmed, we just use the *set_posisions* function to control the robot arm to pick up the selected block. Similarly, when another point is clicked, its image coordinates will be transformed into world coordinates, then passed in *grab_or_put_down_a_block()* function with *isgrab* variable set to False. Then the tobot arm is going to go to the clicked position and open its gripper to place the block.

*4) **State Machine Functions and Tasks**:* To better perform a designated task, We created a series of functions to help control the manipulator. The following are description of the most important ones.

- **Auto_pick and Auto_place function**

The auto_pick and auto_place functions are the core functions in our tasks, which will control the maniputalor to pick or place a single block. given the location of the block in world frame, it will compute whether the end effector could reach that position vertically using inverse kinematics computation function, which will increase the accuracy of the pick and place actions. If the manipulator can not reach it perfectly horizontally, we would modify the target location a little bit closer and decrease $\phi$ a little. If the target place is unreachable after even 5 repeated actions above, we will try horizontal pick if the point is still within the working space. Generally when picking or placing a block, the gripper will reach a place above the target location and slowly goes down. For placing action we also implement dichotomy to place the block step by step and detect the variation of the torque in the motors simultaneously. If the variation of the sum of the torque for each motor is greater than a certain threshold, we assume that the block has contact with a surface and the gripper will stop declining and release the block.

- **safe_motion function**

We found that the manipulator couldn't execute the set_positions or set_joint_positions functions sometimes in the experiments. After several attempts and trial and error, we conclude that the reason is the speed and moving distance limitation of each motors. So we implement interpolation method to make sure there is enough way points for the manipulator to track in its motion. it one of the joint angle will move move than 20 degrees, we will interpolate at least one way point. We also compute the moving time and acceleration time of each part to ensure smooth motion.

For events in the competition, we created a state for each event and the following section explains the logic for each event:

- **Event 1: Pick'n sort!**

The logic we have for this task is basically a while loop, which completes when all blocks detected are all in negative plane. This termination condition is valid because the setting of this task is that all the blocks will be placed in the positive half plane. For each loop, we detect the blocks on the board. Then we traverse through the list of detected blocks, differentiate small and blocks by the *.block.side* attribute, and pick 'n place small and big blocks in the left-negative and right-negative plane respectively. If there are blocks that are stacked together, the first cycle will relocate the block on the top, and the next cycle will take care of the de-stacked blocks.

- **Event 2: Pick'n stack!**

For event 2, we have a quite similar logic as event 1. We handle the placed blocks with loops: in the first loop, we relocate all blocks stacked on another; in the second loop, we pick and place blocks to their target positions. The only difference is that when we place the blocks, we increment the depth of the target position as we iterate through the blocks instead of incrementing the (X, Y) coordinates.

- **Event 3: Line'em up!**

For this event, blocks are not placed only in the positive plane or a designated area. To make sure we do not place a block at a pre-occupied location or ignore blocks in any area, we run our *pick_n_sort()* function first. Therefore, we can have all the blocks categorized by size: big blocks on the right-negative plane; small blocks on the left-negative plane. Then we detect the blocks on board again and sort them by color in this order: 1.red; 2.orange; 3.yellow; 4.green; 5.blue; 6.violet. The last step is to traverse through the sorted block list and place the blocks at its target position, with the X coordinate of the target position incrementing after each block is placed.

- **Event 4: Stack'em high!**

In event 4, although blocks are initially placed anywhere on the board as well, since we're stacking up blocks at two locations, we won't need a rather large "safe zone" as we need in event 3. Therefore, we first used the similar loop logic to clear up blocks that are stacked up initially. Then we detect the blocks again and sort the detected block list by color. Finally, we pick and place the blocks at the target position, incrementing their Z coordinates after each block is placed.

- **Bonus Event: To the sky!**

In event 5, our aim is to build a tower of at least 10 blocks. The first step is to choose a proper location for the foundation of the block tower, and we choose (200,225) on the horizontal plane to make sure that higer points are within its working space and the end effector could reach them with dexterity. We let the manipulator look for and pick the blocks on the plane automatically instead of replenishing the blocks continuously, so it will detect all the blocks on the plane at the beginning of the task. The manipulator will place the first 5 blocks vertically and place other blocks horizontally to ensure its dexterity. After placing every block, the gripper will repeat the open the close action to calibrate the orientation each block and thus increase the stablilty of the tower. The manipulator will raise up before and after placing a block to avoid hitting the tower.

## III. Results

### A. *Computer Vision*

#### 1) *Camera Calibration:*

- **Intrinsic Calibration**:

The following table compares the average intrinsic matrix we got from running ROS camera calibration package and from echoing the factory information.

By comparing the intrinsic matrix we obtained and the factory intrinsic matrix, we did not find too significant difference. The focal length in x-direction is around 10 mm off, and the focal length in y-direction is around 4 mm off. The coordinate of the principal point also has around 10 mm off within x and y axes respectively. We used the intrinsic matrix we got from manual calibration in the following tasks because we trusted it more.

TABLE I: Intrinsic Matrices Comparison

| Method | Intrinsic Matrix |
|---|---|
| **Manual Calibrated** | $\begin{bmatrix} 918.2490 & 0.0000 & 636.4534 \\ 0.0000 & 912.0612 & 365.2384 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$ |
| **Factory** | $\begin{bmatrix} 908.3550 & 0.0000 & 642.5927 \\ 0.0000 & 908.4041 & 353.1265 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$ |

- **Extrinsic Calibration**:

From Figure 11, we can see the detection of AprilTags are quite accurate. The following table compares the extrinsic matrices we got from rough measurement approach and the AprilTags Calibration approach. We can see that there's no significant error in the rotational part of the matrix, but values from the translational part are quite off. The source of error is that in rough measurement, we got the translational part based on the three assumptions.

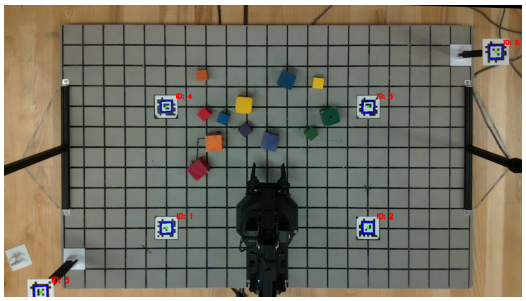

Fig. 11: AprilTags Detection

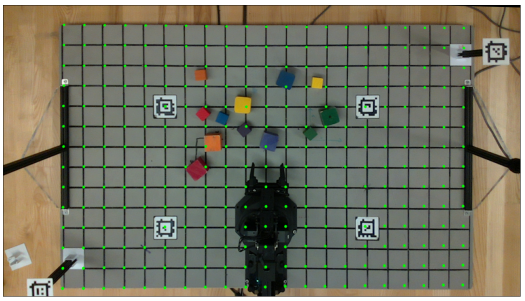

Fig. 12: Grids of Points Projection

The way we verified the correctness of calibration was hovering mouse cursor onto different locations on the board and comparing the outputs with their actual locations. For locations which were supposed to be around 0mm in height, at first we found the [x,y] coordintaes are rather accurate, while the [z] coordinate varied a lot at different locations across the board. We fonund the cause was that when transforming the coordinates, we calibrated only with 4 AprilTags at 0mm height on the board, which lead to an inaccurate z coordinate after transformation. We then added 2 AprilTags with 150 mm height, which enhanced calibration performance.

TABLE II: Extrinsic Matrices Comparison

| Method | Extrinsic Matrix |
|---|---|
| **Rough Measured** | $\begin{bmatrix} 0.9999 & 0.0094 & -0.0426 & 0.0000 \\ 0.0000 & -0.9763 & -0.2164 & 365.0000 \\ -0.0436 & 0.2162 & -0.9754 & 1000.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$ |
| **Apriltags Calibrated** | $\begin{bmatrix} 0.9993 & -0.0277 & -0.0264 & 31.2425 \\ -0.0201 & -0.9673 & 0.2528 & 104.1185 \\ -0.03254 & -0.2521 & -0.9672 & 1049.4826 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$ |

*2) **Workspace reconstruction**:* To validate our calibration outcomes, we conduct a meticulous examination by traversing the mouse cursor along individual horizontal and vertical lines on the workspace board. This scrutiny aims to assess the deviation in the y-axis values when solely traversing the x-axis, and vice versa. The resultant discrepancies are found to be relatively negligible, typically within the range of 1 to 3 millimeters.

However, for achieving a higher degree of accuracy in extrinsic calibration, an alternative approach is employed. This involves the detection of grid points on the board and the determination of the transformation using the cv2.solvePnP method, which offers greater robustness when dealing with outlier point correspondences.

*3) **Block detection**:* The picture below shows the final result of our block detection function. We can see from the figure that we were successful to detect different blocks and have them marked by attributes such as orientation, color and shape.

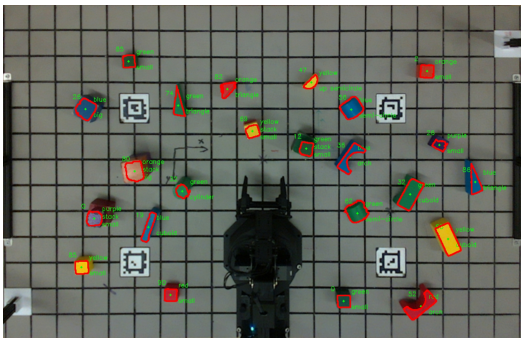

Fig. 13: Block detection result

From Figure 13, it is evident that we are capable of identifying all categories of objects, encompassing their pose, color, and stacking status. In terms of color recognition, our approach of initial clustering followed by determining the minimum distance boasts exceptional precision; we have never encountered identification errors in both experimental and competition settings. Regarding object category recognition, with the exception of the four furthest corners of the workspace, we can accurately identify all objects, including semicircular shapes. Due to the time-consuming nature of clustering across subregions, our algorithm achieves a frame rate of only 2 frames per second, indicating some limitations in real-time performance.

## B. Robot Control

*1) **Forward Kinematic Solution**:* The DH table of the manipulator is shown below, and the result of two forward kinematics method ensure the correction of the computation.

*2) **Inverse Kinematic Solution**:* The result of the first 4 joint angles through inverse kinematics is shown below.

TABLE III: DH Table

| a | $\alpha$ | d | $\theta$ |
|---|---|---|---|
| 0.0 | 1.57 | 103.91 | 1.57 |
| 205.73 | 0.0 | 0.0 | 1.33 |
| 200.0 | 0.0 | 0.0 | -1.33 |
| 0.0 | 1.57 | 0.0 | 1.57 |
| 0.0 | 0.0 | 175.0 | 3.14 |

$$
\begin{aligned}
s &= z_c - l_1 \\
\theta 1 &= arctan2(-x_c, y_c) \\
\theta 2 &= 90 - \alpha - arccos\frac{r}{\sqrt{r^2+s^2}} \\
&\quad - arccos\frac{l_4^2 - l_5^2 + r^2 + s^2}{2l_4\sqrt{r^2+s^2}} \\
\beta &= arccos\frac{l_4^2 + l_5^2 - r^2 - s^2}{2l_4 l_5} \\
\theta 3 &= 90 + \alpha - \beta \\
\theta 4 &= \phi - \theta 2 - \theta 3
\end{aligned}
\tag{4}
$$

Where $l_1$, $l_2$,$l_3$,$l_4$ and $l_5$ corresponds to the length of each link, $\alpha$ is 11.25 degrees. x,y,z are the coordinate of the wrist joint, which could be computed by subtracting the last link length from the end effector position, r is the Euler distance of the projection point from the origin. Joint5 will be defined arbitrarily by the orientation of the block.

*3) **Path Planning**:*

- **Teach and Repeat:**

We taught our robot to cycle swapping blocks at locations (-100, 225) and (100, 225) through an intermediate location at (250,75). We were able to cycle 10 times and the following plot is the joint angles of the robot arm over time for 1 cycle. The following picture is the plot of joint angles throughout one cycle.

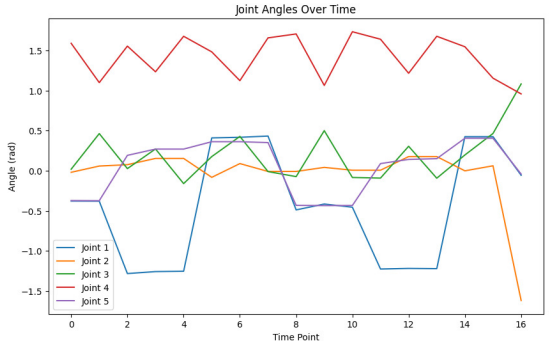


Fig. 14: Plot of Joint Angles in Teach & Repeat

- **Click to Grab/Drop:**

A demo of running the Click to Grap/Drop action can be viewed from this link. From the video, we can see that our algorithm managed to control the robot arm to both pick up and place down precisely at designated location.

*4) **State Machine**:* The following section describes the result we got from competition events.

- **Event 1: Pick'n sort!**

For Event 1, we challenged level 3. For the first two tries, we both exceeded the time limit. Then we adjusted the speed of joint movement and was able to complete the task in time in our third try.

- **Event 2: Pick'n stack!**

For Event 2, we also challenged level 3. We were not able to finish stacking all 6 blocks in our first try. So we increased the speed of joint movement and was able to complete the task in time in our second try.

- **Event 3: Line'em up!**

For Event 3, we challenged level 2. Due to several detection mistakes, we were not able to line up all the big blocks but we lined up all the small blocks, although with a bit flaws in neatness.

- **Event 4: Stack'em high!**

For Event 4, we challenged level 2 as well. We experienced the issue of not detecting all 6 colors for both big and small blocks multiple times so our program halted before the robot arm started to stack in color order. The best result among the several tries we did is we stacked all big blocks in correct order but small blocks were not stacked up because of lack of precision when placing the first few blocks.

- **Bonus Event: To the sky!**

For the bonus event, we had 2 tries in total. We stacked 11 blocks in our first try. We did not modify our code and went on for another try. We were able to stack 12 blocks in the second try.

## IV. Discussion

One of the major challenges we encountered is that as the arm extended towards distant target locations, the influence of gravity caused deviations from the intended trajectory. To address this issue, we implemented the *Target_Pos_Compensation()* function as a compensation mechanism in *kinematics.py*.

Additionally, due to our visual algorithm's adoption of clustering across multiple subregions, the algorithm can only achieve a frame rate of 2 frames per second, indicating significant room for improvement in terms of real-time performance.

The *Target_Pos_Compensation()* function addresses gravitational effects by adjusting the world position coordinates. It calculates offset values for the x, y, and z coordinates based on the object's height and its distance from the origin. These offsets ensure that as the arm moves towards distant targets, gravitational deviations are minimized.

## V. Conclusion

After introspecting the results we got from checkpoints and competition, we concluded a few more aspects that we can still get improved on. Due to potential depth map errors that can reach up to 5mm in the four corner positions, there is a risk of recognition failure for semicircles positioned further away.

In this report, we elaborated on our approaches to camera calibration, including intrinsic calibration, rough extrinsic calibration, and the extrinsic calibration with AprilTags for enhanced accuracy. We have demonstrated the transformation of pixel coordinates into the reconstructed workspace. Moreover, we explained our implementation of a robust block detection system that identifies various geometric shapes. Last but not least, we described our path planning and state machine solutions that we used in the competiton.

## References

[1] J. E. Mebius, "Derivation of the euler-rodrigues formula for three-dimensional rotations from the general formula for four-dimensional rotations," 2007.

[2] X. Jin and J. Han, *K-Means Clustering*. Boston, MA: Springer US, 2010, pp. 563–564. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_425

[3] M. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. Wiley, 2005. [Online]. Available: https://books.google.com/books?id=wGapQAAACAAJ

[4] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.