ROB 550 ArmLab Report - Group 5 (PM)

Yung-Ching Sun, Rui Li, Sarah Adams
{ycs, ruili, adamssh}@umich.edu

***Abstract*—Computer vision and robotic arm kinematics and control play crucial roles in robotics. The integration of these technologies has become increasingly prevalent across various applications. In the ROB 550 ArmLab, we explored the integration of computer vision and robotic arm control by implementing camera calibration, block detection with OpenCV, and forward and inverse kinematics for a 5-DOF robotic arm, culminating in a pick-and-place task. To validate our approach, we integrated computer vision, kinematics, and arm control to tackle complex challenges such as sorting, stacking, and lining up blocks in rainbow order. The competition results demonstrated the robustness and effectiveness of our system.**

I. Introduction

Robotic arms are a common form of robotic technology in industry, research, and avocation. There are many different types of arms that are implemented for a wide variety of uses. One of the most common uses for a robotic arm is to handle repetitive tasks based on information input and an end goal [1].

This ArmLab focuses on implementing methods to integrate the Computer Vision with a Realsense Sensor L515 and the kinematics and control of a 5-DOF, 7-motor RX200 arm to organize blocks and operate a launching mechanism. Specifically, computer vision and robotic arm control must collect data about the workspace and then execute a movement plan to achieve a specified goal. This document describes how the camera calibration, block detection, and kinematics methods were implemented to enable the arm to autonomously detect the target blocks and their positions in the workspace using data collected by the camera. It also outlines how the arm plans its pose and trajectories to manipulate blocks in various configurations, avoid collisions, perform pick-and-place tasks at desired locations, and sort, lineup, and stack blocks.

II. Methodology - Computer Vision

A. Camera Calibration

Camera calibration is used to map image coordinates to real-world coordinates in robotic vision applications. This subsection details the calibration process for the Intel RealSense camera, including intrinsic and extrinsic parameter estimation via ROS tools and fiducials.

1) Intrinsic Camera Calibration

The intrinsic parameters of the RealSense camera were obtained using the ROS camera calibration package with a checkerboard target. To calibrate, the checkerboard was moved across the field of view, a process which was repeated three times and result-averaged for accuracy. Alternatively, the factory intrinsic matrix can be retrieved directly from the factory-calibrated values via the **/camera_info** ROS topic, which also provides distortion coefficients. This matrix defines the camera's focal lengths and optical center, essential for mapping pixel coordinates to real-world depth estimates.

2) Extrinsic Camera Calibration (Manually)

A preliminary extrinsic matrix was calculated using physical measurements. The distances between the camera and the origin of the workspace along the X, Y, and Z axes were measured to determine the translation vector. We approximated the camera's orientation with smartphone protractor application to measure its tilt angle relative to the workspace. By combining the rotation matrix and translation vector, we were able to form the extrinsic matrix. Our hand-measured extrinsic matrix is shown in Table III.

Now, the image coordinates can be transformed to world coordinates by the following equations:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = Z_c K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} ; \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = H^{-1} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{1}$$

where K is intrinsic matrix, H is extrinsic matrix, and $[u, v, 1]^T$, $[X_c, Y_c, Z_c, 1]^T$, $[X_w, Y_w, Z_w, 1]^T$ are coordinates in image pixel frame, camera frame, and world frame, respectively.

3) Extrinsic Camera Calibration (Apriltag)

When using AprilTags for extrinsic calibraiton, the accuracy was significantly improved. The four AprilTags fixed on the workspace board provide automated and precise estimation, reducing errors from manual measurement. We detected the centers of the four AprilTags in the image, and based on their IDs, we appended their centers to form the *image_points*. The *world_points* were assigned according to the IDs, representing the actual positions of the AprilTags in the world frame. We then input the *world_points*, *image_points*, intrinsic matrix K, and camera distortion coefficients into *cv2.solvePnP()*

to obtain the rotation vector and translation vector. The rotation vector was then converted into a rotation matrix using *cv2.Rodrigues()*. Finally, the rotation matrix and translation vector were combined to form the extrinsic matrix.

4) Workspace Projection

After extrinsic calibration was completed, we applied a homography transformation to map the workspace into a top-down view. We extracted four AprilTag detection center as source points and select (355, 510), (835, 510), (835, 210), and (335, 210) as destination points to compute homography matrix. Additionally, to account for the resolution difference between the image and the physical workspace, we scaled these destination points using a ratio $= (720/650) * 0.975$ ensured that the image and workspace were properly aligned by compensating for slight deviation in camera positioning and resolution. We then input the detected AprilTag pixel coordinate and the destination points into *cv2.findHomography()* to compute homography matrix. Finally, we applied the homography transformation using $cv2.warpPerspective$ to warp the image, so the workspace board appears flat and appropiately scaled on GUI, the result of the projection can be seen in Figure 7. w

B. Block Detection

Accurate block detection requires the identification of block contours, centers, colors, orientations, and shapes. This section describes the methodology for detecting and classifying blocks using both RGB/HSV and depth information.

1) Block Contour

Block contours were extracted using depth-based segmentation. The depth frame was preprocessed by applying a depth threshold to isolate objects within a specified height range, filtering out irrelevant background and objects taller than the target cube blocks. A binary mask was then applied to remove regions outside the workspace and robotic arm interference zones. Finally, we used OpenCV's cv2.findContours() to extract contours, which detects block boundaries.

2) Block Center

We used OpenCV image moments that were computed from the extracted contours to detect the center of blocks. The centroid formula is:

$$Cx = \frac{M10}{M00}, Cy = \frac{M01}{M00} \tag{2}$$

where $M10$ and $M01$ are the first-order moments, and $M00$ is the zeroth-order moment (area). This centroid serves as a reference point for mapping the block's position in the depth frame, allowing conversion from image coordinates to world coordinates using the intrinsic and extrinsic matrices.

3) Block Color

For Block color detection, we performed HSV color space as a result of its ability to separate color information from brightness variations, making it more robust under different lighting conditions. The HSV space allows for more effective thresholdings to reduce the impact of shadows and reflections. We started with a predefined HSV range for each block color, including red, orange, yellow, green, blue, and violet, with red requiring two separate ranges due to its hue wrapping around the spectrum. These thresholds were refined through multiple iterations, adjusting the saturation and value components to improve the detection accuracy under varying illumination. The detection process involves extracting the median HSV value from each block's contour and comparing it against the optimized thresholds. The final method ensures stable color classification, minimizing errors caused by lighting variations.

4) Block Orientation

The orientation of the block was determined by fitting a rectangle with minimum area bounding around the detected contour using cv2.minAreaRect(). This function provides the bounding box's rotation angle, which indicates the block's orientation relative to the camera. The orientation angle θ is extracted and displayed for each block, this angle ensuring accurate alignment with the gripper during the arm manipulation.

5) Block Shape

Block shapes were classified based on contour properties, bounding box aspect ratio, circularity, and corner detection. The classification procedure follows these steps: aspect ratio test, circularity test, corner detection for squares, size-based square categorization, and reduce false positive. For aspect ratio , the bounding box's width and height were used in Eq 3. We choose threshold between 0.9 to 1.1 for square shape and rectangle otherwise. We calculate circularity in Eq 4. A value near 1.0 indicated a circular block, while lower values suggested non-circular shapes. To prevent misclassification of circles as squares, a secondary check using Harris Corner Detection to verify if the block had four corners. We measured squares by size, 35x35mm for large and 25x25mm for small blocks, to differentiate them. The algorithm can different square and rectangle shape, but for a vertical cylinder, its cycle shape can be detected as small square depends on where it is located. The method we used to reduce this false positive is to apply the height since both large and small block are below a certain height. A square shape above the height is considered as either vertical cylinder or cuboid.

$$Aspect\ Ratio = \frac{max(w,h)}{min(w,h)} \tag{3}$$

$$Circularity = \frac{4\pi * Area}{Perimeter^2} \tag{4}$$

III. Methodology - Control and Kinematics

A. *Teach and Repeat*

The primary goal of the teach and repeat task is for the bot to follow a path by recording the required waypoints and executing a playback. Functionality was added to the *control_station.py* and *state_machine.py* to perform teach and repeat for the block swapping task. The *Record Joint Position* button and state will store the joint angles of the waypoints into *self.recorded_joint_positions* when clicked. When *Execute* button is clicked, the *set_position()* function will be used to playback the recorded joint angles, and the gripper will open and close at the specified number of waypoints to pick and place the blocks. The waypoints were chosen to swap the two blocks through three positions without allowing the arm to collide with any block during path movements. The joint angles we teach and repeat for this task is shown in Figure 11.

B. *Forward Kinematics*

Forward Kinematics determines the position and orientation of the end effector in the workspace given the joint angles in the configuration space. For the RX200 arm, the configuration space $\mathbf{q} = [\theta_1^*, \theta_2^*, \theta_3^*, \theta_4^*, \theta_5^*]$ represents the joint angles of the arm, where θ_i represents base, shoulder, elbow, wrist, and wrist rotate angle, respectively. The end effector pose in the workspace is represented as $\mathbf{x} = [x, y, z, \phi, \theta, \psi]$, where $[x, y, z]$ denotes the position and $[\phi, \theta, \psi]$ represents the orientation relative to the base frame, in our implementation, we use the Z-Y-X Euler angle rotation to represent the orientation.

1) *Forward Kinematics Equation*

Rigid body motion can be represented by homogeneous transformation matrix $H = [R\ d;\ 0\ 1]$, where R is a rotation matrix, and d is the translation vector. The pose of the end effector can then be obtained by sequentially multiplying the transformation matrices for each joint. Pose of the end effector in the base frame:

$$H = A_1(q_1)A_2(q_2)\ldots A_n(q_n) = \begin{bmatrix} R_n^0 & o_n^0 \\ 0 & 1 \end{bmatrix} \tag{5}$$

where A_i in our project represents the transformation matrix from the joint frame $i-1$ to the joint frame i.

2) *DH Table*

We applied Denavit-Hartenberg (DH) convention to define joint frames and perform forward kinematics. Each A_i transformation is determined by four DH parameters: $[\theta_i, d_i, a_i, \alpha_i]$, representing the joint angle, joint offset, link length, and link twist. The relative transformation between consecutive joint frames is given by Eq. (6), which is implemented in the *get_transform_from_dh()* in *kinematics.py*.

TABLE I
DH Table

	a_i (mm)	α_i (rad)	d_i (mm)	θ_i (rad)
1	0.00	-1.57	103.91	$1.57 + \theta_1^*$
2	205.73	0.00	0.00	$-1.33 + \theta_2^*$
3	200.00	0.00	0.00	$1.33 + \theta_3^*$
4	0.00	1.57	0.00	$1.57 + \theta_4^*$
5	0.00	0.00	152.575	θ_5^*

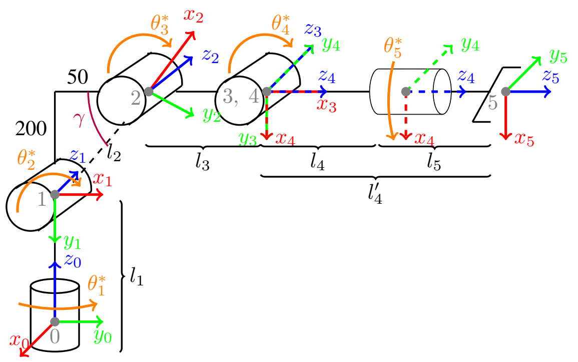

Fig. 1. DH Frame Schematic

$$A_i = Rot_{z,\theta_i} Trans_{z,d_i} Trans_{x,a_i} Rot_{x,\alpha_i}$$
$$= \begin{bmatrix} c\theta_i & -s\theta_i & 0 & 0 \\ s\theta_i & c\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c\alpha_i & -s\alpha_i & 0 \\ 0 & s\alpha_i & c\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

To obtain the DH parameter table for the RX200 arm, we first identified its links and joints, as shown in Figure. 1. Then, we set the z-axes for each frame corresponding to the positive direction of joint's rotation and identified the x-axes and origins using the relation between each z_i and z_{i-1}. We set the global frame at joint 0 frame and end effector frame at joint 5 frame. Next, we can obtain our DH parameters by determined how the frame rotate in z, translate in z, translate in x, and translate in x from $i-1^{th}$ frame to i^{th} frame. Note that we set the end effector frame in the middle of the gripper; thus, the link lengths are $[l_1,\ l_2,\ l_3,\ l_4,\ l_5] = [103.91,\ 205.73,\ 200.00,\ 65.00,\ 87.575]$ according to Figure. 12, and $l_4' = l_4 + l_5 = 152.575$. Additionally, the offset γ between joint 2 and joint 3 is $\gamma = atan2(200, 50) = 1.33$. The DH table we determined for the RX200 arm is shown as Table I.

C. *Inverse Kinematics*

Inverse Kinematics (IK) calculates the arm's joint angle values in the configuration space that allow the end effector to reach a desired position and orientation in the workspace. We use geometrical approach to find the joint angles from the given end effector pose. The code is in the *ik_geometric()* function in *kinematics.py*.

Using the geometry relationship shown in Figure. 2, we can simply compute the base angle θ_1 (line 195):

$$\theta_1 = \arctan(-x/y) \tag{7}$$

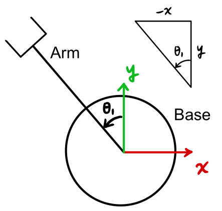

Fig. 2. A schematic of RX200 from bird eye view.

Since the shoulder joint θ_2, elbow joint θ_3, and wrist angle joint θ_4 are co-planar, we can use the 3-link RRR Arm IK method to find θ_2, θ_3, and θ_4. First, refer to Figure. 3 we can see that the wrist joint coordinate relative to the base joint would be (line 197 to 199):

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} -l_4\cos(\phi)\sin(-\theta_1) \\ -l_4\cos(\phi)\cos(-\theta_1) \\ l_4\sin(\phi) - l_1 \end{bmatrix} \tag{8}$$

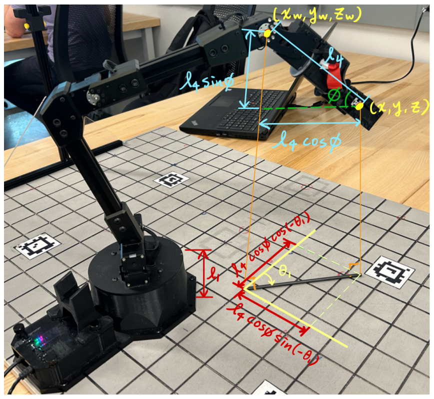

Fig. 3. A schematic of wrist end coordinate.

Now, we can use the 3-link Arm IK closed-form solution to find θ_2, θ_3, and θ_4. In the new defined arm plane (Figure. 4), the horizontal distance from shoulder to the wrist joint is the shortest distance from the projection of the wrist end to the shoulder joint in X-Y plane. The vertical distance is the height difference between the wrist end and the shoulder joint along the base Z-axis. Thus, the horizontal r and vertical distances s can be expressed as follow (line 201 and 202):

$$r = \sqrt{x_c^2 + y_c^2} \tag{9}$$

$$s = z_c \tag{10}$$

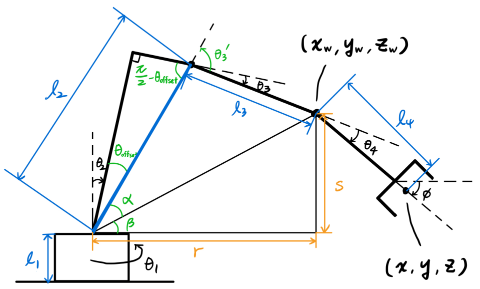

Fig. 4. A schematic of our IK geometrical solution

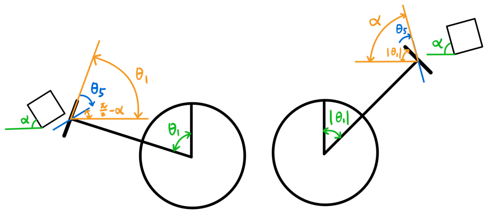

Fig. 5. A scheme of how to determine wrist rotate angle θ_5 from base angle θ_1 and block orientation α. The left one is when $\theta_1 > 0$, and the right one is when $\theta_1 <= 0$.

Using the geometrical closed-form solution, we can calculate θ_3 as follow (line 204 to 209, and 218):

$$\theta_3' = \cos^{-1}\left(\frac{r^2 + s^2 - l_2^2 - l_3^2}{2 l_2 l_3}\right) \tag{11}$$

$$\theta_3 = \theta_3' - \frac{\pi}{2} + \theta_{offset} \tag{12}$$

where $\theta_{offset} = atan2(50, 200)$ is the angle between the l_2 we defined and the real arm shoulder link (line 193). Additionally, we check if the value in arccos in Eq. (11) is valid (between -1 and 1) (line 205 to 208). Then, we can calculate θ_2 (line 211 to 213 and 219) and θ_4 (line 221):

$$\beta = atan2(s, r) \tag{13}$$

$$\alpha = atan2(l_3\sin\theta_3', l_2 + l_3\cos\theta_3') \tag{14}$$

$$\theta_2 = \frac{\pi}{2} - \theta_{offset} - \alpha - \beta \tag{15}$$

$$\theta_4 = \phi - \theta_2 - \theta_3 \tag{16}$$

To perform a smooth and accurate pick-and-place tasks later, we utilize the block orientation detected by the block detection function to adjust the wrist rotate angle θ_5, making it align with the block's orientation. Refer to Figure. 5, if the target block's orientation is α, the wrist rotate angle can be determined by (line 227 to 240):

$$\theta_5 = \begin{cases} \theta_1 - (\frac{\pi}{2} - \alpha), & \text{if } \theta_1 > 0, \\ \alpha - |\theta_1|, & \text{otherwise.} \end{cases} \tag{17}$$

D. Click to Grab and Drop

This task allows users to click on the control panel to set the grasp location and then click another position to set the drop location. The gripper will move to the grasp position, pick up the block, and place it at the drop position.

The first step is to record the grasp and drop positions. After starting camera calibration and block detection, the *calibrateMousePress()* function in *control_station.py* captures the first and second mouse click position in the pixel frame, converts it into world coordinates, and stores it in *self.sm.grab_position* and *self.sm.drop_position*, respectively. If the grab position corresponds to a detected block, its orientation is stored in *self.camera.theta_for_grab*, which is later used in inverse kinematics to adjust the wrist rotate angle θ_5.

To execute the click to grasp and drop task, we define a new state and function, *execute_pick_and_place()*, in *state_machine.py*. This function retrieves the target end effector grasp and drop position $[x, y, z]$ and sets target target orientation to $[\pi/2, 0, 0]$. The end effector follows these waypoints to perform the pick-and-place task: (1) move to 40 mm above the grasp position, (2) lower to the grasp position and close the gripper, (3) lift up 85 mm to make sure it won't collide with other blocks in the workspace when it's moving, (4) move to 85 mm above the drop position, (5) lower to the drop position and open the gripper, (6) lift up 40 mm again and go back the the initialize state.

For each waypoint, we sent the desired end effector pose into *self.rxarm.get_ik_joint()* to compute joint angles, and *self.rxarm.set_position()* moves to the arm. Once all the waypoints are executed, the task is successfully completed.

E. Competitions

We create new buttons and corresponding states for each event to execute the task, which are *sort_and_stack()*, *line_them_up()*, and *to_the_sky()*, respectively. For Event 1 and Event 2, we employed similar strategies and shared function to complete the tasks. (1) **Block Status Identification**: Our block detection approach mentioned in II-B filter out non-square distractors and classifies blocks by size, then store the orientation, color, size, position, and height of large and small square blocks in *self.camera.detected_large_blocks* and *self.camera.detected_small_blocks* dictionaries. (2) **Sorting by Color**: The *sort_blocks_by_color()* function in *state_machine.py* sort the blocks in rainbow order using the key in detected blocks dictionaries. (3) **Unstacking Blocks**: If a block's detected height exceeds a single block's length (we set 45 mm for large and 35 mm for small blocks as thresholds), it is identified as stacked. The *unstack_blocks()* function guide the arm to pick up and put aside stacked blocks.

1) Event 1 (Sort and Stack)

This event is completed by unstacking the stacked blocks, sorting the blocks in the detected block dictionaries by color, assigning target drop positions for stacking (large at (255, -25, 36(i-1)) and small at (-200, -50, 25(i-1)) for each i^{th} block), and pick and place the blocks in rainbow order with detected block center and target drop positions.

2) Event 2 (Line Them Up)

We raise the arm for the camera to have a full view to the workspace, unstack the stacked blocks, assign target drop position in order to lining them up, and execute pick-and-place sequentially in rainbow order. Additionally, a key strategy we implemented was rotating the wrist perpendicular to the lineup direction when dropping blocks. This prevent the gripper from colliding with already placed blocks during the lineup process.

3) Event 3 (To The Sky)

The strategy we used in this event is to pick up the block at (250,-25, 0) and assigned the drop positions at (0, 175, z_i), where $z_i = 41(i-1)$ for i^{th} block. After picking up the block, arm lift up to straight, turn the base to the direction that allows the arm align with the drop position, drop the block, lift the arm straight again, and turn the base back to the grap position.

4) Event 4 (Freethrow)

The basketball launcher is a catapult. There were four primary considerations to the initial catapult design: sizing, variability, arm operability, and simplicity. The launcher was sized to fit in the required space and achieve the required force and pointing for accurate shots. A basic catapult was augmented to be variable with multiple instances of features like the mount bar track in the lever and the attachment features for the loading mechanism so that the result of the shot could be tuned post-manufacturing. For robot arm use, the design incorporates large plates for arm contact to load and release. To simplify assembly, the design is a set of interlocking pieces manufactured with close tolerances to be sanded until the fit is just snug enough for operation.

The final launcher design is as a spring-loaded catapult with multiple teeth for variable loading. The catch teeth are held in place under tension from rubber bands which can be released by pressing a push plate to pull the catch mechanism back away from the crossbar. The launcher was designed in Onshape and then 3D printed in the lab. An isometric CAD view of the launcher is shown in Figure 6.

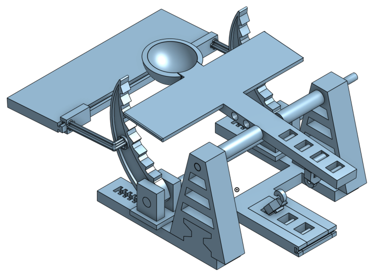

Fig. 6. The basketball launcher is a custom catapult design.

TABLE II
CAMERA INTRINSIC MATRICES COMPARISON

Method	Intrinsic Matrix
Factory	$\begin{bmatrix} 900.54 & 0 & 655.99 \\ 0 & 900.90 & 353.45 \\ 0 & 0 & 1 \end{bmatrix}$
Checkboard	$\begin{bmatrix} 910.11 & 0 & 646.84 \\ 0 & 912.81 & 348.23 \\ 0 & 0 & 1 \end{bmatrix}$

IV. RESULTS AND DISCUSSION - COMPUTER VISION

A. Camera Calibration

1) Intrinsic Camera Calibration

The intrinsic matrices obtained from two methods are shown in Table II. We performed three checkerboard intrinsic matrix calibrations and averaged the results, which showed slightly higher focal lengths (about 1.3%) and shifts of less than 10 pixels in both x and y directions. While factory calibration is stable and precise, the checkerboard method reflects real-world conditions but is prone to errors like imperfect corner detection and environmental variations. Therefore, we chose to use the factory calibration for the project since the camera setup remains unchanged.

2) Extrinsic Camera Calibration

The extrinsic matrices from manual measurements and the PnP algorithm differ notably, with a 27mm deviation in the Y-axis and 25.5mm in Z-depth as shown in Table III. These discrepancies might come from manual measurement errors and noise in the PnP method. Although there is some discrepancy in the PnP method, the PnP automatic extrinsic calibration with AprilTags is still more robust and automated than hand measurements since if the camera was moved. Therefore, we decided to use the PnP automatic extrinsic matrix to perform calibration in the project.

TABLE III
CAMERA EXTRINSIC MATRICES COMPARISON

Method	Extrinsic Matrix
Manual	$\begin{bmatrix} 0.9966 & 0.0259 & -0.0036 & 15.0000 \\ 0.0262 & -0.9899 & 0.1391 & 155.0000 \\ 0.0000 & 0.0000 & -0.9903 & 1030.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$
Automatic	$\begin{bmatrix} 0.9989 & 0.0074 & -0.0012 & 12.6852 \\ 0.0083 & -0.9974 & 0.0715 & 182.1489 \\ -0.0118 & -0.0716 & -0.9973 & 1004.4834 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$

3) Workspace Projection

To project the workspace flat and centered on the GUI, we used the method described in II-A4 to find our homography matrix, the result is as follow:

$$H_{warp} = \begin{bmatrix} 1.1732 & -0.0343 & -149.3866 \\ -0.0037 & 1.1429 & -79.3426 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \quad (18)$$

The top-left 2x2 submatrix handles the rotation, scaling, and shearing relative to the origin image. The bottom-left 1x2 submatrix indicates no perspective distortion, assuming the workspace is flat. The top-right 2x1 submatrix is the translation term, indicating that there is -149.3866 in the x and -79.3426 in the y being shifted. To verify our calibration, we projected grid points onto the image frame and compared them with actual positions on the board, as shown in Figure 7. The figure shows that the projected grid closely matched the expected position, confirming our transformation validity. We also routinely adjusted the camera orientation and performed calibration to ensure our automatic calibration functions properly. See the demo video here: Demo Video.

After all of the calibration steps, we notice that the z-coordinates in world frame reported on the control panel have some offset along both x- and y-axes. Therefore, we measured the offset and applied a linear correction to ensure the z-coordinate remains consistent at every x and y position:

$$\Delta z_{wrt\_x} = (0 - (-5))\frac{405 - x}{405 - (-407)} \quad (19)$$

$$\Delta z_{wrt\_y} = (0 - (-14)))\frac{430 - y}{430 - (-134)} \quad (20)$$

$$z_{corrected} = z_{origin} + \Delta z_{wrt\_x} + \Delta z_{wrt\_y} \quad (21)$$

To quantitatively evaluate our camera calibration, we stacked different number of blocks [0, 1, 2, 4, 6] in positions [(0,175), (-300,-75), (300,-75), (300,325)]. We then compared the actual world coordinates with the

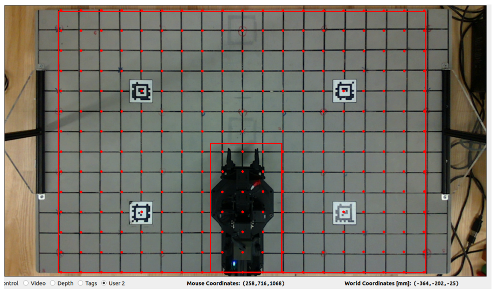

Fig. 7. Grid points after camera calibration and projection.

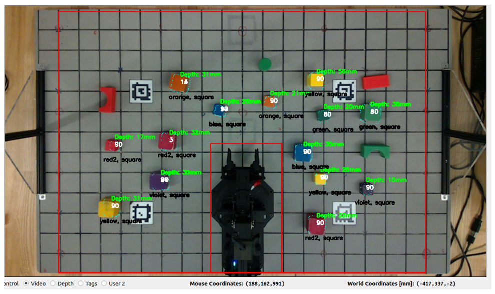

Fig. 8. The screenshot of detected blocks and labels after filter out non-square blocks.

world coordinate reported on the control panel after our calibration. Table VI shows the mean absolute error (MAE) in x, y, and z with different number of stacked blocks. The overall MAE of our calibrated world coordinates is 6.35mm in x, 4.5mm in y, and 5.75mm in z. This indicates that our calibration results are accurate.

B. Block Detection

Our block detection algorithm can detect both large and small cubic blocks and label the center position and height, orientation angle, color, and shape for each of them. Figure8 shows only cubic blocks are used, and the distrators are filtering out. To verify the accuracy of the block detector, we placed 39 blocks with spacing across the board. We aligned the block center to the grid points for center truth, then we used block center coordinate read from the detector and prepared a heatmap for detection error as shown in Figure 9. The average error is 6.85 mm for large blocks and 6.09 mm for small blocks. The errors are smaller in the center than in the corners, as the camera is located at the center (at the top of (0, 275)). Small blocks have less detection error than large ones, likely due to spacing. Large blocks are more prone to misclassifying the sides as the top when identifying contours, while smaller blocks, being shorter, are less likely to have this type of error. Additionally, higher errors are observed in areas closer to the arm, possibly due to placement inaccuracies caused by the arm's influence. Overall, our block detection can effectively

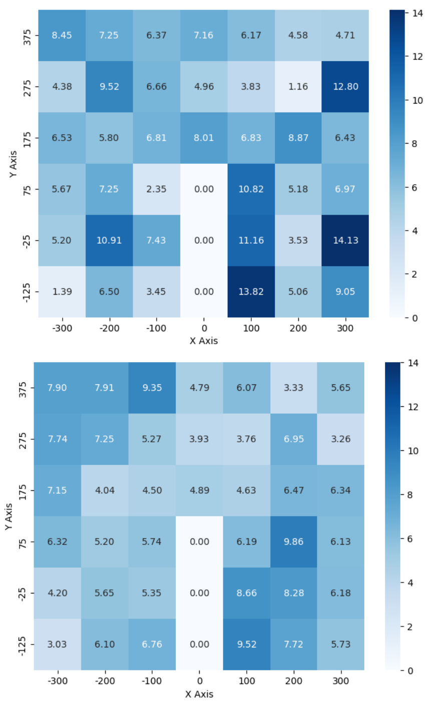

Fig. 9. The heatmaps of detection error for both the large (top) and small (bottom) blocks shows that the highest error was incurred in the bottom right section of the workspace.

identify the color, size, center, orientation, shape, and height of blocks.

V. Results and Discussion - Control and Kinematics

A. Teach and Repeat

The teach and repeat method successfully cycled the blocks between the two locations. As the execution continued to cycle, there some offset in the pick and place locations began to appear. The cycle was successfully performed 9 times. On the 10th cycle, the arm missed a grab. The joint angles are plotted over time for one cycle in Figure 10, and the end effector locations for those points are plotted in Figure 11.

B. Forward Kinematics

To verify our DH table, we move the arm to 7 points in the workspace and record the end effector position reported by our forward kinematics function and actual end effector position. Table IV shows the mean error and mean absolute error (MAE) between the reported and actual end effector positions. Both mean error and MAE are within 10 mm, which is an acceptable level of accuracy. This result verifies that our DH table and forward kinematics implementation are correct. Possible sources of error could be inaccuracy in the joint encoders or manually measurement inaccuracies.

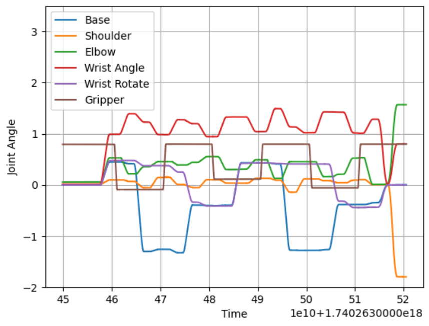

Fig. 10. The plot of joint angles over time for 1 cycle of executing teach and repeat task.

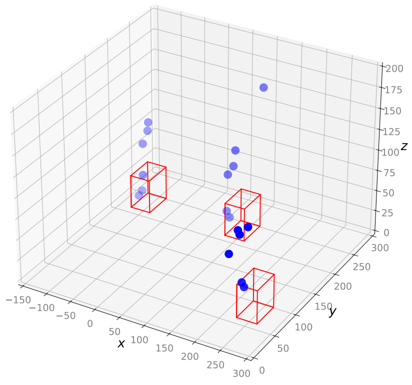

Fig. 11. The end effector positions during the teach and repeat process were calculated from the forward kinematics process.

C. Inverse Kinematics

To verify the correctness of our IK, we use FK for validation. Specifically, since we have validated the accuracy of our FK approach, we assume our FK output end effector pose as the ground truth. We placed the arm in several positions and recorded the FK output end effector poses, input these poses into our IK function to compute the closed-form solutions for each joint angle. Then, substituted the angles back into the FK to compute the end effector poses. Finally, we check if the original FK end effector poses match the FK end effector poses obtained after IK to confirm the correctness of our IK approach. The end effector pose from both approaches was successfully matched when we moved the arm to different positions, indicating that our IK is correct.

TABLE IV
FORWARD KINEMATICS ERROR

Error Function	x (mm)	y (mm)	z (mm)
$Mean\ Error$	0.13	−3.79	−3.82
MAE	5.16	4.21	1.21

TABLE V
COMPETITION RESULTS

Event	Success		Failure/Deficiency
	Practice	Competition	
1	Level 3	Level 3	1 block grab miss
2	Level 3	Level 1	Small on large
3	14 blocks	No attempt	Need fine-tuning
4	Level 1	Level 1	Arm not used

D. Click to Grab and Drop

This task was successfully implemented to move a block from one position to another using two mouse clicks on the control station image. The wrist rotate joint also successfully turned to the angle align with block orientation when grabbing. A demo video: Demo Video.

E. Competitions

The results of the competition event implementations are summarized in Table V. In practice, all requirements for Level 3 were achieved in Event 1 and 2, except that the code could not work for setups where small blocks were stacked on large blocks as the large blocks are searched for first and were not detected accurately under the small blocks. Additionally, the color detection misinterpreted the overlapping blocks as a combination of both colors. Furthermore, during our practice, we we discovered an offset in the arm's base, which led to failed grasps. To address this, we applied a 5-degree correction to θ_1 in our IK function, significantly improving the performance. In Event 3, we implemented the algorithm, but did not have enough time for fine-tuning and debugging. As a result, we did not attempt this event during the competition. However, our practices show that our strategies are feasible.

For Event 4, we did not use arm in the competition as it do not have enough torque to depress the loading mechanism and could not reach the center of the release plate to execute a straight shot. The arm was able to pick up the basketballs and place them in the launcher, however, this process was slow and more points could be earned by firing completely manually, resulting in 60 scored baskets. The design lab improvements to earn more points involve increasing shot frequency, accuracy, and aiming control by using motors for fire and pan/tilt adjustments.

VI. Conclusion

In this ArmLab project, the experimental data show that our system has high accuracy in camera calibration, block detection, kinematics, and control. While we did not complete all competitions, practice results indicate that our system and strategy have potential to accomplish the challenges. We also identified areas for improvement during the competitions. Our block detector, which relies on contours to detect color, shape, and size, struggled with crowded or stacked blocks, leading to detection errors. Future work could focus on refining detection methods and improving robustness. The arm was also able to execute trajectories based on those detections and our IK solution, either through waypoints or path-planning rules. In future, more efficient path planning rules could be implemented.

References

[1] Devonics. (2024) The impact of robotic arms on society. [Online]. Available: https://www.devonics.com/post/the-impact-of-robotic-arms-on-society

[2] T. Robotics. (2024) Reactorx-200 drawings and cad files. [Online]. Available: https://docs.trossenrobotics.com/interbotix_xsarms_docs/specifications/rx200.html#drawings-and-cad-files

VII. Appendices

TABLE VI
World Coordinate Mean Absolute Error After Calibration

Stacked Blocks	x (mm)	y (mm)	z (mm)
0	3.00	2.25	0.375
1	2.75	3.25	0.375
2	3.5	4.75	1.00
4	3.00	4.00	1.50
6	3.25	3.25	1.375

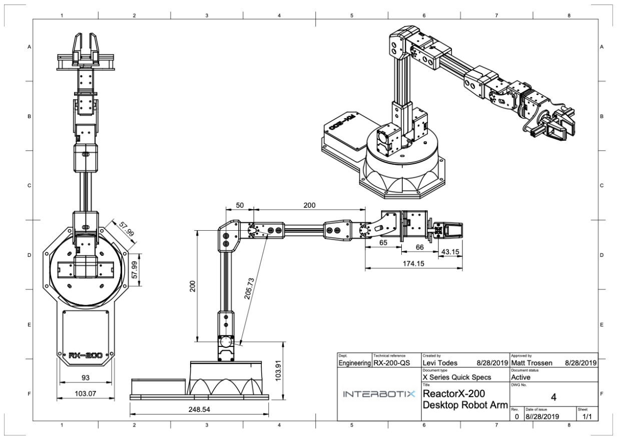

Fig. 12. The official technical drawing for ReactorX-200 Robot Arm. [2]

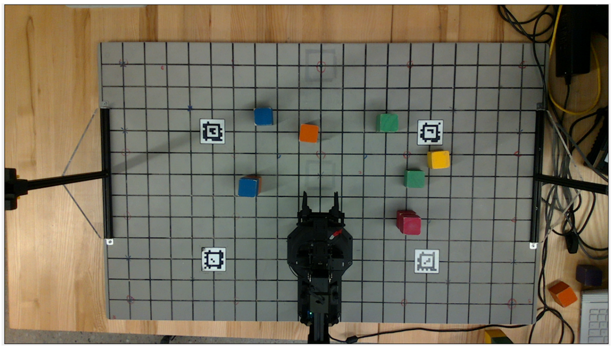

Fig. 13. Video frame before calibration and projection.

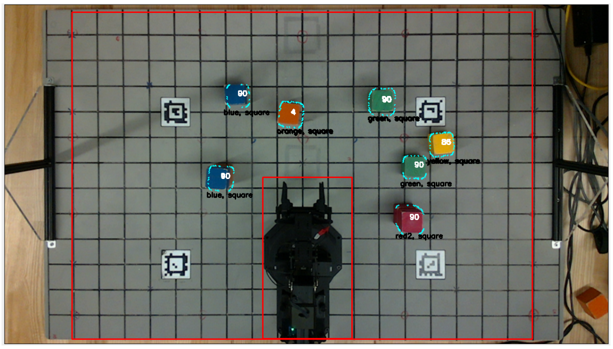

Fig. 14. Video frame after calibration and projection.

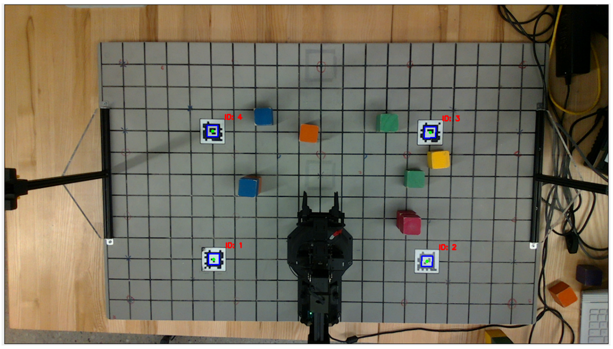

Fig. 17. Apriltag detection result before calibration and projection.

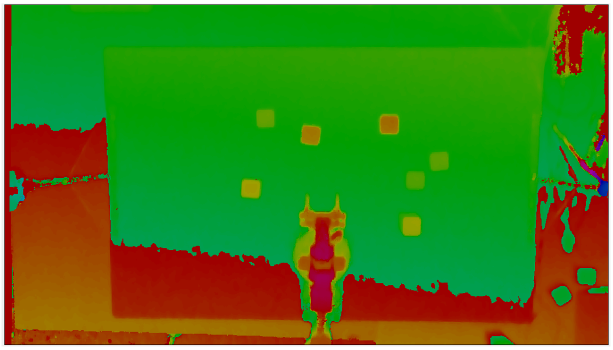

Fig. 15. Depth frame before calibration and projection.

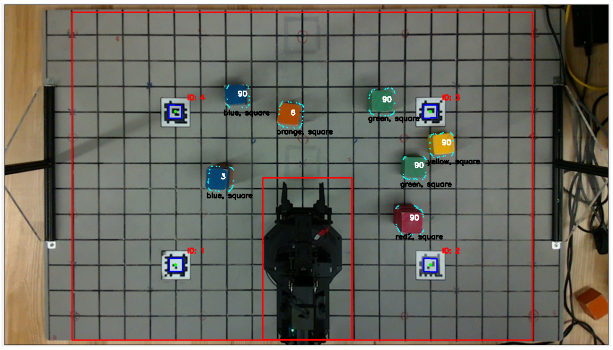

Fig. 18. Apriltag detection after calibration and projection.

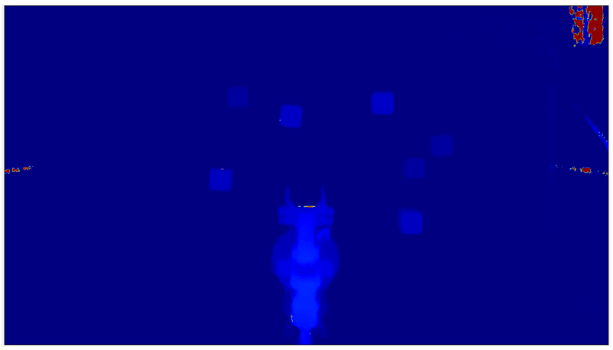

Fig. 16. Depth frame after calibration and projection.

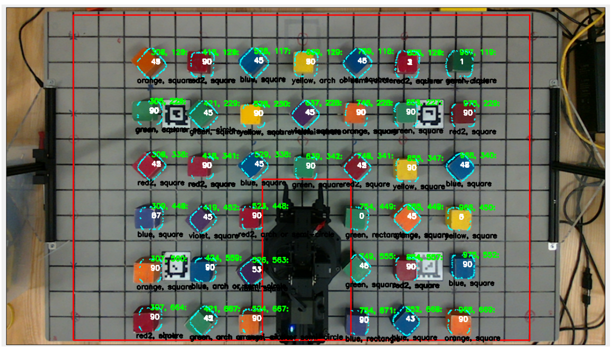

Fig. 19. Block detection result with large blocks for location error calculation.