ROB 550 ArmLab Report Team 5 (Afternoon)

Anirudh Aatresh, Christopher Nesler, Joshua Roney
{aaatresh, neslerc, jproney}@umich.edu

***Abstract*—With good dexterity and vision, a robotic manipulator can undertake a variety of repetitive tasks such as sorting and stacking with considerable accuracy. In this report, we study algorithms based on computer vision and kinematics that can make these feats a reality. These algorithms include functionality to create and execute teach-and-repeat trajectories, perform forward kinematic and inverse kinematic pose generation, detect blocks and block colors, and autonomously sort and stack large and small blocks. We analyze the performance of our implementations through data collected during experiments and suggest further improvements for future work.**

I. Introduction

Multiple degree-of-freedom manipulator arms are a form of robots that are very widely used in industrial and academic settings, alike. In this paper we present our work with the RX200 arm, in which we execute teach-and-repeat behavior, and use color- and LIDAR-based imaging to implement block detection and several automated sorting and stacking tasks.

II. Methodology

A. Camera Calibration

The camera model consists of internal and external parameters, and the process of estimating these parameters is known as camera calibration. The former consists of parameters such as the focal length f, optical axis offsets x_0 and y_0, axis skew s, lens distortion, etc., which are collectively described by the intrinsic matrix K. Cameras are used in various applications and are mounted in a variety of ways. The extrinsic matrix must be designed to describe application-specific external parameters.

1) Intrinsic matrix calibration: A factory-calibrated version of the intrinsic matrix is stored in the camera and can be extracted using a checkerboard calibration procedure. Following the procedure depicted in [1] and [2], we performed four trials of intrinsic matrix calibration and compared their average to the factory calibration intrinsic matrix in III-A1.

2) Extrinsic matrix calibration: Using the physical dimensions of the workspace, we performed a manual calculation to determine a rough extrinsic matrix, which was a transformation between the camera and workspace reference frames. For this transformation, we determined the translational component by measuring the distance between the origins of the workspace and camera frames. The rotation matrix was obtained by considering an inversion in the Y and Z axes of the camera frame with respect to the workspace frame. Furthermore, this extrinsic matrix coupled with the intrinsic matrix can be used to find a transformation between pixel $[u, v]$ coordinates and workspace $[x_w, y_w, z_w]$ coordinates with depth d through the relations:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = z_c K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} ; \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = H^{-1} \begin{bmatrix} x_c \\ y_c \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 976 - d \\ 0 \end{bmatrix} \tag{1}$$

where, K is the intrinsic matrix, H is the extrinsic matrix and $[x_c, y_c, z_c]$ refers to a point in the camera frame. Depth d at pixel coordinates $[u, v]$ is obtained from the depth image captured by the LIDAR sensor.

To keep the calibration procedure simple, yet accurate, we implemented a semi-automated *click to calibrate* procedure. We clicked the centers of four AprilTags [3] and two other points using the mouse pointer and recorded their pixel coordinates (2D coordinates). Using their established 3D workspace coordinates, we found the extrinsic matrix for this camera configuration using a perspective N-point solver. Using the `solvePnP` method from the OpenCV library, we input these 3D - 2D point pairs along with the camera's intrinsic matrix and distortion coefficients, which were obtained during the calibration procedure mentioned in section II-A1.

The resulting extrinsic matrix was then put into eq. (1) to convert between workspace, camera and pixel coordinates. Although our $X-Y$ coordinates were fairly accurately determined using the inverse of the extrinsic matrix, we found that the z_w coordinate was off by about $50\ \mathrm{mm}$. To fix this deviation, we instead performed depth calibration using the formula: $z_w = 976 - d$.

B. Block Detection

To find blocks within the workspace, we masked off areas in the image which we would not consider viable for block detection. Three rectangular regions were defined in the image in which blocks may be found. These rectangles include the portion of the workspace to the left of, above, and to the right of the robot (as seen from view in Fig. 1). This eliminates the portion of the

image beyond the workspace surface, as well as most of the RX200 in its sleep pose (see Fig. 1). The image was also filtered to exclude any pixels corresponding to depth measurements over 960 mm. This prevents the majority of the workspace surface from being considered by the algorithm, reducing false positive contours.

After the masks were applied, the `FindContours` OpenCV method was used to detect contours in the depth image. We screened each of the detected contours for eligibility as blocks using the following criteria. First, if a contour was less than 200 pixels (or greater than 5000) in area, it was deemed too small (or large) to be a block. Second, a bounding rectangle of minimum area was fit around each contour using the `minAreaRect` OpenCV method. If the ratio of this bounding rectangle's width and height was less than 0.5 or greater than 2.0, the contour was eliminated for not being sufficiently square. The remaining contours were considered blocks and each contour's centroid was stored for use as a destination pose via inverse kinematics. The orientation of each block, also used by the inverse kinematic trajectory planner, was found by converting the orientation provided by OpenCV's `minAreaRect` method to radians.

The contours detected in the depth image were then used to identify the corresponding pixels in the RGB image. We converted the RGB data for all pixels in each contour to HSV. Then we took the mean hue of each contour and checked whether it fell within any of a set of ranges we assigned to each potential color (red, orange, yellow, green, blue, violet). We tuned the color hue ranges by manually obtaining the color on the top surface of a block of each color from our workstation's live output image. This accounted for the true setup of our workspace (i.e. influence from our camera, lighting, etc.).

A modified version of this procedure was implemented for cases in which destacking of blocks may be necessary, such as the competition tasks. In this modified procedure, before searching for blocks as described above, contours were found and evaluated at several "slices", each corresponding to a thin range of depth values. Each slice was centered about the expected depth of a particular stack (i.e., two, three, or four small blocks only or large blocks only). This eliminates the issue presented by the side wall of a stack being included in a detected contour, making it insufficiently square to be considered a block by the above method. Then, if contours were found in a slice, and they met the other criteria above, they would be considered blocks located at the $X-Y$ coordinates of their centroid, and the Z-coordinate of the corresponding slice.

To evaluate the performance of our (unstacked) block detection, large blocks were placed on the workspace table in four rows of six at known locations and orientations, as seen in Fig. 1. Each row (located at Y = 325, 175, 25, and -125 mm) had blocks placed at X = -400, -250, -100, 100, 250, and 400 mm. Each row contained one block of each of the aforementioned colors, with their position shifting two spots to the left in each subsequent (descending row) to evaluate different colors in different portions of the workspace. Finally, the blocks were placed with orientations alternating between 0 and $\pi/4$ rad, except where prevented by workspace geometry. The $X-Y-Z$ centroid coordinates, computed orientation, contour size in pixels, and detected color were recorded and evaluated for accuracy. We generated a heatmap of the centroids' positional errors as measured by the two-norm of the X, Y, and Z errors. True block height was assumed to be 38 mm for the purpose of this calculation.

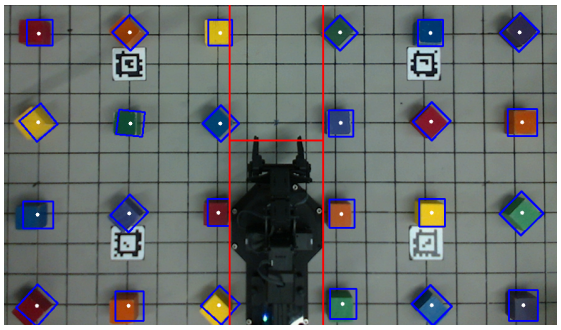

Fig. 1. Array of large blocks used for block detection algorithm accuracy. Centroids (white) and contour bounding boxes (blue) superimposed on RGB image of workspace.

C. Teach and Repeat

We modified the user interface to include several buttons that enable the operator to teach the RX200 a desired trajectory and have it replicate that behavior. A "Start Teach" button initializes an array for storing joint states and disables torque on the arm, allowing the operator to freely maneuver it into a desired configuration. Two buttons labeled "Pose Grip: Open" and "Pose Grip: Closed" allow the user to indicate which state the end-effector should be in at the next recorded pose. The "Record Pose" button adds the current joint angles to the array of poses, along with the desired gripper state. Once all desired poses have been recorded, the "End Teach" button stores the array of robot configurations in a file labeled 'Poses.csv'. These poses, or any manually-entered list of poses stored in a file of this name, can be loaded by pressing the "Recital" button, and executed using the already-present "Execute" button. The "Execute" state in the state machine iterates through a list of given waypoints, and uses the `set_positions` method of the `RXArm` class to move the arm to a given waypoint from its current joint state with a default time of 2 seconds. In cases where only initial and final waypoints

are given, we create intermediate waypoints in between them to make the entire motion smoothly follow the desired trajectory.

We evaluated the performance of the teach-and-repeat feature by training the arm to swap the positions of two blocks, located at (-100, 225) and (100, 225) on the workspace surface. We used an intermediate holding location of (250,75) to temporarily store one block while the other was manipulated. This process was repeated as many times as possible before error accumulation led to a failure to successfully pick and place the blocks.

This implemented teach-and-repeat function was used for tuning the RX200 arm's PID parameters. To avoid overworking the motors we elected to use an I gain of 0 for all motors. We adjusted the P gains of motors until they were enough to reach the target angles and started to oscillate around them. The D gains were then increased until the oscillation ceased and there was no more overshooting.

D. Forward Kinematics

We used the Product of Exponentials method (PoX) for forward kinematics on the RX200 arm. This method takes in the angle for each joint on the robot and calculates the resulting position and orientation of the end-effector. First, we identify the reference configuration of the robot, in which each joint has a value of 0 rad (see Fig. 2). We then form a 4x4 transformation matrix, as in eq. (2), that represents the end-effector position as a 3x1 vector, p, and orientation relative to the base frame of the robot as a 3x3 matrix, R. In this reference configuration the end-effector orientation is aligned with the world axes so the rotation portion of the transformation matrix is an identity matrix and the translation portion is where it is located in world coordinates.

$$g_{st}0 = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -4.5 \\ 0 & 1 & 0 & 429.15 \\ 0 & 0 & 1 & 301.91 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

Next we form a 6-dimensional vector, known as a twist ξ, to represent each joint. The twist is found by identifying the unit axis of rotation of joint i, ω_i, and an associated point on that axis, q_i. The top half of the twist, v_i, is the cross product of the negative rotation axis and the point, $-\omega_i \times q_i$, and the bottom half is the axis of rotation for that joint (eq. (3)). The twists calculated for each joint in the reference configuration (Fig. 2) are listed in eq. (4) - (8).

$$\xi_i = \begin{bmatrix} -\omega_i \times q_i \\ \omega_i \end{bmatrix} \tag{3}$$

$$\xi_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T \tag{4}$$

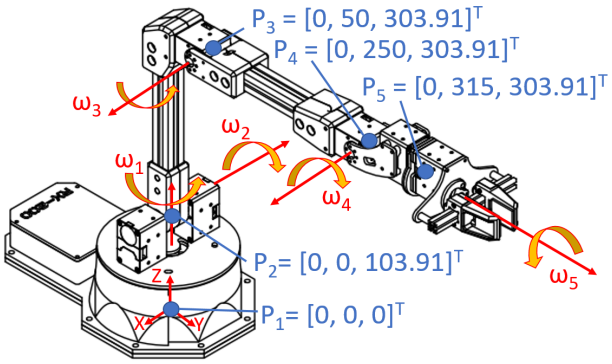

Fig. 2. Schematic of RX200 arm annotated with points, robot frame coordinate system, and screw vectors. Schematic from [4].

$$\xi_2 = \begin{bmatrix} 0 & -103.91 & 0 & -1 & 0 & 0 \end{bmatrix}^T \tag{5}$$

$$\xi_3 = \begin{bmatrix} 0 & 303.91 & -50 & 1 & 0 & 0 \end{bmatrix}^T \tag{6}$$

$$\xi_4 = \begin{bmatrix} 0 & 303.91 & -250 & 1 & 0 & 0 \end{bmatrix}^T \tag{7}$$

$$\xi_5 = \begin{bmatrix} -303.91 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}^T \tag{8}$$

Each twist then has its matrix exponential calculated using eq. (9) obtained from equation 2.36 in Murray, Li, Sastry (MLS). [5].

$$e^{\hat{\xi}\theta} = \begin{bmatrix} e^{\hat{\omega}\theta} & (I - e^{\hat{\omega}\theta})(\omega \times v) + \omega\omega^T v\theta \\ 0 & 1 \end{bmatrix} \tag{9}$$

In which ω is the axis of rotation for the joint, v is the top half of the twist, the variable I is the 3x3 identity matrix, θ is the angle of the joint, and $e^{\hat{\omega}\theta}$ is the rotation matrix of the joint. The rotation matrix $e^{\hat{\omega}\theta}$ is found using Rodrigues' Formula (eq. (10)) which is found as equation 2.14 in MLS [5].

$$e^{\hat{\omega}\theta} = I + \hat{\omega}\sin\theta + \hat{\omega}^2(1 - \cos\theta) \tag{10}$$

In which I is the 3x3 identity matrix, θ is the joint angle, and $\hat{\omega}$ is the skew-symmetric matrix of the joint's rotation axis ω (eq. (11)).

$$\hat{\omega} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \tag{11}$$

Once the joints' twist exponential matrices are calculated they can the be multiplied, in joint order, with the reference configuration matrix to calculate the end-effector's new pose in a 4x4 matrix $g_{st}\theta$ (eq. (12))

$$e^{\hat{\xi}_1\theta_1}e^{\hat{\xi}_2\theta_2}e^{\hat{\xi}_3\theta_3}e^{\hat{\xi}_4\theta_4}e^{\hat{\xi}_5\theta_5}g_{st}0 = g_{st}\theta \tag{12}$$

We represent the orientation as an XYZ Euler angle rotation, also referred to as Roll-Pitch-Yaw. This representation breaks a rotation matrix down into 3 consecutive rotations about the world X-axis, Y-axis, and Z-axis by angle values of ψ, θ, and ϕ, respectively. The values

of these angles are solved for by comparing the given rotation matrix against the XYZ Euler Angle rotation matrix (eq. (13)) in which $c\theta$ is short for $\cos\theta$ and $s\theta$ is short for $\sin\theta$. The resulting output of the forward kinematics is a 6-dimensional pose vector containing the $X-Y-Z$ coordinates and the XYZ Euler angles of the end-effector in the world frame (eq. (14)). This vector's contents were displayed in the GUI to show the end-effector's current pose.

$$\begin{bmatrix} c\theta c\phi & -c\psi s\phi + s\psi s\theta c\phi & s\psi s\phi + c\psi s\theta c\phi \\ c\theta s\phi & c\psi c\phi + s\psi s\theta s\phi & -s\psi c\phi + c\psi s\theta s\phi \\ -s\theta & s\psi c\theta & c\psi c\theta \end{bmatrix} \quad (13)$$

$$pose = \begin{bmatrix} x_w & y_w & z_w & \psi & \theta & \phi \end{bmatrix}^T \quad (14)$$

E. Inverse Kinematics

The arm's inverse kinematics were solved using PoX, similar to forward kinematics, and the Paden-Kahan (PK) subproblems obtained from [5]. The following equations are implemented in the 'kinematics.py' file of the code and will be referenced by line. The inverse kinematics function $'IK\_pox'$ begins at line 170 by taking a desired pose vector of the form shown in eq. (14) and converts it into a 4x4 transformation matrix using the function $'pose\_to\_T'$ located at line 426. The function calculates rotation matrices of the given Euler angles using the $'scipy.linalg.expm'$ function and multiplies them together to form an XYZ Euler Angle matrix (eq. (15)). The coordinates of the pose are used to form the first 3 rows of the 4^{th} column, p. A row of zeros and a one are added to complete the 4x4 homogeneous transformation matrix known as g_d or $g_{st}\theta$, following the form in eq. (2).

$$R_{tot} = R_z(\phi)R_y(\theta)R_x(\psi) \quad (15)$$

When g_d is formed from the desired pose, the PoX equation is manipulated to begin solving for the values of θ. It starts by multiplying both sides of eq. (12) by the inverse of the reference configuration matrix $g_{st}0$ to single out the joint transformation matrices and results in a new matrix g^1 seen in line 193 of the code (eq. (16)).

$$e^{\hat{\xi}_1\theta_1}e^{\hat{\xi}_2\theta_2}e^{\hat{\xi}_3\theta_3}e^{\hat{\xi}_4\theta_4}e^{\hat{\xi}_5\theta_5} = g_{st}\theta g_{st}^{-1}0 = g^1 \quad (16)$$

The value of joint angle θ_1 can be solved for using geometry (lines 200-201 of the code) since the waist joint is the only joint that controls what angle the entire arm faces on the x-y plane. The arctan2 difference between the target $X-Y$ coordinates (x_d, y_d) and the reference $X-Y$ coordinates (x_o, y_o) is found and returns the value of θ_1 needed to rotate the arm to face the target coordinate (eq. (17)). This value is clamped between 2π and -2π to ensure it provides a realistic rotation. Lastly, the twist exponential transformation matrix of joint 1 as well as its inverse are calculated using this found value for θ_1 using eq. (9). Both sides of eq. (16) were multiplied by the inverse matrix to achieve a new matrix g^2 in lines 207-209 of the code (eq. (18)).

$$\theta_1 = \arctan 2(y_d, x_d) - \arctan 2(y_o, x_o) \quad (17)$$

$$e^{\hat{\xi}_2\theta_2}e^{\hat{\xi}_3\theta_3}e^{\hat{\xi}_4\theta_4}e^{\hat{\xi}_5\theta_5} = (e^{\hat{\xi}_1\theta_1})^{-1}g^1 = g^2 \quad (18)$$

By multiplying both sides of eq. (18) by a known point that rests on both the rotation axes of joints 4 and 5 $p4$, the transformation matrices for joints 4 and 5 can be removed since a transformation matrix multiplied by a point on the rotation axis will equal the same point (eq. (19)). Then by subtracting both sides by a point on joint 2's rotation axis (eq. (20)), the setup for PK subproblem 3 is complete and 2 possible values for θ_3 can be solved for.

$$e^{\hat{\xi}_2\theta_2}e^{\hat{\xi}_3\theta_3}e^{\hat{\xi}_4\theta_4}e^{\hat{\xi}_5\theta_5}p_4 = e^{\hat{\xi}_2\theta_2}e^{\hat{\xi}_3\theta_3}p_4 = g^2p_4 \quad (19)$$

$$e^{\hat{\xi}_2\theta_2}||e^{\hat{\xi}_3\theta_3}p_4 - p_2|| = ||g^2p_4 - p_2|| \quad (20)$$

The PK subproblem 3 (implemented in the function $'PK3'$ at line 306 of the code) involves rotating about a single axis to a given distance. The distance (δ) is calculated as the squared norm of vector $g^2p_4 - p_2$. The solution to this subproblem is obtained from MLS pg. 102-103 [5]. The goal of this subproblem involves rotating the point p_4 around axis 3 until it is the distance δ away from point p_2. Up to two solutions exist for the value of θ so a logic statement was added at line 229 to choose the smaller value of the valid values to maintain an elbow-up position. With the value of θ_3 chosen, the corresponding transformation matrix for joint 3 can be calculated along with its inverse (lines 238-240 of the code). Returning to eq. (19), the value of $e^{\hat{\xi}_3\theta_3}p_4$ can be calculated to a point p_4' and the result is the setup for PK subproblem 1 (eq. (21)).

$$e^{\hat{\xi}_2\theta_2}p_4' = g^2p_4 \quad (21)$$

The PK subproblem 1 involves rotating a point about an axis until it reaches another given point and is implemented at line 398 of the code. In this case, point p_4' is being rotated about joint 2 until it reaches point g^2p_4. The solution to this subproblem was obtained from MLS pg. 99-100 [5]. The solution yields only one possible solution. Similar to joints 1 and 3, the transformation matrix of joint 2 and its inverse are calculated at lines 256-257. The inverse of the first 3 joints' transformation matrices are multiplied to both sides of eq. (16) at line 261 to single out the final joint angles to calculate, θ_4 and θ_5 and results in a new matrix g^3 (eq. (22)).

$$e^{\hat{\xi}_4\theta_4}e^{\hat{\xi}_5\theta_5} = (e^{\hat{\xi}_3\theta_3})^{-1}(e^{\hat{\xi}_2\theta_2})^{-1}(e^{\hat{\xi}_1\theta_1})^{-1}g^1 = g^3 \quad (22)$$

Multiplying both side of eq. (22) by a point on joint axis 5 and not joint axis 4 (p_5) will single out joint 4's matrix and result in another setup for PK subproblem

1 which will yield one solution for θ_4 (eq. (23)). The transformation matrix for joint 4 and its inverse are calculated and the inverse is multiplied with g^3 to produce a new matrix g^4 (eq. (24)). Lastly, θ_5 can be solved with PK subproblem 1 by multiplying both sides of eq. (24) with any point not on joint 5's rotation axis, we used origin of the world frame p_0 (eq. (25)).

$$e^{\hat{\xi}_4\theta_4}e^{\hat{\xi}_5\theta_5}p_5 = e^{\hat{\xi}_4\theta_4}p_5 = g^3p_5 \tag{23}$$

$$e^{\hat{\xi}_5\theta_5} = (e^{\hat{\xi}_4\theta_4})^{-1}g^3 = g^4 \tag{24}$$

$$e^{\hat{\xi}_5\theta_5}p_0 = g^4p_0 \tag{25}$$

The resulting joint angles are returned as a 5-dimensional vector. Error catching was added to the function in the event that a joint angle could not be calculated due to the desired pose being outside of the robot's reachable workspace. If any joint angle value is calculated as a NaN, the angle is unachievable and the function exits and returns a $None$ while printing which joint angle could not be solved.

F. Click To Grab

A click-to-grab feature was implemented to allow a user to click a position displayed on the GUI and command the robot to grab the object at that location then return to its origin pose. The user would then click another location on the GUI to command the robot to place the object there then return to its origin pose. Each click would retrieve the pixel coordinates of that location and convert it to world coordinates using eq. (1). The given world coordinates and a prior location set slightly above the given location were run through the inverse kinematics described in section II-E for joint angle positions to achieve. These joint angles were sent through the `set_positions` method of the `RXArm` class as waypoints to plan the trajectory.

G. Task Automation

1) Event 1: Pick 'n Sort: For this event, we use the block detection system described in section II-B to find the locations of all the blocks lying within the positive-Y half plane (valid workspace). This entire pick and sort sequence is repeated indefinitely until no more blocks exist in the valid workspace. At the start of every pick and sort sequence, a snapshot of the workspace is taken and processed using our block detector framework. This system also returns the area of each contour associated with a specific block and we use this data to classify blocks as large or small based on an area threshold (1000). Moreover, using the orientation data from `minAreaRect` method, the end-effector of the arm is oriented correctly to permit more robust grasping when coming in from the top. To increase the size of the workspace, we used a horizontal grasp algorithm to come in from the side when attempting to grasp a block.

Once all blocks were detected, the centroid of each block contour was used as input to the inverse kinematics to determine a valid joint configuration. The arm was made to pick up the block and move to an intermediate pose before commencing the drop sequence. It then drops the block in the fourth quadrant of the workspace if the block was classified as large and in the third quadrant if small. The drop location was updated after each drop to prevent the arm from trying to place a block at a location where a previously placed block already exists. After an attempt has been made to pick and sort each block in the captured snapshot, the entire sequence is repeated until no more blocks are found in the subsequent snapshot.

2) Event 2: Pick 'n Stack: We adopt a similar strategy to pick up blocks as the first event (section II-G1). However, the block placement is done such that the small and large blocks are placed on separate stacks in the third quadrant of the workspace. When performing the dropping sequence, the height at which each block was dropped was incremented after each drop and the drop height was reduced from that of event 1 (section II-G1) to be more gentle while stacking.

3) Event 3: Line 'em Up: We used the color detection component of the block detector (section II-B) to determine the color of each block. Once the blocks' colors were determined, we sorted each block based on their color following the **ROYGBV** precedence, where **R**ed was given rank 1 and **V**iolet was given rank 6. The *quicksort* [6] algorithm was used to sort this list, which was then mapped to a predefined list of color-specific locations. This list was then iterated through, leading to each block being placed in a line in their respective locations. By using a predefined location list for each color, we made sure that sufficient space was left (in the ROYGBV order) for blocks that were missed in the first snapshot and captured in subsequent iterations. The location list was created such that two parallel lines were created, one for each size in the third quadrant.

A destacking algorithm was implemented to deal with cases when blocks were stacked in the workspace. In our implementation of this algorithm, blocks were removed from the stack and placed in the fourth quadrant in a grid. Once all stacks were destacked, blocks were then lined up in order in the third quadrant.

4) Event 4: Stack 'em High: The color sorting algorithm used in section II-G3 was adopted here as well. However, we iterate through the sorted list to create separate color sorted (in ROYGBV order) stacks in the third quadrant. The dropping sequence used here is similar to that used in event 2 (section II-G2).

III. Results

A. Camera Calibration

1) Intrinsic matrix calibration: The average intrinsic matrix obtained by performing four trials of the checkerboard calibration procedure can be seen as K_{avg} in eq. (26) along with the distortion coefficients in eq. (27).

$$K_{\text{avg}} = \begin{bmatrix} 954.6327 & 0.0000 & 629.4831 \\ 0.0000 & 968.4867 & 386.4730 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \tag{26}$$

$$\text{dist. coeffs} = [0.1505, -0.2453, 0.0002, -0.0014] \tag{27}$$

The standard deviation across the four trials of each element of the intrinsic matrix was calculated and can be found in the appendix (VI). There is a slight deviation in values of parameters pertaining to the focal length and x_0 offset. The most deviation was observed in the y_0 offset value. Some of the sources of error in the intrinsic matrix calibration procedure are variations in lighting, the straightness of lines on the checkerboard and the care taken to complete the entire calibration procedure.

The factory calibration intrinsic matrix can be seen as K_{factory} in eq. (28)

$$K_{\text{factory}} = \begin{bmatrix} 904.3176 & 0.0000 & 644.0140 \\ 0.0000 & 904.8245 & 360.7775 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \tag{28}$$

These two matrices (K_{avg} and K_{factory}) differ significantly, especially in f_x and f_y terms. We trust the intrinsic matrix obtained through our calibration procedure as it depicts the transformation matrix obtained from the most recent operation of the camera compared to the factory calibration matrix which would have been recorded during manufacture. The newly obtained calibration matrix would take into consideration any imperfections that occur due to deterioration through use.

2) Extrinsic matrix calibration: This nominal extrinsic matrix can be found in eq. (29).

$$H_{\text{nominal}} = \begin{bmatrix} 1 & 0 & 0 & -14.1429 \\ 0 & -1 & 0 & 194.4616 \\ 0 & 0 & -1 & 978 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

The extrinsic matrix obtained by the *click to calibrate* procedure can be found as H_{cal} in eq. (30).

$$H_{\text{cal}} = \begin{bmatrix} 0.9999 & 0.0070 & -0.0043 & 48.9035 \\ 0.0068 & -0.9989 & -0.0450 & 180.2612 \\ -0.0046 & 0.0449 & -0.9989 & 1020.4982 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \tag{30}$$

The nominal and calibrated extrinsic matrices differ in values, especially in their translational components. Their rotation matrices however are more similar. The absolute difference between these two matrices can be found in the appendix (VI). The sources of error between these two matrices can be deviations in measurement of 3D points in the workspace and the fact that the camera might have moved or been disturbed between experiments. This would mean that the matrix obtained through a calibration procedure would be more accurate than the nominal extrinsic matrix.

In order to verify the accuracy of our calibration procedure, we compared the workspace coordinates obtained through the application of eq. (1) with that of a tape measure. The errors in $X - Y$ coordinates are given in Table I after applying a correction formula (eqs. (31), (32), (33)) that compensates for errors. The values obtained before compensation can be seen in the appendix (VI). The error in Z is the error in depth measurement by the LIDAR, which was found to be 2.25mm. These errors lie within 10mm and were considered within tolerable limits for future experiments.

TABLE I
ℓ_2 NORM (IN mm) OF ERROR IN WORKSPACE $X - Y$ COORDINATES WHEN THE NUMBER OF BLOCKS ON THAT COORDINATE ARE VARIED.

Blocks	(0 , 175)	(-300 , -75)	(300 , -75)	(-300 , 325)
0	3.16	1.41	4.47	1.00
1	5.00	7.28	5.38	1.41
2	5.83	8.06	7.28	3.60

$$x'_w = \begin{cases} 1.0526 x_w & \text{if } x_w < 0 \\ 1.0638 x_w + 2.15 & \text{if } x_w \geq 0 \end{cases} \tag{31}$$

$$y'_w = \begin{cases} 1.0526 y_w - 9.21 & \text{if } y_w < 175 \\ 1.0922 x_w - 14.4444 & \text{if } x_w \geq 175 \end{cases} \tag{32}$$

$$z'_w = 976 - d \tag{33}$$

B. Block Detection

The color detection algorithm correctly identified 91.67% of all blocks in the test array. Two violet blocks were misidentified as having a color of "none", meaning their centroid's mean hue did not fall within the defined violet color range. The root mean square error (RMSE) and standard deviation for the X, Y, Z, and θ components of the centroid pose are reported in Table II.

C. Teach and Repeat

The RX200 arm successfully completed four cycles of swapping the two blocks' locations via an intermediate pose using the trajectory we taught it. The individual joint angles for one complete swapping cycle are shown

TABLE II
ROOT MEAN SQUARE ERROR AND STANDARD DEVIATION FOR CENTROID COORDINATES AND BLOCK ORIENTATION AS ESTIMATED BY BLOCK DETECTION ALGORITHM.

	RMSE	Std. Dev.
X (mm)	5.6778	4.7860
Y (mm)	5.5783	5.4004
Z (mm)	2.8431	2.4672
θ (rad)	0.0272	0.0278

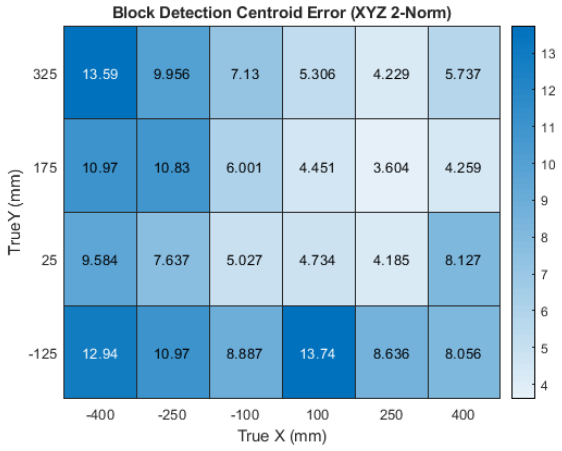

Fig. 3. Heatmap for ℓ_2 norm of XYZ centroid error (mm) on array of large blocks. Data from setup in Fig. 1.

in 4. When used as the input for our forward kinematic model, these joint trajectories generate the positional trajectory of the end-effector when it executes the block swap task (Fig. 5 in the appendix (VI)).

The final PID parameter values used for competition are listed in Table III.

D. Forward Kinematics

The forward kinematics functions developed were integrated into the display of the GUI used when operating the robot. The functions were fed the joint angle values as provided by the encoders of the joints' motors and the resulting coordinates and XYZ Euler angles of the end-effector were displayed. The accuracy of tracking our position using the encoders was assessed by comparing the outputs of the kinematics equations to known coordinates and orientations. The first test was performed by positioning the tip of the gripper at 4 known coordinates

TABLE III
FINAL PID PARAMETERS USED FOR EACH JOINT

	P	I	D
Waist	800	0	6400
Shoulder	1800	0	1000
Elbow	1500	0	3000
Wrist Angle	800	0	2400
Wrist Rotate	640	0	3600

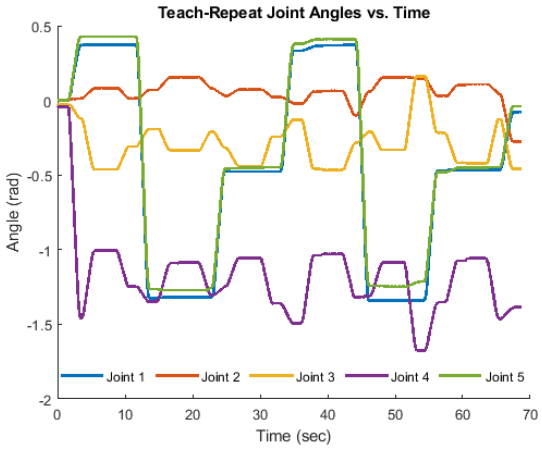

Fig. 4. Plot of angular position (rad) for RX200 base (joint 1), shoulder (joint 2), elbow (joint 3), and wrist (joints 4 and 5) across one teach-and-repeat block swap cycle.

atop 2 large blocks, measured to be 75 mm high, and angled $-\pi/2$ rad about the X-axis and $-\pi/4$ rad about the Z-axis. The expected results would have matching coordinates and orientations with the position the robot was held in with a Z value slightly higher than 75 mm since the chosen end-effector point was slightly inward (about 10 mm) from the tip of the gripper. The results are shown in Table IV.

TABLE IV
FORWARD KINEMATICS RESULTS FROM PLACING THE TIP OF THE GRIPPER ATOP 2 LARGE BLOCKS ROTATED $-\pi/2$ rad ABOUT THE X AXIS AND $-\pi/4$ rad ABOUT THE Z AXIS

	(-300,-75)	(300,-75)	(150,275)	(-150,275)
X (mm)	-300.00	301.00	156.00	-144.00
Y (mm)	-69.00	-76.00	272.00	280.00
Z (mm)	85.00	83.00	79.00	84.00
Roll (rad)	-1.57	-1.57	-1.57	-1.57
Pitch (rad)	0.02	0.02	0.01	0.02
Yaw (rad)	-0.78	-0.78	-0.86	-0.80

The second test for the forward kinematics positioned the gripper against the board rather than atop 2 large blocks. The angle of the gripper was changed to have a $-pi/2$ rad rotation about the X and Z axes. 2 known points were selected to test and the gripper was positioned on the chosen points. The desired results would have matching $X-Y$ coordinates and orientations with a Z coordinate slightly higher than 0. The results are shown in Table V.

Table VI lists the Root Mean Squared Errors (RMSE) and standard deviations of the coordinates and orientation estimates from forward kinematics from the aforementioned tests. As seen in Tables IV and V, errors in the estimate increase the further out the robot stretches from the base and lead to higher errors in the coordinates but accurate orientation estimates.

TABLE V
FORWARD KINEMATICS RESULTS FROM PLACING THE TIP OF THE GRIPPER AGAINST THE BOARD ROTATED $-\pi/2$ rad ABOUT THE X AND Z AXES

	(0,175)	(0,375)
X (mm)	5.67	16.00
Y (mm)	180.00	377.00
Z (mm)	5.00	2.00
Roll (rad)	-1.57	-1.57
Pitch (rad)	0.02	0.02
Yaw (rad)	-1.488	-1.47

TABLE VI
ROOT MEAN SQUARE ERROR AND STANDARD DEVIATION FOR POSE ESTIMATE FROM FORWARD KINEMATICS.

	RMSE	Std. Dev.
X (mm)	7.7583	5.6712
Y (mm)	4.0825	3.6697
Z (mm)	4.6547	3.1411
Roll (rad)	0.0008	0.0000
Pitch (rad)	0.0187	0.0041
Yaw (rad)	0.0617	0.0648

E. Task Automation

1) Event 1: Pick 'n Sort: In our implementation of Pick 'n Sort, we successfully completed level 2 - our robot arm was able to sort 6 blocks randomly placed across the valid workspace into a pile of large and small blocks in the fourth and third quadrants respectively. The destacking algorithm mentioned in section II-B was used here, and the topmost block was directly placed into its respective quadrant. However, we exceeded the time limit in each run with this implementation. Our incorporation of faster movement times between waypoints was ready, but we did not test this due to a lack of time. We received a score of $210/300$ for this task.

2) Event 2: Pick 'n Stack: Our algorithm to solve this task was capable of stacking 5 of 6 blocks in the level 2 configuration of this event and obtained a score of $150/300$. The main issue that we were running into was imprecise gripping actions that led to an unstable stack of blocks being formed. As more blocks were added, it became less stable and collapsed. However, unlike event 1, we were unable to incorporate the destacking algorithm here to deal with stacks in the workspace.

3) Event 3: Line 'em Up: In this event, we could achieve a successful placement of blocks in the right order for all except three blocks, a large green block in the first quadrant and a stack of red and blue small blocks in the second quadrant. The main issue that we faced with the green block was that we were unable to reliably reach the block at its distance. For the stack of small blocks, we encountered an issue of picking up both blocks and, after setting them down (destacking), detecting a single large contour and considering them to be a large violet block (as our color detection algorithm averages the contour's colors – red and blue). However, we managed to correct these issues by implementing a stretched grasp algorithm, where the $X - Y$ of the intermediate pose before grasping is reduced by a factor (0.9), but the grasping poses' $X - Y$ is increased by a factor of 1.05. We also increased the grasping pose's Z to prevent grasping multiple small blocks. We were unable to evaluate this in competition due to a lack of time and received a score of $300/400$.

4) Event 4: Stack 'em High: In this event, we faced a similar issue as event 2 in section III-E2, wherein our algorithm was unable to successfully stack blocks beyond a certain height. We noticed that our wrist joint motor was outdated with a weaker model and defective by struggling to rotate past a specific angle. This was an additional limitation causing issues when trying to stack more than 4 blocks. These issues made stacking blocks in a certain order harder as a collapsed stack would confuse our algorithm and break the order in which the blocks were to be stacked. We received no points for this task.

IV. DISCUSSION

Our camera calibration procedure was semi-automated, and could potentially be affected by imprecise clicks. This could be fixed with an automatic calibration system that uses the AprilTags' [3] data to find the extrinsic matrix. Furthermore, we could also incorporate a distortion reduction procedure into the calibration loop to reduce distortion errors by using alternate calibration methods such as grid matching.

In task automation, our event 1 algorithm was able to address the level 3 scenario, but was not fast enough to do so within the time limit. An improvement here would be to test our implementation of quick movement between waypoints. The main issue we faced in event 2 could be alleviated by a more accurate grab and place operation for precise placement of blocks onto a stack.

An improvement in event 4 would be to fix the stacking algorithm (mentioned in section III-E2) and implement the corrected stretched grasping algorithm used in event 3 (section III-E3). Apart from these, fixing the wrist joint motor and possibly using a more powerful one would help perform much better in this event.

V. CONCLUSION

In this report, we have designed an autonomous system that is capable of identifying blocks of interest and performing manipulative operations on them. We have detailed the mathematical basis behind our kinematic algorithms and have analyzed the accuracy of our system using data collected during experimentation. Our implementation was successful in achieving the desired solutions to some problems, and improvements have been proposed for the shortcomings in our methods.

REFERENCES

[1] "Rob 550 armlab," https://docs.google.com/document/d/1Mi2uipPK5PiONEMRMiNAcIw_yYsrcscGaCcnMEwnzPQ/edit, wN 2022.

[2] "Camera calibration - ros wiki," https://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration.

[3] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2011, pp. 3400–3407.

[4] "Reactorx-200 desktop robot arm," https://www.trossenrobotics.com/reactorx-200-robot-arm.aspx.

[5] R. Murray, Z. Li, and S. Sastry, *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994. [Online]. Available: https://doi.org/10.1201/9781315136370

[6] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, p. 321, jul 1961. [Online]. Available: https://doi.org/10.1145/366622.366644

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

[8] C. Cattaneo, G. Mainetti, and R. Sala, "The importance of camera calibration and distortion correction to obtain measurements with video surveillance systems," in *Journal of Physics: Conference Series*, vol. 658, no. 1. IOP Publishing, 2015, p. 012009.

VI. APPENDICES

A.

Standard deviation in the elements of the intrinsic matrix obtained across four trials:

$$K_{\text{std}} = \begin{bmatrix} 25.9690 & 0.0 & 19.4612 \\ 0.0 & 24.1282 & 105.8529 \\ 0 & 0 & 0.0 \end{bmatrix} \tag{34}$$

B.

Absolute elementwise error between H_{nominal} and H_{cal}:

$$H_{\text{abs err}} = \begin{bmatrix} 0.00003 & 0.0070 & 0.0043 & 63.0464 \\ 0.0068 & 0.0010 & 0.0450 & 14.2003 \\ 0.0046 & 0.0449 & 0.0010 & 42.4982 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \tag{35}$$

C.

TABLE VII
ℓ_2 NORM OF ERROR IN WORKSPACE $X - Y$ COORDINATES WHEN THE NUMBER OF BLOCKS ON THAT COORDINATE ARE VARIED WHEN THE CORRECTION FORMULAS (31), (32) ARE NOT APPLIED FOR COMPENSATION.

Blocks	(0, 175)	(-300, -75)	(300, -75)	(300, 325)
0	3.00	19.10	19.10	22.02
1	2.00	19.79	15.62	19.79
2	4.00	8.06	15.26	23.34
4	6.32	13.41	17.46	27.58
6	8.54	11.66	19.31	29.06

D.

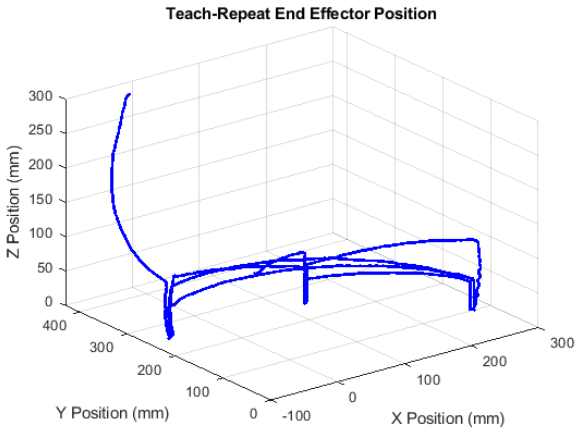

Fig. 5. Plot of 3-dimensional position of end-effector during one cycle of teach-and-repeat block swapping via intermediate position.