ROB 550 Armlab Report

Max Wu, Ashwin Saxena, Ian Stewart
{mdwu, ashwinsa, icstewar}@umich.edu

***Abstract*—Robot arms are used in many applications for their ability to perform precise, repeatable actions. In order to command a robot arm to perform these actions, it must have an understanding of its orientation and position within space, as well as an external sensing system to be able to detect objects of interest to interact with. For the purpose of this project, we programmed a system consisting of a 5 DoF arm and a depth camera to pick up, stack, and sort colored blocks. To carry out object detection, intrinsic and extrinsic calibrations of the workspace were used based off the pinhole camera model, and openCV was used to detect block color, position, and size. Forward and inverse kinematic equations were developed to understand the robot's current position as well as determining what joint angles would be needed to command the robot to move its end effector to any given new position. Combining these methods with a basic logic structure, the robot arm was able to perform in multiple events that required some level of autonomy in detecting, sorting, stacking, and placing blocks throughout its workspace. Our implementation outperformed most other groups, placing second overall in the final competition.**

I. Introduction

Many fields have adopted robotic arms, with medical institutions utilizing them to help doctors perform precise surgical operations, manufacturing operations using them to perfectly place the same objects in the same spot thousands of times every day, and space vehicles using them to adjust equipment or interface with objects of interest without requiring a human be present in a potentially deadly environment. Due to their popularity, one of the main projects of Robotics 550, a course offered at the University of Michigan, involves students utilizing a robotic arm to complete a series of challenges.

The arm in question is an Interbotix ReactorX 200 Robot Arm (Fig. 1), which has 5 degrees of freedom and terminates in a gripper that can be used to manipulate objects of up to 150 grams. Additionally, an Intel Realsense L515 LiDAR depth camera is utilized, which has a standard camera paired with a solid state LiDAR camera, enabling the distance to any object seen by the camera to be determined up to 9 meters with an error of under 14 milimeters. All physical workstation assembly was completed by the course staff.

The underlying software utilizes the open-source Robot Operating System, or ROS, implemented via the Python programming language. Most of the setup of ROS and the basic code structure was also completed by the course staff.

The purpose of this project is to take the barebones GUI that was provided and build on it over the course of roughly five weeks, at the end of which time we would need to compete in several challenges against other teams of students in the course. The main objectives are to calibrate the Realsense camera to the workspace, be able to record specific joint positions for the robot to return to at a later time, determine the end effector location within the workspace via forward kinematics, determine what joint positions would need to be in order to reach a novel position via inverse kinematics, allow manual control of the robotic arm via a simple single-click interface, and finally to allow complex automated control of the robotic arm to complete the designated challenges for the final competition.

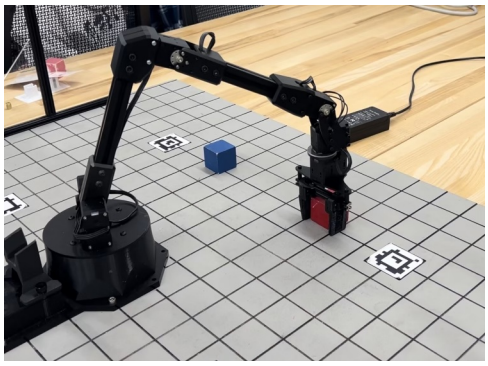

Fig. 1. RX200 Arm

II. Methodology

A. Camera Calibration

In order for the Realsense camera to be useful, it first needs to be calibrated for the workspace, enabling it to provide correct and easily usable depth information. The first step to this process is determining the intrinsic matrix of the camera - the set of mathematical transformations that represent the physical process by which the camera sensor observes a scene in the camera frame through a lens $[X_c, Y_c, Z_c]$ and then outputs a grid of pixels that represent the scene with two-dimensional coordinates $[u, v]$. The intrinsic matrix takes the form of Eq. 1, where f is the focal length of the camera

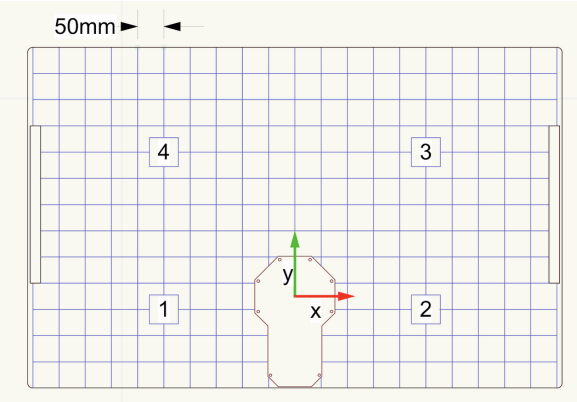

Fig. 2. Workspace coordinate frame

measured in pixels, a is the aspect ratio of the sensor, and u_0 and v_0 are the number of pixels from the start of the sensor (pixel $[0, 0]$) to the point on the sensor closest to where the pinhole of the camera is located in the x and y directions, respectively.

$$IM = \begin{bmatrix} f & 0 & u_0 & 0 \\ 0 & af & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1}$$

The camera's factory calibration settings, which included the intrinsic matrix shown below in Eq. 2, were found to be adequate for our purposes. More details on this are explained in the Discussion section.

$$IM_{factory} = \begin{bmatrix} 900.72 & 0 & 652.29 & 0 \\ 0 & 900.19 & 358.36 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2}$$

B. Workspace Reconstruction

Once the camera's intrinsic matrix was determined, the extrinsic matrix is needed to convert the camera frame coordinates into the defined world frame coordinates from Fig. 2, enabling direct mapping of an object in the workspace to the pixels in the final image at which it will appear and vice-versa. The extrinsic matrix is defined as the mathematical representation of taking objects in one coordinate system in 3D space and considering its position relative to the lens of a camera viewing them.

The extrinsic matrix is expressed as a 4x4 matrix in $SO(3)$ where R and T are the rotation matrix and translation vector from the world frame to the camera frame, respectively.

$$EM = \begin{bmatrix} R & | & T \\ 0 & | & 1 \end{bmatrix} \tag{3}$$

Initially, the extrinsic matrix was determined by taking manual measurements using a tape measure and a gyroscope from a smartphone.

$$EM = \begin{bmatrix} 0.999 & -0.028 & 0.0056 & 4.89 \\ -0.026 & -0.977 & 0.214 & 187.88 \\ 0 & -0.214 & -0.977 & 1044.38 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

In order to get a more accurate measurement of the extrinsic matrix of the camera, as well as to allow the camera to be easily recalibrated should its position relative to the workspace change, an automatic calibration routine was developed that is controlled by the press of a button in the GUI. This was accomplished utilizing four AprilTags placed at known locations on the board, which are detected by the `apriltag_ros` package. We then utilized the `cv2.solvePnP` function to determine the extrinsic matrix that, when paired with our predetermined intrinsic matrix, would result in the tags being seen by the camera at the positions the camera had observed them at.

The resultant extrinsic matrix was generally extremely accurate within the x and y axes, with at most around ±4 millimeters of error on the surface of the board. However, the z value varied greatly over the flat board, giving around ±20 millimeters of error. It was determined that this was likely due to the difference in physical location between the lens of the RGB camera responsible for the image that was detecting the AprilTags and the "lens" of the LiDAR depth camera responsible for determining how far away things are from the camera. This was manually corrected by taking the pixel locations of the four corners of the workspace in the RGB camera and the depth camera and using the `cv2.warpPerspective` function to align them before calibrating, which improved the accuracy in the z direction by about 50%.

$$EM = \begin{bmatrix} -0.999 & 0.028 & -0.028 & 3.702 \\ 0.0334 & 0.980 & -0.195 & -150.02 \\ 0.0223 & -0.196 & -0.981 & -936.29 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

To minimize the error even further, it was observed that the remaining disparity was highly correlated with the y coordinate of the point, with points higher up in the workspace's y dimension having very accurate z measurements and points with lower y values having roughly proportionally lower z values. As such, the maximum difference in z between points at the top and bottom of the workspace was measured and a manual correction factor that was based off of the y coordinate of the point was added, which brought the error down to around ±4 millimeters over the surface of the board.

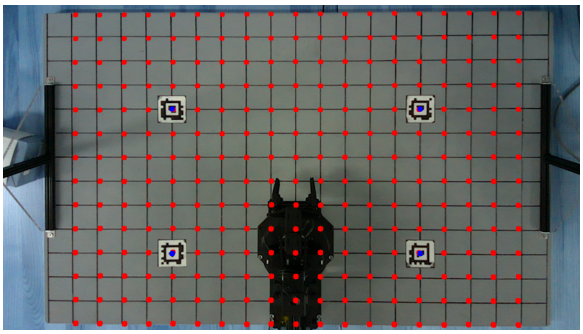

Fig. 3. Grid points in world frame projected onto transformed image

Once this has been completed, the camera's intrinsic and extrinsic matrices were used to perform a perspective transform on the image, known as a homography transform, to essentially zoom the image displayed to the user through the GUI in on only the area of interest and make it appear as if the camera has a perfect top-down perspective on the workspace. To accomplish this, the `cv2.findHomography` function was used to generate a matrix that mapped the location of the detected AprilTags with IDs 1 through 4 to the pixel coordinates $(375, 550)$, $(925, 550)$, $(925, 230)$, and $(375, 230)$, respectively.

$$HM = \begin{bmatrix} 1.1476 & -0.1469 & -79.8910 \\ 0.0359 & 1.0722 & -90.0849 \\ 0 & -0.0002 & 1 \end{bmatrix} \tag{6}$$

Our two calibration matrices and our homography matrix were validated by projecting a grid of points defined by the world coordinates of the actual grid line intersections within the workspace into the camera image on the GUI, as shown in Fig. 3. As can be seen, these line up well, and so we know that to take image coordinates $[u, v, d]$ and determine corresponding world coordinates $[x, y, z]$, we simply reverse the process, multiplying our image coordinates by the inverse of the homography matrix, then by the inverse of the extrinsic matrix, and finally by the inverse of the extrinsic matrix.

C. Forward Kinematics

In order to fully define the end effector (EE) position (Eq.8) of the RX200 arm from the joint angles in configuration space $\mathbf{q} = [q_1, q_2, q_3, q_4, q_5]$, where each q represents an angle of a different joint on the arm, a series of forward kinematic equations need to be developed. The position of the EE can be represented with six parameters, spatial coordinates in 3D space $[x, y, z]$ and three Euler angles $[\phi_z, \theta_y, \psi_z]$ in the ZYZ configuration to show orientation. The Denavit-Hartenberg (DH) parameters uniquely define each joint

TABLE I
RX200 ARM DENAVIT-HARTENBERG PARAMETERS

	L_i	α_i	d_i	θ_i
J_1	0.0	$\pi/2$	103.9	$\theta_1 - \pi/2$
J_2	206.2	π	0	$\theta_2 + \pi/2 + 0.245$
J_3	200.0	0	0	$\theta_3 - 1.326$
J_4	0.0	$\pi/2$	0	$\theta_4 + \pi/2$
J_5	0.0	0.0	174.2	θ_5

and link through four parameters $[\theta_i, d_i, a_i, \alpha_i]$, where θ_i is the joint angle, d_i is the joint offset, a_i is the link length, α_i is the link twist. Table 1 represents the RX200 Arm in its DH parameters for each joint J_i. Each row is a homogeneous transformation matrix that represents the new joint from the previous joint as shown in Eq. 7. Fig. 4 shows the DH frame of every joint.

$$A_i = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} c_{\alpha_i} & s_{\theta_i} s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i} c_{\alpha_i} & -c_{\theta_i} s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}$$

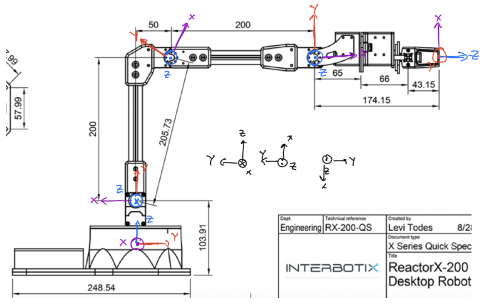

Fig. 4. Schematic of the arm highlighting DH frames at each joint

The complete forward kinematics (FK) equations representing the EE pose from the world coordinate system can be obtained by multiplying the homogeneous matrix of each A_i as shown in Eq. 8.

$$H = A_1(q_1)...A_n(q_n) \tag{8}$$

The FK equations were then validated by reporting the EE pose on the control station GUI and manually moving the RX200 to known coordinates of the board and determining the accuracy. Running the experiment on multiple coordinates, the resultant accuracy of the FK is around 3mm. Taking into account the variation of the camera calibration accuracy, the FK accuracy was deemed to be within the acceptable range.

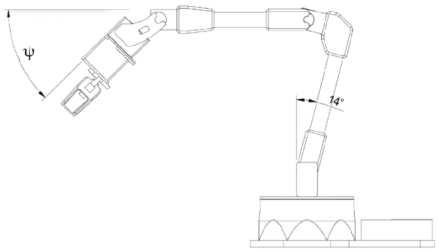

Fig. 5. ψ, the angle of the wrist with respect to the horizontal axis

However, one issue to note that affects the accuracy between the joint angles and the EE pose is gravity. Due to motor backlash and possibly loose joints, there is a lot of end play in the arm, causing up to a nearly 45mm difference between intended EE position and actual EE position. This is a nonlinear phenomenon as the effect of gravity causes a Z-offset that varies depending how far the arm is out from its base. The effect of gravity on the arm's position is discussed in the Results section.

D. Inverse Kinematics

With Forward kinematics defined, a set of inverse kinematic (IK) equations is now needed to translate a desired EE pose into a set of joint angles for the RX200. With IK, there can be multiple configurations that the RX200 arm can be in for a single EE pose (elbow up, elbow down, reverse elbow up, reverse elbow down, etc.), but not all positions are necessarily physically valid (for example, some may involve rotating the joints farther than they can physically be rotated). The following derivation of the IK equations will solve a set of joint angles for each configuration, with basic logic assigning which valid configuration has the highest "priority" when traversing the playing field.

The general process of deriving the IK equations begins with determining the wrist pose of the arm from the EE pose. Eq. 9 obtains the wrist pose in X, Y, Z coordinates using the length of the wrist L_3, and the rotation matrix from the base frame to the EE frame R_{05}. The rotation matrix R_{05} can be obtained from the set of Euler angles of the EE pose from the FK equations and then transformed into a rotation matrix using a SciPy spatial transform function, `from_euler`. The Euler angle of the EE pose is affected by the angle of rotation of the base joint θ_1 and ψ, a user-defined angle representing the wrist joint angle with respect to the absolute horizontal, as shown in Fig. 5.

$$O_{WC} = [X, Y, Z]' - L_3 R_{05}[0, 0, 1]' \tag{9}$$

With the wrist pose determined, the RX200 can then be simplified into a 2 DoF arm where the elbow joint, θ_2, and the shoulder joint, θ_1, can be geometrically determined using the law of cosines (Fig. 6).

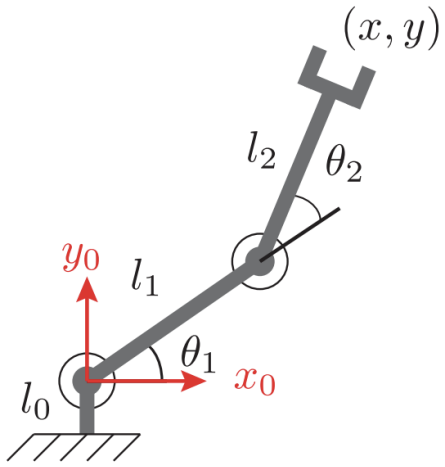

Fig. 6. 2 DoF Arm to solve for the elbow and shoulder joint angles

To generate a closed-form solution for this arm, the elbow joint θ_2 must be first solved (Eq. 10). There can be two θ_2 for a given wrist pose which will be denoted as "elbow up" or "elbow down" configuration. The elbow up configuration will have a set of $theta$ that has the elbow sticking up (opposite to Fig. 6), while the elbow down configuration has the elbow pointing down (as in Fig. 6). Eq. 10 solves for the angles in the down configuration. To obtain the angle for the up configuration, just multiply the angle by -1.

$$\theta_2 = cos^{-1}(((x^2 + y^2) - l_1^2 - l_2^2)/(2l_1 l_2)) \tag{10}$$

Once the elbow angle has been determined, the shoulder angle in the down configuration can then be calculated as denoted in Eq. 11. To obtain the angle in the up configuration, subtract this angle from π.

$$\theta_1 = atan2(y, x) - atan2(l_2 sin\theta_2, l_1 + l_2 cos\theta_2) \tag{11}$$

Solving for both θ_1 and θ_2 will determine the position of the wrist, but not the orientation. To determine orientation, the wrist angle θ_3 and the base angle θ_0 can be solved as shown in Eq. 12 and Eq. 13.

$$\theta_0 = -atan2(x, y) \tag{12}$$

$$\theta_3 = -\psi - (\theta_1 + \theta_2) \tag{13}$$

These angles then need to be translated into an angle command for each servo motor as opposed to an angle measured from the horizontal and normal of the arm length. Referring back to Fig. 4 as the $[0, 0, 0, 0, 0]$ position in terms of joint angle, the θ angles can be

manipulated to result in the required form through Equations 15 and 16 (all angles are in the down configuration, and Eq. 14 accounts for the slight angle offset between the shoulder and elbow link). Note that the only angles that need to be manipulated are the shoulder and elbow angles, as the wrist and base angles are with respect to the horizontal plane.

$$ELBOWANGLE = atan2(50, 200) \tag{14}$$

$$\theta_{shoulder} = \pi/2 - \theta_1 - ELBOWANGLE \tag{15}$$

$$\theta_{elbow} = \pi/2 + \theta_2 - ELBOWANGLE \tag{16}$$

All of these equations were done in a MATLAB Script with 3D visualization to validate the equations before implementing them in the code base. A 3D plot of the arm pose can be found in Appendix A. The final output of the IK equations can then be lumped into a 1x5 array $[\theta_{base}, \theta_{shoulder}, \theta_{elbow}, \theta_{wristangle}, \theta_{wristrotate}]$ where $\theta_{wristrotate}$ will be discussed in the Path Planning section.

E. Path Planning

With the FK and IK equations derived, the logic for path planning using click to grab and place can be developed. The main motion plan for click to grab/place consists of four waypoints, which are automatically generated when the user clicks on a given location in the GUI. The first waypoint is to position the arm right above the position of interest. The second waypoint is lowering the arm until the grabber is at the height of the position of interest, where an object may be grabbed or released. The third waypoint is a small offset in X, Y, Z from the previous position to prevent moving other blocks nearby. Finally, the fourth waypoint is moving the arm into a neutral position suspended in the air. This basic motion plan is a general framework that prioritizes collision avoidance with other objects over speed. Due to how generally-applicable this plan is, it can be utilized for the different autonomous modes required for our class competition.

Before these waypoints can be executed, we must first ensure that the motion that has been generated is in fact physically possible. To that end, there is first a distance check that takes in the clicked waypoint in the base frame and compares it to the maximum reach of the arm $L_1 + L_2 + L_3 = 580mm$. If the euclidean distance of the waypoint exceeds the reach of the arm, it will prompt the user to click on another waypoint instead of calculating the IK equations for a set of joint angles.

If the waypoint passes the initial distance check, it will then proceed to calculate the IK equations given a user defined ψ angle. If a manual ψ angle is not defined, the arm will default set ψ to 90 degrees if the clicked waypoint is within the wrist pose distance, otherwise ψ is set to 0 degrees. The IK function in python generates a 4x5 array where each row indicates a set of joint angles based on each arm configuration (elbow up, elbow down, reverse elbow up, reverse elbow down). Due to the nature of the tasks the arm will need to perform, the elbow up configuration is the first prioritized configuration to prevent any collision with preexisting stacked blocks. The reverse elbow up configuration is then prioritized next, with the elbow down configurations following behind. The wrist rotate angle is a static rotation that is the negative of the base angle to ensure that the gripper will grab blocks by its faces when oriented along the coordinate frame.

This process is repeated for each waypoint in the motion plan; however, with waypoint two, manual Z offsets need to be added and tuned to help combat the effect of gravity depending on what block size the arm is grabbing. The final waypoint's "neutral" position is manually determined by taking the previous base angle and setting the other joint angles to raise the arm until it is very nearly completely vertical. This prevents any possible collision before it moves to its next clicked position.

F. Block Detection

In order to have true autonomy, a block detection algorithm was necessary to identify blocks based on size and color and locate them in the world frame. The OpenCV library was used to create the block detection algorithm. Upper and lower HSV ranges were tuned for each of the six colors and pixels were masked for each color. The masks were tuned by saving them and comparing with the actual image to see which pixels were being filtered. Fig. BLANK shows an example of what the mask for yellow pixels might look like.

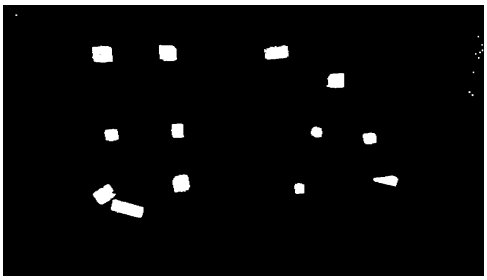

Fig. 7. Tuned yellow mask has white pixels wherever it detects pixels within the yellow range

Once the mask was created, the cv2 `findContours` function was used to find contours for block shapes. Bounding boxes were then calculated using the

`cv2.minAreaRect` function, which returned a bounding rectangle by virtue of providing its center, height, width, and rotation off the positive x-axis.

To limit false positive detection and enhance robustness of the detector, we calculated the area of each rectangle to check whether it was within a reasonable area range for a block. There was also a problem with blocks sometimes being detected as multiple superimposed rectangles. To prevent this, the centers of each block of the same color were checked against each other to ensure they weren't representing the same block.

The AprilTags on the workspace were also being detected as false positives, primarily as blue or purple blocks. Tuning the blue and purple ranges did not help very much, so the depth measurement from the camera was used to eliminate any detected contours that were less than 10mm in height.

G. State Machine

The state machine was used extensively for accomplishing various tasks throughout the lab. Different modes were made to accomplish different tasks and each mode is going to be discussed briefly below:

1) **Teach and Repeat**
   The teach and repeat mode on the state machine is designed to allow the user to manually adjust the joint angles of the robot arm and then save those joint positions as a series of waypoints, which can then be executed by the robot arm to play the pre-recorded motions back.
   a) Save Waypoint: The save waypoint state reads the joint angles for all five of the motors and saves it as an array along with the state of the gripper.
   b) Execute: The execute state sequentially sets the waypoint values as the joint angles of motors with a predefined wait period between each saved waypoint. If the user wants to run the sequence again, they simply have to click the execute button again.
2) **Stop Motion**
   The stop motion state was created to interrupt execute and other event modes mid-execution.
3) **Mouse Mode**
   The mouse mode state is used for saving user mouse coordinates as destination points for the pick and place mode.
4) **Pick And Place**
   As discussed in the path planning section, this mode is used to allow the user to pick up and place objects using only the mouse cursor.
5) **Pick and Sort**
   This mode uses the block detection algorithm to detect blocks and sort them based on their size. To account for stacked blocks in the initial environment, once the robot finishes the first pass on sorting, it re-scans the board for any "new" blocks and executes the sorting process again until it does not detect any block in the positive half-plane of the board.
6) **Pick and Stack**
   This mode first utilizes the block detection algorithm to determine the color and size of blocks in the positive half plane. It then places all of the detected small blocks away from the positive half plane, while stacking all the large blocks first. Once the large blocks are stacked three high, it stacks using the remaining small blocks.
7) **Line Up**
   This mode first unstacks all blocks from the workspace, staging all large blocks into the left half plane and all small blocks into the right half plane in locations that are easily reachable by the robot arm. It then scans the workstation for six staged large blocks, one of each color, and six staged small blocks, one of each color. Once this has been confirmed, the robot will place the blocks of each size into lines in rainbow color order, and then push the edges of each line towards the center to create a neat arrangement.

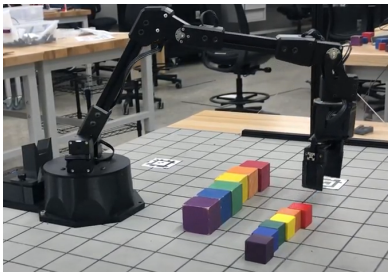

Fig. 8. Desired final configuration for the Line Up state

8) **Stack High**
   This mode works very similarly to Line Up, but instead of placing the blocks into predefined positions to create a horizontal line, it uses the pick-and-place logic to create vertical stacks of each size in rainbow order.

The non-specific states such as pick and place and execute were designed to be as general and modular as possible, which enabled the competition-specific states (Pick and Sort, Line Up, Stack High, and Pick and Stack) to be easily configured for their specific purpose by building on top of these general states.

III. Results

A. Block Detect

To verify the accuracy of the block detector, about 60 blocks of different shapes, sizes and colors were staged on the board and the algorithm was ran on the blocks in different lighting conditions. The contours of each block were re-projected back on the image as can be seen in Fig. 9. The ROYGBV blocks have a 100% accuracy in the world frame and the only non-detected blocks were the black and pink blocks, which were not to be used in the final competition and therefore were intentionally not detected.

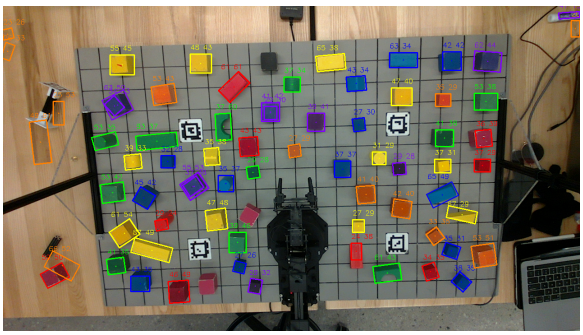

Fig. 9. Blocks outlined by their identified color superimposed on the ground truth image

B. Gravity Offset

TABLE II
Gravity Effect at Multiple Positions

Joint Angle $[\theta_{shld}, \theta_{elb}, \theta_{wrst}]$	**Expd. Z** *(mm)*	**Msrd. Z** *(mm)*	ΔZ *(%)*
[0, 0, 0]	303.9	274.7	-9.6
[0, 0, $-\pi/2$**]**	129.8	115.2	-11.3
[0.6, 0.7, 0]	272.8	229.9	-15.8
[-0.8, -0.8, $-\pi/2$**]**	108.2	106.3	-1.8

To determine the effect that gravity has on the arm, an experiment was run where the arm was set to multiple joint positions ranging between far from the base and close to the base, and the expected Z and measured Z was recorded. Table 2 shows the recorded positions using only the shoulder, elbow, and wrist angle, keeping the base and wrist rotate constant.

To accommodate for this, blocks were prioritized to be placed closer to the base rather than farther out, and a static Z offset added was added to have the gripper grab blocks on the board consistently without over or undershooting. When stacking, we found that adding an additional Z offset, proportional to the height of the block, helped enable automated stacking at significantly higher heights than possible without this additional offset.

C. State Machine

The Teach and Repeat state was the least sophisticated state of the state machine. Since it simply used preset waypoints, when motions were repeated over time the placement of the blocks wasn't perfect and error accumulated until eventually the robot lost control of the blocks it was moving. In one test where the robot "juggled" two blocks back and forth across the board, the robot managed to do about 7 cycles before the robot would begin to miss picking up the block. The Fig. 10 shows the values of the joint angles for any given cycle of this test, but because there was some inherent inaccuracy in the actual resulting location of the EE when given those static commands, in addition to some randomness in how the block ends up landing on the workspace when dropped, repeating this exact same profile would eventually fail.

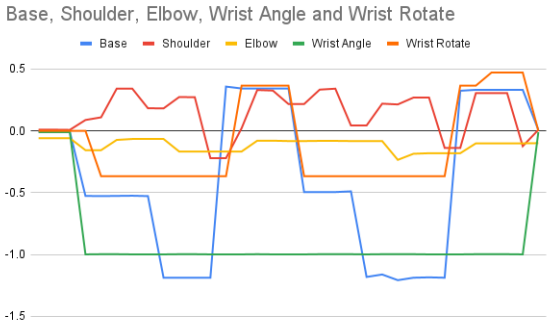

Fig. 10. Values of each joint angle while executing a cycle of exchanging blocks

That is not to say that Teach and Repeat was useless, however - in fact, it proved invaluable for the Line Up mode, where the pick and place logic could be used to autonomously grab the blocks from locations that varied from run to run while Teach and Repeat waypoints could be used to guarantee that blocks would always be placed in the exact same spot, within error. This gave us a highly repeatable set of lines and allowed us to teach the robot to push the ends together to straighten out the lines and close the gaps between the blocks as they had been placed on the board.

IV. Discussion

A. *Camera Calibration*

In order to experimentally determine the values of the camera's intrinsic matrix, the `camera_calibration`

package included in ROS was utilized. This requires holding a checkerboard pattern of a known size at a variety of different positions within the camera frame. The program records the position of the points between all of these boxes (Fig. 11) and then allows the camera to be calibrated and the intrinsic matrix to be determined based on the recorded data.

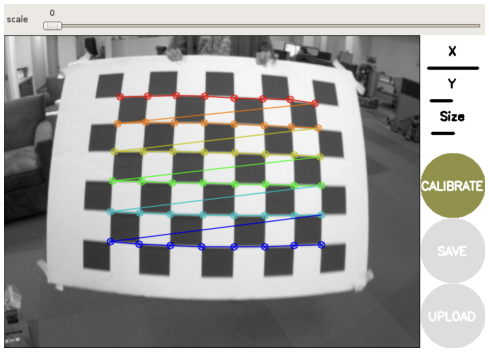

Fig. 11. Calibration GUI in ROS

Shown below are the matrix found by averaging several runs of the `camera_calibration` package (IM_{avg}), and a matrix ($\Delta\%$) showing the percentage differences between the values in the factory-supplied matrix (Eq. 2) and the experimentally-determined matrix. As can be seen in Eq. 18, the factory calibration was not off by more than 5% in any element from the experimentally-determined matrix.

$$IM_{avg} = \begin{bmatrix} 919.66 & 0 & 621.88 & 0 \\ 0 & 931.66 & 340.61 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (17)$$

$$\Delta\% = \begin{bmatrix} -2.10 & 0 & 4.66 & 0 \\ 0 & -3.50 & 4.95 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (18)$$

Although the difference between the two matrices is very small, we decided to use the factory intrinsic matrix, as the experimentally-determined matrix had a larger run-to-run variance than we trusted. Lens distortions and uncontrollable sample sizes might be doing more harm than good, and gathering more samples per calibration attempt would only lead to significant increases in processing time to determine an intrinsic matrix.

B. *Gravity Offset*

Gravity had a significant impact on the performance of the IK during the events. As observed in Table 2, if the arm was far out from the base when placing or reaching a block, there is at least a 10-15% reduction in the expected Z. This relationship is also not linear, so static Z offsets can not completely eliminate this disturbance.

One possible way to eliminate this would be to create a manual calibration curve that applies the correct Z offset at various points throughout the workspace - however, this would require extensive testing to create. An alternative solution would be to include some sort of feedback loop, but we didn't have accurate enough sensing of the arm's true position to enable this.

C. *Block Detection*

While the block detector is really robust in identifying rectangular blocks based on the color, it performs poorly with cylinders, wedges, and arches. Due to the limitation of the the `cv2.minAreaRect` function, only rectangles could be identified. This wasn't a huge concern since distractor blocks were only used for the hardest level of two of the events. If there was more time, a better block detection algorithm could have been implemented which could factor in non-rectangular shapes. By detecting non-rectangular blocks, the robot could have attempted Level 3 for the Line Up and Stack High events.

Furthermore, there were some edge cases where our block detector did not perform perfectly, such as when a small block was placed on top of a large block. Our system would detect both blocks, and, because the center of each would be reported as being at the same height, we needed to manually ensure that small blocks were moved first in our competition algorithms. Given more time, it would have been better to adjust our block detection algorithm so a small block on top of a large block would cause the large block underneath not to be detected as grabbable until the top block had been moved.

V. Conclusion

A combination of a fairly reliable calibration scheme, a highly repeatable pick and place algorithm, and a very accurate block detection algorithm allowed our system to perform very well in the competition. It completed all of the events, netting the maximum possible score in two of them, and placed second among the eighteen teams in terms of points.

References

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

[2] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

VI. Appendices

Appendix

A. MATLAB End Effector 3D Plot

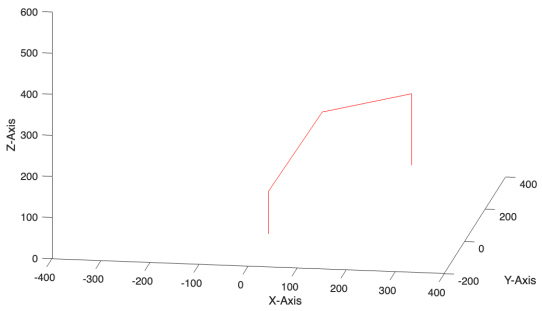

Fig. 12. 3D plot in MATLAB used to validate IK equations