ROB 550 Armlab Group 5 Report

Ryan Aspenleiter, Andrew Chen, Andy Qin
{ryanasp, chendrew, andyqin}@umich.edu

I. Introduction

The Armlab is a project revolved around a 5-Degree of Freedom (DOF) robotic arm, the RX200, and an RGB camera and solid-state LIDAR sensor, the Real Sensor L515. The goal of this project is to enable the robotic arm to manipulate objects by using computer vision, kinematics, and path planning. This lab can be broken down into three key learning points:

Acting - Foward Kinematics (FK) and Inverse Kinematics (IK)

Sensing - 3D image calibration and object detection using color and depth information

Reasoning - Path planning and state machine design

By employing these techniques, the robot was able to detect blocks and classify them on color, size, and orientation. The robot was then able to manipulate the blocks by sorting and stacking them based on the tasks.

II. Methodology

A. Basic Motion and Camera Calibration

To begin, we implemented a "teach-and-repeat" functionality. In this mode, the robot was placed into a no-torque mode and was then put into a series of positions where we would record the joint angles at each position. The robot could then play back those positions. To test this, we used this functionality to teach the process of swapping blocks at locations (-100, 225) and (100, 225) through an intermediate location at (250,75). We were able to continue this cycle 10+ times.

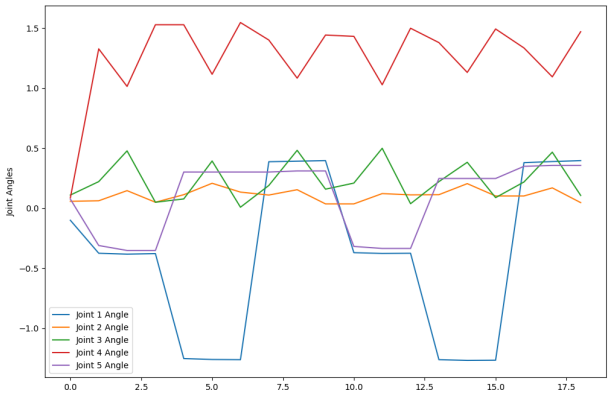

Fig. 1. Joint angles over time for one cycle of teach-and-repeat

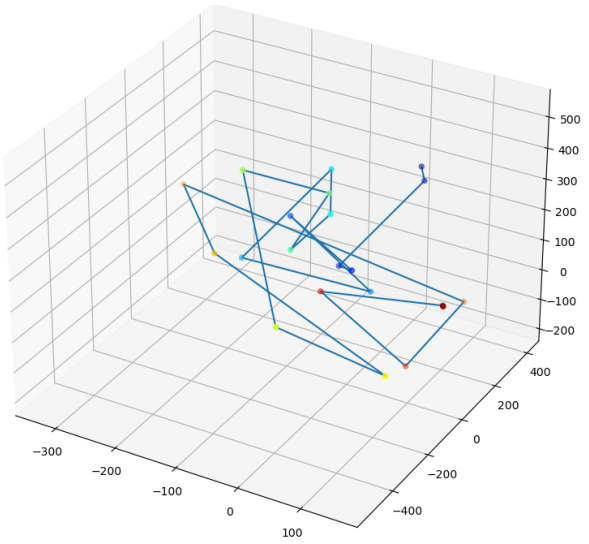

Fig. 2. End effector position over time for one cycle of teach-and-repeat

We then moved on to finding intrinsic and extrinsic matrices of the camera. We used a checkerboard and the ROS camera calibration package to measure the intrinsic matrix five times in order to find an average intrinsic calibration and compare it to the factory intrinsic calibration.

$$K_{calculated} = \begin{pmatrix} 1051.721 & 0.0 & 656.594 \\ 0.0 & 1058.565 & 349.314 \\ 0.0 & 0.0 & 1.0 \end{pmatrix} \quad (1)$$

$$K_{factory} = \begin{pmatrix} 911.962 & 0.0 & 658.318 \\ 0.0 & 912.346 & 357.746 \\ 0.0 & 0.0 & 1.0 \end{pmatrix} \quad (2)$$

From inspection, the measured and factory matrices look to be very similar with slight differences due to sources of error. Some of these errors include lighting, measurement noise, image distortion, and hardware distortion. The biggest source of error comes from limited views of the checkerboard since the calibration was calculated based on a certain limited number of positions of the checkerboard. We trust our own calibration over the factory calibration since it was calibrated with the exact setup that we used for this lab. There may be several factors that have changed since the factory calibrated it including hardware distortion and lighting.

To find a baseline estimate for the extrinsic matrix, we physically measured the distance of the camera from the origin of the world axis. We also assumed perfect 180-degree angle rotations in the z and y-axes to keep the values clean.

$$Rt_{nominal} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 350 \\ 0 & 0 & -1 & 1000 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

B. April Tags

In order to get a more accurate measurement of the extrinsic matrix, we can use Apriltags on the board and their known locations to calibrate the camera. We start off by making sure the camera can recognize and label the Apriltags by drawing a green dot in the center and a blue box around the perimeter of each detected Apriltag. These drawings act as a sanity check to make sure the detections are accurate. Even with Apriltags that we could move around, we were able to see how accurate the drawings were as long as the entire Apriltag was in the field of view of the camera. Using these Apriltag detections, we were able to create an auto-calibrate functionality that calculates the extrinsic matrix so that we are able to get world coordinate locations based on pixel locations and also apply a projective transform to change the perspective of the camera to look directly above the board instead of from an angle.

To find the extrinsic matrix, we first have a list of model points, which are known 3D points in the world coordinate system of the centers and corners of each Apriltag. We then use the detections of the Apriltags to get image points, which are corresponding 2D points in the pixel coordinate of the centers and corners of the Apriltags. With a list of 3D world points and their respective pixel locations, OpenCV has a function called cv2.solvePnP() that takes model points, image points, and the intrinsic matrix as input and outputs the correct extrinsic matrix. With both the intrinsic and extrinsic matrices calculated, these equations transform pixel coordinates into world coordinates:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = Z \cdot \mathrm{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \mathrm{Rt}^{-1} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{5}$$

Equation (4) converts pixel locations to a camera coordinate system using the intrinsic matrix (K) and depth information from the LIDAR sensor (Z). The camera coordinate system is then converted to world coordinate system using the extrinsic matrix (Rt). This implementation can now tell us the world coordinates of any location on the screen based on the pixel location.

To change the perspective of the camera, there is a simple projective transform equation.

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = H_{\text{warp}} \begin{bmatrix} u_0 \\ v_0 \\ 1 \end{bmatrix} \tag{6}$$

In this equation, H_{warp}, is a transformation matrix that maps the original pixels to a new location, which changes the perspective of the camera. In order to find H_{warp}, we get a list of known pixel locations (u_0 and v_0) and create a corresponding list of desired pixel locations (u' and v'). A very useful function in OpenCV is cv2.findHomography that finds H_{warp} given the known pixel locations and desired pixel locations. In this specific case, the u_0 and v_0 are Apriltag detection locations and the u' and v' are hard-coded to positions that move the Apriltags to certain locations on the screen so that our workspace is centered. H_{warp} is a 3x3 matrix. The top left 2x2 matrix is the rotation matrix and the first two entries of the last column form the translation vector. The last row of the homography matrix impacts the skew and ensures that the matrix is homogeneous.

It is important to note when to use (u_0 and v_0) and when to use (u' and v'). When given u_0 and v_0 locations to draw on the screen, we must convert them to u' and v' before using OpenCV so it is drawn in the correct new perspective. However, when trying to find world coordinates given u' and v' screen locations, we must convert them to u_0 and v_0 using H_{warp}^{-1}. This is because equation (4) is based on using the original pixel coordinate system.

Using the Apriltags to find the extrinsic matrix rather than using the nominal extrinsic matrix is important since it can be calculated precisely even when slight external variables are changed. In addition, it provides a more precise measurement of the world coordinate which is important when the goal is to be able to manipulate small objects. To verify the accuracy of the auto-calibration, we found 20 known world coordinate points and tested to see if the correct world coordinates are outputted when the mouse cursor moves over the known points on the screen. The outputted values consistently gave accurate points with a tolerance of +/-10 mm. In addition, we created grid point projections in order to qualitatively see the accuracy of our calibration.

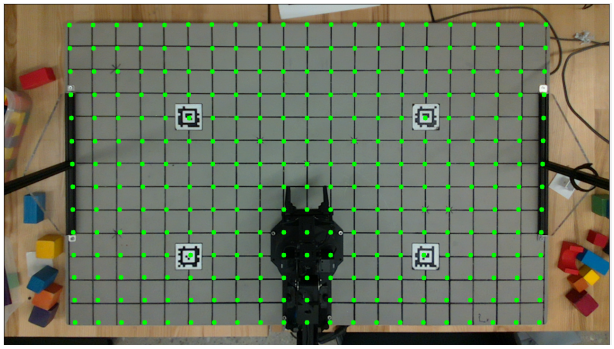

Fig. 3. Grid point projections

From Figure 3, we can see that the calibration is relatively accurate. However, the accuracy decreases the further we move away from the center. Lens distortion appears to be the primary source of error, although noise and lighting conditions may have also contributed to that error.

C. Kinematics

1) Forward Kinematics: Forward Kinematics was implemented to derive the homogeneous transformation matrix that contained the world position and orientation of the end effector with each of the five joint angles as input parameters. Specifically, a Denavit-Hartenberg (DH) table was derived to represent the five links of the arm, where:

θ: rotation along z-axis
d: translation along z-axis
a: translation along x-axis
α: rotation along x-axis
θ_i: rotation of each joint

TABLE I
DH TABLE

Link Number	θ (rad) Joint Angle	d (mm) Joint Offset	a (mm) Link Length	α (rad) Link Twist
1	θ_1	0	0	0
2	0	103.91	0	$-\beta - \theta_2$
3	0	205.73	0	$\beta - \pi/2 - \theta_3$
4	0	200	0	$-\theta_4$
5	$-\theta_5$	154.15	0	0

To deal with the segment where the arm is 90-degree bent, we merged the two links into one link with a length of the hypotenuse and an extra link twist of β, where $\beta = \text{atan2}(50, 200)$.

To verify our Forward Kinematics function, we first extracted the end effector information from the transformation matrix, namely the x, y, z, and ϕ, θ, ψ where the orientation uses the common ZYZ Euler angle convention.

$$H = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_{14} \\ r_{21} & r_{22} & r_{23} & t_{24} \\ r_{31} & r_{32} & r_{33} & t_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (7)$$

where

$$x = t_{14}; y = t_{24}; z = t_{34}$$
$$r_{33} = \cos\theta$$
$$r_{13} = \cos\phi\sin\theta; r_{23} = \sin\phi\sin\theta$$
$$r_{31} = -\sin\theta\cos\psi; r_{32} = \sin\theta\sin\psi$$

Thus, we dealt with the two conditions of $\sin\theta = \sqrt{1 - r_{33}^2}$

If $\sin\theta > 0$:

$$\theta = \text{atan2}(\sqrt{1 - r_{33}^2}, r_{33})$$
$$\phi = \text{atan2}(r_{23}, r_{13})$$
$$\psi = \text{atan2}(r_{32}, -r_{31})$$

If $\sin\theta < 0$:

$$\theta = \text{atan2}(-\sqrt{1 - r_{33}^2}, r_{33})$$
$$\phi = \text{atan2}(-r_{23}, -r_{13})$$
$$\psi = \text{atan2}(-r_{32}, r_{31})$$

Then we manually drive the end effector to four set positions, record the derived coordinates, and compared with the ground truth. We achieved great accuracy overall. The larger error at last was because the position was too far away so that it could only be reached with a certain orientation, thus height was restricted.

TABLE II
GROUND TRUTH VS RECORDED FK RESULTS (x, y, z)

Ground Truth	Recorded Results	Avg. Error %
(0, 175, 100)	(-0.54, 176.48, 100.48)	0.66%
(-300, -75, 25)	(-291.99, -75.05, 26.06)	0.50%
(300, -75, 50)	(290.7, -75.67, 51.42)	1.68%
(300, 325, 10)	(287.75, 315.54,13.16)	12.86%

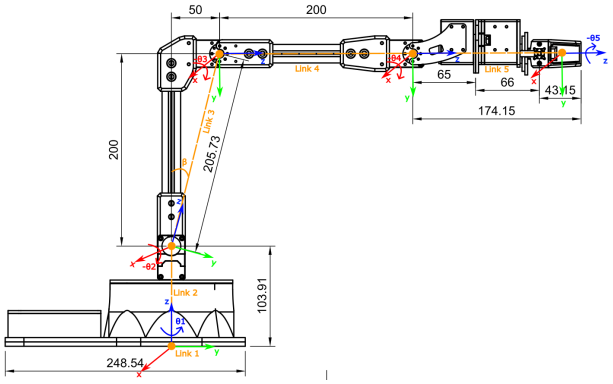

Fig. 4. Forward Kinematics Schematics

2) Inverse Kinematics: Inverse Kinematics was implemented to derive the five joint angles from a given position and orientation of the gripper in the world frame to actuate the arm. It was essential when performing tasks like following a trajectory and picking up blocks at a certain location.

The 1st and the fifth joint angles have a direct relationship with ϕ and ψ of the Euler angles that can be derived after subtracting an offset angle. θ, however, represents the sum of the rest of the joint angles and needs to be dissected.

$$\theta_1 = \phi - \pi/2$$
$$\theta_5 = -\psi - \pi/2$$
$$\theta_{234} = \theta - \pi/2$$

After getting rid of θ_1 and θ_5, we transform the arm into a 2D plane with the origin at the 1st joint. The arm is now a 3-link RRR problem. The end effector now has a coordinate of (x_e, y_e), where:

$$x_e = \sqrt{x^2 + y^2}$$
$$y_e = z - 103.91$$
$$\theta_e = -\theta_{234}$$

Now we decouple the problem by finding the twist center (Joint 4). This gives us l_c which will form a triangle with l_2 and l_3 to help solve θ_2 and θ_3. We will come back to θ_4 later.

$$x_c = x_e - l_4 \cos\theta_e$$
$$y_c = y_e - l_4 \sin\theta_e$$
$$l_c = \sqrt{x_c^2 + y_c^2}$$

Next we use the law of cosine to derive the angles between l_2 and l_3, and l_3 and l_c, both of which have a direction relationship with θ_2 and θ_3. Specifically:

$$\theta_3 = \cos^{-1}\left(\frac{l_2^2 + l_3^2 - l_c^2}{2 l_2 l_3}\right) - \beta - \pi/2$$
$$\theta_2 = \delta - \gamma - \theta_3$$
$$\theta_4 = \theta_e - \theta_2 - \theta_3$$

where:

$$\delta = \text{atan2}(y_c, x_c)$$
$$\gamma = \cos^{-1}\left(\frac{l_c^2 + l_3^2 - l_2^2}{2 l_c l_3}\right)$$

Recall that $\beta = \text{atan2}(50, 200)$. Now all five joint angles have been derived. The Inverse Kinematics function was verified in "Click and Grab" and in competition when keeping the gripper vertically downward to fetch blocks.

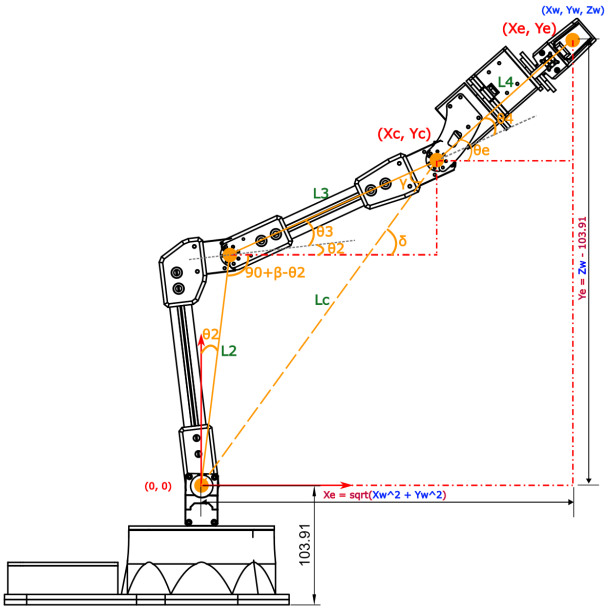

Fig. 5. Inverse Kinematics Schematics

D. Movement Planning

For the arm to successfully grab and place blocks in a variety of configurations, a general movement plan had to be created. Depending on the position given to the arm, there are two possibilities. If the given position is close enough to the arm's base, the arm can approach the position from above and grab the position in a way that is easier to manage. If the position is instead too far to approach from above, the arm will instead approach the position from the side. This option allows the arm to reach a wider range of positions but risks failure in grabbing those positions due to the angle in which it approaches.

If the arm is provided an orientation to grab the given position at (such as with information about a block detection), it orients its wrist to align with the given orientation. This can only be done if the arm can approach from above, as approaching from the side requires the wrist to be aligned with the rest of the arm.

To test the movement plan, we implemented "Click and Grab" functionality into our arm. In summary, if a pixel position is clicked on in the control station, the arm will attempt to grab or release at that position, depending on the number of previous clicks.

E. Block Detection

All of the competition events in Armlab involved the use of 25mm and 40mm cubes. These cubes could be either red, orange, yellow, green, blue, or purple. To succeed in the competition, we implemented a detector using our workstation's RGB camera that could detect

and distinguish these blocks.

First, the detector converts the camera's RGB image to the HSV scheme. We found that when compared to RGB, our colors are easier to separate in HSV since they are mostly dependent on hue. From there, the detector thresholded the HSV image for each of the different colors we wanted to detect using our found hue, saturation, and value numbers. The detector then applies a series of morphological filters to the images. We found that erosion and dilation with a 5x5 kernel followed by erosion and dilation with a 7x7 kernel is effective for both removing noise and cleaning up edges of blocks. After that, each contour can be found via cv2.findContours().

Lastly, we enforce a series of conditions to our contours to remove false positives. Using cv2.minAreaRect(), the detector calculates the minimum area rectangle for each contour to approximate size, shape, and position. Detections are then discarded if any of the following conditions apply:

- the rectangle's center falls outside the approximate range of the grey gameboard
- the rectangle's area is too small or too large to be a block
- the ratio of the rectangle's longer side to its shorter side is too large to be a square block

If a detection passes our criteria, it is classified as either small or large depending on whether its minimum rectangle's area is larger than our threshold. The detector then generates a new detection object with information on the detection's attributes and adds it to a global list of detections. The list is cleared and updated after each active frame.

We verified our block detector's accuracy by placing many color blocks in a predefined configuration spanning most of the workspace and saw how the detection output compared to the real configuration. The colors of the blocks at each space was determined with a random number generator in order to make the evaluation as objective as possible.

As seen in Figures 6 and 7, the block detector correctly determines the color, size, and positions of all of the blocks in the test. However, it is important to note that the block detector is not currently designed to filter out distractor objects by default. To handle situations where distractor objects are present, we needed to find the expected positions of each block manually and add an additional condition to filter out detections based on distance from these expected positions. For a closer look at the detection results, consult Figure 8 in the Appendix.

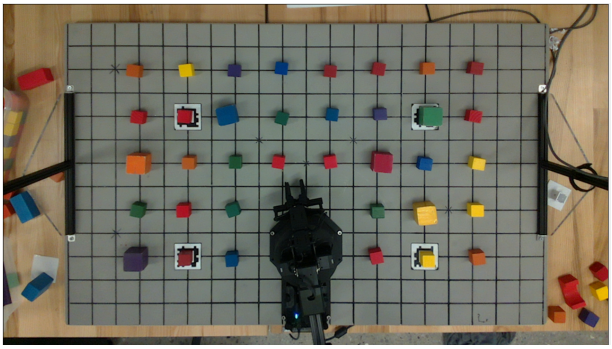

Fig. 6. Block Detector Evaluation Configuration

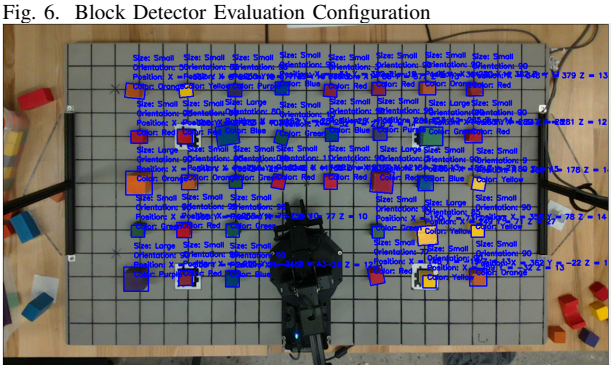

Fig. 7. Block Detector Evaluation Configuration with Detection Labels

F. Competition

The overall performance of our arm was tested with a series of four competition events. It is important to note that each event could be completed at one of three levels, with tasks from higher levels being more complex than the tasks from lower levels.

1) Event 1: Pick 'n sort!: In this event, the arm is tasked with sorting blocks based on size by placing small blocks to the left of the arm in the negative half-plane and placing large blocks to the right of the arm in the negative half-plane. At level three, the level in which our arm competed, there are nine blocks to sort of random sizes and colors with some blocks possibly stacked two high. 30 points are gained for each block picked and dropped correctly and (10*level) points are gained for completing the task in 180 seconds.

To compete for this event, the arm stayed in its "task" state so long as there was at least one block detected in the positive half-plane. While in this state, the arm would grab the block that is closest to the arm and place it on the side corresponding to its detected size. The arm kept track of how many blocks of each size it sorted and placed each new block such that it would not be on top of an existing block. See Algorithm 1 in Appendix for more detailed pseudocode.

2) Event 2: Pick 'n stack!: In this event, the arm needs to stack blocks three high. At level three, the level in which our arm competed, there are six blocks

(three small and three large) of random colors with some blocks possibly stacked two high. 30 points are gained for each block picked and stacked and (10*level) points are gained for completing the task in 120 seconds.

To compete for this event, the arm stayed in its "task" state so long as it recorded that less than six blocks had been stacked. Initially and after three blocks had been stacked, the arm would choose a new position to begin a stack on. The candidate positions were on top of Apriltags to ensure that no blocks were already on them at the time of the event starting. The arm would then grab a block that is at least 70 mm away from either stack position, prioritizing large blocks, and place it on the stack position. See Algorithm 2 in Appendix for more detailed pseudocode.

3) Event 3: Line 'em up!: In this event, the arm must line up all blocks of the same size in lines in rainbow color order. At level three, the level in which our arm competed, there is one small block and one large block of each color (12 blocks in total) as well as a set of distractor objects the arm needs to avoid. 30 points are gained for each block in the correct place and order. However, only 10 points are gained for each block that is out of order or is more than three centimeters from the line but is still clearly "lined up". In addition, the lines of blocks have to be shorter than 30 cm long. 20 bonus points are given for the neatness of each line and 100 bonus points are given for completing the task with the presence of distractor objects. This task must be completed in 600 seconds or less.

The logic for this event was similar to the logic from the first event. The primary difference is that each block would be placed in a different, predetermined location depending on its detected size and color. See Algorithm 3 in Appendix for more detailed pseudocode.

4) Event 4: Stack 'em high!: In this event, the arm has to stack all blocks of the same size in stacks in rainbow color order. At level one, the level in which our arm competed, there is only one small block of each color (six blocks in total). The first block is worth 10 points and each additional block is worth (n*10) points, where n is the number of blocks in the stack at the time the additional block is added. 40 point are awarded for each completed stack in the correct order. This task must be completed in 600 seconds or less.

To compete for this event, the arm stayed in its "task" state so long there were either less than six small blocks or six large blocks stacked. The arm notes the closest block it can detect and then determines if there is a block that matches the color corresponding to the next position on either stack, prioritizing the small blocks. If such a block is found, it determines if it can grab the block from above, moving it to a predefined "bad" location if it cannot. This "bad" location allows the arm to more easily grab the block in future iterations and potentially reveal the next color blocks for the stacks. If the arm can grab the block from above, it will move it to either the small block stack position or large block stack position. Otherwise, if no such block exists, the arm moves the closest block to a predefined "bad" location so it becomes easier to grab in future iterations and potentially reveal the next color blocks for the stacks. See Algorithm 4 in Appendix for more detailed pseudocode.

III. Results

For the competition, we scored 1036 points in total with perfect scores for Events 1 and 2, and a near-perfect score for Event 3. Event 1 was relatively simple, and we were able to get a perfect score easily. For Event 2, we were mainly worried about stacking blocks on top of a small block since balancing was inconsistent for us. Fortunately, all of the stacked blocks were organized enough to not topple over. For Event 3, we lost 4 points due to imperfect line qualities. We were able to get the blocks to line up in the correct order, but the spacing between each block varied and some of the blocks had slight rotations in them. We largely skipped Event 4 due to time constraints, so we have no code for the detection and manipulation of the arch and semi-circle blocks. We placed 4th overall in the competition for our lab time.

IV. Discussion

As seen from the results, the arm accomplished Events 1 and 2 at the highest difficulty level perfectly. The state machine logic for these events did not require especially precise movement, so our arm's performance was sufficient to complete them. Event 3 required precise placement of blocks into lines to receive the maximum score. Although our arm placed blocks in the correct color order, it left a small amount of space between the blocks in lines as a result of imperfect placement. A perfect score for Event 3 could be achieved with additional experimentation with the choice of intrinsic matrix (factory vs manual calibration) and slower arm movement. Although our arm performed well in competition, there are a number of ways in which it could be improved upon in the future.

To be competitive for Event 4, we would need to improve the block detector so that it could detect and grab semi-circle and arch blocks. Both of these types of blocks share the same colors as the standard blocks, so no changes would be needed with our HSV numbers. Unlike standard blocks, however, the shapes of new block types appear to be different depending on their orientation. Our existing block detector's logic would not be able to meaningfully categorize and filter non-rectangular shapes. The introduction of template matching to our block detection process might prove useful

when attempting to detect these shapes. The current block detection also does not use the depth camera in any form, so introducing depth as an additional filter could dramatically simplify other elements of the block detector.

A general issue for our arm came from inconsistencies in terms of where it would grab and place blocks. Some of this is due to the inherent give of the motors in the arm, but some errors may come from our choice to use the factory intrinsic matrix over our manually calibrated intrinsic matrix. The speeds at which the arm operated may have also contributed to issues in block placement. Generally slower speeds could improve the precision of where the arm moves, but experimentation is needed to determine how much speed we could afford to lose while completing tasks under the given time limits.

There are several particular block configurations that are especially difficult for the arm to handle. If a small block is started on top of a large block of the same color, the block detector will only detect the large block. Because depth information is not used, the contours of the small block would become effectively invisible to our detector.

In addition, special logic would need to be introduced to recognize an arch block or semi-circle block in an orientation such that they look like rectangular blocks from the camera's view. Depth information could again be useful, but another approach to differentiating blocks like this could be to have the arm manually reorient the block so information about its other faces could be ascertained by the block detector.

Although the block detector can successfully detect blocks stacked very high up, the arm would struggle to grab those blocks. The logic currently used to determine whether a given position is reachable would mark such a location as reachable from above. Depending on the height of the position, the higher position to approach from might not itself be reachable. This would cause the arm's movement to fail.

V. Conclusion

Most of the objectives of this lab were achieved. We successfully controlled the arm to complete given tasks in the manual and three competition events with decent accuracy. We are also confident to complete Event 4 if given extra time.

Perception wise, the block detector was proved to be reliable in most cases. Control wise, the kinematic functions were verified. Improvements could be made in block detection, specifically color classification and orientation detection. In addition, depth sensor information could be used in the future to improve grip precision and extra logic is needed for blocks of special shapes.

VI. Appendices

Algorithm 1 Task 1 Level 3

$smallblocks = 0$
$largeblocks = 0$
while There is at least one block detected that is in the positive half-plane **do**
  $currentdetection$ = closest detection to the arm that is in the positive half-plane
  if $currentdetection.size = "Large"$ **then** ▷ We need to make sure that we always grab the highest block in a stack
    if There is at least one small block $alternatedetection$ detected above the large block **then** $currentdetection = alternatedetection$
  grabdetection($currentdetection$)
  if $currentdetection.size = Small$ **then**
    $newposition = [-150 - (smallblocks \mod 3) * 50, -50 - (50 * int(smallblocks/3)), -20]$
    $smallblocks = smallblocks + 1$
  else
    $newposition = [150 + (largeblocks \mod 3) * 50, -50 - (50 * int(largeblocks/3)), -10]$
    $largeblocks = largeblocks + 1$
  placedetection($newposition$)

Algorithm 2 Task 2 Level 3

$currentstackposition = None$
$usedcandidateposition = None$
$candidatepositions = [(250, -25), (-250, -25)]$
$blocksstacked = 0$
while $blocksstacked < 6$ **do**
  if $currentstackposition = None$ **then**
    $currentstackposition =$ a candidate position where no block is detected within 70 mm from it ▷ We prioritize stacking small blocks first
  $detection$ = a detection whose size is large, not below another block, at least 70 mm away from $currentstackposition$, at least 70 mm away from $usedcandidateposition$ if it is not None
  $detection$ = a detection whose size is small, not below another block, at least 70 mm away from $currentstackposition$, at least 70 mm away from $usedcandidateposition$ if it is not None
  grabdetection($detection$)
  placedetection($currentstackposition$)
  if $blocksstacked \mod 3 = 0$ **then**
    $usedcandidateposition = currentstackposition$
    $currentstackposition = None$

Algorithm 3 Task 3 Level 3

$colorsequence = ["Red", "Orange", "Yellow", "Green", "Blue", "Purple"]$
while There is at least one block detected that is in the positive half-plane **do**
  $currentdetection$ = closest detection to the arm that is in the positive half-plane
  if $currentdetection.size = "Large"$ **then** ▷ We need to make sure that we always grab the highest block in a stack
    if There is at least one small block $alternatedetection$ detected above the large block **then** $currentdetection = alternatedetection$
  grabdetection($currentdetection$)
  $index = colorsequence$.index($currentdetection.color$)
  if $currentdetection.size == Small$ **then** $newposition = [-125 - index * 29, -50, -20]$
  else$newposition = [125 + index * 44, -50, -10]$
  placedetection($newposition$)

Algorithm 4 Task 4 Level 1

$targetsmall = 0$
$targetlarge = 0$
$baddetection = 0$
$smallstack = (-250, 25)$
$largestack = (250, 25)$
$colorsequence = [\text{"}Red\text{"}, \text{"}Orange\text{"}, \text{"}Yellow\text{"}, \text{"}Green\text{"}, \text{"}Blue\text{"}, \text{"}Purple\text{"}]$
while $targetsmall < 6 \vee targetlarge < 6$ **do**
    $closestdetection =$ detection closest to arm
    $currentdetection = detection$ where $(detection.size = \text{"}Large\text{"} \wedge colorsequence.index(detection.color) = targetlarge)$
    $currentdetection = detection$ where $(detection.size = \text{"}Small\text{"} \wedge colorsequence.index(detection.color) = targetsmall$
    if $currentdetection \neq None$ **then**
        if cangrabfromabove($currentdetection$) **then**
            grabdetection($currentdetection$)
            if $currentdetection.size = \text{"}Small$ **then**
                placedetection($smallstack$)
                $targetsmall = targetsmall + 1$
            else
                placedetection($largestack$)
                $targetlarge = targetlarge + 1$
        else
            grabdetection($currentdetection$)
            $newposition = [100 + baddetection * 45, 75 - (50 * int(self.bad_detection/3)), -10]$
            $baddetection = baddetection + 1$
            placedetection($newposition$)
    else
        grabdetection($currentdetection$)
        $newposition = [100 + baddetection * 45, 75 - (50 * int(self.bad_detection/3)), -10]$
        $baddetection = baddetection + 1$
        placedetection($newposition$)

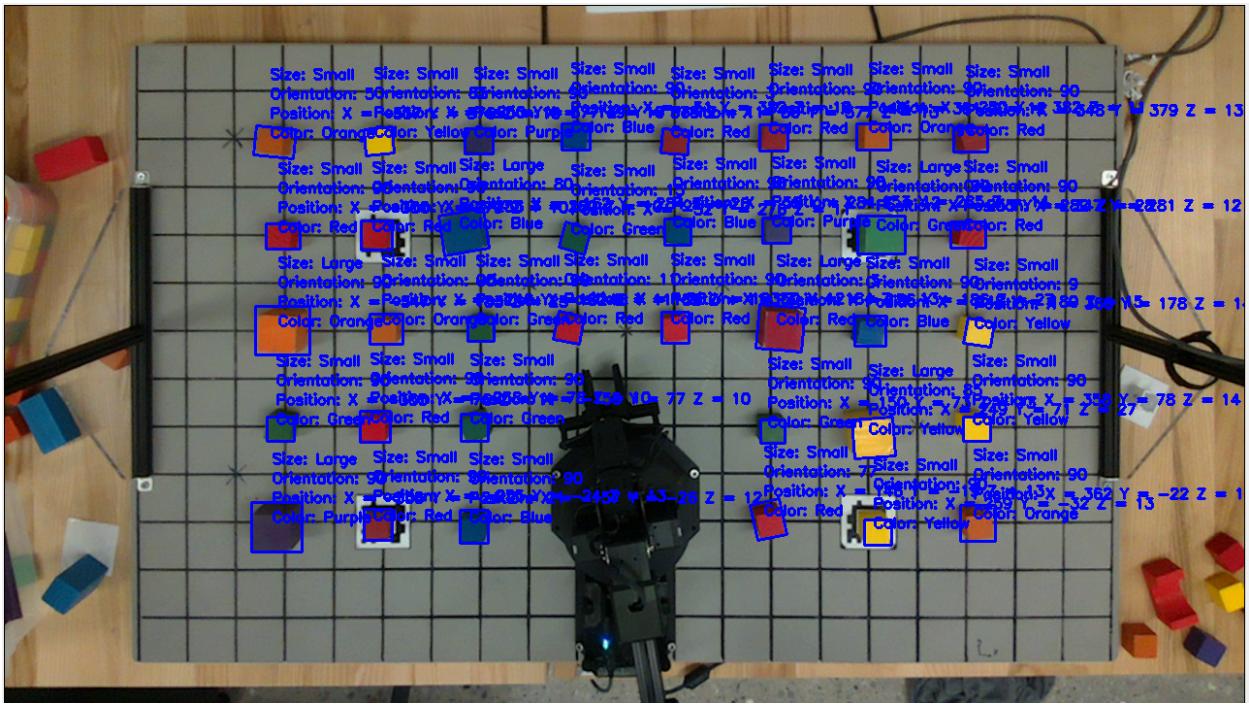

Fig. 8. Enlarged Block Detector Evaluation Configuration with Detection Labels