ROB 550 Armlab Report

Parker Bradshaw, Daniel Cherenson, Jinjia Guo
{psbradsh, dmrc, jinjaguo}@umich.edu

***Abstract*—This report details the Armlab project in ROB 550 in which we implemented a range of algorithms on the ReactorX-200 (RX-200) Robot Arm to perform tasks in a class-wide competition. The algorithms include teach-and-repeat, camera calibration, forward- and inverse-kinematics, computer vision, and state machines for each competition task. We also designed a basketball launcher mechanism that uses the RX-200 arm to load and shoot a basketball towards varying sizes of hoops.**

I. Introduction

In this project, we focus on building autonomy for a 5-DOF robotic arm, shown in Figure 1, with three main components: acting, sensing, and reasoning. In the acting module, we implement forward and inverse kinematics modeling of the manipulator and develop a grasping strategy. For sensing we utilize depth camera sensors and perform 3D image calibration to gather spatial data. Additionally, object detection is implemented using OpenCV [1]. The reasoning component of the project involves path planning as well as the design of state machines to manage the robot's decision-making.

Fig. 1. ReactorX-200 5-DOF Arm [2]

II. Basic Motion and Camera Calibration

A. Teach and Repeat

We began development with an algorithm to teach the robot how to move along a prescribed path. To teach arm waypoints the user physically moves the robot arm to a desired position and saves the set of joint positions in a list via a button in the graphical user interface (GUI). The user can also specify if the gripper should open or close at each location. Once the user finishes creating waypoints, the robot executes the trajectory by sending joint position commands to the servos.

We taught the robot a trajectory for swapping two blocks at locations (-100 mm, 225 mm) and (100 mm, 225 mm) via an intermediate location at (250 mm, 75 mm), all in the world frame. This trajectory ran for up with five cycles before enough error had accumulated in picking up and placing the block, making it impossible to grab the block at the prescribed location. Figure 2 below plots joint angles over time during one cycle of this block swap trajectory.

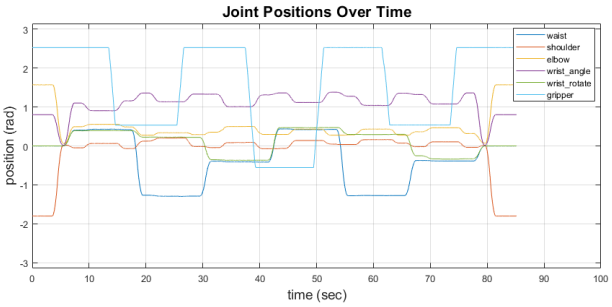

Fig. 2. Plot of individual joint angles during a repeated block swap routine

B. Intrinsic Camera Calibration

Given an object detected in an image, we use the pin-hole camera model to find its location in the workspace coordinates. In this model, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z_c} \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{c|c} I & 0 \end{array} \right] \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

In this formula, (X_c, Y_c, X_c) are the coordinates of a 3D object in the camera coordinate frame. (u, v) are the pixel coordinates of the projected point, (u_0, v_0) is the principal point at the center of the image, and (f_x, f_y) is the focal length in pixels. The intrinsic matrix (or camera

matrix) is independent of the observed scene. This matrix captures the internal characteristics of the camera—such as focal length, principal point, and skew which do not change unless the camera's internal configuration (like focal length) are modified. Once the camera is calibrated and the intrinsic matrix is estimated, it can be reused for different scenes.

The factory camera intrinsic matrix is

$$K_{factory} = \begin{bmatrix} 902.19 & 0 & 662.35 \\ 0 & 902.39 & 372.23 \\ 0 & 0 & 1 \end{bmatrix}$$

For the camera intrinsic parameter matrix K, we can use the checkerboard calibration. We perform 3 to 5 calibrations and calculate the average resulting values to obtain the final intrinsic matrix. Using the ROS *camera_calibration* program, we calibrated the camera four times and recorded the following camera intrinsic matrix as the average

$$K_{ROS} = \begin{bmatrix} 983.24 & 0 & 673.31 \\ 0 & 985.46 & 412.60 \\ 0 & 0 & 1 \end{bmatrix}$$

A visualization of the ROS *camera_calibration* is given below in Figure 3

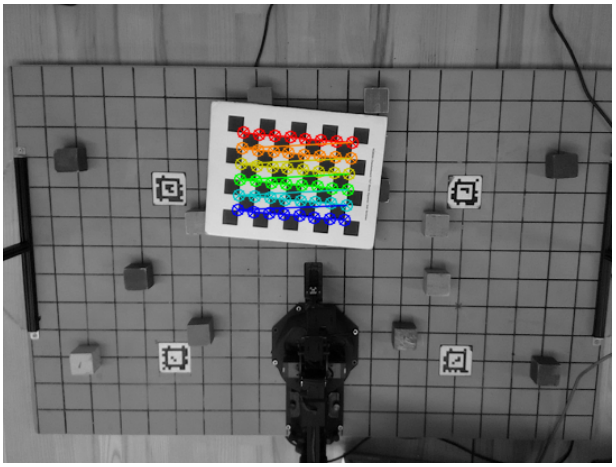

Fig. 3. Calibration by the ROS *camera_calibration* program [3]

The matrices have the same standard structure, as expected, but have slightly differing entries in f_x, f_y, u_0, and v_0. These are relatively very small differences from the stock camera calibration.

$$\sigma_{K_{ROS}} = \begin{bmatrix} 67.03 & 0 & 40.10 \\ 0 & 67.23 & 17.35 \\ 0 & 0 & 0 \end{bmatrix}$$

We decided to use the factory camera intrinsic matrix. Potential sources of error in the calibration process are inconsistent lighting or shadows in the camera field of view and motion blur as the user is moving the checkerboard pattern.

C. Manual Extrinsic Calibration

The extrinsic matrix, H_{ext}, describes how the camera is positioned and oriented relative to the world or scene. The extrinsic matrix transforms points from the world coordinate system to the camera's coordinate system by the following matrix equations

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = H \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Thus we have:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = P \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

in which

$$P = KH$$

The 3D point in the world coordinate system is first transformed to the camera coordinate system through the extrinsic matrix H_{ext}, and then transformed to the pixel coordinate on the image plane through the intrinsic matrix K.

For the camera's extrinsic matrix, we used the physical measurements of the lab setup to get a rough extrinsic matrix for the camera. The translation t was obtained by measuring the height from the camera to the designated origin of the workspace. Using the accelerometer values, obtained from the *realsense-viewer* program, we estimate the rotation angle of the camera and thus the rotation matrix R. After obtaining the translation and rotation matrix, we have the full external parameter matrix of the camera.

For the manual extrinsic calibration process, we made the following assumptions about the orientation of the camera relative to the workspace

1) X axis of the world frame is parallel to the X axis of the sensor;
2) YZ plane of the camera frame is parallel to the YZ plane of the world;
3) Camera frame is at the front surface of the sensor.

Assumptions 1 and 2 dictate that the only rotation between the world and the camera frame is about the X axis. Using the accelerometer data in the camera Y and Z frame, we computed the roll angle of the camera to be 9.4°. Since the Z axis of the camera points downwards and the Z axis of the world frame points upwards, we applied a further rotation of 180° about the X axis to arrive at a roll angle of 189.4°.

We then measured the distance from the camera surface to the workspace origin in the world coordinate frame. Since the homogeneous transform goes from the world frame to the camera frame and since the translation t happens after the rotation, the translation must be expressed in the camera frame. We applied the negative inverse rotation to the measured translation t and the resulting extrinsic homogeneous transformation matrix with rotation and translation is

$$H_{ext} = \begin{bmatrix} 1 & 0 & 0 & -9 \\ 0 & -0.987 & 0.163 & 225 \\ 0 & -0.163 & -0.987 & 1032 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

D. Automatic Extrinsic and Depth Calibration

To automatically calibrate the camera extrinsics, we used known world locations of four AprilTags and the Perspective N-Point (PNP) algorithm. PNP pose computation takes in measured image coordinates from the AprilTag detector and known world coordinates of the AprilTags, solving for the rotation and translation that minimizes the reprojection error of four AprilTags from 3D-2D point correspondences. The detection and 3D pose estimation of the AprilTags are done by the *apriltag* ROS node. With the 3D coordinates of tags in the camera frame returned by the *AprilTagDetectionArray* message, we compute the corresponding 2D coordinates of 4 tags by multiplying it with the intrinsic matrix and then undo the scaling by $\frac{1}{Z_c}$.

The 3D coordinates of 4 tags in the world frame are

1) (-250, -25, 0)
2) (250, -25, 0)
3) (250, 275, 0)
4) (-250, 275, 0)

Since the camera is not directly above the center of the board, we applied a perspective transform called homography to display the workspace in the GUI as if viewed from above. We need depth calibration because the depth frame and the RGB color frame are not perfectly aligned, there are millimeter deviations, especially at the edges of the frames. The source points for calculating the homography matrix are the Apriltag positions, whose positions in pixels are known. The implementation of the homography matrix is based on the sample code *homography_transform.py* script from the *armlab_opencv_examples* repository. In addition to the three-dimensional coordinates of the object, we also need the depth information of the object successfully grab an object in the later tasks of the lab.

After all the matrices have been calculated, we can complete the reconstruction of the workspace. First, the homography matrix is determined using the *cv2.findHomography()* function using the reference points in the image frame, and then a transformation is applied to change the perspective from "trapezoidal" to rectangular. The reference points we chose for the homography computation are (384, 504), (896, 504), (896, 216), and (384, 216), all in image coordinates, where the origin is in the top left of the screen, positive X is to the right and positive Y is down on the GUI. We chose these reference points since they centered the workspace grid on the screen and did not show much of the area outside the board. One example of the homography matrix after calibration is

$$H = \begin{bmatrix} 1.05 & -0.125 & -50.648 \\ -0.0166 & 0.931 & -29.48 \\ 0 & 0 & 1 \end{bmatrix}$$

The third row is comprised of 0, 0, and 1, which shows that the homography transformation is affine and can be represented as a skewed rotation and a translation. The translation is given by the H_{13} and H_{23} elements, and the skewed rotation is given by the 2x2 rotation matrix formed with H_{11}, H_{12}, H_{21}, and H_{22}.

Figure 4 below shows the projection of red dots onto the workspace grid points. Each red dot represents a known location in the world frame transformed through the extrinsic, intrinsic, and homography matrices to appear in the GUI window.

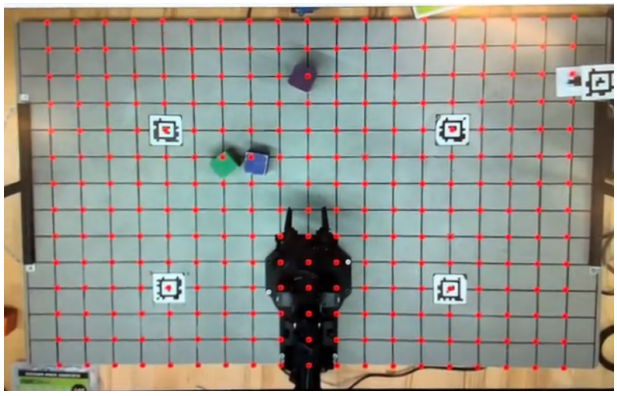

Fig. 4. Projected grid points on the workspace

By observing the error in grid point projection, we determined that the PNP method using the *SOLVEPNP_AP3P* solver flag was the optimal solution for calibration. The grid points are all within a few pixels of the real world locations, except for in the far corners of the board where a few millimeters of inaccuracy is observed. Since we would be conducting most of our block grabbing and stacking operations away from the far corners, we decided that this was within the allowable calibration error.

III. Forward and Inverse Kinematics

A. Forward Kinematics

The forward kinematics (FK) objective is to determine the position of the joints in the global frame, given the position of each joint in the robot frame. Either the Denavit-Hartenberg (DH) table or the Product of Exponentials (PoX) can be used to implement FK in that part.

The DH convention is a standard method for describing the geometric relationship between joints and links in manipulators and robot systems [4]. It is widely used to solve forward and inverse kinematics problems in robotics. It simplifies complex three-dimensional kinematic problems into a series of relatively simple transformations by representing each link with four standardized parameters. From these four parameters, a transformation matrix can be constructed where each row in the DH table represents a homogeneous transformation matrix, detailed as follows:

$$A_i = Rot_{z,\theta_i} Trans_{z,d_i} Trans_{x,a_i} Rot_{x,\alpha_i}$$

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

The complete FK equation is assembled from the individual homogeneous transformations.

$$A_1(q_1)A_2(q_2)...A_n(q_n) = \begin{bmatrix} R_n^0 & O_n^0 \\ 0 & 1 \end{bmatrix}$$

where R_n^0 represents the orientation of the end effector with respect to frame 0 and O_n^0 represents the location of the end effector in frame 0.

In this 5-DOF robot arm, the frame of each joint is defined in Figure 5 below. The frame numbers are defined such that link i connects frame $i-1$ to frame i.

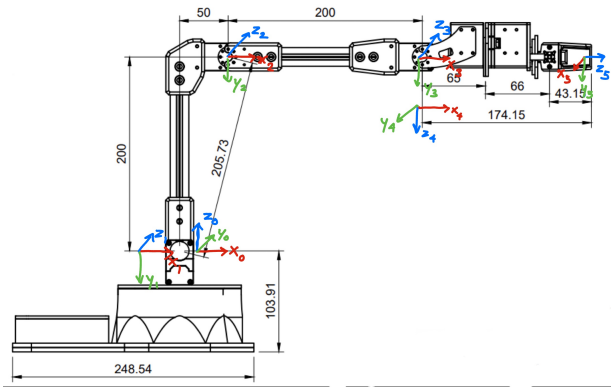

Fig. 5. Frame definitions on the RX-200

Frames 0 and 1 are coincident and frames 3 and 4 are coincident. Using the frame definitions and link parameters from the above figure, we can derive the following DH table in Table I.

TABLE I
Denavit-Hartenberg (DH) parameters

link	θ_i (rad)	d_i (mm)	a_i (mm)	α_i (rad)
1	θ_1^*	0	0	$-\frac{\pi}{2}$
2	θ_2^*	0	205.73	0
3	θ_3^*	0	200.0	0
4	θ_4^*	0	0	$-\frac{\pi}{2}$
int.	$\frac{\pi}{2}$	0	0	$\frac{\pi}{2}$
5	θ_5^*	174.15	0	0

We have used an intermediate transformation with a fixed joint angle to go from the wrist pitch to wrist roll DOFs. This would be replaced with a wrist yaw DOF in a full 6-DOF robotic arm.

We applied the following servo angle command offsets when going between D-H convention frames and the actual RX-200 servo position command conventions:

TABLE II
Servo offsets

Base θ_1^*	90°
Shoulder θ_2^*	-74.14°
Elbow θ_3^*	70°
Wrist Pitch θ_4^*	-3°
Wrist Roll θ_5^*	0°

These offsets redefine the 0° position for the servos to be in the orientation shown in Figure 5, with the exception of the base joint that has its offset to align the zero position with the x axis of the world frame.

We also added a z offset of 103.9 mm to the end effector position in the world frame to account for the distance of the base to the shoulder joint. This could have also been a parameter in the D-H table, but the end result is identical.

We verified the accuracy of our DH table by moving the end effector to various known locations and viewing the estimated end effector position in the GUI. At first, we had a few sign errors in the table based on incorrect frame definitions. After fixing them and testing, we saw that we were within millimeters of the correct location and were satisfied with the performance of the forward kinematics calculations.

B. Inverse Kinematics

Given a desired end effector position O_5^0 and orientation R_5^0, we used inverse kinematics to find the

joint angles θ^* that will achieve the desired inputs. The following steps describe the code in *kinematics.py* lines 199-257.

First we ensure that the desired position is reachable based on the maximum extension of the arm. We then decoupled the orientation of the wrist from the rest of the arm. To reduce the dimension from 3D to 2D, we computed a yaw angle of the base and rewrote the position in a 2D frame aligned with the yaw angle

$$\theta_1^* = \text{atan2}(O_{5,y}^0, O_{5,x}^0)$$

We then transformed O_5^0 into O_5^1 with the transformation matrix R_0^1, which is purely a function of θ_1^*

$$O_5^1 = R_0^1 O_5^0$$

In this representation, there are only two non-zero elements in the x and z axes, so the problem is now reduced to two dimensions. We then subtracted a vector representing the length of the wrist $l_4 = 174.15$ mm, to find the desired position of O_3^1. Then, we computed the joint positions θ_2^* and θ_3^* using the analytical solutions to 2-link revolute-revolute (RR) inverse kinematics with the following equations

$$\theta_3^* = \cos^{-1}\left(\frac{(x^2+z^2) - l_2^2 - l_3^2}{2 l_2 l_3}\right)$$

$$\theta_2^* = \text{atan2}(z, x) - \text{atan2}(l_3 \sin\theta_3^*, l_2 + l_3 \cos\theta_3^*)$$

where $l_2 = 200$ mm and $l_3 = 205.73$ mm, and x and z are the two dimensional coordinates of O_3^1. Since inverse cosine has two solutions, we take the negative solution, called "arm up", so that the arm does not crash into the table.

Then, the wrist pitch angle is just $\theta_4^* = \theta_d - \theta_3^* - \theta_2^*$ where θ_d is the desired end effector pitch angle. Finally, the wrist roll angle θ_5^* is set to 0 if the desired pitch angle is not -90°. A pitch angle of -90° indicates a singularity in the Euler angles that cannot disambiguate roll and yaw, meaning the wrist roll angle can control the yaw angle. This is useful for positioning the gripper to pick up blocks. In this case, we define $\theta_5^* = \psi_d + \theta_1^*$ where ψ_d is the desired yaw angle of the end effector.

IV. Block Detection

A. Block Detector

We implemented a block detector that detects red, orange, yellow, green, blue, and purple blocks and determines their position in the workspace. Outputs are the block's color, size (large or small), and orientation (rotation angle). The general procedure for detecting blocks' location in a depth frame is the following:

1. Extract images from the video stream.

2. Divide the ROI (region of interest) area in the image and perform mask processing as shown in Figure 6.

3. Perform graphics processing on the area, including denoising, etc.

4. Perform color segmentation based on HSV and traverse to find the outlines of all objects.

5. Determine the size based on the depth information and the size of the outline area.

6. Use top surface segmentation to only consider the top object when stacking objects.

7. Calculate the rotation angle of the object between 0-90°.

8. Store all information in the dictionary.

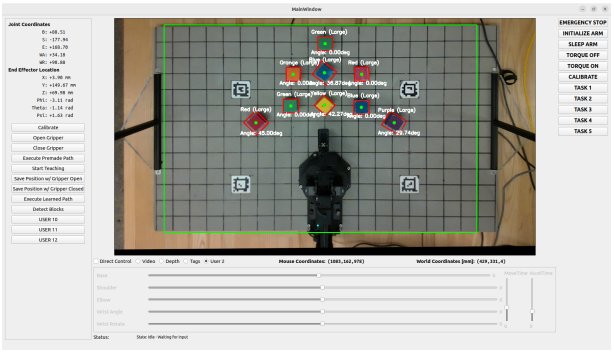

Fig. 6. ROI and final results on the board

HSV stands for Hue, Saturation, and Value. Hue is the basic attribute of color, which is what we usually call color names, such as red, yellow, etc. Saturation refers to the intensity of color. The higher the value, the more intense the color, and the lower the value, the grayer it becomes. Each value ranges from 0 to 100%. HSV describes color as a point in a cylindrical coordinate system. The center axis of the cylinder is black at the bottom, white at the top, and gray gradients in between. The angle around this axis corresponds to "hue", the distance to this axis corresponds to "saturation", and the height along this axis corresponds to "hue" or "brightness". The exact values of different HSV thresholds are shown in the table below:

TABLE III
HSV PARAMETERS OF DIFFERENT COLOR

color	HSV High Threshold	HSV Low Threshold
red	[150,100,80]	[180,255,255]
green	[50,100,60]	[90,200,255]
blue	[99,93,73]	[105,255,255]
yellow	[20,180,160]	[50,255,255]
orange	[5,120,157]	[13,209,255]
purple	[105,55,25]	[200,200,120]

In classifying block size, we use the minimum area of large blocks ($850px^2$) as the threshold to ensure that

all large blocks can be correctly classified. In addition, we ignore all areas with an area less than $100px^2$, which eliminates some false block detections. The angle of each block from 0 to 90 is returned by *cv2.minAreaRect()*.

Since blocks can be stacked, we need to be able to locate the top surface of each block/stack for picking and placing. We found that when the object is not on the top surface, the detected outer contour of the object is a wide rectangle instead of a square. In this case, we can use the aspect ratio to filter out the sides of blocks detected which have their top faces covered by another block. The result is shown below:

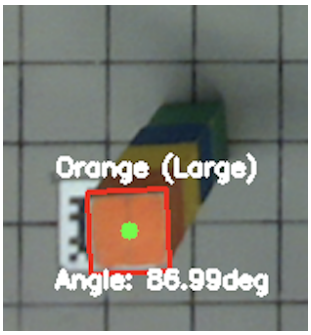

Fig. 7. The surface segmentation

We will now evaluate the effect of the entire block detector. First, we use a heat map to evaluate the position detection. As can be seen from the error color and the bar on the left side of the image in the figure below, even for the block with the largest error, the difference between its true position and the detected position is within 2 mm. We can conclude that the position detection effect under this algorithm is effective.

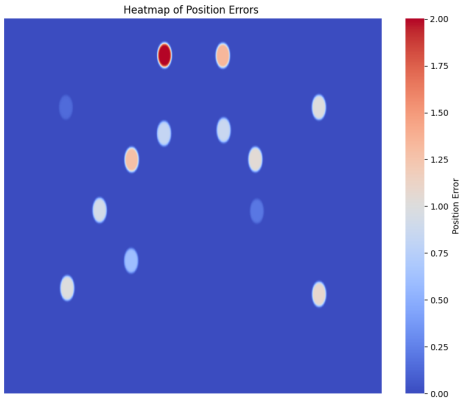

Fig. 8. The HeatMap

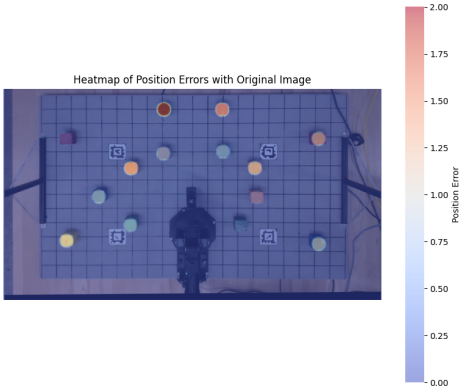

Fig. 9. Heat map with Original Image

In addition to the position, the color and size of the blocks were also evaluated. Three blocks of each type (red large, red small, green large, green small, blue large, blue small, yellow large, yellow small, orange large, orange small, purple large, purple small) were placed on the board. The confusion matrix in Figure 10 was used to evaluate the results.

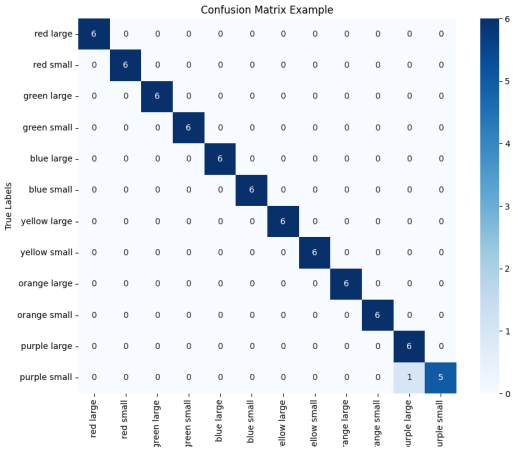

Fig. 10. Confusion Matrix of color and size detection

As can be seen from the above figure, except for a small purple block that was mistakenly detected as a large purple block, the other results were very accurate. The reason for the failure of purple block detection is that the colors of different purple blocks are not very uniform, and often vary greatly, resulting in errors in the detection results.

To improve the robustness of the detection method, we use data enhancement. For each different object, we enlarge, reduce, rotate, add noise, and flip its image.

The effect of data enhancement is shown in Figure 11 . Our implementation uses the OpenCV functions *cv2.erode*, *cv2.dilate*, and *cv2.GaussianBlur*. The user can continuously adjust HSV thresholds to adapt the result to all lighting conditions encountered.

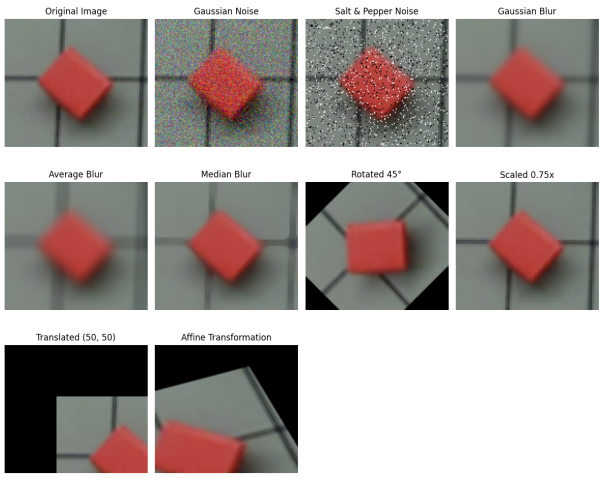

Fig. 11. Data Augmentation for Red Block

V. Logic and Final Competition

A. Click To Grab & Place

Putting together the camera calibration and inverse kinematics tasks, we added click-to-grab functionality. The user clicks a block in the GUI and the gripper moves to that spot and grabs the block. Then the user clicks a location in the GUI and the gripper releases the block at that location. At a high level, the algorithm is as follows:

1) Receive click from GUI and save the mouse click's pixel coordinates.
2) Convert the pixel coordinates to world coordinates using the inverse homography, inverse intrinsic, inverse extrinsic matrices, and depth information.
3) Calculate a pre-move pose and a post-move pose that are set distances above the desired grab pose. This ensures the arm approaches the block at a good angle.
4) Compute the joint positions for the pre-move, grab/drop, and post-move poses and execute them in order. Either close or open the gripper depending on the current gripper state.
5) When the user clicks again, repeat steps 1-4 to drop the block.

B. Competition Tasks

For events 1-3, we stored block information in a dictionary data structure of key-value pairs. The keys were a tuple of block distance from the origin and color. The values were a tuple of x-y-z location, angle, and size. We reconstructed this dictionary whenever the arm was in a position that did not obstruct any of the blocks on the board. The reason to store blocks in a dictionary is that we were able to sort them based on either distance or color, which would both be needed for the various events. This meant we only needed to design a single block detecting algorithm for all events.

1) Event 1 - Sort 'n stack: We used the block detector to discriminate between large and small blocks and to stack them in color order. We sorted the block dictionary by color and stacked the blocks in a specified location based on their size. We received the highest possible score on this level, showing that our block detector and sorting logic was working well.

2) Event 2 - Line 'em up: We used the block detector again to discriminate between large and small blocks, this time placing them in horizontal lines ordered by color. We again used the sorted dictionary to place each block in a set location based on its color, with a small offset between them. To gain maximum points, we could have used the arm to compress the lines of blocks to reduce the distance between them. Despite this, we still scored decently and were able to place the blocks in the correct order in mostly straight lines.

3) Event 3 - To the sky: We used the block detector to find blocks and stack them in a specified location. We placed a single block in the same place every time to minimize grabbing error. We achieved a 12 block tower, showing that our inverse kinematics was precise enough in the vertical dimension to place blocks accurately. We could have improved our score by altering the placing logic when at higher heights to reach the physical limit of the arm - 14 blocks. We could have gone even higher by creating a smaller stack and adding that to the larger stack.

VI. Basketball Launcher

The design lab section of Armlab requires a mechanism which launches small foam basketballs into a hoop. This mechanism must launch using stored potential energy which is recharged only using force applied by the arm. It can either be aimed by hand, using two small servos, or by the arm itself with increasing preference respectively.

The mechanism we designed is based around a catapult which stores energy in a linear spring. Figure 12 shows the launching motion of the catapult arm. A funnel allows the arm to easily drop balls into the mechanism and a tube connected to the funnel feeds the balls into the ready to fire position of the catapult arm. The catapult and funnel mechanisms are mounted through a 3d printed rolling element bearing. A 30 degree yaw around this bearing is shown in Figure. The other side

of this bearing is static and mounted to ground to rigidly locate the mechanism in a consistent position.

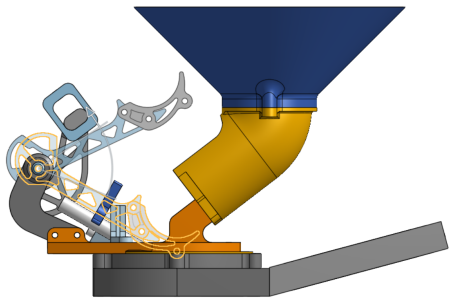

Fig. 12. Side view of launching mechanism showing motion of the catapult.

A Hold Down Release Mechanism (HDRM) holds the catapult down and allows the arm to release stored energy, launching the ball. To release the mechanism the arm simply has to lightly touch the long blue colored trigger piece shown in Fig. 13. The HDRM can be reset by moving the catapult arm down past its ready position, latching the HDRM around it. During Event 4 this was accomplished by hand, but our team also demonstrated automated reset using only the arm. A string running through a special channel, if pulled, would pull the catapult arm down past the HDRM resetting it. This string could be tied to a block allowing the arm to pull the string and arm the catapult in an automated manner, which would result in a higher score.

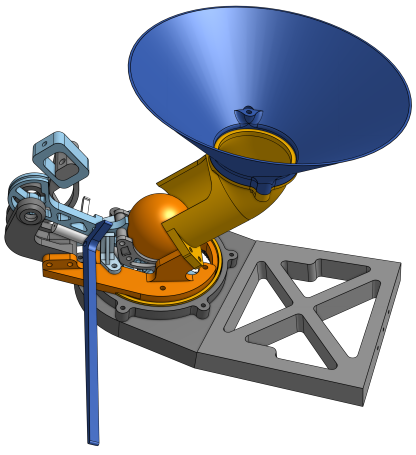

Fig. 13. Perspective view of launching mechanism showing 30 degree yaw.

REFERENCES

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

[2] Reactorx-200. [Online]. Available: https://docs.trossenrobotics.com/interbotix_xsarms_docs/specifications/rx200.html

[3] camera_calibration. [Online]. Available: https://wiki.ros.org/camera_calibration

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