ROB 550 Armlab

Rahul H Kumar, Tyler Smithline, Xiujin Liu
{rahulhk, tylergs, jeanliu}@umich.edu

***Abstract* — This paper describes an approach to using a 5 DOF robot arm to autonomously manipulate objects within a workspace. This process requires the integration of many sub-tasks, including camera calibration, computer vision, forward kinematics, inverse kinematics, and path planning. By developing methods to accomplish each of these tasks, we have programmed the robot arm such that it can identify and distinguish blocks of different sizes and colors and move them to target destinations. It can also accomplish more complicated tasks such as stacking and sorting.**

I. Introduction

Serial robotic arms have enormous potential to increase the efficiency, accuracy, and safety of tasks that are traditionally difficult for humans. In order for robotic arm systems to accomplish practical tasks within a workspace, they must combine vision and manipulation functionality. We set out to develop a math-based computational approach to program a robot arm with computer vision and manipulation capabilities.

Our system had a number of high-level goals. First, it needed to be able to determine and represent the locations of objects within the work space in a world coordinate system. Next, it needed to detect blocks of different colors, sizes, and shapes. Additionally, the system needed to determine the necessary joint angles to move the arm to the desired locations in the workspace. Finally, a state machine is required to determine when to execute specific tasks and a Graphical User Interface is necessary for intuitive control from the robot operator.

This project builds on decades of prior work in robot kinematics and computer vision. Additionally, it makes use of existing frameworks and packages such as Robot Operator System 2 (ROS2) and OpenCV.

The success of the work laid out in this paper will be evaluated by the ability of the robot to perform a set of block manipulation tasks. Several examples of the system's performance on a variety of tasks will be given.

II. Methodology

A. Computer Vision

1) Camera Calibration: In order to determine where objects are within the workspace, it is necessary to determine a mathematical relation between the image produced by the camera and the world frame. This task requires knowledge of two relations: The first between camera coordinates and image coordinates called camera intrinsic matrix, and the second between camera coordinates and world coordinates called extrinsic matrix.

The camera used in this study produces an RGB and depth value for each pixel represented by (u, v, d) coordinates within the image frame. Each (x, y, z) coordinate in the workspace corresponds with a pixel coordinate and depth. This relationship is illustrated by a diagram of the pinhole model seen in (Fig. 8) in the appendix.

The intrinsic matrix converts the coordinates in the camera frame to image frame pixel coordinates with (1).

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z_c} \begin{bmatrix} \alpha & s & u_o \\ 0 & \beta & v_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & | & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{1}$$

In (1), the α and β represent the focal length of the camera in the camera coordinate's x and y directions. The values u_o and v_o represent the offset between the camera's principal axis and the center of the image plane. The principal axis is the line perpendicular to the image plane that passes through the pinhole of the camera. Finally, s is the axis skew of the camera, which is zero for a true pinhole camera.

The intrinsic camera parameters were determined using the ROS camera calibration module described in [1]. We moved a checkerboard of known size through the workspace at different orientations, distances from the camera, and skew angles. The module extracted key points at the intersections of the checkerboard squares and the distances between them to information to determine the intrinsic parameters of the camera. We performed this calibration method five times, and calculated the average intrinsic calibration (2).

$$\begin{bmatrix} 865.99 & 0 & 636.59 \\ 0 & 870.47 & 355.65 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

The factory calibration taken using the linux command [*ros2 topic echo /camera/color/camera_info*] is (3).

$$\begin{bmatrix} 902.57 & 0 & 660.20 \\ 0 & 903.04 & 377.16 \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Since the camera's coordinate system does not line up with world coordinates, an extrinsic matrix allows us to transform world coordinates to camera coordinates and vice versa using the inverse extrinsic matrix. This relationship is given by (4) below.

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} & \mathbf{R} & & \mathbf{T} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{4}$$

The extrinsic matrix consists of a 3×3 rotation matrix (**R**) and 3×1 translation matrix (**T**) that are composed to form a 4×4 homogeneous transformation matrix.

In our study, we calculated the extrinsic method using two methods. The first method was through hand-measuring the distance and angle offsets between the center of the world frame and center of the camera frame. We measured the following offsets:

$\Delta x(mm)$	$\Delta y(mm)$	$\Delta z(mm)$	$\theta_x(^\circ)$	$\theta_y(^\circ)$	$\theta_z(^\circ)$
10	162	1040	-9	0	0

Using a standard rotation matrix about the x-axis for **R**, we produce the nominal extrinsic transformation matrix:

$$H_{nom} = \begin{bmatrix} 1 & 0 & 0 & 10.0 \\ 0 & -0.9816 & 0.1908 & 162.0 \\ 0 & -0.1908 & -0.9816 & 1040.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

The second method we used to obtain an extrinsic matrix transformation is through calibration using April Tags [2] at known locations. This method uses the ROS April Tag module to extract the image coordinates (u, v) of each April Tag in the workspace. These are then drawn on the control station GUI as seen in (Fig. 9) in the Appendix. The visualization made it apparent that the detections are quite accurate and also made it easy to validate when one of the four April Tags was not being detected due to poor lighting conditions, or if an unintended tag was being detected in the workspace.

This method also requires knowing the world coordinates (x_w, y_w, z_w) of the April Tags. Then the OpenCV function solvePnP [3] can be used by calling: [*ret, rvecs, tvecs = cv2.solvePnP(world_points, image_points, camera_matrix, distCoeffs = dists)*] The function takes the list of world points, image points, the camera intrinsic matrix, and distortion coefficients as parameters and outputs the rotation vector and translation vector. It is then trivial to convert these values into a homogeneous transformation matrix. This method is used every time the calibration routine is ran, and yields slightly different results each time. One instance of the extrinsic matrix produced is given in (6).

$$H_{cal} = \begin{bmatrix} 1.000 & -0.007 & 0.004 & 47.56 \\ -0.007 & -0.976 & 0.220 & 116.94 \\ 0.002 & -0.220 & -0.976 & 1055.61 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

2) Workspace Reconstruction: The extrinsic and intrinsic matrices defined previously can be utilized to convert pixel coordinates (u, v, d) to world coordinates (X_w, Y_w, Z_w). This relationship is given in two parts by (7), where **H** and **K** are the extrinsic and intrinsic matrices of the camera system.

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \mathbf{H}^{-1} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}, \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = Z_c(u, v)\mathbf{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{7}$$

These equations constitute the $pixel\_to\_world()$ function.

We improved the accuracy of the calibration by adding distortion coefficients to the SolvePnP function. These coefficients were obtained by running the linux terminal command [*ros2 topic echo /camera/color/camera_info*] [4]. The distortion coefficients are: $[0.168, 0.542, 0.001, 0.001, 0.522]$. The results of this change are discussed later on.

Since the camera is positioned non-parallel to the workspace, the image appears as a trapezoid. In order to fix this, we used a homography to warp the image into a rectangle. A homography matrix **H** warps each point within the image as shown in (8).

$$w \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ v_0 \\ 1 \end{bmatrix} \tag{8}$$

We multiply out these matrices in (9) and solve for u' and v' in (10) to see what each element of the homography represents.

$$\begin{aligned} wu' &= h_{11}u_0 + h_{12}v_0 + h_{13} \\ wy' &= h_{21}u_0 + h_{22}v_0 + h_{23} \\ w &= h_{31}u_0 + h_{32}v_0 + 1 \end{aligned} \tag{9}$$

$$\begin{aligned} u' &= \frac{h_{11}u_0 + h_{12}v_0 + h_{13}}{h_{31}u_0 + h_{32}v_0 + 1} \\ v' &= \frac{h_{21}u_0 + h_{22}v_0 + h_{23}}{h_{31}u_0 + h_{32}v_0 + 1} \end{aligned} \tag{10}$$

From these equations, we can see that u' and v' are both linear combinations of u_0 and v_0 where the values

of homography matrix **H** are the coefficients.

The values of the 3×3 matrix **H** are calculated using the OpenCV function *cv2.findHomography(src_pts, dest_pts)* where *src_pts* are the pixel coordinates of the April Tag centers and *dest_pts* is a set of arbitrary points that form a rectangle with the same aspect ratio as the original April Tag locations. These values were manually tuned such that the workspace grid extends nearly to the top and bottom of the image. Then the warp is applied to the image using the function *cv2.warpPerspective(image, homography)*. The results of this function can be seen with estimated grid-points projected onto the image in (Fig. 4).

3) Block Detection: The approach taken to detect blocks of different shapes, sizes and colors in this task involves usage of classical computer vision methods. First, the *cv_image* obtained from the image listener callback function and is used to obtain the current video frame as an image. A copy of this image is passed through the calibration procedures described in the previous section and then a warp perspective transform is applied on it to obtain a rectangular overhead view of the workspace. The transformed image is then converted from RGB color space to HSV. This is done as HSV color space is better for colored block detection than RGB space because it separates color information (hue) from brightness (value), making it more robust to variations in lighting conditions. At the same time, a helper program is used to set the HSV range for different colors given in the workspace. Sliders are used to determine HSV value thresholds for different colors as shown in (Fig. 1). These recorded ranges are stored by color in a dictionary for future use.

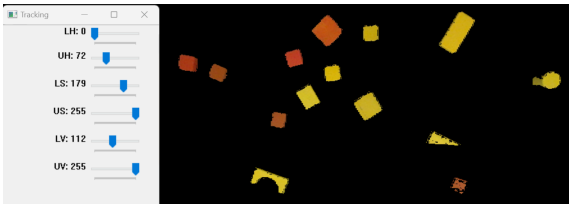

Fig. 1: HSV tracker

Color masks are generated individually using the *inRange()* function in opencv. Additionally, the pixels containing the robot and the objects in the workspace outside the board are filled with zeros so that they do not affect the detection process. Binary thresholding is applied on the masked image and the *find_contours()* function in OpenCV is used to find contours, which are essentially curves in the image that enclose a blob of a single color. This function returns a vector of points that constitute different contours and all the vectors are stored in a single matrix to allow for easy access of individual contours. The obtained contours are looped through to access individual blobs.

$$\mu_{pq} = \sum_x \sum_y x^p y^q I(x, y) \tag{11}$$

Individual contours whose area is more than the minimum area threshold are used for further calculations. The contour area is found using the image moment formula shown in 11. Setting p and q both to zero will give us the area of the region enclosed by the contours. When the area is found to be greater than the minimum threshold, the centroids of the contour are calculated by setting one of p and q to zero to get the x and y coordinates respectively. The shape of the contour is estimated using the *approxPolyDP()* function. The return value of this function is the number of sides in the detected blob, which is used to estimate and record the shape. Similarly, *minAreaRect()* function is used to obtain the corners of the shape and its orientation angle. The function returns the four corners of the smallest rectangle that encloses the contour. The euclidean distance between each set of adjacent points is calculated and the aspect ratio of the rectangle is found. The angle and shape are recorded along with the centroid of the block and returned from the block detector function. The algorithm 1 shown represents the pseudocode of the block detection methodology discussed above. The algorithm also deals with cases where blocks of different heights are stacked on top of each other by using a simple height thresholding method. This algorithm is implemented by looping multiple times from a set height to zero. The block detections are recorded only when the z value of the world coordinate is greater than the current height threshold. The height threshold is decreased iteratively until it reaches zero and breaks the loop.

B. Robot Control

1) Forward kinematics: The objective of forward kinematics (FK) is to determine the position of the end effector in the world frame given the angles of each of the joints in the configuration space. We approached the implementation of forward kinematics using the Denavit-Hartenberg (DH) method.

Based on the technical drawing for the RX200 arm in (Fig. 11) in the appendix, we know the lengths of each link of the arm. In addition, we know the directions of positive joint rotation from using the direct control feature of the provided control station interface. Combining the known robot parameters with the instantaneous joint angles from the *rxarm.get_positions()* function, we present the *FK_dh()* function that determines the position of the end effector in the world frame.

To implement the forward kinematic function, the first step is to determine the DH-parameters for the robot arm:

d, θ, a, and α. By simplifying the technical drawings into five links connected by five rotary actuators, we can assign a coordinate frame to each rigid body as seen in (Fig. 2). Applying DH conventions from [5], we can determine the DH-parameters as shown in (Table I). The parameters d_i, θ_i, a_i, and α_i can be described as distance along z_{i-1}, angle from x_{i-1} to x_i about z_{i-1}, distance along x_i, and angle from z_{i-1} to z_i about x_i respectively. Then, we use the function *rxarm.get_positions()* to get the instantaneous joint angles for each joint: θ_1, θ_2, θ_3, θ_4, θ_5, completing our DH table for a specific instance in time.

With these DH-parameters, we can find the matrix for each coordinate system transformation from (12). A total homogeneous transformation matrix is produced if we post-multiply the rotation matrix for each stage together. In addition, we need to transform from world frame to the robot base coordinates used in DH parameters. Therefore, we pre-multiply by matrix M (13), which rotates the frame about the z-axis. Combining these operations produces the homogeneous transformation matrix in (14).

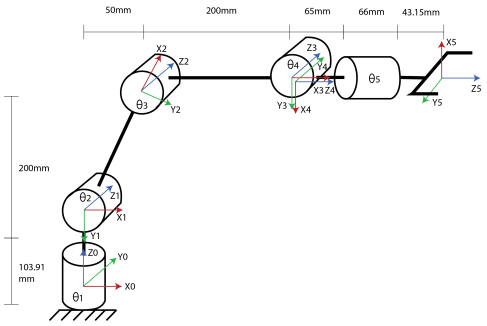

Fig. 2: Axes Used for DH Parameter Generation

joint	d	θ	a	α
1	0	$-\pi/2$	103.91	θ_1
2	205.73	0	0	$-\arctan(4)+\theta_2$
3	200	0	0	$\arctan(4)+\theta_3$
4	0	$\pi/2$	0	$\pi/2+\theta_4$
5	0	0	174.15	$\pi+\theta_5$

TABLE I: DH Parameters

$$A_i = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12)$$

$$where: s_{\theta_i} = \sin\theta_i, c_{\theta_i} = \cos\theta_i$$
$$and\ s_{\alpha_i} = \sin\alpha_i, c_{\alpha_i} = \cos\alpha_i$$

$$M = \begin{bmatrix} \cos\pi/2 & -\sin\pi/2 & 0 & 0 \\ \sin\pi/2 & \cos\pi/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (13)$$

$$H = M * A_1(q_1) * A_2(q_2) * A_3(q_3) * A_4(q_4) * A_5(q_5) \quad (14)$$

To extract position information of end effector from final homogeneous transformation matrix, we can use the relations of (15) referencing the 4×4 transformation H.

$$\begin{gathered} x = H[0][3], y = H[1][3], z = H[2][3] \\ \theta = \arccos(H[2][2]) \\ \psi = \arccos(-H[2][0]/\sin(\theta)) \\ \phi = \arcsin(H[1][2]/\sin(\theta)) \end{gathered} \quad (15)$$

2) Inverse kinematics: The objective of inverse kinematics (IK) is to determine the joint positions required to move the end effector to a specified location in the world frame. It will take a desired position and orientation of end effector and output the necessary joint angles to move the end effector to that position. The algorithm is implemented inside the *IK_geometric()* function, which takes the position and wrist angle of the end effector as parameters: $x, y, z, \psi, block\_angle$, and returns the necessary angle for each joint: θ_1, θ_2, θ_3, θ_4, θ_5. Since we want the gripper to grasp and place the blocks vertically, we always set ψ equal to $\pi/2$, and ϕ is determined from the orientation of the blocks.

To be more specific, we start by finding base angle θ_1 seen in (Fig. 10) in the appendix, which is calculated using the x and y coordinates of the end effector using (16). For θ_2, θ_3, θ_4, we can treat the arm as a 3-link RRR arm like the one in (Fig. 3), which allows us to produce the set of equations (17) with known parameters: l_1, l_2, l_3 (link lengths), and x, y, z, θ (position and orientation of the end effector). Finally, we use a numerical solver method (scipy.optimize.fsolve) to get the unknown values: θ_1, θ_2, θ_3 (joint angles).

One consequence of using the numerical method is that some joint configurations will be outside of robot's reachable workspace, which will cause some issues. We use recursion to solve this problem. We manually set up a joint angle allowable range constraint and if we get solutions outside of our specified range, we will try to solve the system again and under slightly different conditions. To avoid infinite recursion for an invalid pose we limit the maximum recursive depth to 25 cycles, but this rarely happens in practice.

$$\theta_1 = \arctan 2(x, y) \quad (16)$$

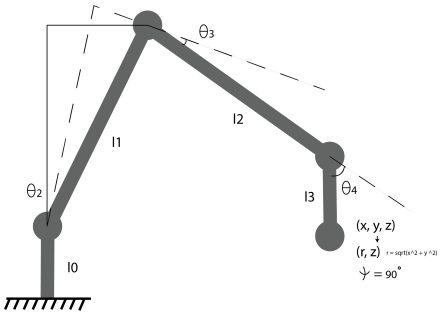

Fig. 3: 3-link RRR Arm

$$
\begin{gathered}
l_0 + l_1\cos(-\theta_2 + \arctan(4)) + l_2\cos(-(\theta_2+\theta_3)+ \\
l_3 cos(-(\theta_2+\theta_3+\theta_4)) - z = 0 \\
l_1\sin(-\theta_2 + \arctan(4)) + l_2\sin(-(\theta_2+\theta_3)+ \\
l_3\sin(-(\theta_2+\theta_3+\theta_4)) - r = 0 \\
\theta_2+\theta_3+\theta_4-\psi = 0 \\
where: r = \sqrt{x^2+y^2}
\end{gathered}
\tag{17}
$$

3) Path planning: Three major challenges encountered by the path planning algorithm is as follows: First, the wrist rotation angle of the robot must be set so that the blocks are picked up in a smooth manner and the arm can reach as far as possible in the workspace. This problem is tackled by offsetting the wrist rotation by 90 degrees when the pickup position is less than -45 degrees or more than 45 degrees, as shown in (algorithm 2) in the appendix. This ensures that the gripper does not hit the block during the pickup process. Second, way-points need to be set before and after pickup and drop locations so that the arm can avoid collisions with other block stacks when carrying out its trajectory. This is done by using appropriate pickup and drop offsets customized for each task. Finally, The moving time is set based on the the maximum angular displacement of all the joints while between two way-points. The moving time is calculated using $time = max\_disp/max\_vel$ such that the arm movement is as fast as possible at each step of the trajectory.

4) State machine: The state machine communicates with the control station, camera node and rxarm node to implement different states for different tasks. The various states implemented in the state machine are as follows:

- The states 'idle', 'manual' and 'estop' were previously defined to implement the fundamental actions of the robot so that other tasks could be performed using it.
- The 'execute' state sets the joint positions to some predefined waypoints that can be used to test the movement of the manipulator.
- The 'recordPoints', 'playback' and 'clear' states are used in the teach and repeat tasks. The path planning algorithm explained in the previous section is used to set the velocity of the arm such that it is directly proportional to the maximum displacement. This state also plots the joint positions after implementation of the 'playback' state.
- The 'calibrate' state finds the extrinsic matrix using the april tag positions and *solvePnP()* function and implements the warping of the image frame using the calculated perspective transform so that the workspace is visible in a rectangular overhead view.
- The event states are defined such that they use the block detection and the IK functions described previously to conduct tasks of autonomous pick and place challenges described in the competition.

III. Results

A. Computer Vision

The results of the combined intrinsic, extrinsic, and perspective warp are visualized in (Fig. 4). The figure shows that the workspace grid stretches beyond the dots at high magnitudes of x and y, indicating that the image does not have perfectly even spacing between lines.

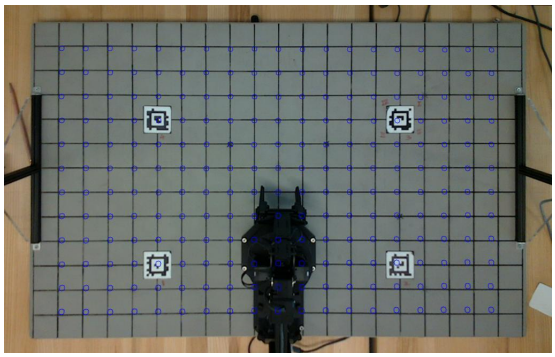

Fig. 4: Grid-points Projection on Warped Image

To verify the extrinsic calibration, we compared the results of the $pixel\_to\_world()$ function to the known world coordinates. Evidence of the accuracy of the calibration was obtained by measuring the (X_w, Y_w, Z_w) coordinates through the GUI for stacks of large blocks at the following points: $(0, 175), (-300, -75), (300, -75), (300, 325)$. This data is tabulated in Table II in the appendix and shows that Root-Mean-Square Error is 5.6 mm averaged over all measured locations, and 4.3 mm for locations on the floor of the workspace. This should be within an allowable range of error.

The output of the block detection algorithm is shown in (Fig. 5). While the blocks of all colors and sizes were

detected successfully as required by the objective, there were a few cases when the algorithm did not perform as well as expected. These are explained later on.

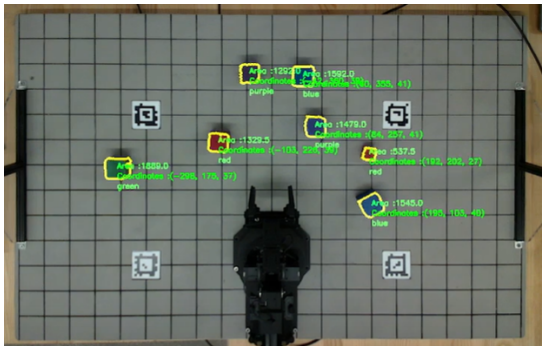

Fig. 5: Block Detection Output

To verify the obtained results from block detection centroids, an experiment was conducted where blocks were placed at known positions in the workspace. After converting the block detection results from image plane coordinates to world coordinates, they were compared to the known coordinates of the blocks, as visualized in (Fig. 6). As seen in the figure the two plots overlap reasonably well, demonstrating that the block detection algorithm measures accurate centroid positions. It is observed that the error of block detection grows when the blocks are moved further from the board's center.

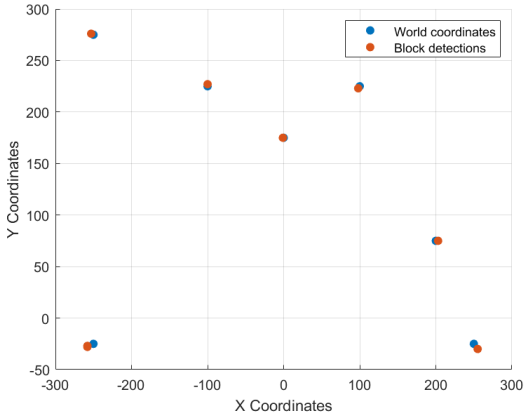

Fig. 6: Block Detection Accuracy Test Results

The shapes of the blocks were estimated using the *approxPolyDP()* method in the previous section and scale ratio calculated using adjacent points from *minAreaRect()* function in opencv. While cube, cuboid and triangle blocks were recognised correctly, the algorithm made false predictions for semi-circle blocks. The areas of the detected blocks were used to categorize the object into small or large blocks. The difference in area between blocks of different sizes was large enough that there was no overlap in the size ranges used for classification. The block detection algorithm was tested under different lighting conditions and on blocks of different reflection coefficients but of the same color. This test proved to require tuning of the HSV color bounds to fit the blocks with high reflections into the classification algorithm. The block detector also works for cases where blocks of different sizes are stacked on top of each other due to the height thresholding approach explained in the methodology. Overall, the block detection algorithm was implemented successfully and used for subsequent autonomous manipulation tasks.

B. Robot Control

Generally speaking, the robot was successful in navigating to all locations needed to pick up and drop off blocks. For the blocks within the QR code region, the success rate of the numeric solver was 100%. For blocks near the robot arm, the solver experienced a higher percentage of failures since the numerical method could not always reach a valid answer. The inverse kinematic results also demonstrated some error, which was apparent from the block not always being centered within the gripper at locations where the calibration and block detections were validated to be accurate.

The path planning algorithm worked well for most of the tasks, with a few outliers where the performance can be improved. The speed of the arm changes proportional to the displacement between two way-points which is expected. The parameterized vertical offsets around pickup and drop points ensures that the arm can finish all of it's given tasks without colliding with other blocks or stacks of blocks in the workspace. However, there exists a few edge cases where this approach either fails or proves inefficient. When a single block is placed behind a very tall stack of blocks that is in the trajectory of the arm, the manipulator might collide with the tall stack. This challenge is still encountered even when using the height thresholding approach explained in previous sections. Additionally, the wrist angle (ψ) of the arm at the pickup and drop locations is sometimes not exactly 90 degrees as required by the program because some positions are not reachable at this angle by the numerical IK solver. In these cases, the solver finds a a wrist angle as close to the desired value as possible, allowing the manipulator to complete the given task with negligible error.

The states described in the previous section were implemented successfully. The 'execute' state made the robot follow the predefined trajectory in under 30 seconds. The teach and repeat task was optimized using the desired velocity and moving time as described in the path planning section and the task was tested to be able to repeat it's actions for more than 10 times while swapping blocks among three positions. (Fig. 7) shows the joint

position after three iterations of the teach and repeat task. The calibration state was successful in getting the extrinsic transform and warping the image to overhead view. For each of three events, custom states and algorithms were implemented to autonomously complete the given tasks using block detection, Inverse Kinematics, and path planning methods discussed previously.

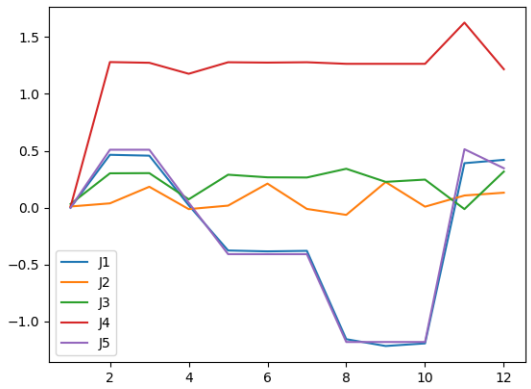

Fig. 7: Teach and Repeat Graph for 3 Iterations

C. Competition Results

Our robotic arm system was validated through a competition with five different events. We chose to compete in three of the five events, opting for the highest level of difficulty for each task. In the first event, Pick'n sort, we successfully sorted nine blocks of varying colors and sizes and received (300/300) points. In the second event, Pick'n stack, we successfully stacked the small and large blocks and received (210/210) points. Finally, we were able to sort large and small blocks by rainbow color while avoiding distractor blocks and scored (497/500) points, losing 3 points for imperfect lines.

IV. Discussion

A. Camera Calibration

Throughout the competition events, it was evident that there was some error in the calculation of world coordinates for detected blocks. In this section, we discuss the decisions made in the calibration method and some of the potential sources of error.

Compared to the manual intrinsic, the factory intrinsic matrix has a f_x and f_y that are closer in value, meaning that the factory calibration portrays the camera as closer to a perfect pinhole model than the manual calibration does. The manual intrinsic matrix data varied substantially between calibration trials and also differed from the factory calibration. One source of error may have been inconsistencies in checkerboard printing that resulted in color and size variance between squares. Additionally, changes in lighting between trials may have produced slight deviations in the localization of keypoints on the checkerboard. Overall, the factory intrinsic matrix produced more accurate world coordinates in *pixel_to_world()* because the factory calibration was likely performed with more consistent lighting, more than five trial, and may have included more positions and orientations. Due to these differences, we trusted the factory calibration more than the manual calibration and chose to use it for the workspace reconstruction algorithms.

In terms of the extrinsic calculation, the calibrated extrinsic performs much better than the manual one. One potential reason is that the calibrated extrinsic matrix accounts for small amounts of rotation about the y and z axes that were not considered in the hand-calculations. Additionally, the nominal matrix has high amounts of human error from taking measurements by hand, and would not be sufficient for pixel-to-world conversion.

While the automatic extrinsic calibration is better than the manual one, it is not perfect. One attempt to mitigate calibration error was to solve for the extrinsic with distortion coefficients; This improved the pixel-to-world accuracy but some error remains. There are many possible causes. For starters, the accuracy of the calibration depends on the ability of the camera to accurately locate the April Tags. Changes in lighting or vibration of the camera between calibrations may result in a mismatch between the calibration conditions and actual working conditions. Finally, since the intrinsic camera matrix is passed to *solvePnP()*, any error in calculating the intrinsic will propagate into the extrinsic calculation.

An additional metric to evaluate our camera calibration is the grid-point of dots projected on the image as seen in (Fig. 4). Since the dots are an evenly spaced array, it is evident from the figure that the calibration produces highly accurate results near the origin but the accuracy of the pixel-to-world conversion drops off further away. This is supported by our measurements of block world coordinates through the GUI. The data in (Table II) reveals that the average RSME for the blocks centered at $(0, 175)$ is 2.49 mm but the average of blocks at $(300, 325)$ is 6.57 mm. These results suggest that the extrinsic calibration is accurate since the centers of the grid and workspace align, but the intrinsic focal length parameter has some error.

B. Block Detection

The shape estimation part of the algorithm works by calculating the number of sides in the detected polygon initially. This will successfully classify triangles, arches

and cylindrical blocks. However, cubes, cuboids and semi-circular blocks all have 4 sides from the bird's-eye view. The algorithm tackles this problem by calculating the aspect ratio which is used for classification. However, there exists cases where the algorithm throws false positive results in classifying semi-circular arches as the threshold values for classifying blocks into cubes or semicircles are very close.

The size classification of the blocks works well for most of the tested cases since the large blocks have roughly three times the area of small blocks. The problem of stacked blocks of different sizes was dealt with by using the height thresholding approach discussed in the previous sections. This allowed for stacking of small blocks on big blocks as the block at the greater height would be recorded first and then the block detection code would run again for the lower block. However, a few tests did detect a big block under a small block as a second small block at the same location due to z values of the respective centroids being very close to each other. This problem was dealt with by decreasing the step size of the height threshold iteration. Another problem was that the algorithm would fail to detect blocks under shadows or with strong specular reflections. The reason for this is that the value and saturation of the block color changes when the lighting conditions change. This was fixed by tuning the value and saturation bounds for the recorded colors. The hue value remain the same under different light intensities. It is worth noting that the red colored blocks required wrapping of hue value to the (0-10) range and hence required a bitwise 'and' operation of two HSV masks.

C. Forward Kinematics and Inverse Kinematics

In order to determine the accuracy of the Forward Kinematic results, we used direct control mode to move the robot arm to known positions such as the April Tags. We then compared the FK estimate to the world coordinate to determine the FK calculation accuracy. This testing revealed consistent error in pose estimates, but we fixed it by adding offsets to the measured encoder angles to account for constant error that may have arisen from sensor drift.

For Inverse Kinematics, a singular configuration occurs when the end effector is located on the axis of z_0, causing infinite solutions for θ_1 and result in an unsolvable problem. To avoid this condition, it may have been beneficial to solve the Inverse Kinematic problem in closed form by using geometry. Compared to our approach of numerically solving a system of equations, which is usually non-closed and has more than one answer, it's faster to compute, and provides explicit knowledge of degeneracies and singularities. In addition, instead of using a symbolic method, we used a numerical method to solve the IK problem, which resulted in the system occasionally returning an invalid pose outside of the robot boundary. To solve this problem and get valid results, we had to use a recursive solution that attempted to solve the IK problem under slightly different conditions. Even so, it still failed occasionally. A numerical solving method could have helped our algorithm avoid failures and work more efficiently.

D. Path Planning

The vertical offset approach to adding waypoints in between pickup and drop locations so that the arm can avoid collision sometimes fails in edge cases described in the results section. This is because the offset might not be enough for the gripper to move over the stack and execute the grasping process. The height thresholding process discussed previously helped in solving this problem as the arm would complete the part of the task dealing with the blocks with greater height, i.e. the ones on top of the stack before it tried to grasp the lower blocks, thus avoiding collision. As discussed earlier, some poses in the workspace could not be reached by the robot. Hence an iterative IK solver was used. Whenever the numerical IK failed to reach a certain point, a new destination was fed which was slightly offset from the desired point, so that the robot could still complete the task. This resulted in wrist rotation not being exactly at the desired angle at some positions in the workspace.

V. Conclusion

In this paper, implementation of a framework for an autonomous 5 DOF manipulator is discussed. Various algorithms used to tackle robot control and computer vision problems are explained. A classical approach to robot perception is taken, and the Denavit-Hartenberg method and a numeric solver are used to implement inverse and forward kinematics respectively. The software designed was implemented on a RXArm in ROS2 ecosystem and tested for various edge cases such that the robot performs to it's maximum ability in executing the given tasks in the workspace.

The future scope for this project may include the addition on learning-based methods for perception as this would avoid the external tuning required in case of changing light conditions that was discussed earlier in the report. Additionally, learning based algorithms could increase accuracy of shape detection and improve collision avoidance in trajectories by using previous block location data. This paper showcases robot perception and manipulation fundamentals that form the backbone of industrial robotics; With improvements such as learning-based methods we will have a strong groundwork for endless robotic arm applications.

REFERENCES

[1] Camera calibration. Accessed: 2023-10-18. [Online]. Available: https://navigation.ros.org/tutorials/docs/camera_calibration.html

[2] E. Olson, "Apriltag: A robust and flexible multi-purpose fiducial system," University of Michigan APRIL Laboratory, Tech. Rep., May 2010.

[3] (2023) Perspective-n-point pose computation. [Online]. Available: https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html

[4] (2023) Understanding topics. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

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

[6] Y. Ding. (2023) Lecture slide: rotations & transformations, forward kinematics, inverse kinematics, basic image processing, numerical inverse kinematics.

VI. APPENDICES

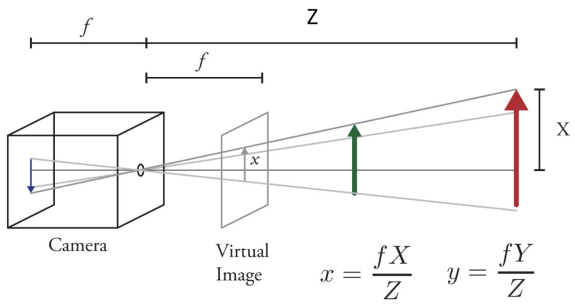

Fig. 8: Pinhole Camera Model

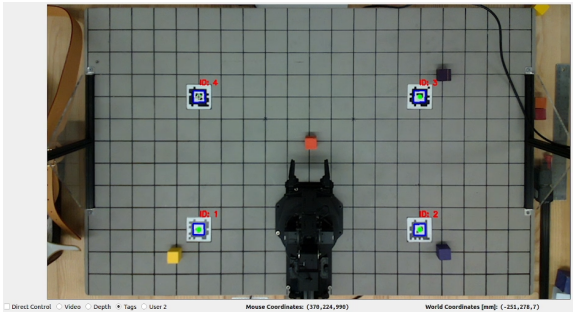

Fig. 9: April Tag Location Visualization

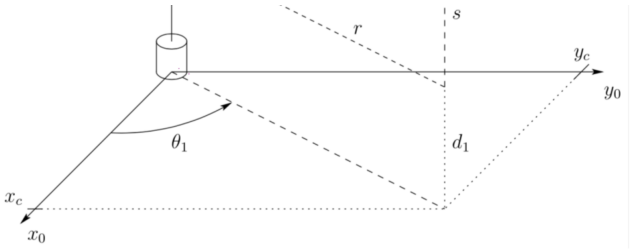

Fig. 10: Calculation for θ_1

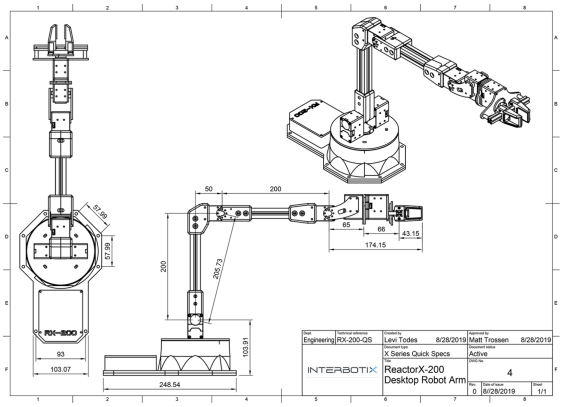

Fig. 11: Technical Drawing of the RX200 Arm

Algorithm 1 Block Detection

Require: $video\_frame, HSV\_bounds[], min\_area, shape\_dict$
Ensure: $annotated\_frame$
  procedure BLOCKDETECTOR
    $mask \leftarrow \text{inRange}(video\_frame, HSV\_bound[color])$
    $mask \leftarrow \text{threshold}(mask)$
    $contours \leftarrow \text{findContours}(mask)$
    for cnt **in** $contours$ **do**
      $area \leftarrow \text{moment}(0,0)$
      if $area > min\_area$ **then**
        $center\_x \leftarrow \frac{\text{moment}(1,0)}{\text{moment}(0,0)}$
        $center\_y \leftarrow \frac{\text{moment}(0,1)}{\text{moment}(0,0)}$
        $sides \leftarrow \text{approxPolyDP}(cnt)$
        $shape \leftarrow shape\_dict[sides]$

Algorithm 2 Path Planning

Require: $source, dest, p\_offset, d\_offset, des\_vel$
Ensure: $trajectory$
  procedure PATHPLANNING
    if $\arctan 2(\text{dest}(y), \text{dest}(x)) > 45$ **then**
      wrist_rot $\leftarrow$ wrist_rot $+ 90$
    if $\arctan 2(\text{dest}(y), \text{dest}(x)) < -45$ **then**
      wrist_rot $\leftarrow$ wrist_rot $- 90$
    w_point1 $\leftarrow$ source
    w_point1$(z) \leftarrow$ source$(z) + p_offset$
    w_point2 $\leftarrow$ dest
    w_point2$(z) \leftarrow$ dest$(z) + d_offset$
    for j **in** $joint\_pos$ **do**
      max_disp $\leftarrow \max(|j - j[-1]|)$
    moving_time $\leftarrow \frac{\text{max\_disp}}{\text{des\_vel}}$
    $trajectory \leftarrow$ IK(source, w_point1, w_point2, dest)

World (mm)	Measured (mm)	Stack Ht.	RMSE (mm)
(0,175,0)	(0,177,4)	0	2.58
(0,175,38)	(0,177,39)	1	1.29
(0,175,76)	(0,179,77)	2	2.38
(0,175,152)	(0,181,155)	4	3.87
(0,175,228)	(0,179,228)	6	2.31
(-300,-75,0)	(-305,-81,0)	0	4.51
(-300,-75,38)	(-305,-82,35)	1	5.26
(-300,-75,76)	(-307,-83,72)	2	6.56
(-300,-75,152)	(-304,-81,150)	4	4.32
(-300,-75,228)	(-305,-82,226)	6	5.10
(300,-75,0)	(308,-82,-1)	0	6.16
(300,-75,38)	(312,-82,33)	1	8.52
(300,-75,76)	(313,-82,71)	2	9.00
(300,-75,152)	(314,-82,147)	4	9.49
(300,-75,228)	(317,-80,224)	6	10.49
(300,325,0)	(306,328,2)	0	4.04
(300,325,38)	(308,327,37)	1	4.80
(300,325,76)	(310,327,74)	2	6.00
(300,325,152)	(311,328,150)	4	6.68
(300,325,228)	(316,330,227)	6	9.70

TABLE II: Actual vs. Measured World Coordinates