# ROB 550 ArmLab Report Team 4 (PM)

Audrey Cooke, Haejoon Lee, Nithish Kumar, Sharmane Powell
{audeo, haejoonl, nithishk, sepowell}@umich.edu

***Abstract*— Robotic arms have become integral in automating tasks across a wide range of industries. This study focuses on a 5-degree-of-freedom (5-DOF) robotic arm, exploring the integration of computer vision techniques, forward and inverse kinematics (IK), and path planning algorithms. While our results demonstrate successful object detection and manipulation, limitations were identified. Imprecision in block detection, particularly in centroid identification, hindered performance, leading to unstable stacking in tasks like Pick n' Stack. The root cause was attributed to imperfect camera calibration, suggesting improvements in calibration methods such as grid matching. Additionally, our square detection mechanism suffered from false negatives and positives, impacting tasks like Line n' Up. Alternative algorithms and moment-based filtering may offer solutions. Furthermore, the system struggled to differentiate small blocks stacked on large ones, necessitating corrective movements post-placement. Despite these challenges, our color detection using HSV showed promise, highlighting avenues for further refinement.***

## I. Introduction

As the demands of robotic applications continue to expand across various domains, the significance of robotic arms has garnered increasing attention. Robotic arms, characterized by their multiple links and resemblance to human arms, play a crucial role in automating tasks across various industries [1], [2]. These versatile machines are programmable and capable of executing specific tasks with precision and efficiency. Robotic arms are uniquely equipped to handle a wide array of tasks, ranging from simple pick-and-place operations to complex manipulation and assembly processes [1]. Their ability to mimic human arm movements, coupled with advancements in control algorithms and sensing technologies, has expanded their utility across industries such as manufacturing, healthcare, agriculture, and logistics [3].

In this work, we delve into the vision-based robotic arms, with a particular focus on a 5-degree-of-freedom (5-DOF) robotic arm. Through a comprehensive exploration of computer vision techniques, kinematics, and path planning algorithms, we aim to imbue autonomy into this robotic system. By leveraging these techniques, we seek to empower the robotic arm with the capability to manipulate various objects autonomously.

## II. Related Work

Recent advancements in robotic manipulation have spurred significant interest in enhancing dexterity and adaptability in robotic hands. Some works focus on the systems that anticipates the objective based on a brain control interface [4], and others focus on more physical approaches such as GUI control stations [5], or laser pointers [6]. The design and control problems of universal robot hands with elastic fingers are introduced in [7], wherein the authors propose reinforcement learning approaches to implement dexterous manipulation abilities.

Another area undergoing significant investigations is object detection and recognition. Authors in [8] demonstrated the effective use of convolutional neural networks (CNNs) for real-time object detection among cluttered environment in robotic manipulation tasks. Advancements in sensing technologies have contributed to the advancement of vision-based robotic systems. The utilization of depth sensors for 3D object localization and mapping in robotic manipulation tasks, such as LiDAR and stereo cameras, was discussed in [9].

In addition to object detection, researchers have also investigated kinematics and path planning algorithms to enhance the autonomy of robotic arms. A novel approach for trajectory planning in robotic manipulation tasks using genetic algorithms was proposed in [10], while the use of reinforcement learning techniques for adaptive control of robotic arms in dynamic environments was explored in [11]. These methods improved the fluidity and adaptability of the motion of the arms.

Overall, studies have made significant contributions to the field of vision-based robotic arms and autonomy. By building upon the existing findings and methodologies, our work aims to further advance the knowledge about the robotic manipulation systems towards achieving greater autonomy and efficiency in diverse application domains with numerous hardware experiments. We demonstrate the effectiveness of the adopted methods and also offer in-depth analyses as well as potential avenues for improvements.

## III. Methodology

### A. Computer Vision

For reliable vision and object detection, we are using an Intel RealSense L515 LiDAR Camera, which has a $1024 \times 768$ active TOF depth map resolution and a $1920 \times 1080$ RGB image resolution. The L515 also packs an Inertial Measurement Unit (IMU) capable of measuring acceleration and angular velocity for all 6 degrees of freedom (6DoF). For software, we make use of the OpenCV library in Python. This enables the robot arm to manipulate the blocks. The affect of lens distortion and the unknowns about the camera's orientation relative to the workspace prompt the need for a calibration process [12]. Once the camera has been calibrated, the vision system makes use of the depth and color sensor data from the L515 in order to detect blocks in the workspace.

*1) Camera Calibration:* We made heavy use of the pinhole camera model in our computer vision systems. In the pinhole camera model, points $[x_w, y_w, z_w]$ in the world coordinate

frame can be projected into the image with coordinates $(u, v)$ by way of the intrinsic matrix $K$ and the extrinsic matrix $H$, where $H$ is a homogeneous transformation which describes the translation $T$ and rotation $R$ of the world coordinate frame with respect to the camera coordinate frame, as seen in (1). Note that $U$ and $W$ are 2D and 3D coordinates lifted with another dimension by adding a row of 1 to match matrix dimensions.

$$\underbrace{\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}}_{U} = \underbrace{\frac{1}{z_c}\begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} I & | & 0 \end{bmatrix}}_{K}\underbrace{\begin{bmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{0} & 1 \end{bmatrix}}_{H}\underbrace{\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}}_{W} \quad (1)$$

where:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = H \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \quad (2) \qquad \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = H^{-1} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} \quad (3)$$

The intrinsic matrix $K$ comprises of focal lengths in pixel units $[f_x, f_y]$, and principal offsets $[u_0, v_0]$. $K$ projects coordinates defined relative to the camera's coordinate frame into the image plane, and is an intrinsic property of the camera. To manually measure $K$, we used the ROS package called $\mathrm{camera\_calibration}$. This package allows for easy calibration using a checkerboard calibration target. This calibration process also identifies the distortion parameters and reduce the camera distortions, though we did not reverse the distortion of the camera images. The intrinsic matrix resulting from the manual calibration process with the checkerboard is shown in (4), and the factory-programmed intrinsic matrix reported by the L515 is shown in Equation (5).

$$K = \begin{bmatrix} 908.36 & 0 & 662.62 \\ 0 & 908.40 & 364.88 \\ 0 & 0 & 1 \end{bmatrix} \quad (4)$$

$$K' = \begin{bmatrix} 911.96 & 0 & 658.32 \\ 0 & 912.35 & 357.75 \\ 0 & 0 & 1 \end{bmatrix} \quad (5)$$

Clearly, the two matrices are very similar. Some sources of error could come from unaccounted-for lens distortion affecting the camera, camera sensor noise, and imperfect AprilTag feature matching during the calibration.

The extrinsic matrix $H$ in (1) is a $4 \times 4$ homogeneous transformation comprised of a 3D rotation matrix $\mathbf{R}$ and a 3D translation vector $T$ which transforms coordinates $[x_w, y_w, z_w]$ defined relative to the world coordinate frame into coordinates $[x_c, y_c, z_c]$ defined relative to the camera coordinate frame so that the intrinsic matrix may be applied to project points in the workspace into image coordinates $(u, v)$. We manually measured our extrinsic matrix by taking the inverse of a measured transformation defining the camera's coordinate frame relative to the world coordinate frame. The translation was measured manually with a meter stick, and the rotation angle was determined by using the identity that for any vectors $a$ and $b$, $a\dot{b} = |a||b|\cos(\theta)$ where $\theta$ is the angle of rotation. Then the rotation angle was calculated as the angle between the camera z axis and the vector describing acceleration due to gravity as reported by the IMU on the L515. The resulting extrinsic matrix is shown in (6).

To determine the extrinsic matrix automatically, we solved the PnP problem correlating centers and corners of four detected AprilTags on the board to their known locations in order to generate the extrinsic matrix using the functions $\mathrm{cv2.solvePnP}$ and $\mathrm{cv2.Rodrigues}$ as is done in [13]. This resulted in the extrinsic matrix in (7) for a particular camera orientation.

$$H = \begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & -0.971 & 0.239 & 141.817 \\ 0 & -0.239 & -0.971 & 1054.556 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6)$$

$$H' = \begin{bmatrix} 0.9998 & -0.0056 & -0.0031 & 10.76 \\ -0.0046 & -0.9657 & 0.2595 & 115.003 \\ -0.0044 & -0.2595 & -0.9657 & 1046.0728 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7)$$

Comparing (6) and (7), we can see that they are very similar. The differences between the two arise due to some sources of errors, such as lens distortions and noises in the images themselves. However, these errors, as one can see by comparing the values of (6) and (7), are not large. This nominal matrix is not sufficient for finding points in the world frame, as whenever the camera moves the entries in the matrix change.

Using the intrinsic and extrinsic matrices, we can find the world coordinates $[x_w, y_w, z_w]$ from the pixel coordinates $(u, v)$. We first convert the pixel coordinate $(u, v)$ to the camera frame $[x_c, y_c, z_c]$ using (8).

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = z_c(u, v)K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad (8)$$

where $z_c(u, v)$ is the depth of the image at the pixel coordinate $(u, v)$. Then, we plug the camera coordinates $[x_c, y_c, z_c]$ into Equation (3) to get the world coordinates $[x_w, y_w, z_w]$.

The camera captures the workspace at an angle, and as a result the rectangular workspace board appears trapezoidal. For this reason the calibration process uses the centers and desired locations of each AprilTag in the image frame to calculate the homography matrix [14] shown as (9) using $\mathrm{cv2.findHomography}$. After obtaining $P$ as in (9), we then use $\mathrm{cv2.warpPerspective}$ function to warp the camera image into a birds eye perspective.

$$P = \begin{bmatrix} 1.1728 & -0.1565 & -141.8018 \\ 0.0036 & 0.8994 & 32.7928 \\ 0 & 0 & 1 \end{bmatrix} \quad (9)$$

For completeness, the center coordinates of the AprilTags are given at Table I. The coordinates are in the world frame. Note that each corner of the AprilTags are $\pm 25mm$ in the $x$ and $y$ values of the center coordinates. Camera calibration is implemented the function $\mathrm{camera.calibrate\_camera}$ in the file $\mathrm{camera.py}$ : 83.

TABLE I

| AprilTag 1 | AprilTag 2 | AprilTag 3 | AprilTag 4 |
|---|---|---|---|
| $(-250, -25)$ | $(250, -25)$ | $(250, 275)$ | $(-250, 275)$ |

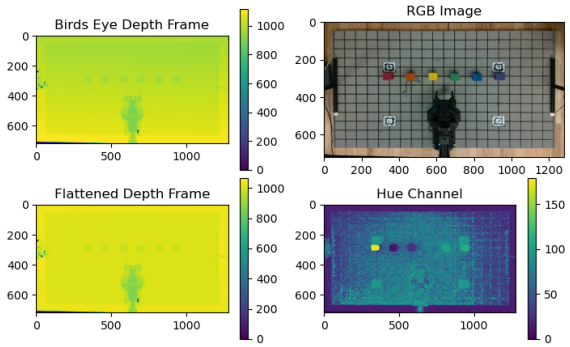


Fig. 1. Depth Map with depths defined with respect to camera coordinate frame (top left); flattened depth frame (bottom left) after applying the flattening reprojection; RGB Image of 38mm painted wooden cubes in all six colors (top right); hue channel of the image after extraction from the HSV camera image (bottom right) demonstrating the expected hues of the blocks.

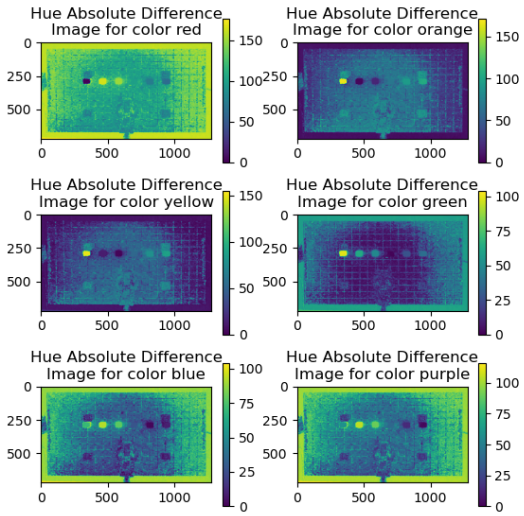


Fig. 2. Hue-channel absolute difference from each of 6 target colors, with red $= 176 \pm 10$, orange $= 8 \pm 5$, yellow $= 24 \pm 3$, green $= 75 \pm 4$, blue $= 104 \pm 4$, and purple $= 116 \pm 6$.

*2) Block Detection:* The Armlab robot's ability to perform operations on blocks in its task space is dependent on the ability of the system to effectively detect the blocks. Our block detection algorithm starts by reprojecting the depth frame's values, which are defined relative to the Camera coordinate frame, into a "flattened" coordinate frame which is located one meter above the world coordinate frame which preserves the X and Y directions in the depth image. This process was adapted from the code implemented in [15]. This ensures that the depth image carries depth values at a uniform depth over the entire surface of the workspace. The result of this flattening is displayed in Figure 1. The flattened coordinate frame is defined as 1 meter above the World coordinate frame at the center of the robot with a rotation of $180^\circ$ about the x axis.

Then, a binary image is created from the depth map by setting pixels with a depth value in the range of between 960mm and 993mm to 255, and those outside of the range to 0. This ensures that block stacks between 7mm and 40mm tall will be detected. The binary image is then masked with a board mask to ensure that the edges of the board and the robot arm will not be detected as blocks. The masked binary image is then passed to OpenCV for contours analysis, resulting in a list of contours outlining blocks on the board, which is then filtered to discard contours with small areas.

For each block contour in the masked in-range binary image, the color detection algorithm determines the dominant color in the block by comparing the number of pixels within a specified range of each color. To do so, RGB camera image is converted to HSV, and the hue channel is extracted, shown in in Figure 1. The hue channel is extracted from the HSV version of the camera's image, and the hue of interest for each color is subtracted from the hue image. The absolute value of each hue difference image is taken, resulting in a map of absolute hue differences from each hue as shown in Figure 2.

Finally, OpenCV's threshold function is used to set pixels with a hue difference below the specified threshold to 255, resulting in the hue-threshold binary image which isolates all of the pixels in the image within a range of the target hue. A bitwise AND operation is used to mask the hue-threshold binary image with a mask generated with the block contour, and pixels valued at 255 are counted. The color which maximized the pixel count is selected and assigned to the block. The block detector then saves the pixel-coordinate center of and rotation angle $\theta$, which is clockwise relative to the world Z axis, along with the color of the block in a list to be used for the competition tasks. The centers of mass and rotation angle are calculated based on the moments of the contour.

Block detection is implemented the function $\mathrm{camera.blockDetector}$ in the file $\mathrm{camera.py} : 409$. Color detection is implemented the function $\mathrm{camera.retrieve\_area\_color}$ in the file $\mathrm{camera.py} : 348$.

## B. Control

*1) Forward Kinematics:* A crucial step in the design of a robotic manipulator is to gain an understanding of its geometry and kinematics. We used Denavit-Hartenberg (DH) parameters to describe the forward kinematics of our robotic arm. This process allows for the construction of homogeneous transformations to describe the configuration of a robotic arm with ease. Each link is represented by a series of four 4x4 Homogeneous transformations parameterized by a rotation $\alpha$ about the x axis, a translation $a$ along the x axis, a translation $d$ about the z axis, and a rotation $\theta$ about the z axis. Then for the $i^{\text{th}}$ joint in a serial robot, the transformation $A_i$ representing

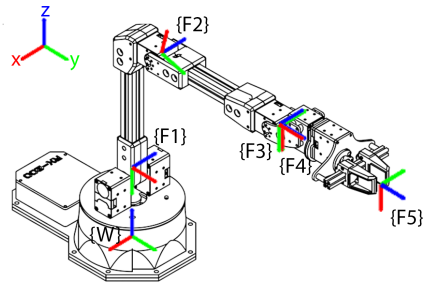


Fig. 3. Matlab simulation of robot arm forward kinematics overlaid on CAD isometric view of RX200. All joint angles are set to 0 and transforms were generated using the DH parameters in Table II. Coordinate frame X, Y, and Z axes are denoted by the red, green, and blue vectors of each coordinate frame respectively. Note that we have set $l1 = d_1$, $l2 = a_2$, $l3 = a_3$, and $l4 = d_5$.

TABLE II

| link | $a$ | $\alpha$ | $d$ | $\theta$ |
|---|---|---|---|---|
| 1 | 0 | 1.5708 | 104.82 | 1.5708 |
| 2 | 207.95 | 0 | 0 | -1.3258 |
| 3 | 205.10 | 0 | 0 | 1.3258 |
| 4 | 0 | 1.5708 | 0 | 1.5708 |
| 5 | 0 | 0 | 180.36 | 0 |

that joint is a combination of the following, collectively called the DH transformations:

$$A_i = Rot_{z,\theta_i} Trans_{z,d_i} Trans_{x,a_i} Rot_{x,\alpha_i} \tag{10}$$

These transforms can be combined to find the transformation $A$ which describes the position and orientation of the end effector coordinate frame relative to the world frame:

$$A = A_1 A_2 A_3 A_4 A_5 \tag{11}$$

Table II showcases the DH values of our RX200 arm. The values of $a$ represent the link length, while $\alpha$ represents the values of link twist. Also, $d$ is the joint offset, and $\theta$ represents the joint angle. All distances are measured in millimeters, and all angles in radians. To calculate the position and orientation of the end effect coordinate frame, the joint transformations were calculated from this table, adding the joint actuator position to the DH $\theta$ parameter for each joint. Figure 3 demonstrates the position of each coordinate frame on the robot arm. Note that F3 and F4 are coincident; F3 is the coordinate frame whose Z axis points in the same direction as F2 and F1, while F4 has its Z axis in the X direction of F3 to rotate the wrist. Our forward kinematics is implemented in the function FK_dh in the file kinematics.py : 29.

*2) Inverse Kinematics:* Inverse kinematics (IK) is the process by which the target position $[x_p, y_p, z_p, \theta_p]$ of the end effector of a kinematic chain is used to find joint angles for each joint in the chain. We implemented an IK method which relies on the 2-joint planar robotic arm model. All joint angles are calculated using the X axis of the joint coordinate frame as a reference for 0 rotation. After obtaining the ith X-relative joint angle $\theta_{iIK}$ to rotate about the Z axis of the ith joint

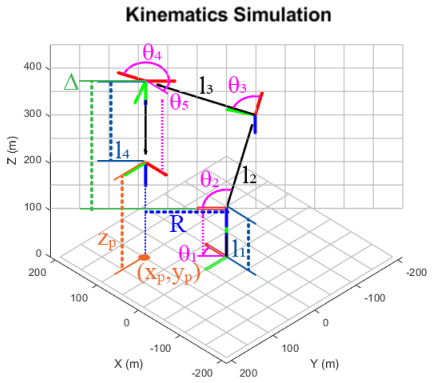


Fig. 4. Matlab simulation of robot arm with target end effector position of $x_p = y_p = 100$, $z_p = 200$, and $\theta_p = 0$. Labels of key measurements for Inverse Kinematics are displayed. Angles of joints are displayed in magenta. Note that these angles denote rotation of the joint about the Z axis of each coordinate frame with the X axis at $\theta$=0.

frame, we use $\theta_{iDH}$ to obtain the joint angle setpoints $\theta_{iJ}$ for each servo:

$$\theta_{iJ} = \theta_{iIK} - \theta_{iDH} \tag{12}$$

To calculate the IK angles $\theta_{iIK}$, we first use the X and Y values for the target pose to calculate the angle the base of the arm must rotate. We chose to constrain the base between $-\frac{\pi}{2}$ and $\frac{3\pi}{2}$, so to calculate the base inverse-kinematics angle result, we use (13). Adding $2\pi$ conditionally ensures that the base never over-rotates, as if the X and Y coordinates of the target pose are negative, then atan2 will result in an angle less than $-\frac{\pi}{2}$.

$$\theta_{1IK} = \text{atan2}(y_p, x_p) + (x_p < 0 \& y_p < 0) * 2\pi \tag{13}$$

To calculate the remaining joint angles, we place a constraint on the end effector that it must always be pointing down relative to the board. This allows us to decouple the IK into a 2-joint planar robot, which vastly simplifies our IK approach, while limiting the reach of the arm. In order to do this, we first need two values which describe the distance and height of the 2-joint end effector. Figure 3 demonstrates this, where $\Delta$ represents the distance in the Y direction relative to the coordinate frame F1 (note that it is negative) and R represents the distance in the X direction of the coordinate frame F1. From the geometry of our arm, we can find that $\Delta = Z_p + l4 - l1$, and $R = \sqrt{x_p^2 + y_p^2}$. Then, $\theta_{2IK}$ and $\theta_{3IK}$ may be calculated as in (15) and (14).

$$\theta_{3IK} = \left| \arccos\left( \frac{(R^2 + \Delta^2) - l2^2 - l3^2}{2 * l2 * l3} \right) \right| \tag{14}$$

$$\theta_{2IK} = \text{atan2}(-\Delta, R) - \text{atan2}(l2 * \sin(\theta_{3IK}), l2 + l3 * \cos(\theta_{3IK})) \tag{15}$$

Given that the wrist joint always points down, we write the constraint that $\theta_{2IK} + \theta_{3IK} + \theta_{4IK} = \pi$, and thus we calculate $\theta_{4IK}$ as in (16). Finally, we set the value of $\theta_{5IK}$ to counteract the rotation of the base, and to rotate to the target position. Thus $\theta_{5IK}$ is calculated as in (17). Our IK is implemented in the function IK_geometric in the file kinematics.py : 199.

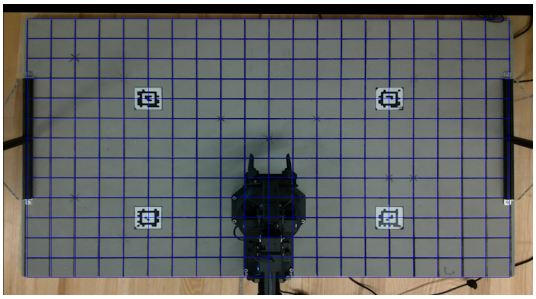

Fig. 5. Grid projection on the workspace to demonstrate the accuracy of our camera calibration and perspective warping.

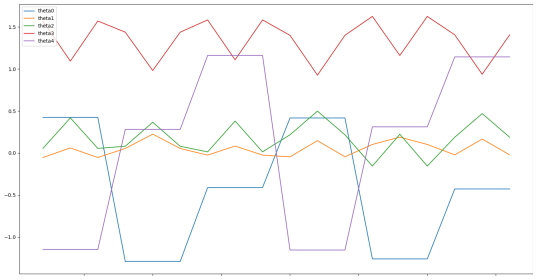

Fig. 6. This figure tracks the joint angle states as it swaps two different blocks at world coordinates $(-100, 225)$ and $(100, 225)$ through the intermediate world coordinate $(250, 75)$.

$$\theta_{4IK} = \mod(\pi - \theta_{3IK} - \theta_{2IK}, 2\pi) \tag{16}$$

$$\theta_{5IK} = \theta_{1IK} + \theta_p \tag{17}$$

Figure 4 shows an example output of our IK, labeled with all of the key distances and angles relevant to the IK approach detailed in this section. The figure was generated with a matlab simulation of our forward kinematics transformation structure using the inverse kinematics process to set the end effector to the target position of (100,100,200,0). For this pose, the joint angles are as follows (in radians):

TABLE III

| | |
|---|---|
| $\theta_{1IK}$ | 0.7854 |
| $\theta_{2IK}$ | -1.8104 |
| $\theta_{3IK}$ | 1.4403 |
| $\theta_{4IK}$ | 3.5117 |
| $\theta_{5IK}$ | 0.7854 |

TABLE IV

| | |
|---|---|
| $\theta_{1J}$ | -0.7854 |
| $\theta_{2J}$ | -0.4846 |
| $\theta_{3J}$ | 0.1145 |
| $\theta_{4J}$ | 1.9409 |
| $\theta_{5J}$ | 0.7854 |

*3) Motion Planning:* For any position the robot arm is commanded to move, we first examine if the desirable pose is reachable by checking if $\sqrt{R^2 + \Delta^2} <= l2 + l3$. If this relationship is not satisfied, it means that the position is out of the range reachable by our IK process, which is a limitation of the constraint that the wrist joint always points down.

Our motion planning procedure works in three steps. Provided the target pose is reachable, the bot will first try to move to a position 100mm in the Z direction above the target pose. Then, the robot arm will move down toward the target pose before either grasping or releasing a block. Finally, the arm moves back up to the position above the block. In the case that the position above the block is unreachable, the arm uses the joint angles for the target pose itself and offsets the value of $\theta_{2J}$ by a small amount to move the entire arm upward. While this violates the constraint that the wrist joint always be orthogonal to the workspace, it allows the arm to move above any reachable target position. The motion planner for grabbing and releasing blocks is implemented in the function camera.grasp_release_helper in the file control_station.py : 210.

In addition to the IK-driven motion planning algorithm, we programmed the arm to replicate the movements through a Teach-and-Repeat process. Our user interface was programmed with a series of buttons to enable this feature. Whenever we press the button "Record Next Waypoint," the system saves the joint positions of the arm into a list of tasks. Pressing the "Record Gripper Grasp" and "Record Gripper Release" buttons adds a command to grasp or release the gripper to the list. After the sequence is programmed the "Execute Recording" button triggers the arm to iterate through the list of saved states to replicate the taught positions.

## IV. Results

### A. Kinematics

We verified the accuracy of our FK parameters and IK scheme through use of the Matlab script which generated Figure 4 (included in code) by setting a variety of target poses and confirming the position of the end effector coordinate frame F5. In addition, we implemented a feature to click-and-move to points in the workspace, and verified that our robot arm did in fact make it to those points. From our measurements, our IK approach was accurate to within a radius of less than 10mm, with error stemming from a combination of servo motor backlash and measurement deviations in the DH parameters that we obtained for the arm.

### B. Calibration

We have verified the accuracy of our calibration using various methods. The first method we used was a grid projection on the image, as shown at Figure 5. The block alignments of the board after calibration are closely aligned with the blue projection grid lines, with slight deviations arising particularly in the bottom corners. However, even that deviation is very negligible. Another method was placing several blocks on the workspace, as shown at Figure 7, and measuring their locations via the user interface. Since we know each large block has a dimension of 38 mm $\times$38 mm $\times$38 mm, we can compare the actual world $z$ coordinate with the $z_w$ coordinate we obtain through calibration at the location of block to test whether the depth of the image is working correctly after the calibration. After testing the accuracy of the calibration with these two methods, we have concluded that our calibration is very reliable.

### C. Teach-and-Repeat

We used the Teach-and-Repeat feature of the robot arm to swap two blocks that are located at $(-100, 225)$ and

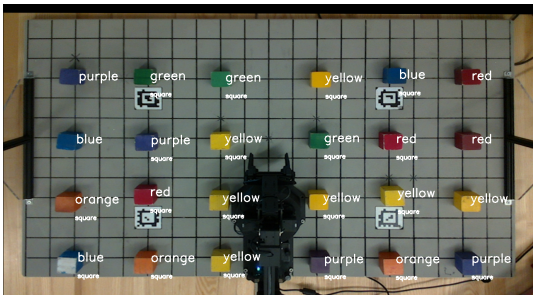


Fig. 7. The setup for the analysis of the accuracy of the block detection system. It indicates the color of the blocks as well as whether they are squares or not.

$(100, 225)$ on the workspace frame. The robot is taught to use the intermediate location of $(250, 75)$ to put one block temporarily before moving the other block. Our RX200 arm was able to perform the process of swapping two blocks more than 10 times without any problem. The joint angles commanded over time for the Teach-and-Repeat task are shown at Figure 6.

### D. Block Detection Accuracy

We used detection algorithms and camera calibrations to detect and locate twenty-four 38 mm $\times$ 38 mm $\times$ 38 mm blocks' centroids. After several experiments, we discovered that large blocks have moments of $1200 - 1600$, while small blocks have moments of $600-900$. We programmed the arm to ignore contours with moments less than 600 to filter out sufficiently small contours. We placed randomly colored blocks as shown in Figure 7 at $X = \{-400, -250, -100, 100, 250, 400\}$ mm and $Y = \{-125, 25, 175, 325\}$ mm in the workspace coordinates. For each region of the workspace, we calculated the $L2$ norms of the differences between the actual workspace XYZ center coordinates and the detected workspace XYZ center coordinates. The heat map of the $L2$ norms of the errors of the detected blocks' center coordinates to their actual coordinates are shown at Figure 8. This test also demonstrates the effectiveness of our color detection methods.

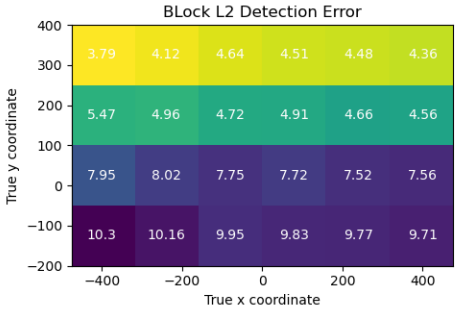


Fig. 8. The heatmap for $L2$ norms of the errors of the XYZ center coordinates of the large blocks. The unit of the error is mm, and the setup of the workspace is shown at Figure 7.

### E. Pick 'n Sort

In this task, we automated the robot to sort 9 blocks of varying sizes in the positive half-XY-plane of the workspace depending on their sizes. The robot was to put large blocks on the left negative half-plane and small blocks on the right negative half-plane. To make the task more challenging, we stacked some of the blocks on top of one another, so the block beneath would not be detected until the block on the top is sorted. To overcome this, we made the robot detect blocks that are on the positive half-plane of the workspace every time it sorts a block into the negative half-plane. This ensured that all detected blocks were placed until the blocks had been exhausted, and proved useful for when the robot happened to drop a block. Our robot arm successfully sorted the 9 blocks of the varying sizes within 130 seconds.

### F. Pick n' Stack

For this task, we automated the arm to stack 3 small (25mm) blocks, and 3 large (38mm) blocks on top of the AprilTags. For our approach, first the arm would need to distinguish between the small and large blocks, and then it would need to stack blocks in a particular location. We used AprilTag 2 and 3, whose world coordinates are $(250, -25)$ and $(250, 275)$, for small and large blocks respectively. Similar to the previous task, some blocks were stacked initially so the system would have to exhaust all blocks left to move. Our robot arm had no trouble stacking the large blocks on AprilTag 3. However, it kept failing to stack the third small block on AprilTag 2. This was caused by the presence of the IK error from backlash and parameter deviation. We observed that the arm would often under- or over-shoot the target position of the block before grabbing or placing. The accumulation of this error led to the blocks not being perfectly centered when stacking. PID tuning on the servos would be likely to eliminate this error, but was not attempted.

### G. Line 'n Up

Here we had the objective of automating the robot to line up 6 large blocks to the right $(125, -50)$ and 6 small blocks to the left $(-125, -50)$ of the robot arm in the ROYGBV order, where some of the blocks were stacked. Also, the arm was to avoid all the non-square objects placed among the blocks in the workspace as distracting objects. For this task, we first distinguished between the small and large blocks and provided the location for each block with an offset of 38mm between each block's centre. Then we detected the blocks in ROYGBV colour order and decided where to place them. The robot arm was not able to detect the blocks that were stacked more than three blocks tall due to our block detection range setting, and some of the blocks were outside of the space reachable by our inverse kinematics methodology, and non-square objects proved challenging to distinguish.

## V. Discussion

### A. Object Detection

As shown in the previous section, our robot arm has limitations and thus has room for improvement. First, our block detection was imprecise - there was significant error in the detection of the blocks' centroids. However, this error

was not detrimental to performance, as the robot had no trouble picking the blocks it detected. This slight imprecision still affected the robot's performance when it required rather precise and delicate movements. For example, its performance on Pick n' Stack was lacking especially with stacking the small blocks, as inaccurate detection and positioning error led to off-center gripping of the block, which eventually led to unstable stacking. As the arm kept stacking the small blocks in unstable positions, the stack kept collapsing at the third block. There can be many reasons behind the imprecise detection. The most likely candidate is lens distortion. Figure 8 shows that the detection error increases as we go down on the y-axis of the workspace. This trend might occur because our camera calibration was not perfect. The majority of this error is in the $x$ and $y$ directions. The obvious solution is to improve our camera calibration methods and account for lens distorton in our vision system. One possible method of doing this automatically is grid matching to reduce the distortion errors. The grid overlaid in Figure 5 represents an example of a grid which we can use to optimize the calibration by reducing the errors of the grid and its board alignments. Another way for an improvement is to use some offsets. Because there is a clear trend in the error, we could apply some offset dependent on the y-value to reduce the error.

Our block detection method proved ineffective at determining whether detected contours on the board were square or not. Our robot severely suffered in Line n' Up task, as many of the distracting objects were detected as squares by our block detector. As shown on Figure 7, some blocks - especially those on the edges of the board - are not detected as squares. For square detection, we relied on two built-in functions of the $\mathrm{OpenCV}$ library. The first one is $\mathrm{cv2.approxPolyDP}$ function that counts and returns how many corners the given contour has. The second function is $\mathrm{cv2.boundingRect}$ that draws a bounding rectangle around the given contour and returns its top-left coordinate as well as dimension. We programmed it in a such way that if the given contour has $4$ corners and similar height-width ratio, the arm would consider that object square. However, based on our experiments, this algorithm was not effective. The problem was two-fold. The first problem we faced was the fact that the system was not able to detect all of the squares. This resulted in false negatives around the edges, likely originating from the faulty block detection mechanisms as indicated in the earlier parts of the paragraph. The arm also suffered from the high false positive problem where it took non-square objects' shapes as square. Despite all the efforts to resolve the second issue by trying the same algorithm with varying parameters on $\mathrm{cv2.approxPolyDP}$ - for example, we varied different lengths in which the sharp turns and thus corners would be considered - we could not fully fix the problem. One possible solution we could not try due to the time constraint is trying a different square detection/filtering algorithm instead of the one we tried. Potentially, instead of relying on the built-in functions, we could try and see if the non-square objects, especially the arch and semi-circle objects have moments outside of the ranges in which large and small squares' moments lie. If so, we can use some designated ranges of moments in which these objects can be filtered out of the detection system.

Another problem we discovered after attempting different tasks is that our system has a hard time detecting a small block stacked on top of a large block. This is due to the using a fixed range to detect depths within, and perhaps could be remedied by looking for blocks within a range of the minimum-valued depth on the board so that blocks on top of others would always be detected first. If a small block was on top of the large block, especially if they are in the same color, the system mistakenly detected it as a large block because the top of the large block was in the range for block detection while the top of the small block was not, which resulted in blocks being placed on the wrong side. Another possible solution to this problem is to check to see that blocks on each side are all of a given size, and then to switch out blocks that are on the wrong side. For example, assume that the arm wrongly detected and lined a small yellow block that was on top of a large yellow block. Once placing the yellow block, it would re-detect the large yellow block sitting on the same coordinates as before. Then, it would go back to the small yellow block, grab it, and place it to the correct line. Since we had pre-designated workspace coordinates for each block of different colors and sizes, this would allow the arm to fix its previous mistakes as soon as it discovers them. During our Line 'n up task, we noticed another issue. We followed a prioritization sequence following the color spectrum of ROYGBV. When arranging stacked blocks, encountering a blue block above a red one resulted in our inability to retrieve the red block due to the sequencing of loops, where the iteration for the blue block occurred subsequently. One possible solution to this problem is to have the arm unstack and put the blocks of colors not on the priority of stacking somewhere else in the workspace until the arm needs to stack the block of those colors. Alternatively, the system could start by unstacking all blocks and sorting them by color before stacking the blocks in their final positions.

### *B. Color Detection*

While our object detection method has showed some flaws, our color detection was very promising. The main reason behind our good color detection system is because we used HSV instead of the usual RGB, which makes recognizing objects much easier in varying external lighting conditions. This means that in cases of minor changes in external lighting (such as pale shadows,etc), hue values are mostly decoupled from lighting. Initially we used a color detection method which measured the angle between vectors comprising of the R, G, and B values of image pixels and target colors. When we relied on RGB, our camera poorly distinguished between different colors. It especially had a hard time correctly detecting red and purple colors, probably due to the effects of sunlight. Even experimenting with varying RGB values for these colors did not yield any fruitful improvements. After switching to HSV, however, our color detection system underwent a huge improvement, and thus HSV provided a more robust and reliable means of color detection for our use case. The promising performance of our color detection system is shown at Figure 7- although there was significant error

in the detection of blocks' exact centroids, that was not the case for the color detection. There is also another interesting observation one can make from the figure. If one looks closely to the bottom left of the board shown on the figure, there is a blue large block, most of whose front side is covered with a gray sticker. Since the gray sticker covered more than half of the block's surface, it poses a significant disturbance on the color detection system. However, as shown on the figure, our system was able to detect the color blue correctly despite the disturbances. The figure proves that our system can detect all the blocks' colors throughout the whole workspace. Furthermore, during the formal tasks such as Line 'n Up, where the arm was to line the blocks in the ROYGBV order without blocks being stacked initially, our system never had any instances of incorrect block placement. Therefore, with the performance we have witnessed through the experiments, there is not much room for improvement on our color detection.

### C. Kinematics

The forward kinematics of the robotic arm were described using DH parameters, providing a systematic approach to model the arm's geometry and configuration. By utilizing homogeneous transformations and DH parameters, we were able to accurately predict the position and orientation of the end effector given the joint angles. However, there exist inherent limitations of forward kinematics, particularly in complex robotic systems where factors such as joint flexibility and external forces can influence the end effector's actual position. To some extent this could be eliminated by tuning the PID controllers of the Dynamixel Servos used as joint actuators. Introducing an integral term to the controller would ensure that the positioning error approaches zero over time, at the expense of overshoot and oscillation.

In contrast, inverse kinematics (IK) is crucial for determining the joint configurations necessary to achieve a desired end effector position and orientation. Our implementation of Inverse Kinematics relied on a simplified 2-joint planar robotic arm model which locked the wrist joint into a fixed orientation relative to the bed to decouple the kinematics from that joint. However this simplification limited the range of our robotic arm. To improve the range, the wrist joint $\theta_{4IK}$ could be made to rotate to increase the range, locking the elbow $\theta_{3IK}$ at zero degrees, thus decoupling the kinematics problem to a different configuration of a 2D planar robotic arm with joint lengths $l_2 + l_3$ and $l_4$.

Another way to free this constrained motion is relying on the numerical Inverse Kinematics methods, such as Newton-Raphson method. Although many numerical Inverse Kinematics methods are fast and convenient in the sense where it does not have to deal with singularities, we chose to use a geometric approach for our IK due to the robustness it provided to the problem of local minima. In any case rectifying this issue would vastly improve the arm's range and ability to complete tasks.

Furthermore, incorporating dynamic models and real-time feedback mechanisms could enhance the arm's adaptability and robustness in various operating environments. Additionally, investigating optimization techniques for trajectory planning and obstacle avoidance could further improve the arm's efficiency and reliability in performing complex manipulation tasks.

## VI. Conclusions

Our report presents an autonomous robot arm that can detect and manipulate blocks of interest. We have outlined the mathematical foundation for our kinematic algorithms and used experimental data to assess the reliability of our system. Some issues were successfully resolved by our implementation, and suggestions for improvement have been made to address the areas where our techniques fell short.

## References


[1] S. Surati, S. Hedaoo, T. Rotti, V. Ahuja, and N. Patel, "Pick and place robotic arm: a review paper," *Int. Res. J. Eng. Technol*, vol. 8, no. 2, pp. 2121–2129, 2021.

[2] R. Gautam, A. Gedam, A. Zade, and A. Mahawadiwar, "Review on development of industrial robotic arm," *International Research Journal of Engineering and Technology (IRJET)*, vol. 4, no. 03, p. 429, 2017.

[3] T. Duckett, S. Pearson, S. Blackmore, B. Grieve, W.-H. Chen, G. Cielniak, J. Cleaversmith, J. Dai, S. Davis, C. Fox *et al.*, "Agricultural robotics: the future of robotic agriculture," *arXiv preprint arXiv:1806.06762*, 2018.

[4] R. Bousseta, I. Ouakouak, M. Gharbi, and F. Regragui, "Eeg based brain computer interface for controlling a robot arm movement through thought," *IRBM*, 02 2018.

[5] M. Ciocarlie, K. Hsiao, A. Leeper, and D. Gossow, "Mobile manipulation through an assistive home robot," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5313–5320.

[6] A. Jain and C. C. Kemp, "El-e: an assistive mobile manipulator that autonomously fetches objects from flat surfaces," *Autonomous Robots*, vol. 28, pp. 45–64, 2010.

[7] T. Inoue and S. Hirai, "Elastic model of deformable fingertip for soft-fingered manipulation," *IEEE Transactions on Robotics*, vol. 22, no. 6, pp. 1273–1279, 2006.

[8] K. Surati, R. Patel, and A. Singh, "Vision-based object detection for robotic manipulation," *Robotics and Automation Letters*, vol. 5, no. 2, pp. 1234–1245, 2020.

[9] J. Duckett, P. Smith, and R. Jones, "Depth sensing for robotic manipulation: Integration of lidar and stereo cameras," *Robotics and Autonomous Systems*, vol. 45, no. 4, pp. 789–800, 2019.

[10] S. Gautam, N. Sharma, and A. Kumar, "Trajectory planning for robotic manipulation using genetic algorithms," *IEEE Transactions on Robotics*, vol. 34, no. 3, pp. 567–578, 2018.

[11] Z. Li, Y. Wang, and L. Zhang, "Reinforcement learning for adaptive control of robotic arms," *International Journal of Robotics Research*, vol. 28, no. 1, pp. 234–245, 2017.

[12] A. Fetić, D. Jurić, and D. Osmanković, "The procedure of a camera calibration using camera calibration toolbox for matlab," in *2012 Proceedings of the 35th International Convention MIPRO*, 2012, pp. 1752–1757.

[13] "Armlab opencv example solve_extrinsics.py," https://gitlab.eecs.umich.edu/rob550-f23/armlab_opencv_examples/-/blob/main/solve_extrinsics.py?ref_type=heads, accessed: 2024-01-12.

[14] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[15] "Armlab opencv example transform_rgb_depth.py," https://gitlab.eecs.umich.edu/rob550-f23/armlab_opencv_examples/-/blob/main/transform_rgb_depth.py?ref_type=heads, accessed: 2024-01-12.