Development of a 4-DOF Serial Robotic Manipulator for Pick-and-Place Applications

Kai Nakamura, Owen Sullivan, and Evan Carmody

***Abstract*—This paper presents our development of a 4-DOF serial robotic manipulator designed for pick-and-place tasks. We discuss robot kinematics, camera calibration, transforming 2D pixel coordinates into 3D world coordinates, object detection and classification, and object localization challenges. We conclude with insights about our final system and discuss is applicability in the field of robotics.**

I. Introduction

Our primary goal was to create a ball-sorting robot using a robotic arm and a camera. A variety of colored, 3D-printed balls are arbitrarily placed on the workspace, and the robot is tasked with locating the balls and determining the trajectory to grab and sort the different colors. This project required us to explore and implement forward kinematics, inverse kinematics, trajectory generation, and a robust computer vision system.

The robotic arm we used was the OpenMANIPULATOR-X, an open-source and low-cost serial manipulator [1]. The OpenMANIPULATOR-X has four revolute joints and four degrees of freedom. The camera we used was a standard USB webcam [2]. The camera we picked used a wide-angle fisheye lens which we had to account for in Section II-C.

II. Methodology

A. Forward Kinematics

In order to control our robotic arm and move it around the workspace, we needed to solve the forward kinematics of our robot so we could determine the end-effector position given the current joint angles. We used the Denavit-Hartenberg (DH) convention to model the forward kinematics of our arm [3]. We started by assigning frames of reference to each of the joints (F_1, F_2, F_3, and F_4) as well as the base of the robot (F_0) and the end-effector (F_5) as shown in Fig. 1. Note that the frames F_0 and F_1 coincide with one another. From these frames, we found the DH parameters of the OpenMANIPULATOR-X and used them to calculate the transformation matrices between each of the joints (Table I).

TABLE I: Denavit-Hartenberg (DH) Parameters of the OpenMANIPULATOR-X

T_i^{i+1}	$\theta\ [rad]$	$d\ [mm]$	$a\ [mm]$	$\alpha\ [rad]$
T_1^2	θ_1^*	96.326	0	$\frac{\pi}{2}$
T_2^3	$\theta_2^* + \tan^{-1}(\frac{128}{24})$	0	130.23	0
T_3^4	$\theta_3^* - \tan^{-1}(\frac{128}{24})$	0	124	0
T_4^5	θ_4^*	0	133.4	0

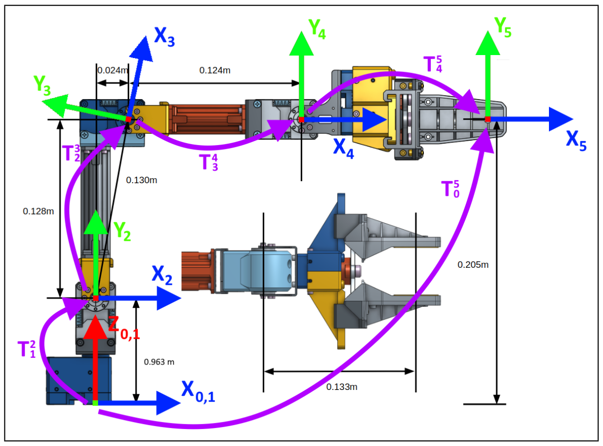


Fig. 1: Denavit-Hartenberg (DH) Reference Frames of the OpenMANIPULATOR-X

Multiplying all of the transformation matrices together will produces a single transformation matrix between the base frame of the robot to the end-effector (1). This final transformation matrix is the forward kinematics solution for our robot since it contains the end-effector's position and orientation relative to the base frame of the robot. Equation 2 shows the forward kinematics solution for the position of the end-effector. Note that s and c are short for sin and cos, and that θ_{23} and θ_{234} are short for $\theta_2 + \theta_3$ and $\theta_2 + \theta_3 + \theta_4$.

$$T_0^5 = T_0^1 \cdot T_1^2 \cdot T_2^3 \cdot T_3^4 \cdot T_4^5 \tag{1}$$

$$\vec{p}_{ee} = \begin{bmatrix} c(\theta_1)[133c(\theta_{234}) + 124c(\theta_{23}) + 130s(\theta_2)] \\ s(\theta_1)[133c(\theta_{234}) + 124c(\theta_{23}) + 130s(\theta_2)] \\ 96.3 + 133s(\theta_{234}) + 124s(\theta_{23}) + 130s(\theta_2) \end{bmatrix} \tag{2}$$

B. Inverse Kinematics

Forward kinematics is great for figuring out where the end-effector is from the given joint angles. But in order to pick up balls at a given position, we need to be able to work backwards to find the desired joint angles. This is where inverse kinematics comes into play.

Unfortunately, calculating the inverse kinematics is a lot more complicated because there might be multiple or even infinite solutions for a single position. There might even be no solutions if the position isn't reachable. But, the general idea is to use trigonometry to calculate possible joint angles, and then plug those values back into the forward kinematics to see which configurations actually work.

Since the given position has three coordinates (x, y, and z) and there are four unknown joint angles, the angle α is also provided to make the problem solvable where α is the angle between the end-effector and the xy-plane.

From Fig. 2:

$$\begin{aligned} r &= \sqrt{x_c^2 + y_c^2} \\ r\cos(\theta_1) &= x_c \\ D_1 = \cos(\theta_1) &= \frac{x_c}{r} \\ \theta_1 &= atan2(\pm\sqrt{1 - D_1^2}, D_1) \end{aligned} \tag{3}$$

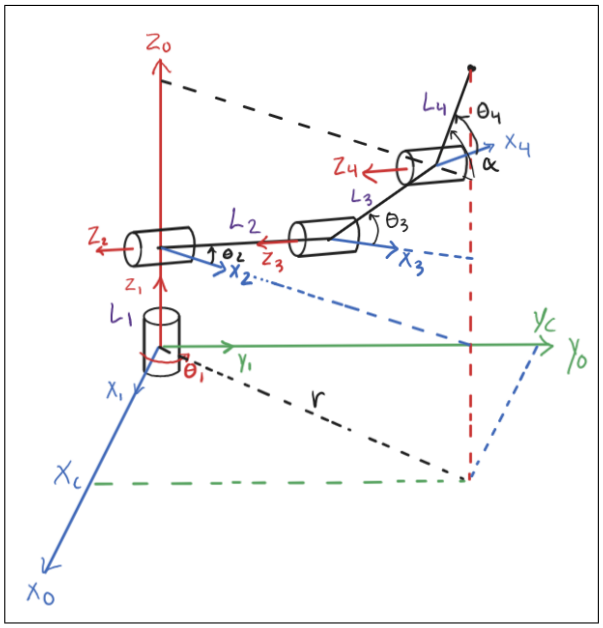


Fig. 2: Diagram of the Robot at an Arbitrary Pose

From Fig. 3:

$$\begin{aligned} x_{24} &= r - L_4\cos(\alpha) \\ z_{24} &= z_c - L_1 - L_4\sin(\alpha) \end{aligned}$$

Using the Law of Cosines:

$$\begin{aligned} \cos(\pi - \theta_3) &= \frac{L_2^2 + L_3^2 - (x_{24}^2 = Z_{24}^2)}{2L_2L_3} \\ D_3 = \cos(\theta_3) &= -\frac{L_2^2 + L_3^2 - (x_{24}^2 = Z_{24}^2)}{2L_2L_3} \\ \theta_3 &= atan2(\pm\sqrt{1 - D_3^2}, D_3) \end{aligned} \tag{4}$$

Using the Law of Cosines:

$$\begin{aligned} D_\beta = \cos(\beta) &= \frac{L_2^2 + (x_{24}^2 = Z_{24}^2) - L_3^2}{2L_2\sqrt{x_{24}^2 = Z_{24}^2}} \\ \beta &= atan2(\pm\sqrt{1 - D_\beta^2}, D_\beta) \end{aligned} \tag{5}$$

$$\begin{aligned} D_\phi = \cos(\phi) &= \frac{x_{24}}{\sqrt{x_{24}^2 + y_{24}^2}} \\ \phi &= atan2(\pm\sqrt{1 - D_\phi^2}, D_\phi) \end{aligned} \tag{6}$$

$$\theta_2 = \phi - \beta \tag{7}$$

$$\theta_4 = \alpha - \theta_2 - \theta_3 \tag{8}$$

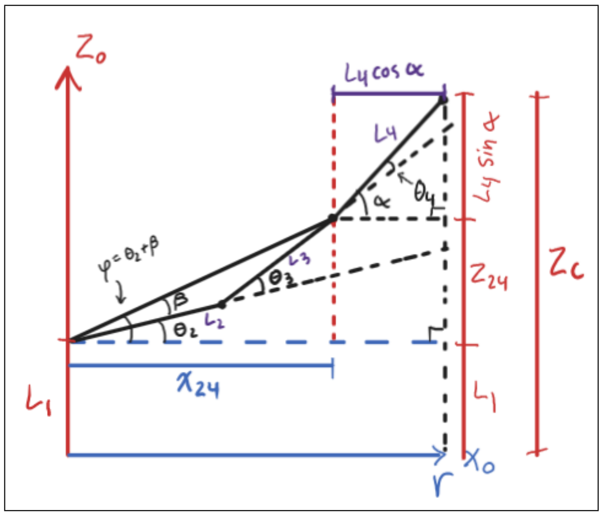


Fig. 3: Diagram of Joints 2, 3, and 4 from the XR-Plane

Equations for θ_1, θ_2, θ_3, and θ_4 all use $atan2$ to solve for all possible solutions of inverse kinematics. Not all candidates are valid solutions, in fact most are not. To determine which combinations of joint angles are actually valid, we plug the candiates back into the forward kinematics to see if the joint angles bring the end-effector to the desired position.

C. *Camera Calibration*

Proper camera calibration is necessary to ensure accurate and reliable visual perception of the workspace. The first step of the camera calibration process is to remove any distortion that the camera lens applies to the image. The camera we used has a wide-angle fisheye lens that produced significant distortion near the edges of the image. This distortion can make objects appear stretched or warped which causes inaccuracies in object detection and localization. Notice how the straight red lines in Fig. 4 don't line up with the edges of the checkerboard.

To correct for this distortion, we used MATLAB's Camera Calibrator app to calculate our camera's intrinsic parameters and remove the effects of lens distortion. The Camera Calibrator uses several calibration images each taken from a different perspective to estimate various parameters such as the focal length and lens distortion coefficients.

The Camera Calibrator detects the intersection points of the checkerboard pattern and uses these to establish a correlation between 2D pixel coordinates and 3D world coordinates. The

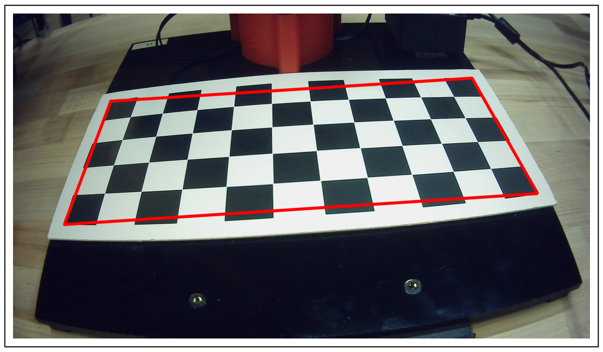


Fig. 4: Distorted image taken by a fisheye lens camera

app estimates the camera parameters by minimizing reprojection error which is the difference between the observed image points and their projected world coordinates. For our set of calibration images, the Camera Calibrator app was able to estimate camera parameters that reduced the mean reprojection error per image down to 0.51 pixels (Fig. 5).

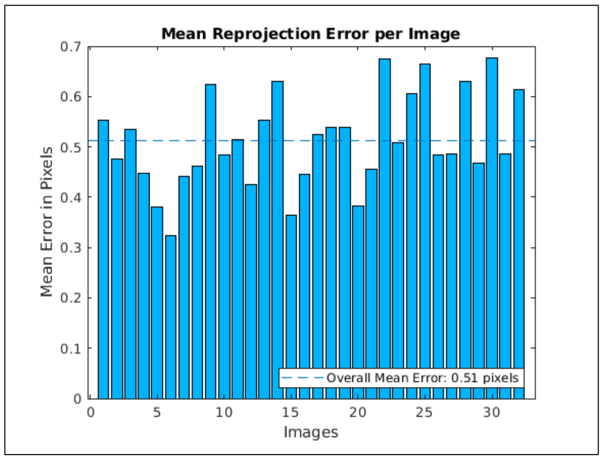


Fig. 5: Mean reprojection error of calibration images

The Camera Calibrator's estimated locations of the camera from the calibration images can be seen in Fig. 6. These estimations closely mirror the real world positions that our calibration images were taken from as we moved our camera around the checkerboard.

Once we had calculated the intrinsic parameters of the camera, we could use them to remove the fisheye lens distortion. Notice how the straight red lines in Fig. 7 now line up with the edges of the checkerboard, unlike how they did previously in Fig. 4.

Using the parameters of our calibrated camera, we could then create a transformation matrix that converts 2D pixel coordinates from an image and translates them into 3D world coordinates that lie on the xy-plane of the checkerboard. This transformation matrix is called, $T_{image}^{checker}$, since it maps pixel coordinates of the image to world points on the checkerboard.

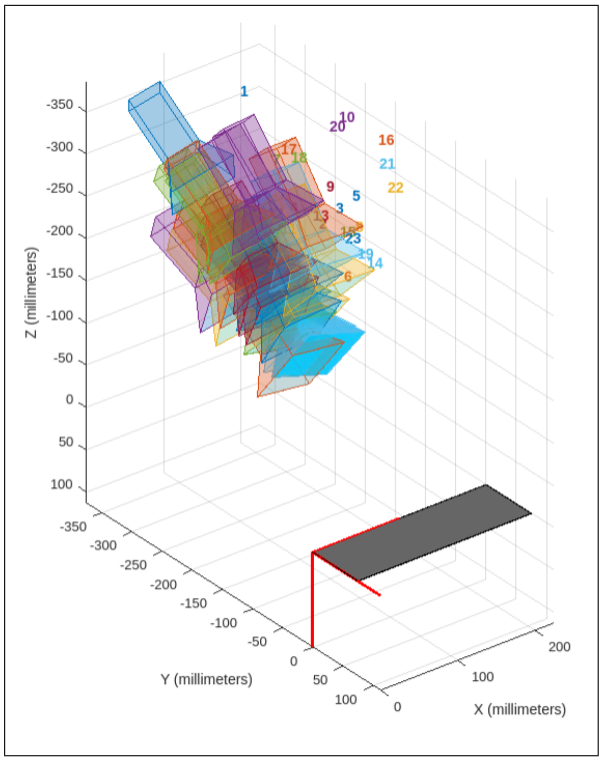


Fig. 6: Estimated locations of the camera from the Camera Calibrator app

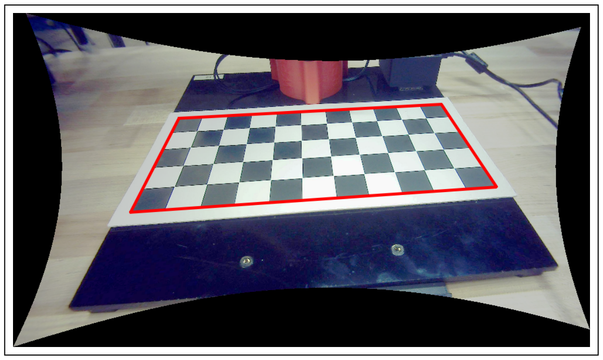


Fig. 7: Undistorted image of the checkerboard

D. *Camera-Robot Transformation*

In order for the robot to pick up an object, it has to know where the object is in respect to its origin. However, world points obtained from the calibrated camera lie in the reference frame of the checkerboard, $F_{checker}$, and not in the reference frame of the robot, F_0 (Fig. 8). Points with respect to the checkerboard can be transformed into points with respect to the robot by finding the transformation matrix that maps frame 0 of the robot to the checkerboard frame, ${T_0}^{checker}$.

The transformation matrix between the robot and checkerboard can be calculated by finding the rotation and translation between the robot and checkerboard. Equation 9 shows the rotation matrix, $R_0^{checker}$, obtained by visual inspection of the robot frame and checkerboard frame (Fig. 8). Equation 10 shows the translation matrix, $\vec{p}_0^{\,checker}$, obtained by measuring the distance between the origin of the robot and the origin of the checkerboard. Equation 11 shows the final transformation matrix, $T_0^{checker}$, obtained by combining $R_0^{checker}$ and $\vec{p}_0^{\,checker}$.

$$R_0^{checker} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{9}$$

$$\vec{p}_0^{\,checker} = \begin{bmatrix} 113 \\ -95 \\ 0 \end{bmatrix} \tag{10}$$

$$T_0^{checker} = \begin{bmatrix} 0 & 1 & 0 & 113 \\ 1 & 0 & 0 & -95 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11}$$

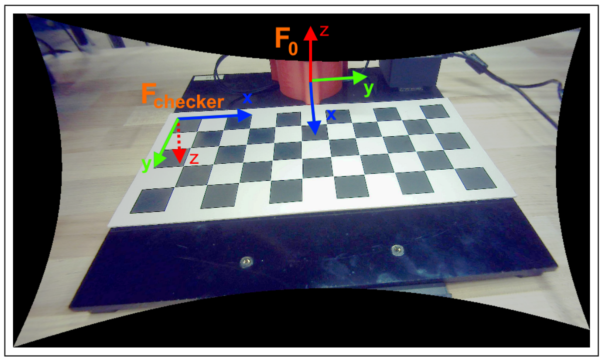


Fig. 8: Reference frames of the robot (F_0) and the checkerboard ($F_{checker}$)

With $T_0^{checker}$ from Section II-D and $T_{image}^{checker}$ from Section II-C, 2D pixel coordinates from the image can now be fully converted into 3D world coordinates in the reference frame of the robot (12).

$$\vec{p}_0 = T_0^{checker} \cdot T_{checker}^{image} \cdot \vec{p}_{image}$$

$$\vec{p}_{image} = T_{image}^{checker} \cdot T_{checker}^{0} \cdot \vec{p}_0$$

$$\vec{p}_{image} = T_{image}^{0} \cdot \vec{p}_0 \tag{12}$$

E. Object Detection and Classification

Detecting objects from an image and classifying them requires several stages of image processing to extract meaningful information that can be used for decision-making. Each step of an image processing pipeline performs a distinct operation that enhances or extracts desired information or reduces irrelevant details.

Our robot was designed to pick up small colored balls (red, orange, yellow, green, and gray) and sort them based on their color (Fig. 9). To achieve this, we created an image processing pipeline that could extract the pixel coordinates of each ball and classify its color. The gray colored balls proved challenging to detect since they blend in with the white and black checkerboard, unlike the other brightly colored balls. This required us to develop a robust image pipeline capable of identifying all five colors.

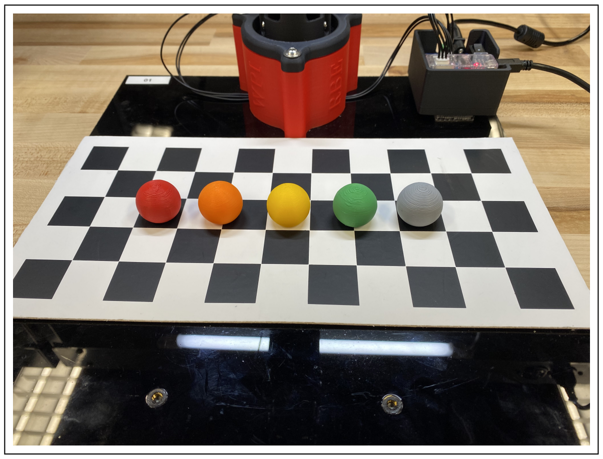


Fig. 9: Red, orange, yellow, green, and gray colored balls

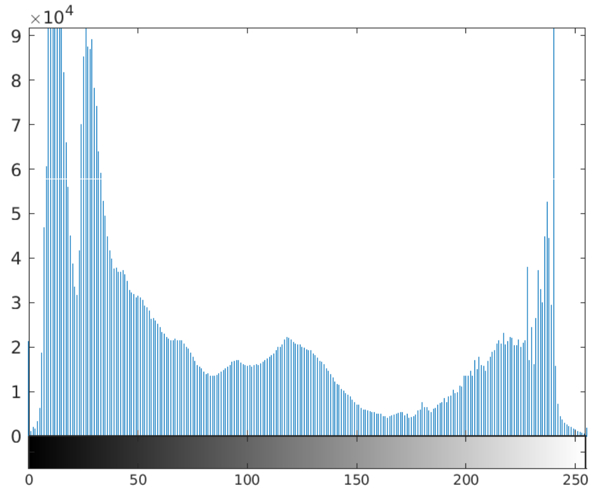
However, there are some objects that even the most robust image pipelines can't detect. We were able to identify the gray balls because they can still be differentiated against the white and black checkerboard. It would be near impossible to detect a white ball or black ball since it would completely blend in with the checkerboard. Different colors that are very similar can also cause difficulties if they are not easily discernible from one another.

1) Brightness Equalization: The first step of our pipeline is to equalize the brightness of the raw image. Later stages in the pipeline rely heavily on predefined color thresholds which can be thrown off if the brightness of the image changes due to the ambient light in the room. To account for this, we apply a histogram equalization to evenly distribute the brightness of the image (Fig. 10). Brightness equalization will make an underexposed image brighter and an overexposed image darker so that the vision pipeline stays consistent no matter what the lighting conditions are (Fig. 11).

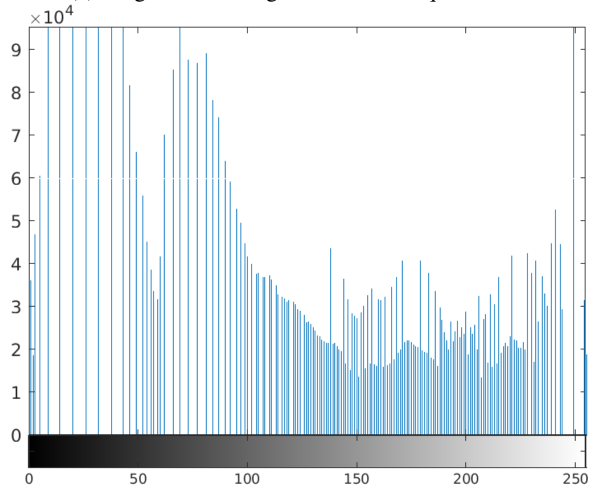
2) Undistortion: We then undistort the fisheye lens using the calibrated camera parameters from Section II-C. It is important to perform this step after the brightness equalization so that the black edges introduced from undistortion don't influence the brightness histogram.

3) Image Masking: Next, the image is color masked to separate out the balls from the background. This operation is done in the HSV color space which we found was the most intuitive to work with. There are two filters applied to the image, one for the colored balls and one for the gray ball. Both filters are then combined with one another to get a mask that contains all five colored balls.

We found the values for our filters by using MATLAB's Color Thresholder app. The colored ball filter removed colors with low saturation and low values to get rid of the black and

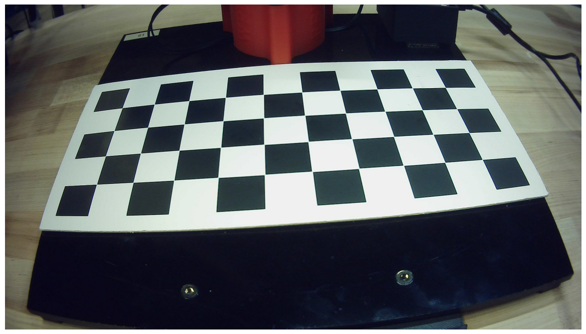


(a) Brightness Histogram Without Equalization

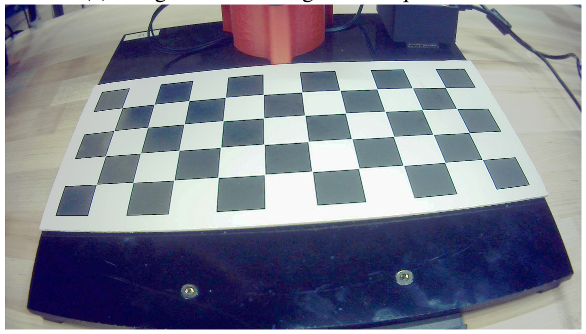


(b) Brightness Histogram With Equalization

Fig. 10: Brightness Histograms With and Without Equalization



(a) Image Without Brightness Equalization



(b) Image With Brightness Equalization

Fig. 11: Images With and Without Brightness Equalization

white checkerboard. This left behind only the brightly colored balls, but also removed the gray balls.

To add the gray balls back in, a second filter band passes only the mid-ranged values found in the gray ball. The outputs of both filters combined creates a mask containing all the balls.

Under certain lighting conditions, some regions of the checkerboard match the gray values found in the ball which creates artifacts in the masked image. However, these artifacts are cleaned up in the next step which is identifying circles.

4) Identifying Circles: To remove noise from the masked image and pick out the colored balls, we used MATLAB's circle finding algorithm. We tried a variety of different approaches such as eroding the image and using edge detection, but we found these methods unreliable. With MATLAB's imfindcircles function we were able to achieve very consistent results even under different lighting conditions.

The circle finding algorithm requires more processing power and so is slower than other methods. However, for our purposes we only needed a single image of the workspace for each run. For live tracking of objects, we found it better to use other methods such as erosion and edge detection for increased speed.

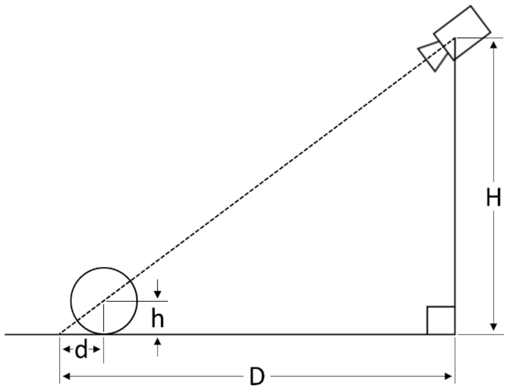
5) Classifying Circles by Color: Next, we take each of the identified circles and determine which of the five colors (red, orange, yellow, green, or gray) it is closest to. The first method we tried was finding the average color of the circle and calculating its Euclidean distance to each of the expected target colors. Then, whichever target color had the shortest distance would be the color of that circle.

We tried this method in HSV, RGB, and L*a*b* color spaces, but none of these produced the results we needed. This method had a lot of trouble differentiating between similar colors and was not reliable enough to be used.

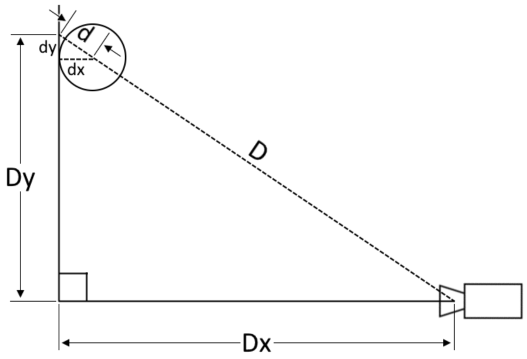
Instead, we decided to use a different approach that was more computationally expensive, but much more reliable. We created five different color filters for each of the target colors. Each of the color filters is applied to the circles one at a time. Whichever color filtered circle contains the most pixels is the most prevalent color of that circle. We found that this method was the most reliable at differentiating between the different colors.

F. Object Localization

At this stage in the image pipeline, we've obtained the location of each ball and classified its color. There is one last problem that needs to be addressed to get the location of the ball. When the 2D pixel coordinates of the image are transformed into 3D world coordinates, the image is projected directly onto the xy-plane of the checkerboard. However, the actual ball sits on top of the checkerboard with some fixed height and is not lying flat on the checkerboard like a piece of paper (Fig. 12).



(a) Illustration of projection error from side



(b) Illustration of projection error from top

Fig. 12: Illustrations of projection error from different perspectives

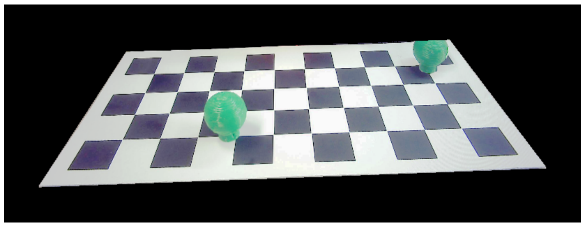
If this projection error is not accounted for, then the robot will attempt to pick up the balls slightly behind where they actually are on the checkerboard. To solve this issue, we calculate the actual position of the balls by using similar triangles:

$$d = D\frac{h}{H}$$

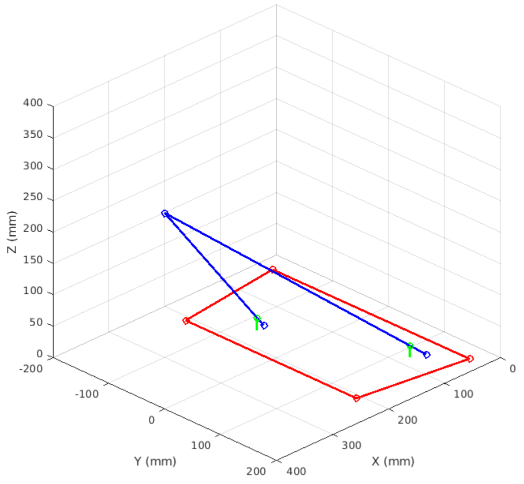
$$d_x = d\frac{D_x}{D} \tag{13}$$

$$d_y = d\frac{D_y}{D} \tag{14}$$

Using the x and y offsets from 13 and 14, the actual position of the object is obtained as seen in Fig. 13.



(a) Image for Object Localization



(b) Object Localization Visualization

Fig. 13: Object localization being run on image after image processing pipeline, with accompanying visualization.

G. Pick-and-Place

At this point, we now had all the components we needed for our pick-and-place system, we just needed to put everything together. First, the arm is moved out of view of the camera and the camera captures an image of the workspace. The image is processed using our vision pipeline to pick out the colored balls (Section II-E). The camera parameters are used to project the pixel coordinates of the balls onto the checkerboard (Section II-C). Then, the checkerboard coordinates are transformed into robot coordinates (Section II-D). A slight adjustment is made for the height of the ball (Section II-F). Inverse kinematics is used to send the robot to pick up the ball (Section II-B). And finally, the arm drops the ball off in the appropriate delivery zone based on its color. With all these steps combined, we finally had a fully-fledged design for picking up and sorting the colored balls.

All of the movements were implemented using joint space cubic trajectories. This was done over task space and/or quintic trajectories because they were computationally faster. The added benefits of smoother acceleration or moving the end-effector in straight lines weren't necessary for our use cases.

III. Results

The methods outlined in this paper resulted in a very successful pick-and-place robot. In addition to this paper, a video was made to document the development of our arm, which can be seen here.

Our arm was successfully able to separate red, orange, yellow, green, and gray balls consistently in a variety of different lighting conditions. It was also able to process an arbitrary number of balls so long as they fit onto the workspace with sufficient room for the end-effector motion.

Additionally, we were also able to implement live tracking of moving targets. By using a tweaked version of our vision pipeline, we were able to speed up the image processing to allow our robot to dynamically track moving balls and pick them up once they stopped moving.

IV. Discussion

Completing this project gave us a glimpse into current and future applications for pick-and-place robotic manipulators. Robots similar to the OpenMANIPULATOR-X are being used in a multitude of other fields. For example, there are many potential uses for such robots in the fields of manufacturing, packaging, automotive, aerospace, healthcare, and even agriculture.

Pick-and-place robots have countless benefits that contribute to their efficiency in these fields. Most notably, these robots are capable of increased and continuous productivity. In comparison to a human performing manual labor, a robot is much more efficient and productive. Robots are able to perform actions with increased precision and accuracy. They are programmed to complete repeatable tasks and are especially practical in the fields of electronics manufacturing and pharmaceuticals, where small components require the utmost accuracy during assembly. This results in increased product quality and in turn, increased customer satisfaction.

Another benefit of large-scale pick-and-place robotics is their adaptability and versatility. Interfaces for industrial robots are made to be relatively simple to program, and because of this, a different end-effector or different programming allows for a robot to be easily adapted for the task at hand. A robotic manipulator designed for assembling small electronic components could be altered to move and package boxes, and then altered again to transport fragile chemical equipment.

Increased versatility, accuracy, and reliability result in increased throughput, reduced cycle time, and heightened cost-efficiency. These aspects are desirable for businesses and companies in every field. The culmination of these characteristics and the continual improvement of robotic manipulators shows the increasing desire for automation in a plethora of fields.

Although sorting colored balls is a relatively simple task in the general sense, working with the OpenMANIPULATOR-X robotic arm and completing this project helped us learn about the great benefits of robotic manipulators.

V. Conclusion

Throughout this project, we were able to explore and apply a variety of techniques related to robotic manipulators. These topics included forward and inverse kinematics, trajectory and path generation, differential kinematics, and computer vision. We began by solving for forward and inverse kinematic solutions of our arm. We then calibrated our camera using MATLAB's Camera Calibrator app to remove fisheye distortion, allowing for accurate conversion of pixel coordinates to world coordinates. Next, we used the transformation matrix between the checkerboard and robot to transform coordinates into the frame of the robot. We then developed an image processing pipeline complete with brightness equalization, distortion correction, image masking, circle detection, object classification. Lastly, we applied object localization to correct for the height of the balls. With everything combined, we were left with our very own pick-and-place robotic manipulator (Fig. 14).

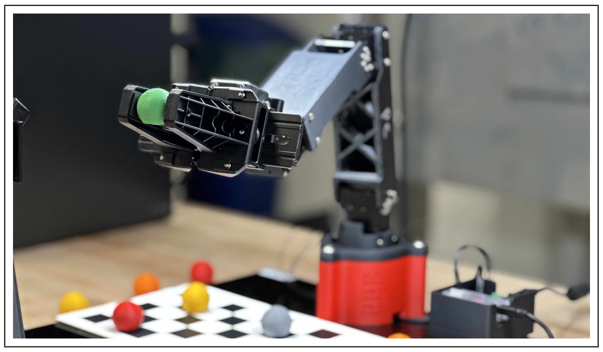


Fig. 14: Our Pick-and-Place Robot in Action

References

[1] Robotis. OpenMANIPULATOR-X Overview, https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/

[2] ELP USB Security Camera Low Light 1080P Sony IMX323 HD Sensor H.264 Voice Recording Pinhole Spy Camera For Video Conference. https://webcamerausb.com

[3] J. Denavit and R. S. Hartenberg, "A kinematic notation for lower-pair mechanisms based on matrices," 1955.