Lab 5: Final Project – Robot Pick and Place System

Nicolas C.B. Graham, Pranav S. Jain, Fiona K. Prendergast

Abstract— **This project aimed to design a robotic pick-and-place system integrating computer vision for object detection and a robotic arm to sort objects based on color and shape. Using inverse and forward kinematics, trajectory generation, and velocity kinematics, the robot successfully performed the task with an estimated accuracy rate of about 99%.**

INTRODUCTION

Robotic systems have completely transformed automation in industries today, with pick-and-place robots playing a huge role in streamlining processes like manufacturing and logistics. These systems do the heavy lifting by identifying, grabbing, and relocating objects with precision, boosting productivity while minimizing human error. For our final project, we built a robotic pick-and-place system using the OpenManipulator-X robotic arm. The key here was integrating computer vision, which allowed the robot to detect and locate objects on its own.

This project draws from concepts we worked on throughout the course. In Lab 1, we focused on getting the basics down—establishing communication with the robot and controlling its joints using MATLAB commands. Lab 2 took things up a notch with forward kinematics, which let us figure out the robot's position in space so it could move to specific target points. Lab 3 was all about inverse kinematics, helping us calculate the joint angles needed for the robot's end-effector to reach a set position. We also used trajectory generation to ensure smooth transitions between points. By the time we hit Lab 4, we were working with velocity kinematics and using Jacobian matrices to control the robot's speed and direction while it followed a desired path.

For the final project, we put all these pieces together to create a full pick-and-place system. We used a camera for intrinsic calibration and applied computer vision techniques to identify objects based on color. From there, the robot had to pinpoint the objects' location, grab them with its gripper, and place them in predefined spots. This required blending kinematic control, trajectory planning, and real-time decision-making. This report breaks down the design, implementation, and testing of the system, and looks at how well it performed and where we could make improvements.

METHODOLOGY

Camera Calibration

The initial step in the calibration process involved performing intrinsic calibration to determine the camera matrix, which is crucial for accurately mapping the 2D image captured by the camera to real-world 3D space. To achieve this, we used a checkerboard pattern with known square sizes, which provided a reference for real-world dimensions. By taking multiple images of the checkerboard from various angles, MATLAB was able to analyze how the checkerboard appeared in different parts of the camera's field of view.

The calibration process calculated the pixel size of the checkerboard in each image and compared it to the actual dimensions of the physical checkerboard. These pixel-to-real-world transformations were essential in determining how the camera perceives depth and scale. Using these transformations, MATLAB computed the camera matrix, which includes parameters such as focal length and optical center. This matrix allows the camera to accurately interpret object positioning in 3D space, completing the intrinsic calibration.

As MATLAB does the intrinsic calibration for us, we look at the mean errors within each image in the form of a histogram given by MATLAB and deselect images with higher mean errors until the overall mean error was less than 0.5. The overall error within our calibration was 0.43 as seen in figure 7.

Ball Detection:

A new class called Image_Handling was developed to manage all of the image processing required by the robot for ball detection. The rationale for creating this class was to modularize the image processing tasks, allowing for efficient handling of different image-based operations, such as filtering, masking, and color segmentation, while also keeping the code clean and reusable. Given the critical role of accurate object detection in pick-and-place systems, this approach enabled us to isolate and debug issues more easily, ultimately improving the performance of the robot's decision-making process.

Once the camera was calibrated, the next step was to mask the empty checkerboard in the workspace. The checkerboard serves as the reference frame for the robot to localize objects. By masking the checkerboard, we ensured that the robot's focus was restricted to this area, preventing it from detecting or responding to any foreign objects or background noise outside the grid. This masking process helped to reduce false positives and improve the accuracy of the ball detection algorithm, especially when multiple objects were present in the scene. The checkerboard was masked by using the detectCheckerboardPoints() method to get inner vertices on the board, converting the corner vertices to checkerboard frame and offseting them by 25 mm to get the outer corners with an extra offset of 11 mm to get the white padding outside the checkerboard before converting back into the image frame. These new values were used to create a new polygon which was then converted into the checkerboard mask.

The use of color was a key element in our approach to detecting the balls. We specifically chose grey balls to test the robustness of the robot's ability to detect darker objects, which pose unique challenges in computer vision. The color contrast between the balls and the checkerboard played a crucial role in determining the success of the detection process. Lighter colors are easier to detect due to their higher contrast, but darker objects tend to blend into the background more easily, making detection more challenging. This prompted the need for more refined color filtering techniques to ensure accurate ball detection in varying lighting conditions.

To achieve this, we used the getColorCoordinates() function in the Image_Handling class to obtain the coordinates of the balls based on their colors. The detection process began by applying a series of filters to the images captured by the camera. The first filter utilized the HSV (Hue, Saturation, Value) color space, which is well-suited for color-based object detection, as it separates chromatic content (color) from intensity, making it more resilient to lighting changes. This color filtering step allowed the robot to distinguish the colored balls from the checkerboard grid and from each other, ensuring the robot was aware of which ball it was picking up based on its color.

After color filtering, we applied an erode filter to the image. The purpose of the erode filter was to remove small noise and separate balls that were positioned close together. By removing outer layers, this filter prevented the robot from mistakenly identifying clusters of balls as a single object. Once the balls were sufficiently separated, a close filter was applied to complete the spherical shapes of the balls, ensuring that any gaps or inconsistencies were filled in. This was followed by a fill filter, which further refined the image, making the balls more defined and ready for final processing.

Finally, the regionprops() function was used to calculate the centroids of the detected balls. This function extracts the properties of labeled regions in the binary image, such as area, orientation, and centroid. The coordinates of the centroids were then transformed into the checkerboard frame, allowing the robot to precisely locate and move toward each ball.

For detecting the grey ball, we encountered additional challenges due to its similarity to the white and black colors of the checkerboard and the shadows cast by the balls, which were also shades of grey. To overcome this, we applied a grey color filter in the L*a*b* color space. This color space proved to be more effective in isolating grey shades compared to HSV, as it separates color information more distinctly. After applying the erode filter to remove noise, we used the imfindcircles() method to detect the grey balls. This method allowed us to filter out grey regions that were not circular, focusing only on the round shapes of the balls and improving the accuracy of grey ball detection.

Ball Detection Error:

As these images are taken at an angle, the centroid returned by the camera has an error of roughly 13mm, changing slightly based on the ball's position on the checkerboard. To calculate and eliminate this error, a vector from the supposed centroid of the ball to the camera is subtracted from the base of the camera in the checkerboard frame. Then, the magnitude of this vector is calculated, the radius of the ball is measured, and is used along with the concept of similar triangles to get this error. Hence, to calculate the real centroid of the ball, a unit vector from the supposed centroid of the ball to the camera base is calculated by dividing the vector by its magnitude which is then multiplied by the error and then added the supposed centroid of the ball.

$$Checker2CamBase = \begin{bmatrix} 112.5 \\ 277 \\ 0 \end{bmatrix}$$

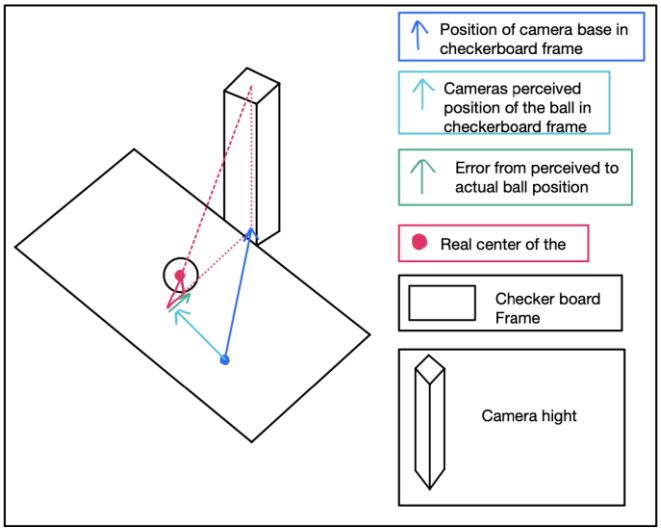

Figure 1: Diagram of real-world ball location calculation.

Other Physical Errors:

After the error of the centroids of the ball has been dealt with, the target position for the arm is set by converting the coordinates from the checkerboard frame to the robot base frame by multiplying the by trans0toChecker transformation matrix.

$$trans0toChecker = \begin{bmatrix} 0 & 1 & 0 & 85 \\ 1 & 0 & 0 & -112.5 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Due to the minor errors in the length of the robot's links, the translation of the base frame of the robot to the checkerboard, the translation of the checkerboard frame to the camera base, the variable slop in the motors of the robotic arm, there was a secondary error that needed to be corrected to get the target position. A vector from the robot's base to the target position was calculated and the unit vector was calculated by dividing the vector by its magnitude. This unit

vector was multiplied by the error and then added to the previous target position to get the correct target position.

Choice of Trajectory Planning

We had the choices of joint space trajectory planning and task space trajectory planning. Each has different benefits and drawbacks. We start using interpolate_jp to start at the home position, once there is a ball detected we use task space trajectory planning to move from home to a position directly above the ball. Once at a pre-determined Z offset above the ball location, we use task space trajectory planning to descend and then grab the ball. Once we have grabbed the ball with the gripper, we use task space trajectory planning to translate along the Z axes to the predetermined Z offset, once at the z offset we use joint space trajectory planning to move from above the ball to the desired drop location, each colored ball with its own drop location.

Using task space trajectory planning allows for the robot's end-effector to move in a straight line which would be especially useful while grabbing the ball and moving away from the checkerboard as the gripper shouldn't disturb the other balls close to the target ball. In other places, any type of trajectory planning could theoretically be used as the robot is moving in a more open space.

Derivation of Forward Kinematics

The forward kinematics of the OpenManipulator-X robot were derived using the Denavit-Hartenberg (DH) convention, which assigns coordinate frames to each joint and defines the geometric relationship between consecutive links.

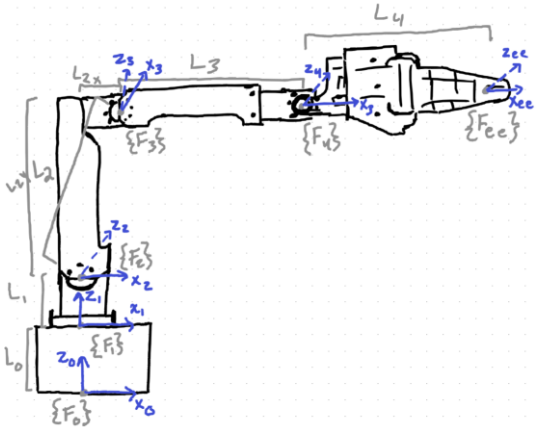

Figure 2: Robot arm in the home position with the frames 0-4 and end effector frame assigned, and the link lengths labeled.

By determining the four DH parameters for each joint—joint angle, offset, link length, and twist angle—we calculated the transformation matrices that describe the position and orientation of each joint relative to the previous one.

Link	theta [deg]	d [mm]	a [mm]	alpha1 [deg]
1	theta1	L1	0	-90
2	theta2 – 79.38	0	L2	0
3	theta2 + 79.38	0	L3	0
4	theta4	0	L4	0

Figure 3: Table of the DH Parameters for each link in the robot arm (not including link 0). The 'theta' and 'alpha' parameters are in degrees and the 'd' and 'a' parameters are in millimeters.

Link	Lengths [mm]
L0	36.076
L1	60.25
L2x	24.0
L2y	128.0
L2	130.231
L3	124
L4	133.4

Figure 4: Table of the lengths of each of the robot links in millimeters from the lengths given in Lab 1. L2 was calculated using L2x and L2y and the Pythagorean theorem.

$$FK\ Trans = \begin{bmatrix} cos(\theta) & -\sin(\theta) * \cos(\alpha) & \sin(\theta) * \sin(\alpha) & a * \cos(\theta) \\ \sin(\theta) & \cos(\theta) * \cos(\alpha) & -\cos(\theta) * \sin(\alpha) & a * \sin(\theta) \\ 0 & \sin(\alpha) & \cos(\alpha) & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

These matrices were then multiplied to compute the overall transformation from the robot's base to its end effector. This final transformation allowed us to determine the end effector's position and orientation in the workspace based on the given joint angles. The accuracy of the forward kinematics was verified by comparing the calculated results with the actual robot movements, confirming that the model is reliable for future tasks like inverse kinematics and trajectory planning.

Derivation of Inverse Kinematics

The inverse kinematics of the robotic arm was derived using a geometric approach to calculate the joint angles required to reach a given position and orientation of the

end effector in space. This method was chosen over the algebraic approach to avoid issues with infinite solutions that arise in a 4-DOF system. By defining the task space as [X, Y, Z, alpha] and the joint space as [q1, q2, q3, q4], the angles were calculated step by step. The process began with determining theta_1 using the law of cosines and MATLAB's atan2 function to find all valid solutions. Next, theta_2 and theta_3 were calculated by forming geometric triangles and again applying atan2 to account for multiple solutions. Finally, theta_4, which determines the wrist angle, was computed based on the desired orientation of the end effector.

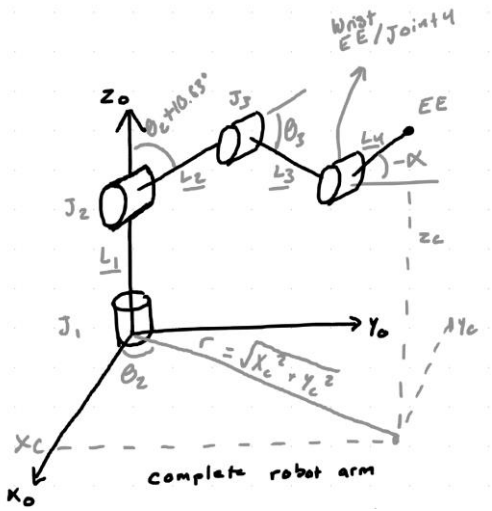

Figure 5: Stick diagram of the 4-DOF robot arm in frame 0 with joints 1-4, links 1-4 and joint values (Θ1, Θ2, Θ3, Θ4) all labeled

This geometric method was implemented in MATLAB using the ik3001() function, which solves for all joint angles given a target position. The results were validated by comparing the calculated joint angles with the robot's forward kinematics function, ensuring the accuracy of the solution. This derivation is essential for precise control of the robot, enabling it to move reliably within its workspace and complete complex tasks.

Derivation of Forward Velocity Kinematics

The forward velocity kinematics of the OpenManipulator-X robot were derived using the Jacobian matrix. The Jacobian provides a linear mapping from joint space velocities to task space velocities, translating how the movement of each joint affects the velocity of the end effector. To calculate the Jacobian, we employed two methods: the partial differentiation method and the cross-product method. In the partial differentiation method, we derived the Jacobian by taking the derivative of the end effector's position with respect to each joint angle. In the cross-product method, the Jacobian was computed by taking the cross product of each joint's axis of rotation with the position vector from the joint to the end effector. Both methods resulted in the same 6x4 Jacobian matrix, which was implemented in MATLAB as the function jacob3001(). Using this matrix, the forward velocity kinematics were computed by multiplying the Jacobian by the joint velocities. This allowed us to calculate both the linear and angular velocities of the end effector in real-time, which was validated by comparing theoretical and actual velocities during controlled movements.

Derivation of Inverse Velocity Kinematics

Inverse velocity kinematics were also derived using the Jacobian matrix, allowing us to calculate the necessary joint velocities to achieve a desired end effector velocity. To accomplish this, the inverse of the Jacobian matrix was used. Specifically, we applied the pseudo-inverse of the Jacobian matrix to convert task space velocities back into joint space velocities, ensuring that the end effector could follow a planned velocity-based trajectory. Only the upper half of the Jacobian was used as we only cared about the position of the end-effector and not its orientation. The pseudo-inverse was necessary because the Jacobian for a 4-DOF robot is not a square matrix, and this method ensured that the computed joint velocities were optimal with minimized errors. By multiplying the desired task space velocity vector by the pseudo-inverse of the Jacobian, we obtained the required joint velocities to achieve the desired movement of the end effector. This approach was implemented in MATLAB through the fdk3001() function, and the accuracy of the inverse velocity kinematics was confirmed by comparing the robot's motion with the expected velocities during trajectory tracking.

Go to Ball

Using this target position and getting the initial position of the robot by setpoint_js() method from the Robot() class, a trajectory is planned, where the robot moves to the target position in 3 seconds. The z-axis of the target position is offset by 50 mm to allow the robot to come over the ball without disturbing any of the other balls. The gripper of the robot is also opened.

Grab Ball

For the grabbing the ball, a trajectory is again planned where the target position has the same X and Y coordinate as the trajectory planned for going to the ball; however, the z-axis of the target position is 10 mm to allow the robot to grab the ball and not hit the checkerboard. After the trajectory is moved by the robot using run_trajectory() method in the Robot() class, the gripper of the robot is closed.

Move Away from Board

After the robot gets the ball, it needs to move away from the board to stop the robot from disturbing the rest of the balls. For this reason, a trajectory is again planned to go to the same target position as when it goes to the ball.

Go to Box

Based on the color of the ball, it has a specified area to be deposited. Hence, a trajectory is planned to that target position. As the target position is manually inserted, the z-axis of the target position of the ball is kept significantly above the colored boxes to make sure that the arm doesn't hit or move the boxes. After the trajectory is completed, the gripper of the robot is opened to deposit the ball in its respective location.

Sorting Approach

The sorting approach involved getting a new image of the checkerboard for each loop, and sorting the balls by color: red, green, orange, yellow, grey. The new image helped to update the position of the balls if they had been moved while the robot was picking up the previous ball. The robot kept picking the same-colored balls until it finished picking all those colored balls before moving to the next color.

System Architecture

Our system architecture for controlling the robotic arm is built around several key classes and methods designed to handle both kinematic calculations and robot control. The core classes include Robot, Traj_Planner, and Image_Handling, each of which serves a distinct function in the system. The Robot class manages the interface with the robot, allowing us to send joint commands and read sensor data. The Robot class also handles forward and inverse kinematics, providing methods such as fk3001() and ik3001() for calculating the robot's end-effector position and required joint angles. The Traj_Planner class enables trajectory generation, using cubic and quintic polynomials to plan smooth movements. Meanwhile, the Image_Handling class processes visual input from the robot's camera, detecting objects and generating coordinates for manipulation. These components work together in a well-defined workflow: image data is captured and processed, kinematic calculations determine the appropriate joint movements, and the robot executes those movements to achieve the desired task. Although we do not use a traditional state machine, the system follows a structured process, transitioning from image acquisition and object detection to trajectory planning, and finally to executing movements, ensuring reliable control of the robot throughout its operations.

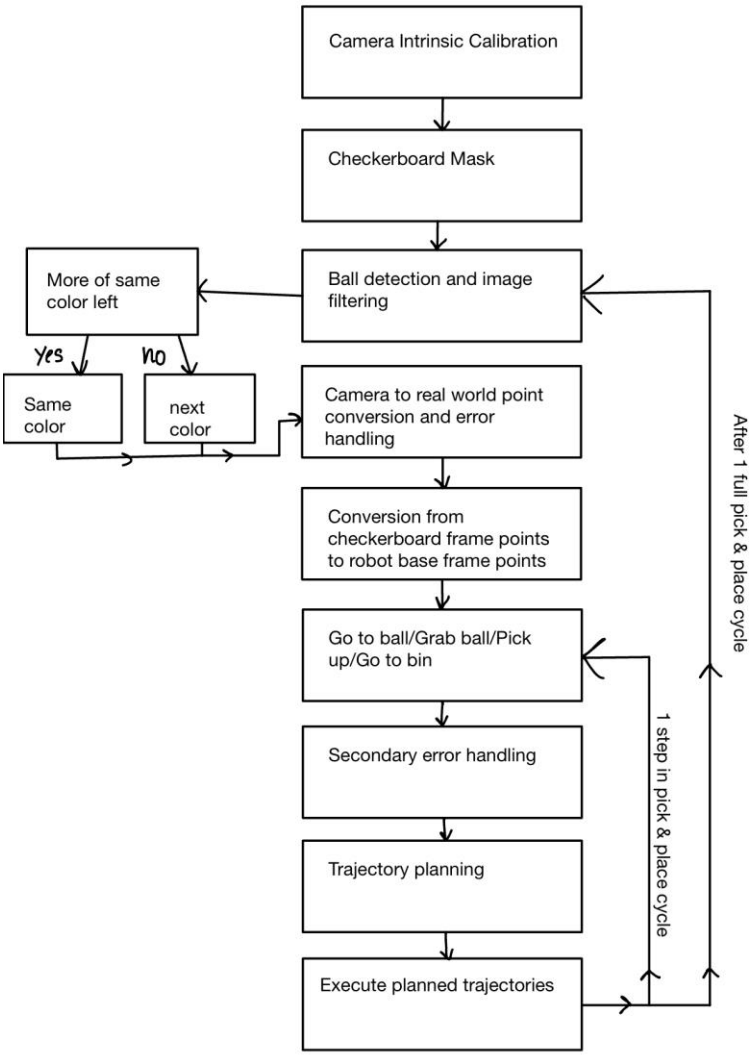

Figure 6: Image showing the system architecture of the code

RESULTS

A. General Images

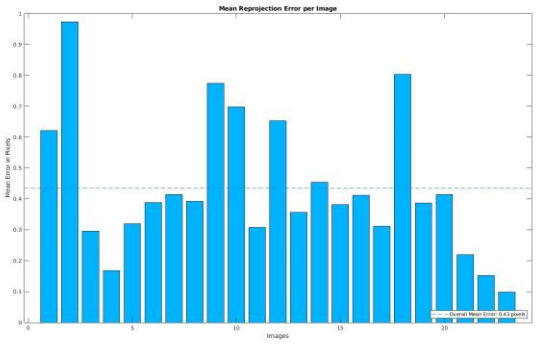

Figure 7: Graph of Errors

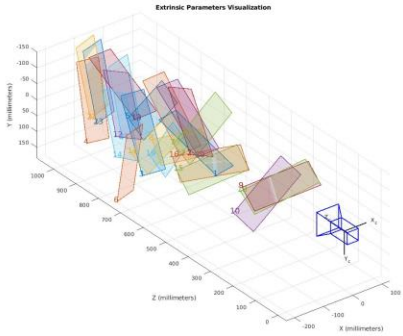

Figure 8: Plot of Extrinsic Parameter Visualization

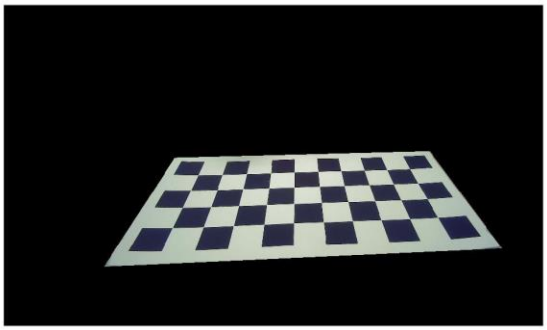

Figure 9: Image from camera with mask of checkerboard

B. Red, Green, Orange, and Yellow Balls

Figure 10: Raw image from the USB Webcam, unmasked, no filters, with one red ball placed

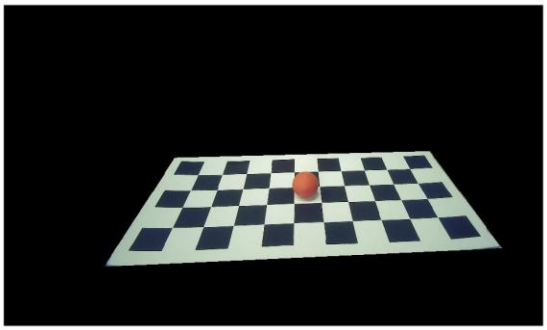

Figure 11: Checkerboard Mask applied to the raw image with red ball.

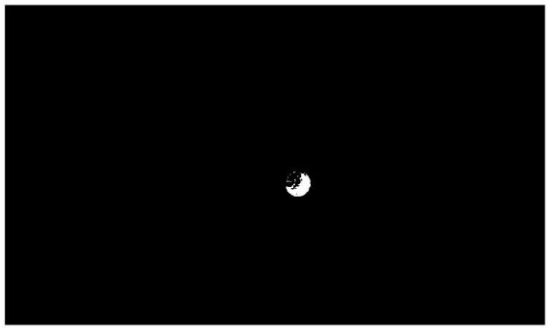

Figure 12: Color filter Applied to the image with checkerboard mask.

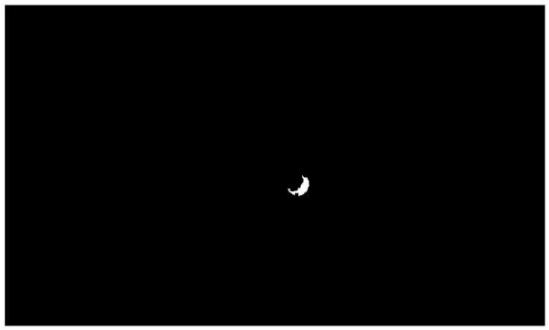

Figure 13: Erode filter Applied to the image with color filter

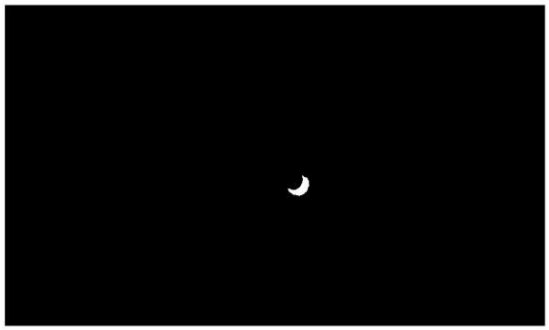

Figure 14: Close and Fill filters applied to the image with erode filter

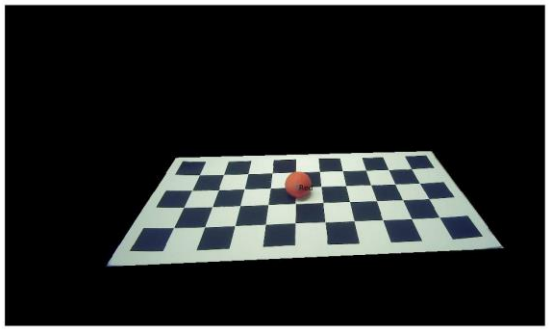

Figure 15: Regionprops() method used to get the centroid of the balls

C. Grey Balls

Figure 16: Raw image from the USB web camera with grey ball

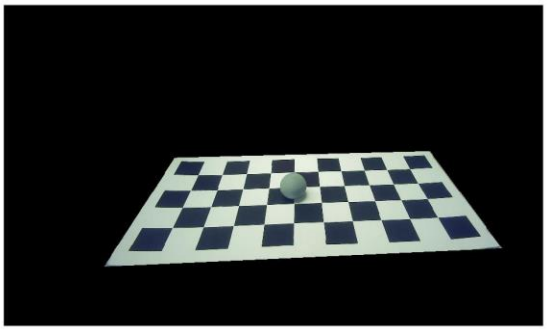

Figure 17: Checkerboard Mask applied to the raw image with grey ball.

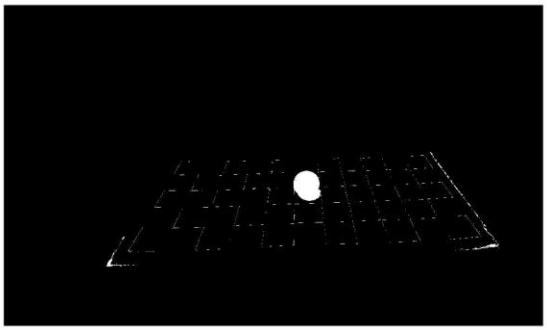

Figure 18: Color filter Applied to the image with checkerboard mask.

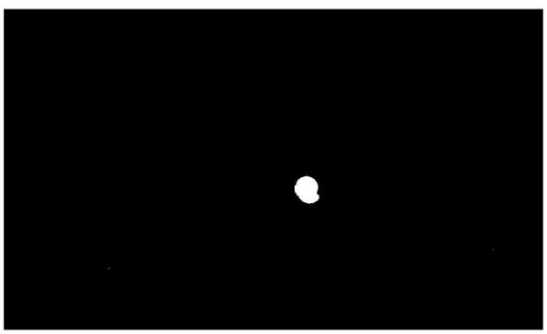

Figure 19: Erode filter Applied to the image with color filter

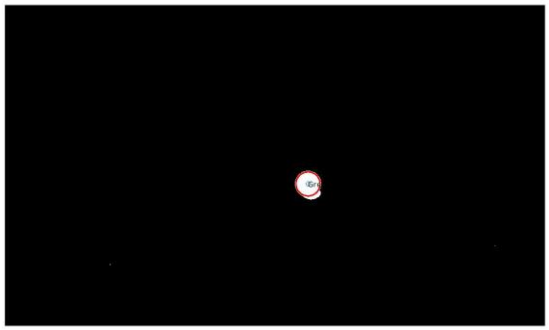

Figure 20: imfindcircles() method used to find circles in the image with eorode filter.

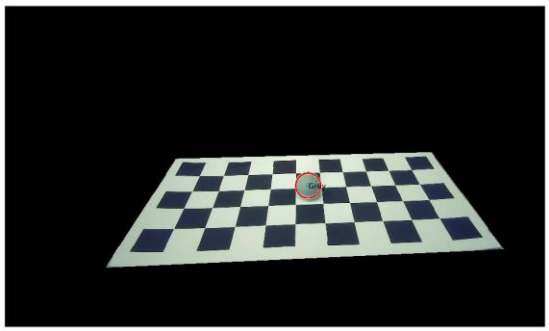

Figure 21: Used the circles to find the coordinates of the center of the circle.

DISCUSSION/ CONCLUSION

The development of our robotic pick-and-place system demonstrated the effective integration of computer vision and kinematic control, enabling the OpenManipulator-X robot to identify, pick, and sort objects based on color with a high level of accuracy (approximately 99%). Our approach utilized key principles from earlier labs, such as forward and inverse kinematics, trajectory generation, and velocity control, all of which worked together to automate the system. The camera, calibrated with intrinsic parameters, enabled reliable detection of objects within the workspace. By using HSV color space and filtering techniques, the system was able to segment balls from the background even in challenging lighting conditions. However, detecting grey-colored balls required additional techniques, such as using the Lab* color space and circular filters for shape detection.

In terms of kinematic control, the system performed reliably. Forward and inverse kinematics allowed the robot to calculate the necessary joint angles for accurate positioning of the end effector. Trajectory planning ensured smooth movement, minimizing collisions with the checkerboard or objects. Despite this, some discrepancies between the planned and actual end-effector positions were observed, likely due to minor errors in the robot's link lengths and joint flexibility. These discrepancies could be

addressed by refining the kinematic model and implementing better error correction techniques.

Another challenge was the detection and avoidance of singularities during trajectory planning. By incorporating the Jacobian matrix and real-time singularity detection, the system successfully avoided problematic configurations, ensuring smooth and reliable movement. Additionally, lighting conditions impacted object detection, requiring frequent adjustments to the vision algorithm. A more robust computer vision system, capable of adapting to varied lighting environments, would significantly improve performance.

Looking ahead, there are several areas for improvement. Enhancing the vision system to be more adaptable under different lighting conditions would increase detection accuracy. Implementing advanced trajectory generation methods, such as septic trajectory, could make movements smoother and reduce the risk of overshooting or jerky motion. Incorporating feedback control would also allow the system to make real-time adjustments, making it more adaptive to environmental changes.

In conclusion, this project successfully integrated key principles from previous labs into a functional robotic system. It provides a strong foundation for future work in robotic automation and highlights the challenges of integrating vision systems with robotic control. With further refinement, this system has the potential to scale for use in real-world industrial applications.

APPENDIX

Final Demo Code Release: https://github.com/RBE3001-A24/RBE3001_A24_Team_17/releases/tag/Lab5Final

Video: https://youtu.be/EvV3xOcC02Q