# RBE3001 Lab 5: Ball Sorting

Hushmand Esmaeili. Author, Logan Rinaldi. Author, and John Robinson. Author

***Abstract*— We used MATLAB to program a 3DOF RRR robotic arm to pick up and sort different colored balls into their respective bins. We calculated the forward and inverse position and velocity kinematics and used this information to perform trajectory planning in the task space. We used MALAB functionality to calibrate the camera. Using the camera, we performed object detection using masking techniques. We performed object classification using the different colors of the ball. We performed object localization by calculating the actual ball location from the camera readings. The final program included a state-machine that checked for balls based on their color, and then if one was detected the end effector moved to that position to pick it up. The robot then moved the balls into their respective bins. The final program also accounted for errors in detecting balls due to obstruction of view, errors when the ball wasn't picked up the first time, and errors when the robot was approaching a singularity. The final program successfully completed the task and worked consistently to pick up and sort different colored balls and was also able to pick up a small random object, a mini rubber duck.**

## I. Introduction

In this project we used a three degrees-of-freedom (3-DOF) robotic arm to sort four different colored balls into their respective bins. We had a camera attached to a post that overlooked a checkerboard field. Using the camera, we had to detect and categorize the balls on the checkerboard by their color. Then, we had to identify the location of the balls with respect to the robot base frame. From there, we had to send the robot arm to a position where the end effector could pick up the ball and then move it to the appropriate bin. We achieved the project by coding a state machine in MATLAB. We utilized calculations from previous labs to plan trajectories, detect singularities, and help get current data about the robot arm.

## II. Methodology

To complete this project we started by looking at the work we had done for the previous labs. The work we had done to find the position kinematics and velocity kinematics served as a base to complete the color ball-sorting project.

The first task, essential to deriving the forward kinematics calculations of the manipulator, was to create a diagram of the robot, assign and attach a proper frame for each joint, and create a table of DH parameters based on assigned frames. To implement the forward kinematics in MATLAB, we used the derived DH parameters and created three methods to perform different parts of the forward kinematics calculation. In the end, we had the fk3001(), which generates the complete homogeneous transformation matrix which represents the position and orientation of the end effector with respect to the base frame.

We derived inverse kinematics of the robot, using the geometric approach, which was essential to the trajectory generation of the robot. Additionally, we obtained the velocity kinematics of the robot, which was used primarily for singularity detection. The details for the forward and inverse kinematics, as well as for the velocity kinematics, were previously reported.

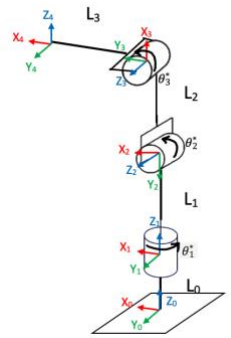
After setting up the forward and inverse kinematics and the velocity kinematics of the robot, we began to work on setting up our computer vision by performing intrinsic calibration of the camera. Once we successfully connected to the camera in MATLAB, we took a series of photographs from multiple angles. Then, we used the MATLAB calibrate feature to complete the intrinsic calibration and obtain the calibration parameters, which were exported to a script.

Once we had completed the intrinsic calibration, we moved on to finding the extrinsic calibration. To do this, we used MATLAB getCameraPose() function to find the pose of the camera, and the pointsToWorld() to find the x, y location of a defined point on the grid. Using this information, along with the transformation from the base frame to the checkerboard, we were able to relate a point that the camera identified to a target that the robot can move to.

Next, we masked the workspace to be able to identify different colors. We used these colors to identify the different balls on the grid. Once we found the x, y location of each ball, we performed object localization. We used a geometric approach to relate the point the camera had identified to the actual location of the ball that the robot could go to grab it. To complete this, we used the camera height, ball height, location from pointsToWorld(), and other intermediate variables to find the actual location of the ball.

Once this was all complete, we were able to create a state machine that looked for and identified a ball by color, move the robot to just above the ball, lower to the ball and lift it up, and finally move the ball to its respective bin. Our state machine also accounted for some errors that may occur, such as if the robotic arm was covering the checkerboard in such a way that a camera could not see a ball. Another error we accounted for was if the trajectory planning of the robot arm would send it to a singularity, in which case we would move the arm and recalculate a different trajectory.

Figure 1. Diagram of the 3-DOF robot arm with frame assignments.




## III. Results

We gathered results regarding the position kinematics, velocity kinematics, image processing, ball sorting, and the overall system architecture.

### A. Position Kinematics

In Lab 2 and 3 we calculated the forward (FK) and inverse (ik) kinematics of the robot arm.[1][2] We used the results from these labs to aid in completing the ball-sorting project.

To calculate the FK of the 3-DOF manipulator, we created a diagram of the robotic arm, showing the links and joints, and assigned coordinate frames for the base, end-effector and joints, following the DH convention, seen in Fig. 1. We also derived the DH parameters of the robotic arm, which can be found in the Appendix.

We used the DH parameters to calculate the intermediate transformations. Finally, we obtained the homogeneous transformation matrix from Frame 0 to Frame 4. This matrix was used to calculate the forward kinematics of the robot and we implemented a function fk3001(), which generates the complete homogeneous transformation matrix that represents the position and orientation of the end effector with respect to the base frame.

Figure 2. 3D stick model of 3-DOF robotic arm in home positions

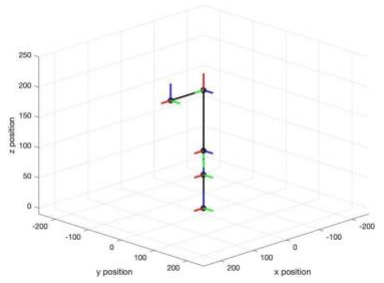
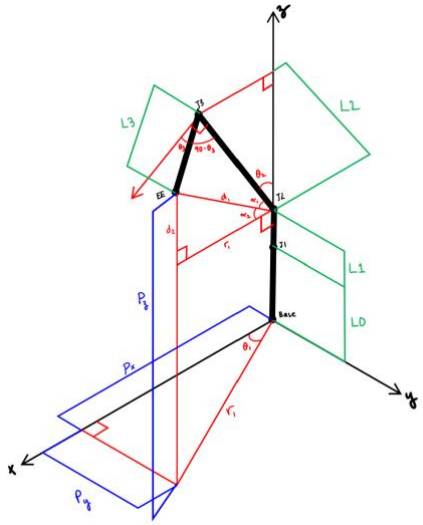



Figure 3. 3D drawring of the labeled 3-DOF robotic arm.




We successfully created a 3D stick model of our robot arm in MATLAB. The 3D stick model contains all the joints and links of the robot, as well as the frame assignment for each joint. To verify that the FK of the robot worked, we visualized the robot using the 3D stick model. Fig. 2 shows a test for the robot's FK.

To find the IK of the 3DOF robotic arm, we created a diagram of our robot, seen in Fig. 3 From our calculations, we solved for each joint angle (1), (2), and (3).

$$\theta_1 = atan2\left(\pm\sqrt{1-D_1^2}, D_1\right), where\ D_1 = \frac{P_x}{r_1} \quad (1)$$

$$\theta_2 = 90 - atan2\left(D_2, \pm\sqrt{1-D_2^2}\right) - atan2(d_2, r_1),\ where\ D_2 = \frac{L_3\, cos(\theta_3)}{d_1} \quad (2)$$

$$\theta_3 = atan2\left(D_3, \pm\sqrt{1-D_3^2}\right), where\ D_3 = \frac{L_3^2+L_2^2-d_1^2}{2L_3L_2} \quad (3)$$

These angle definitions completed our IK.

### B. Velocity Kinematics

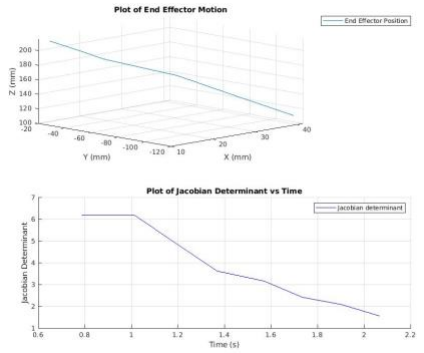
Velocity kinematics uses the Jacobian matrix of the robot to calculate task-space velocities for the robot and identify its singular configurations. We first formulated the 6 by 3 manipulator Jacobian matrix by using the partial derivative-based approach for the upper half of the Jacobian and the z-component of the rotation matrix from the intermediate homogenous transformation matrices for the bottom half. The resulting Jacobian matrix is shown in the Appendix.

We validated the obtained Jacobian matrix by finding the position Jacobian matrix (Jp) and its determinant at two singular configurations. At singular configurations, the determinant of Jp should be 0. The first test was at an internal singularity, with the arm extended upward along the z0 axis, with a joint configuration of q = [0 0 90]. This

[1] "Lab 2: Forward Kinematics". Esmaeili, Rinaldi, Robinson. 2021.

[2] "Lab 3: Inverse Kinematics and Trajectory Generation". Esmaeili, Rinaldi, Robinson. 2021.

Figure 4. Plot of determinant as robot approaches a singularity.




produced a Jacobian determinant of zero. We also tested a boundary singularity, with the arm extended straight outward, with a joint configuration of q = [0 90 -90], which also produced a Jacobian determinant of zero. We plotted the relationship of the determinant and how close the robot is to reaching a singularity in Fig. 4. Using this relationship, we were able to create a method that uses the determinant to detect when the robot is approaching a singularity.

### *C. Image Processing Pipeline*

An essential component of the color-sorting robot was the computer vision (CV) system. The goal of the CV system was to identify objects and localize them with respect to the base frame.

The calibration of the CV system was divided into two parts: intrinsic and extrinsic calibration. The goal of the intrinsic calibration of the camera is to obtain the intrinsic parameters camera matrix, which include the camera's focal length and pixel dimension. These values allowed us to use the camera as a sensor to make measurements.

By taking a series of images from different orientations and running the calibration, we obtained the intrinsic camera parameters, and exported the camera parameters to a script. We also obtained a visualization for the mean projection error per image, as shown in Fig. 5.

Figure 5. Histogram of the mean reprojection error per image.

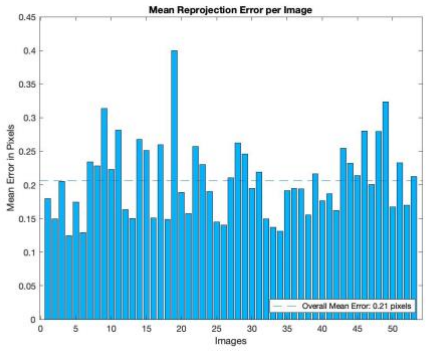
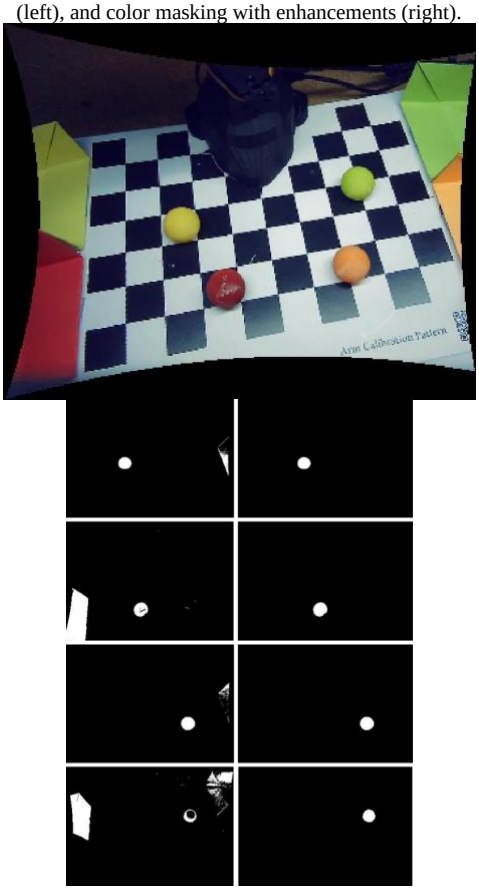



Figure 6. Reference (raw) image from camera (top), basic color masking from MATLAB Color Thresholder (left), and color masking with enhancements (right).

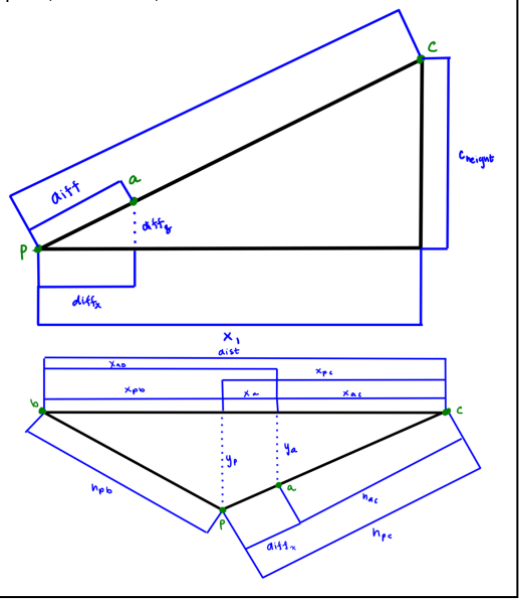


The reprojection error is the distance between the checkerboard points detected in a calibration image, and the corresponding world points projected using the estimated camera matrix. From Figure 5, the overall mean error is 0.21 pixels. The mean error in pixels for each image is for the most part within about 0.3 pixels, except for one image which is about 0.4 pixels.

From the intrinsic calibration process using MATLAB, we also obtained an undistorted image with checkerboard axes, as shown in Fig. 6.

The top image shows the raw undistorted image taken by the camera (Figure 6), pre-processing; all four colored balls are within the workspace of the checkerboard, while the colored bins are outside of the workspace. The images on the left column are from the basic color masking from the MATLAB Color Thresholder app. From the left column images, the first row is from the yellowMask() function, second row from the redMask() function, third row from the orangeMask() function, and the fourth row from the greenMask() function. The refined images from further image processing are shown in the right column of the same figure. Similarly, from the right column, the first row is for the colorMask() for the yellow ball, the second row is for the

Figure 7. X-Z plane of setup (top) and X-Y plane of setup (bottom), where p is projected point, a is actual point, c is camera, and b is the robot base.




red ball, third row is for the orange ball, and the fourth row is for the green ball. These images show almost perfect and complete circles, representing the masked colored balls in the workspace.

When using the pointsToWorld() to get the location of the ball, there was some difference between the coordinate that was returned and the actual location of the ball. To perform object localization, we did some calculations to convert the projected point we got from pointsToWorld() to the actual x-y coordinate that the robot could use to pick up the ball. Fig. 7 shows the x-z plane of the configuration. We found the distance in the x direction, diffx, between the projected point, p, and the actual point, a, (4).

$$diff_x = x_1 \frac{diff}{c_{height}} \quad (4)$$

Using the x-y plane drawing, we solved for the x position of the actual point with respect to the base frame, (5), and the y position with respect to the base frame, (6).

$$y_a = \frac{y_p(h_{pc} - diff_x)}{h_{pc}} \quad (5)$$

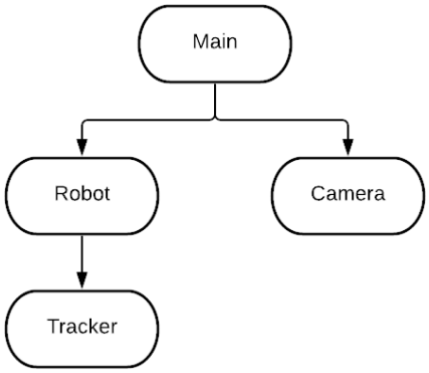
$$x_{ab} = \frac{x_p(h_{pc} - diff_x)}{h_{pc}} + x_p \quad (6)$$

Using the object localization, we were able to successfully find the coordinates of the ball position given the projected point from the camera.

*E. System Architecture*

Our code is separated into four main components: the Main script, and the three classes that fall under it. The main script iterates through the state machine, Fig. 8, and calls functions defined in the Robot and Camera classes. Instances of the Robot and Camera classes are both created in the main script, and an instance of the Tracker class is created in the Robot class. One issue we faced in this process was that

Figure 8. Class structure of final program.




upon creating an instance of the camera class, it would need to calibrate the camera which took an excessively long time and was significantly slowing down our ability to iterate the code. To remedy this, the first time we set up the camera in its new environment we create the camera object and save it to a Matlab file. Every future time we run the script, instead of creating a new camera object, it instead loads it from the file to save time.

In our main script, the robot runs through a state machine, Fig. 9, to look for balls in the camera's field of view. To begin this process, it goes through the reset state in which it moves out of the way of the camera so that the camera can

Figure 9. State machine of final program.

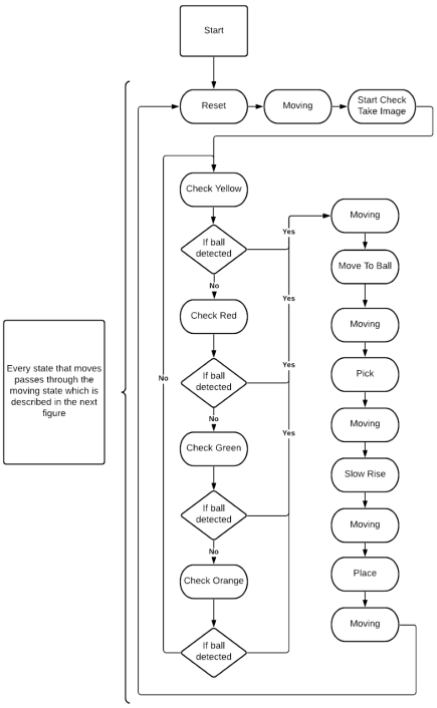
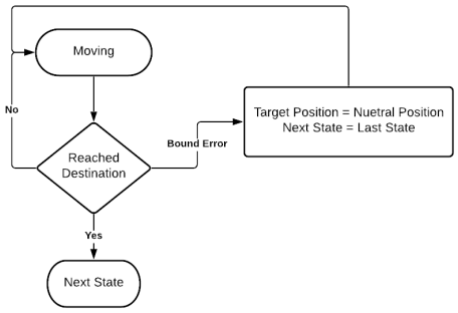


Figure 10. Moving state of final program.




see the ball on the field. It then iterates through looking for balls of each color in order. Once it detects a ball it will run through the state machine that denotes the motion for placing and depositing a ball. It will move to the location of the found ball and place it in the appropriate bin. When it is time for the robot to move it will first set a target destination in the Tracker object, and then move to the moving state.

In the moving state, Fig. 10, the robot is constantly moving to new points generated by the Tracker class until it reaches the target destination. If, in doing this motion, the robot tries to move to a location outside of the bounds of the joints the ik3001 function will throw an out of bounds error. The motion state will then catch this error and send the robot to a neutral position over the center of the board before retrying the previous motion.

## IV. Discussion

For calculating the forward kinematics of the 3-DOF Manipulator, we were able to confirm that the assignment of coordinate frames and hence the table of DH parameters were properly done. We did this by checking that each of the intermediate transformations matched the calculated parameters. More specifically, for each intermediate transformation, we ensured that the rotation about z-axis, translation about z-axis, translation about x-axis, and rotation about x-axis all matched the actual physical transformation of the frames. Though there might have been simpler ways to assign and attach the frames, we found that our approach worked for us.

We also found that all our FK methods worked correctly. We know this because we both checked our outputs against what we would expect based off our knowledge of forward kinematics, as well as through the 3D stick model. By moving the arm around, the stick model correctly visualized the arm, proving that our methods worked correctly.

We verified our inverse kinematics using our forward kinematics equations. Using the inverse kinematics, we were able to write a method that performed quintic trajectory planning in task space. This allowed us to pick a position, time, and velocity and generate a trajectory that the robot could use to traverse to a given point in the task space during a time period with the given initial and final velocity. We ended up using this method heavily in the final ball-sorting project.

For the derivation of the forward velocity kinematics of the robot, the main result for this section was the derivation of the Jacobian matrix. From it, we were able to obtain other results, specifically related to singularity detection. We obtained the Jacobian matrix shown in the Appendix. By examining the upper half of the Jacobian, if we look at the first column, we see that the third row is 0. This makes sense because Joint 1 has no contribution to the linear velocity of the end effector in the z-direction. Similarly, it is expected that all other entries in the upper half of the Jacobian are not zero, since the joints all have contributions to the linear velocities in different components, except for the one mentioned. Additionally, in the bottom half of the Jacobian, it is expected that Joint 1 only has contribution to the angular velocity of the end effector about the z-axis, and Joint 2 and Joint 3 contribute to the angular velocities of the end effector about the x- and y-axes. This is confirmed by the obtained lower Jacobian, where the first column is all 0 except for the last row, and for columns 2 and 3, it is all nonzero except for the last row. By quick inspection, the obtained Jacobian appears to be right.

In validating the obtained Jacobian, by examining its determinant at singularities, one thing we noticed in the internal singularity configuration was that the first column of the position Jacobian was zero. This means that the linear velocity of the end effector with respect to joint 1 was zero in this configuration. This is because Joint 1 and the end effector are in line in this configuration. This result means that no matter the angle of Joint 1, it will not change the velocity of the end effector linearly with respect to Joint 1; they will remain colinear. This position also means that there are infinite combinations of angles for Joint 1 to achieve this position, making the position of the end effector impossible to determine.

For the camera calibration process and the image processing pipeline, we were able to identify objects and localize them with respect to the base frame for our CV system. A few of the key results of this section were the mean reprojection error per image from the intrinsic camera calibration, the undistorted image with checkboard, the image processing pipeline, and the object localization results.

For the intrinsic calibration of the robot, we obtained the mean reprojection error per image histogram (Figure 5). The reprojection error is a measure of the accuracy of the system; it is the distance between a pattern keypoint detected in a calibration image, and a corresponding world point projected into the same image. The higher this error, the less accurate the CV system is. Our calibrated system produced an overall mean error of 0.21 pixels. This means that, on average, the distance between a pattern keypoint and a corresponding world point projected into the same image is off by 0.21

pixels. MATLAB's documentation suggests that this error should not be higher than 1. Our system produced an error way below that threshold, which means that overall, the system should be very accurate. Additionally, we removed any images that had a reprojection error higher than 0.5 pixels. Overall, this ensured that our CV system is accurate.

The second meaningful result was the undistorted image with checkboard. This undistorted image was automatically generated by the provided Camera.m script. There are two things to note from that result. The first thing is that, although the camera captures distorted images due to its fisheye lens, the returned image was successfully undistorted to properly show the checkboard. The second observation is that the checkerboard's keypoints were properly identified in the image, as well as the checkerboard axes with the origin in the right place. Both observations suggest that the calibration process was successful and accurate. The extrinsic calibration of the robot also proved to work once we performed the camera-robot registration.

The object detection and classification sections were essential for the robot to identify and locate a ball by its color. The image processing pipeline that we chose was appropriate and successful in achieving the results that we wanted. As shown in Figure 6, first we captured the undistorted image. The four balls of distinct colors are within the frame. From the left column of this figure, basic color masking, produced by the MATLAB Color Thresholder app, was successful in identifying objects of specific colors. Additionally, to segment the image in the HSV color space meant that we could properly separate the luma (image intensity) from the chroma (color information). For our CV system, this feature is particularly important since it makes it more robust to lighting changes and in removing shadows. One thing to note is that color segmentation in the HSV color space would not work for black balls because the V channel (value) is close to 0.

The images obtained from the color segmentation process still have undesired objects from outside the checkboard workspace, since those objects are of similar colors to the balls, and the balls are not perfect circles, sometimes with holes or black spots in the center. Therefore, we performed further image processing and additional enhancements. The next enhancements were masking the workspace from the checkerboard, and image erosion and image filling to obtain more coherent circles, representing the balls. These steps were essential before we found the centroid of the identified ball, otherwise our algorithm would also attempt to find centroids for the other undesired objects. From the right column of Figure 6, the balls were evidently properly segmented and identified. One thing we noticed is that our green and yellow balls were mixed up at times. We found that the reason for this was because we did the initial color thresholding in a specific lighting condition. When the lighting condition changed, the system was not as accurate. For future experiments and projects, we would want to improve the color thresholding process so that the system works for different lighting conditions.

Finally, from the object localization section, we were able to successfully find the coordinates of the ball position given the projected point from the camera. Using this, we could send the robot to that point using trajectory planning and picking up the ball. A key assumption for this part was that the camera post was exactly centered to the robot; in other words, that the y-component of the camera, with respect to the base frame, was 0. A problem that we ran into was that, although the CV system was able to accurately return the position of a ball with respect to the base frame, when we attempted to move the arm to the ball, the gripper was not able to grab the ball. The reason behind this was not due to the CV system being inaccurate, but that we had not considered that the robot's end effector position is effectively at the side of the gripper, not at the center of the gripper. Hence, through some trigonometry and calculations, we were able to successfully correct this error and grasp the ball.

## V. Conclusion

In conclusion, our team was able to successfully make a pick-and-place, color-sorting robotic arm. We implemented and integrated several main components, including forward and inverse kinematics, velocity kinematics, trajectory planning, and a computer vision system. The robot incorporated a CV system to identify and classify targets, localize them with respect to the base frame of the robot, move the end effector of the robot towards the object, grasp them and then place them in determined drop-off locations, according to their color.

We derived the forward and inverse kinematics of the robot, as well as the velocity kinematics and trajectory planning, in previous labs. Some of the key findings and results were included in this report as they are integral parts of the system.

For the CV system, we had to setup and calibrate the camera. The calibration process was divided into two parts, intrinsic and extrinsic calibration. After this process was done, the camera worked and we had scripts that performed the intrinsic calibration with viable results, and we could reliably relate pixel coordinates to points measured with respect to the robot's base frame. Upon the completion of the camera calibration, we determined and followed an image processing pipeline, which took a frame of the camera and identified and classified the balls by colors. Additionally, we were able to find the centroid of these objects and convert them into positions with respect to the base frame of the robot. Finally, we did further calculations to correct the errors produced due to the projection of the ball onto the grid, so the system could accurately locate the objects.

Once all the components of the system were implemented, we integrated them through a state machine,

which was run in our main code. After many iterations of testing, debugging and improving, we arrived at a fully functional robot system that could sort, pick and place balls of different colors in their respective locations.

## APPENDIX

DH Parameters of 3-DOF Robotic Arm:

| Intermediate Transformations | Link | $\theta$ (in °) | d (in mm) | a (in mm) | $\alpha$ (in °) |
|---|---|---|---|---|---|
| $T_0^1$ | 0 | 0 | $L_0 = 55$ | 0 | 0° |
| $T_1^2$ | 1 | $\theta_1^*$ | $L_1 = 40$ | 0 | −90° |
| $T_2^3$ | 2 | $\theta_2^* - 90°$ | 0 | $L_2 = 100$ | 0° |
| $T_3^4$ | 3 | $\theta_3^* + 90°$ | 0 | $L_3 = 100$ | 90° |

Jacobian Matrix of 3-DOF Robotic Arm:

$$J = \begin{pmatrix} 1.7453\,\sigma_6\,\sigma_9\sin(0.0175\,\theta_1) - 1.7453\,\sigma_8\,\sigma_7\sin(0.0175\,\theta_1) - 1.7453\,\sigma_8\sin(0.0175\,\theta_1) & -1.7453\,\sigma_6\cos(0.0175\,\theta_1) - \sigma_5 - \sigma_4 & -\sigma_5 - \sigma_4 \\ 1.7453\,\sigma_8\cos(0.0175\,\theta_1) + 1.7453\,\sigma_8\,\sigma_7\cos(0.0175\,\theta_1) - 1.7453\,\sigma_6\,\sigma_9\cos(0.0175\,\theta_1) & -1.7453\,\sigma_6\sin(0.0175\,\theta_1) - \sigma_3 - \sigma_2 & -\sigma_3 - \sigma_2 \\ 0 & 1.7453\,\sigma_6\,\sigma_9 - 1.7453\,\sigma_8 - 1.7453\,\sigma_8\,\sigma_7 & 1.7453\,\sigma_6\,\sigma_9 - 1.7453\,\sigma_8\,\sigma_7 \\ 0 & \sigma_1 & \sigma_1 \\ 0 & \cos(0.0175\,\theta_1) & \cos(0.0175\,\theta_1) \\ 1 & 0 & 0 \end{pmatrix}$$

$$\sigma_1 = -\sin(0.0175\,\theta_1)$$

$$\sigma_2 = 1.7453\,\sigma_7\,\sigma_6\sin(0.0175\,\theta_1)$$

$$\sigma_3 = 1.7453\,\sigma_8\,\sigma_9\sin(0.0175\,\theta_1)$$

$$\sigma_4 = 1.7453\,\sigma_7\,\sigma_6\cos(0.0175\,\theta_1)$$

$$\sigma_5 = 1.7453\,\sigma_8\,\sigma_9\cos(0.0175\,\theta_1)$$

$$\sigma_6 = \sin(0.0175\,\theta_2 - 1.5708)$$

$$\sigma_7 = \cos(0.0175\,\theta_3 + 1.5708)$$

$$\sigma_8 = \cos(0.0175\,\theta_2 - 1.5708)$$

$$\sigma_9 = \sin(0.0175\,\theta_3 + 1.5708)$$

Link To Video
Link To Code

| Section | Author |
|---|---|
| **Abstract** | Logan Rinaldi |
| **Introduction** | Logan Rinaldi |
| **Methodology** | Logan Rinaldi, Hushmand Esmaeili |
| **Results** | Logan Rinaldi, Hushmand Esmaeili, John Robinson |
| **Discussion** | Logan Rinaldi, Hushmand Esmaeili, John Robinson |
| **Conclusion** | Hushmand Esmaeili |

## REFERENCES


[1] "Lab 2: Forward Kinematics". Lab report. Worcester, Massachusetts. Worcester Polytechnic Institute. H. Esmaeili, L. Rinaldi, J. Robinson. September 2021.

[2] "Lab 3: Inverse Kinematics and Trajectory Planning". Lab report. Worcester, Massachusetts. Worcester Polytechnic Institute. H. Esmaeili, L. Rinaldi, J. Robinson. September 2021.