Robotics Engineering 3001: Manipulation - Final Project

Lorenzo Manfredi Segato, Filippo Marcantoni, Nikhil Grangaram

***Abstract*— The purpose of this paper is to report on a project which is the culmination of a series of laboratories performed on the OpenManipulator-X robot arm using MATLAB. Throughout these laboratories, the team has explored and mastered forward, inverse & velocity kinematics, trajectory planning, the image processing pipeline and robot statics & dynamics; with the purpose of engineering a robotic arm capable of pick up, sorting and dynamic tracking of different colored balls (and other objects) via an external camera and a gripper at its end effector.**

I. INTRODUCTION

A. Background

As briefly stated in the abstract, various subtopics under the 'Manipulation' umbrella were addressed during this endeavour (see above), and delving into deep detail at this stage is superfluous. However, it is worth noting that, without intimate understanding of kinematics, trajectory planning and image processing - particularly the former two - completion of this enterprise would prove a Herculean task. As the paper unfolds, more of the relevant theory will be outlined.

B. Motivation

The motivation for this project is pure: The ability to demonstrate successful implementation and understanding of the concepts covered in this class - showing progress in both the theoretical (calculations) and practical (MATLAB script writing) aspects of robotics.

The end goal of this project was to design a pick-and-place system for the OpenManipulator-X robot arm, using an external camera and gripper at the end-effector. This system also needed to be able to differentiate objects by color, dynamically track objects, and manipulate random objects

To accomplish this venture, our journey can be discretized into 9 distinct steps, each of which will be addressed in the Methodology section:

1) Forward Kinematics Derivation
2) Inverse Kinematics (Geometric Method)
3) Velocity Kinematics Derivation
4) Camera Setup: Intrinsic and Extrinsic Calibration
5) Object Detection and Localization
6) Object Position Correction
7) Object Sorting
8) Dynamic Tracking
9) Manipulation of a Random Object

Each of these was implemented and realized through object-oriented programming in MATLAB, designing a plethora of functions and scripts that allow for precise and consistent movements, following a specific trajectory and visualization of the same. For example, at each snapshot of the camera, various different masks are applied to the image allowing for only specific colors to be visible. With some additional image processing, we are able to locate the the centroid of the visible objects in the images. With some clever mathematics, we are then able to convert the image frames into the real-world, allowing for physical manipulation of the objects placed in the robot's task space.

All in all, the team was successful in its undertaking, going beyond what was required of the project. The team was able to develop a pick-and-place sorting system, capable of dynamic tracking and unique object pickup (given the constraints of the robot gripper).

II. METHODOLOGY

A. Forward Kinematics Derivation

The first step to controlling our robotic arm was to derive the forward kinematics that allow us to mathematically describe and model the position and orientation of our end effector with respect to the base frame. We accomplished this by following the DH-convention where we derived the DH-parameters from joint 0 to the end effector. Figure 1 shows our DH-table which has been populated with the DH-parameters that we derived and the other subfigure shows the coordinate frames that we assigned by following the DH convention.

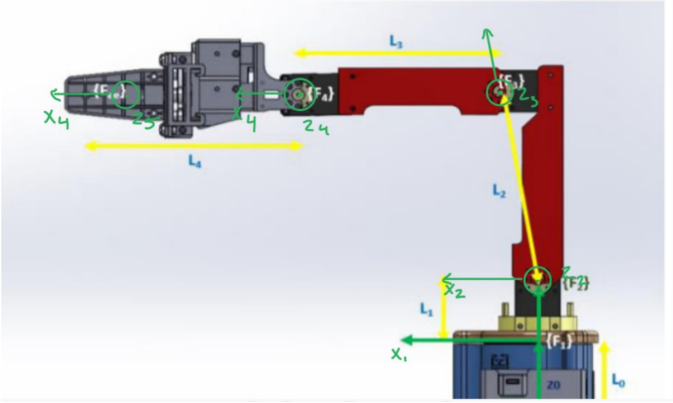
We then feed the DH-parameters we found into the transformation matrix shown below to find the transformation matrix from joint "i" to "i+1":

$$T_i^{i+1} = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The resulting matrices will be shown in Sec.III.A where we will show the individual transformation matrices as well as the transformation matrix from the base to the end effector. Furthermore, in Sec.III.B, we will reveal a software visualization of the robot arm which we found by gathering transformation matrices from the base to each joint which gives us the points of each joint in space. Being able to

Link	θ	d	a	α
1	θ_1^*	L_1	0	α_1
2	$\theta_2^* + \theta_2$	L_2	0	α_2
3	$\theta_3^* + \theta_3$	0	L_3	α_3
4	θ_4^*	0	L_4	α_4

(a) DH Parameters



(b) DH Frames

Fig. 1: DH Information

digitally simulate the robot was incredibly helpful as it provided us a with a tool with which we could check our software implementation before bringing it to hardware.

B. Inverse Kinematics (Geometric Method)

Then, we needed to solve the inverse kinematics of the robot arm to find the required joint angles for any given x, y, z, and α

We chose to solve the inverse kinematics using the geometric method where we solved for each joint angle at a time.

The first step was to find θ_1 which we did so with the help of Figure 2.

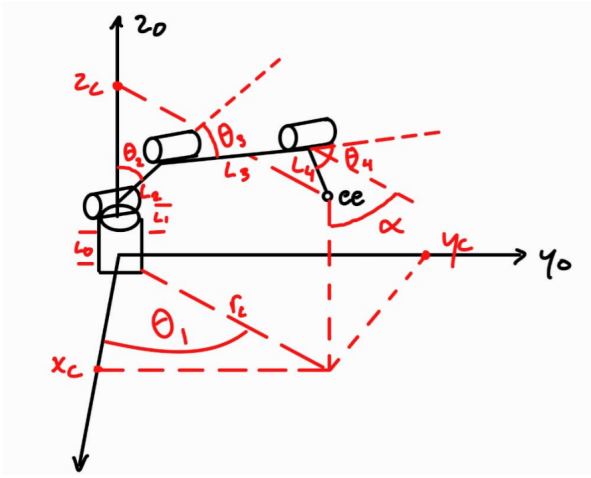


Fig. 2: Isometric Sketch of the Robot Arm

From the figure we know that:

$$r_c = \sqrt{x_c^2 + y_c^2}$$

$$\cos(\theta_1) = \frac{x_c}{r_c} = D_1$$

$$\sin(\theta_1) = \pm\sqrt{1 - {D_1}^2} = C_1$$

$$\theta_1 = \text{atan2}\left(\pm C_1, D_1\right)$$

Now, we can shift our attention to finding $\theta_2, \theta_3, \theta_4$. To accomplish this, we will need to now look at the robot in the z-r plane shown in Figure 3.

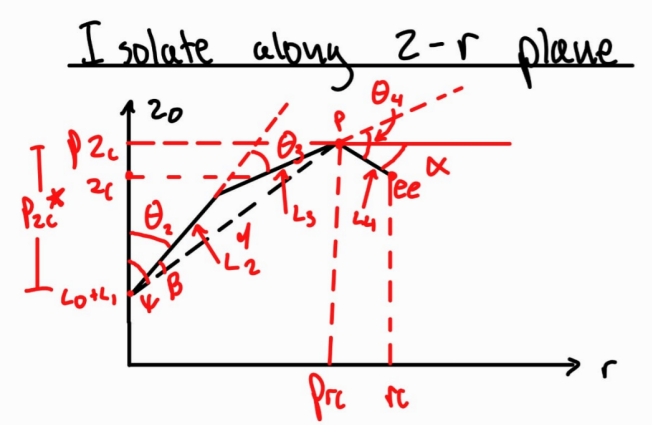


Fig. 3: Sketch of the Arm in the Z-R Plane

From the diagram shown above, we can find that:

$$p_{r_c} = r_c - L_4\cos(\alpha)$$

$$p_{z_c}^* = z_c + L_4\sin(\alpha)$$

$$p_{z_c} = z_c + L_4\sin(\alpha)$$

Now, we can transition to solving for θ_3:

$$\cos(\theta_3) = -\left(\frac{L_2^2 + L_3^2 - (p_{zc}^*)^2 + (p_{r_c})^2}{2L_2L_3}\right) = D_3$$

$$\sin(\theta_3) = \pm\sqrt{1 - {D_3}^2} = C_3$$

$$\theta_3 = \text{atan2}\left(\pm C_3, D_3\right) - \theta_3^*$$

Where θ_3^* is an angular offset for joint 3.
Using, θ_3, we can now solve for θ_2:

$$\psi = \text{atan2}(p_{r_c}, p^*_{z_c})$$

$$\cos(\beta) = \frac{L_2^2 + (p^*_{z_c})^2 + (p_{r_c})^2 - L_3^2}{2L_2\sqrt{(p^*_{z_c})^2 + (p_{r_c})^2}} = D_\beta$$

$$\sin(\beta) = \pm\sqrt{1 - D_\beta^2}$$

$$\theta_2 = \pm(\psi - \beta) - \theta_2^*$$

Where θ_2^* is an angular offset for joint 2.

Now, we just need to solve for θ_4 and since we are given α, we can find θ_4 as:

$$\theta_4 = \alpha - \theta_2 - \theta_3$$

However, we know that we want our robot arm to be in the "elbow-up" configuration. Therefore, we were able to eliminate all redundant computation which solved for irrelevant negative versions of the correct joint angles for our given goal configurations:

$$\theta_1 = \text{atan2}\left(\pm C_1, D_1\right)$$

$$\theta_2 = \beta - \psi - \theta_2^*$$

$$\theta_3 = \text{atan2}\left(C_3, D_3\right) - \theta_3^*$$

$$\theta_4 = \alpha - \theta_2 - \theta_3$$

Note that the value for θ_1 still depends on if or not y_c is positive or negative.

C. Velocity Kinematics

Now that we solved for the inverse kinematics of our robot, we then moved onto deriving the velocity kinematics. We first started by solving for $J_p(q)$ using the time-derivative method. In this method, we took the partial derivatives with respect to each joint and each equation for x, y, and z respectively. Then, we placed those partial derivatives in their matching indices in the $J_p(q)$ matrix.

Then, for $J_w(q)$, we simply used the third column of each rotation matrix which then gave us the corresponding joint's effect on the angular velocity of the end effector.

Please note that the matlab script for deriving this matrix is located in Appendix A and we could not include the matrix in the paper due to the sheer size of all of its elements.

D. Camera Setup: Intrinsic and Extrinsic Calibration

Our team then transitioned to setting up and calibrating our camera. We accomplished this using the Camera Calibrator app which is built into Matlab.

Using this app, our team took 40 images with which to calibrate our robot arm. The images then went through a filtering process where 30 images were shown to be acceptable. In this case, acceptable meant that the x-axis, y-axis, and origin were all properly shown.

We then cut down our images to 20 in an effort to quicken the camera calibration process and were left with 20 high-quality images with properly labeled axis and coordinates like the one shown in Figure 8.

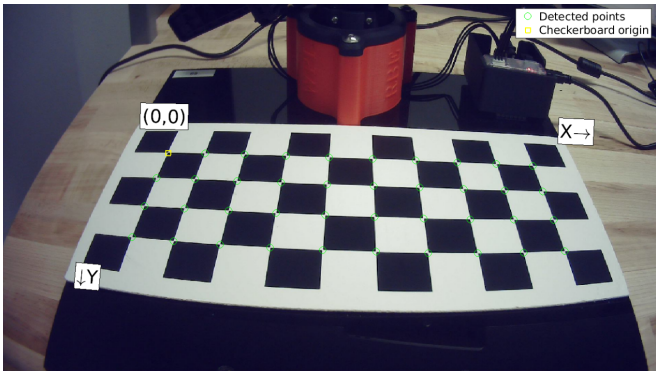


Fig. 4: Mean Error in Pixels

The results of camera calibration will be shown in Sec.III.C

E. Object Detection and Localization

Our focus then shifted to being able to find and localize objects in our camera's field of view.

1) Masking the Complement of the Checkerboard: The first step in accomplishing this goal was to mask out anything in our camera's view that wasn't the checkerboard. This was done in an effort to be able to "block out" harmful and unnecessary noise that could confuse the robot.

We accomplished this by using the function poly2mask and feeding in coordinates using the ginput function. Thus, every time we finished calibrating the camera, we would be prompted to select the 4 corners of the checkerboard and then the mask would be created around those points. The results of this and any other vision-pipeline related result will be shown in Sec.III.D

2) Creating Colored Masks: Then, we needed a way to be able to focus on the "target colors" that our robot is interested in. In this example, the objects were a grey ball, red ball, green ball, orange ball, and yellow ball.

Thus, for each of these "target colors", we created a color mask using the inbuilt colorThresholder application in Matlab. These masks helped us block out any other color that we did not care about and made finding the balls much easier. As previously mentioned, any vision-pipeline related result will be shown in Sec.III.D. Furthermore, please note that we brought all RGB images to the HSV color space

since it made detecting the target objects easier and more accurate.

3) Localizing Objects: Now that we had a method to be able to mask out other colors besides our target objects, we needed a way to detect the objects using the camera and to then convert the position in the camera frame to the position in the robot-base frame.

The first step in this process was to use the Matlab function: imFindCircles. This function then returned the centers and radii of each ball that it detected per mask. This would become very useful later in the project when we wanted to sort each colored ball differently.

Using the information of the centers of each ball, we were then able to use the matlab function pointsToWorld which, by using other information such as our cameraIntrinsics, returned the position of each ball in the checkerBoardFrame. We were then able to correct this and transform the given position into the robot-base frame using the following operation:

$$R_0 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

$$t_0 = \begin{bmatrix} 113 \\ -70 \\ 0 \end{bmatrix}$$

$$T_0 = \begin{bmatrix} R_0 & t_0 \\ \mathbf{0} & 1 \end{bmatrix}$$

$$r_{pos} = T_0^{-1} \begin{bmatrix} worldPt^T \\ 0 \\ 1 \end{bmatrix}$$

Using the aforementioned process, we were able to get all of the center positions of all target objects in the robot-base frame.

4) Other Methods: In this process of arriving at our preferred vision pipeline, we attempted other methods such as trying to use functions such as regionProps to try and ascertain the same knowledge as what imFindCircles was able to provide us.

However, we chose to not pursue any other option due to the cleanliness of code when implementing imFindCircles since it is able to return the centers and radii of each ball in one function call as opposed to the multi-line function calls that other methods needed.

F. Object Position Correction

To illustrate this problem, please take note of Figure 5.

As you can see in this figure, the position returned by pointsToWorld is not our desired return value since it is clearly not the same as the actual center of the ball. Therefore, this requires us to do the following math as a corrective action:

cam_x and cam_y are the incorrect values that are provided by the camera

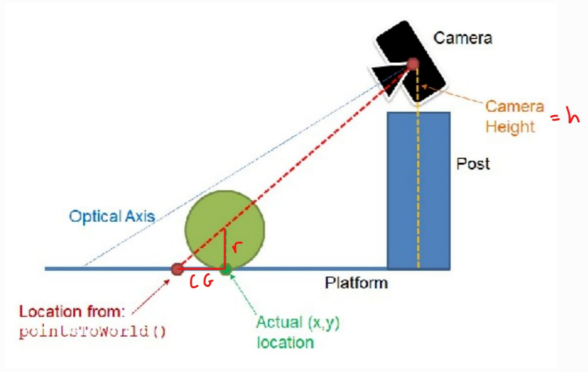


Fig. 5: Diagram for Object Position Correction

We then find the vector from the given ball position to the base of the camera (named "BV"):

$$BV = \begin{bmatrix} cam_{base_x} - cam_x & cam_{base_y} - cam_y \end{bmatrix}$$

Then, we take the magnitude of the vector to save and convert the aforementioned vector into a unit vector:

$$\hat{BV} = \frac{\begin{bmatrix} cam_{base_x} - cam_x & cam_{base_y} - cam_y \end{bmatrix}}{\|BV\|}$$

Finally, we find our correctiveGain and use that as our magnitude with which we can correct our original values by:

$$CG = \frac{r \cdot \|BV\|}{h}$$

$$CV = CD \cdot \hat{BV}$$

$$pos_{actual} = CV + pos_{cam}$$

All of the aforementioned math has been put into a function that we call every time we find a new ball. Also, please note that our code includes an offset in both the x and y directions which are the result of a constant offset that our robot had when being told to go to a goal position.

G. Object Sorting

Now that we can find and correct for the true position of the balls, we turned our sights to sorting them. Our process was very simple:

1) Take a snapshot with the camera
2) Use the aforementioned vision pipeline to find all of the balls
3) Travel to and sort all of the grey balls
4) Repeat for all of the different colored balls
5) Take a new snapshot and repeat the rest of the process

The idea behind this process is that we wanted our robot arm to be able to sort indefinitely, meaning that even if it finishes the "original batch" of object, the user will be able to place more objects on the checkerboard which will get detected in the next "snapshot".

Also, by ignoring any new balls that get placed on the checkerboard besides the "original batch", we ensured that the robot arm would never get disturbed nor confused by the presence of a new object since the object will not get detected until the next cycle. Please note that all of the balls get sorted to different positions which are defined at the top of the matlab script. For example, the sorted position for grey balls is named "greySortedPos".

Furthermore, one key feature to note is that we implemented cubic trajectory planning in the joint space throughout our implementation as it was the simplest to implement and that due to the lack of large movements, our robot would have negligible acceleration and therefore, acceleration did not need to be accounted for by the implementation of a quintic trajectory planner. Thus, throughout our implementation, we used a function that we previously made called interpolate_jp() which performs a cubic trajectory in the joint space given a desired time of travel.

H. Dynamic Tracking

To execute dynamic tracking, we first send the robot arm to position of the object using the corrective action discussed in SEC.II.F. Then, the robot arm pauses until the dynamic tracking is ready to start at which point the arm then follows the object as it moves through the checkboard. In our case, we chose to use a green ball on the end of a black pole to be our "tracked object"

I. Manipulation of a Random Object

In order to manipulate a random object, our first step was to create a new mask for our new object. Thus, since we knew that we needed a mask for the object, we strategically chose the object shown in Figure 6 which is a different color than any of the other objects that we care about sorting. Then, after we found the object, we then repeated similar steps in creating a new colored mask for the object and creating a new function for grasping the monkey stuffed animal.

Then, we adapted our sorting script to sort the monkey stuffed animal last in its sorting sequence. While doing so, we also realized that the stuffed animal was slightly larger than the balls that were orginally sorting and therefore, needed to slightly change the range of radii that imFindCircles accepted for the stuffed animal.

Furthermore, after implementing all of the aforementioned functionality, we generated a State Diagram which is shown in Appendix C.

III. RESULTS

This section will cover all of the results, plots, and figures that are associated with the aforementioned methodologies.

Fig. 6: Random Object - Kipling Monkey

A. Transformation Matrices

To begin, we start with revealing the transformation matrices from each joint to the next where in T_i^{i+1}, "i" is the current joint and "i+1" is the next joint:

$$T_0^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & L_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_1^2 = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & L_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^3 = \begin{bmatrix} \cos(\theta_2) & 0 & -\sin(\theta_2) & 0 \\ \sin(\theta_2) & 0 & \cos(\theta_2) & 0 \\ 0 & -1 & 0 & L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^4 = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & L_2\cos(\theta_3) \\ \sin(\theta_3) & \cos(\theta_3) & 0 & L_2\sin(\theta_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_4^5 = \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & L_3\cos(\theta_4) \\ \sin(\theta_4) & \cos(\theta_4) & 0 & L_3\sin(\theta_4) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

Please note that the matrix from T_0^5 is solvable, however, the sheer size of that transformation matrix does not allow it to properly fit on the page. Thus, we have chosen to omit showing the matrix in its original state and are instead representing it as the product of the consecutive matrices from the base to the end effector.

B. Visualization of the Robot Arm

By using the transformation matrices from the base to each joint, we are able to visualize the robot arm as shown in Figure 7.

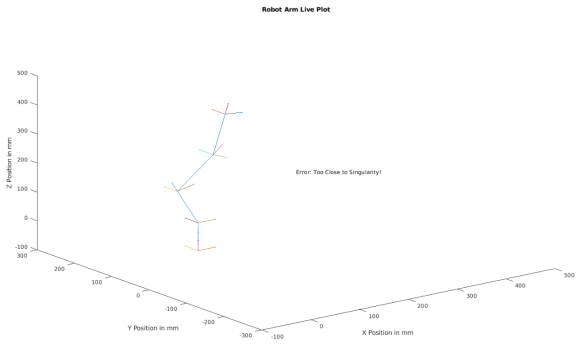


Fig. 7: Digital Visualization of Robot Arm

Please note that this visualization is incredibly computationally expensive due to all of the trigonometric operations as well as it's sheer number of operations. Thus, despite its visual appeal, we chose not to run it during any of our trajectories due to how slow it made our trajectories.

C. Camera Calibration Figures

The following histogram and 3d plot in Figures 8 and 9 are the results of our camera calibration.

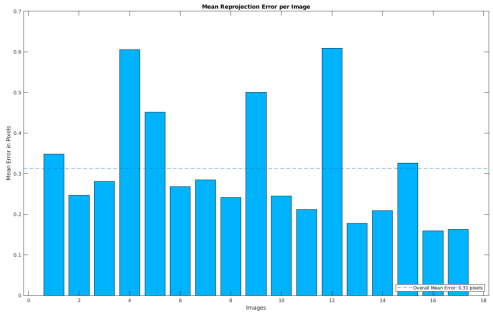


Fig. 8: Mean Error in Pixels

Our original goal was to ensure that the mean error in pixels never exceeded 1 pixel during calibration and as the figure above shows, we have accomplished this goal since the highest mean error is around 0.6 pixels.

This next figure visualizes the extrinsic parameters and helped us as developers "see" what the camera sees while calibrating:

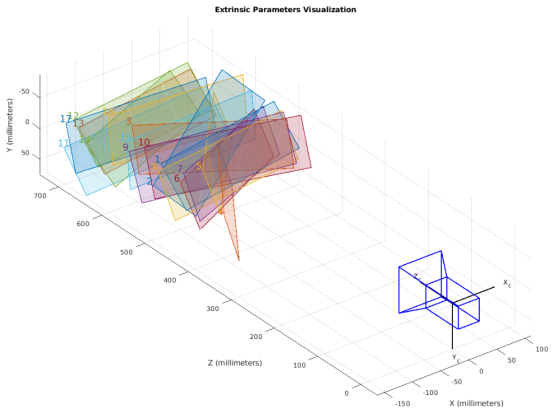


Fig. 9: Extrinsic Parameters Visualization

D. Vision Pipeline

This following section reveals all of the vision pipeline related results and please note that all of our results includes random object which is also recognized by our vision pipeline as a target object after the previously mentioned integration.

1) Original Image: Figure 10 is the original image taken by the camera that gets processed in the later "subsubsections".

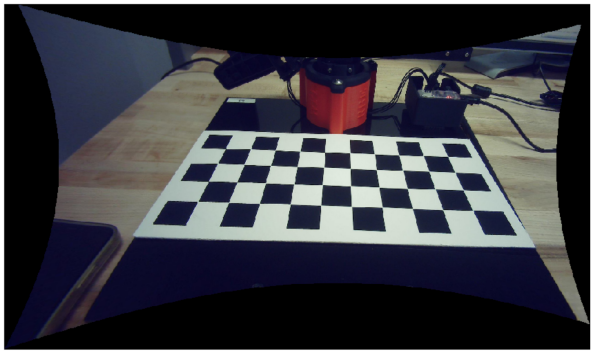


Fig. 10: Original Snapshot from Camera

2) Blocking out the Noise: The first process that the original image goes through is the masking out of anything else that isn't the checkerboard. As previously mentioned, we use the inputs from ginput that gives us the points of each of the corners in the camera frame.

Also, note that there are now objects in the image. This is because one group member places the objecst on the checkboard while the other provides the inputs into ginput.

3) Creating Masked Images: The second step after acquiring the original image is to apply the colored masks for each ball. After doing so, we are left with the masked images like the example of the red and green balls in Figure 12.

4) Localizing Target Balls: Figure 13 shows the end result after using imFindCircles to find the centers and radii of the balls. After finding that information, we overlay drawn circles at the center of where the vision pipeline assumes

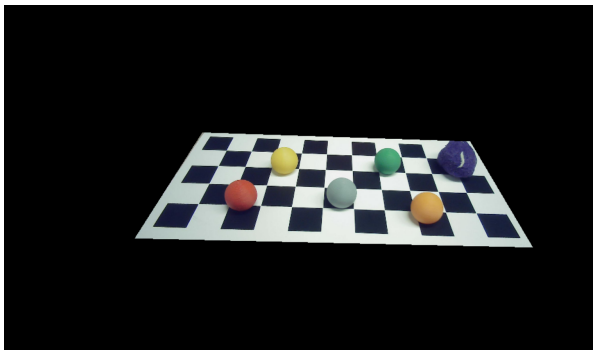


Fig. 11: Masked Checkerboard Image

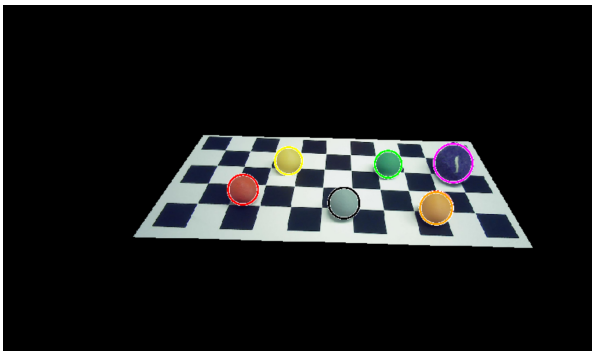
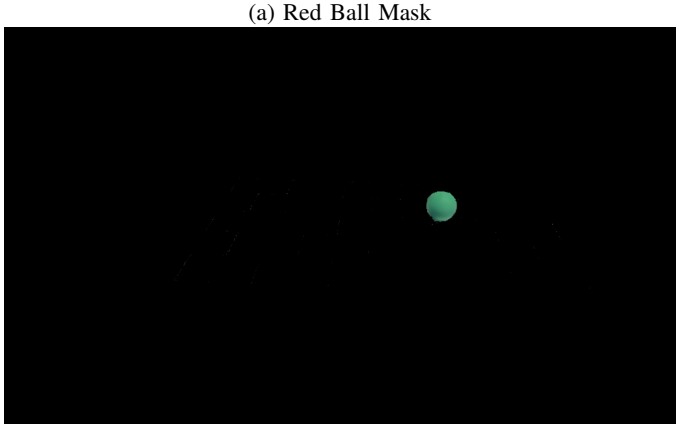


Fig. 13: Detected and Localized Balls

(a) Red Ball Mask



(b) Green Ball Mask

Fig. 12: Masks Images for the Objects P1

a ball is located. By overlaying the circle that the vision pipeline "sees", we can verify that the vision pipeline is returning an accurate result

E. Experimental Results

Our experimental results are incredibly successful with the robot arm always sorting all of the target objects on the checkerboard. While there are scarce issues with the color masking sometimes being unreliable due to changes in lighting conditions, the cyclical nature of the sorting algorithm results in the robot arm always detecting target objects that it might not have detected in its "original batch". If you would like to see a supplement in-depth explanation and a live demo of the sorting and dynamic tracking algorithms, please visit this youtube link: https://www.youtube.com/watch?v=2cv498b8PI4

IV. DISCUSSION

1) Prerequisite Technical Work: As we developed correctly in the past labs during the term the forward kinematics, inverse kinematics, trajectory planning and velocity kinematics of our robotic arm, we were able to implement everything together in a picking and placing system. As we went through the methodology, the first step of our final project was to perform the intrinsic and extrinsic calibration of the camera. We achieved the intrinsic calibration first by using the camera calibration app on MATLAB, which we did by moving the camera around the checkerboard to record different view angles and orientation. After having recorded around forty pictures, we selected twenty of these pictures making sure that the X and Y axis were aligned consistently in each photo, and after we selected the camera model, which is "fisheye", we ran the camera calibration. This step will enhance the program ability to estimate and correct any distortion created by the camera optics. We were able to evaluate the results of our calibration performing an analysis of the histogram of reprojection errors. 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. We tried to achieve a reprojection error smaller than 1 pixel, which did as the largest reprojection error was of 0.6 pixels. After we were happy with our calibration results we exported the camera parameters as a script in matlab to then use it later on in the project.

2) Camera Calibration: We, as well, performed the extrinsic calibration of the camera running the function getCameraPose(), which calculates the extrinsic calibration parameters of the camera and calculates the transformation matrix from the frame of the image to the frame of the checkerboard, and pointsToWorld() which transform from image frame to checkerboard frame. Calculating the transformation matrix between the robot's base frame and the checkerboard's frame, we were able to relate the position of objects within the field of view of the camera to robot task space coordinates. We were also able to visualize the

extrinsic parameters using Figure 9, which helped us "see" what the camera sees while calibrating.

3) Vision Pipeline: We then developed a reliable object recognition system that can identify and label the centroid in image space and color of all of the balls in the usable portion of the workspace. We did so by masking first the workspace of the checkerboard passing in the function poly2mask four arbitrary points as inputs that we select using the ginput function every time the camera is done calibrating. After masking the checkerboard, using the HSV color space in the inbuilt ColorThresholder application in MATLAB, we created a mask for each ball by blocking any other color to just be able to target the ball of interest, which we then exported as matlab functions. We were then able to find the balls by passing as input the masked images for each ball in the function imFindCircles which returns the centers and radii of each ball. Then passing the centers and radii of the balls that were outputted from the function imFindCircles in the viscircles function we were able to outline the edge of each ball.

4) The Sorting Algorithm: We were finally able to develop an algorithm for our picking and sorting system by running a while loop in which: we take a picture, mask the checkerboard, mask each ball that was detected by the camera (sometimes due to the lighting conditions the camera wasn't able to find certain balls on the checkerboard), and after finding the balls we converted the centroids of each ball to the task space x,y using forward kinematics, and corrected their actual location with a function that we implemented ourselves based on the math and calculations discussed in the Section F: "Object Position Correction" of the methodology we were able to determine the pose of each ball in the task space. After we know where each ball is and we determined a certain joint configuration for each ball to be sorted to, our robotic arm will go to the pose of each ball in the task space, will open its gripper at an height of the radius of the balls from the checkerboard, it will close the gripper grabbing the ball and it will then move to the sorted location, which is different for each ball, by using inverse kinematics, and once it will be at that location it will open the gripper dropping the ball.

If a ball has not been detected at first, the program will run picking up every other ball and, at the end of the loop, it will take another picture and it will be able to find the ball that wasn't found at first, and as well any other new ball that was placed before the camera takes the new picture. So we were able to implement a state machine that keeps sorting indefinitely until there are no balls or objects on the checkerboard.

5) Extra Credit Work: For extra credit, we were supposed to be able to pick and sort an object that wasn't a ball, and we decided to use a stuffed monkey toy. We did this by creating a new mask for the monkey, creating a purple mask, and running a similar steps that we did for the other balls. The major difference for the stuffed monkey toy was that its size was a little bigger than the balls', so we just had to slightly change the range of radii that imFindCircles accepted for the stuffed animal. We finally adapted our sorting script to sort the monkey stuffed animal last in its sorting sequence and we were able to perform the first extra credit task.

The second extra credit task was to execute a dynamic camera tracking of the balls, in which we were supposed to move a ball along the checkerboard and the robot had to follow the ball around at a certain height. We did this dynamic camera tracking for the green ball by first sending the robot arm to the position of the ball using the corrective action discussed in Section F of the methodology and, when the dynamic tracking was ready to start, we were able to move the ball around with the robotic arm following it.

V. CONCLUSION

This lab was extremely interesting and rewarding as it gave us the perfect opportunity to not only develop a computer vision pipeline but also to integrate everything we learned throughout the course to solve a real-world problem. Beyond technical skills, this lab also pushed us in terms of documenting and presenting our work by requiring that we create a proper, conference-style paper through which to present our work.

VI. Appendix A: Jacobian Derivation

Listing 1: Matlab Script

```
syms theta1 theta2 theta3 theta4
syms theta5 L0 L1 L2 L3 L4

% DH parameters [theta, d, a, alpha]
dh_params = [
theta1, L1, 0, -90;
theta2 - 79.4, 0, L2, 0;
theta3 + 79.4, 0, L3, 0;
theta4, 0, L4, 0;];

% Compute transformation matrices
T01 = [1, 0, 0, 0;
0, 1, 0, 0;
0, 0, 1, L0;
0, 0, 0, 1];
T12 = dh2matfunction(dh_params(1,:));
T23 = dh2matfunction(dh_params(2,:));
T34 = dh2matfunction(dh_params(3,:));
T45 = dh2matfunction(dh_params(4,:));

% Compute ee transformation matrix
T05 = T01 * T12 * T23 * T34 * T45;
fx = T05(1,4);
fy = T05(2,4);
fz = T05(3,4);

J = sym(zeros(6,4));

%upper half jacobian
J(1,1) = simplify(diff(fx, theta1)); %p11
J(2,1) = simplify(diff(fy, theta1)); %p21
J(3,1) = simplify(diff(fz, theta1)); %p31
J(1,2) = simplify(diff(fx, theta2)); %p12
J(2,2) = simplify(diff(fy, theta2)); %p22
J(3,2) = simplify(diff(fz, theta2)); %p32
J(1,3) = simplify(diff(fx, theta3)); %p13
J(2,3) = simplify(diff(fy, theta3)); %p23
J(3,3) = simplify(diff(fz, theta3)); %p33
J(1,4) = simplify(diff(fx, theta4)); %p14
J(2,4) = simplify(diff(fy, theta4)); %p24
J(3,4) = simplify(diff(fz, theta4)); %p34

%lower half jacobian
J(6,1) = 1;
J(5,2) = 1;
J(5,3) = 1;
J(5,4) = 1;
```

VII. Appendix B: Authorship

Section	Author
Planning	Whole Team
Coding	Whole Team
Experimentation	Whole Team
Abstract	Lorenzo
Background / Motivation	Lorenzo
Methodology	Nikhil
Results	Nikhil
Discussion	Filippo
Conclusion	Nikhil
Video	Filippo

VIII. Appendix C: Class Diagram

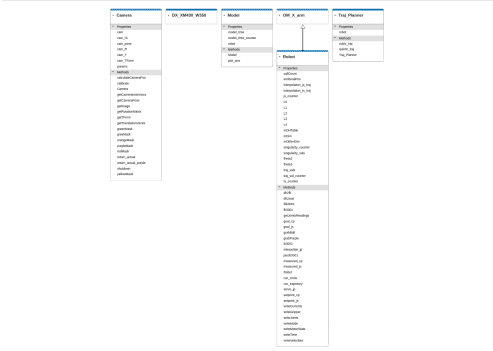


Fig. 14: The Class Diagram Generated by Matlab

References

[1] Purdue Online Writing Lab. *Professional Technical Writing*. [Online]. Available: `http://owl.english.purdue.edu`

[2] The Writing Center at the UW Madison. *Scientific Reports*. [Online]. Available: `http://writing.wisc.edu/Handbook/ScienceReport.html`

[3] The Writing Centre, Queen's University, Canada.

[4] Norton, R.L., *Design of Machinery*, 4th ed., Mc Graw Hill, 2008.