Robotic Object Classification and Manipulation

Elwell, Gavin
dept. of Robotics Engineering
dept. of Mechanical Engineering
Worcester Polytechnic Institute
Worcester, MA, United States
gpelwell@wpi.edu

Behrens Chase
dept. of Robotics Engineering
dept. of Computer Science
Worcester Polytechnic Institute
Worcester, MA, United States
chbehrens@wpi.edu

Hage, Charbel Saeed
dept. of Robotics Engineering
dept. of Mechanical Engineering
Worcester Polytechnic Institute
Worcester, MA, United States
cshage@wpi.edu

***Abstract*—Robot manipulation applies to many industries. While many robots are hard-coded to move in repetitive paths, by implementing computer vision, we were able to make the robot arm more environment-dependent, enabling it to pick up objects from different positions within the workspace. In our case, we were simply sorting balls by color; however, this could be adapted to perform arbitrary pick and place tasks. For example, this could be implemented in an industrial setting, performing quality control checks by sorting out products that do not meet specifications.**

I. Introduction

The goal of this project was to enable the robot to detect, extract, relocate, and sort items within its workspace. To enable the robot arm to perform these tasks, we used the Davit-Hartenberg convention to determine the forward kinematics and the inverse kinematics for the arm. We used Jacobian and differential kinematics to prevent the robot from moving into singularity positions.

The robot is a RRRR (Revolute-Revolute-Revolute-Revolute) robot according to the Davit-Hartenberg convention. The arm is controlled by four DYNAMIXEL servo motors; the first one has its z-axis pointing upward, while the rest have their z-axes pointing out of the page. Fig. 1 depicts the robot arm system.

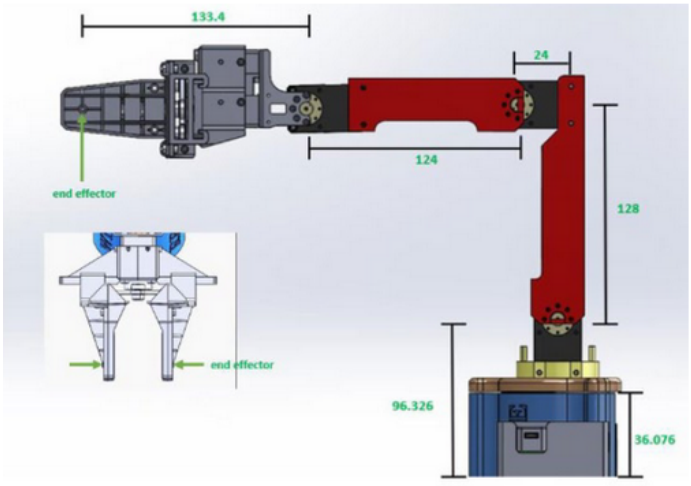

Fig. 1. A diagram depicting the robot arm used.

II. Program Design Decisions

We chose to group all of the image processing-related functionality into an aptly named image processor class. These functions include: generate static mask, color mask, image to robot, detect centroids, and correct centroids.

We group all robot functionality into a robot class. This includes blocking JS move, blocking ts move, and picking up the ball functions. We chose to group the pick up ball functionality with the general robot class because we felt this robot was already specialized to completing pick and place tasks, thus this function could be repurposed for picking up any similarly sized object at the specified location.

Throughout the report, you will be able to see our design decisions for each of these functions in more detail.

III. Internal Motor Behavior

The DYNAMIXEL servo motors contain integrated sensors that allow them to execute controlled motion out of the box. To understand the motion profiles of the built-in motor control, we graphed the output of the angular position of each of the motors as they move from one angle to another, shown in Fig. 2.

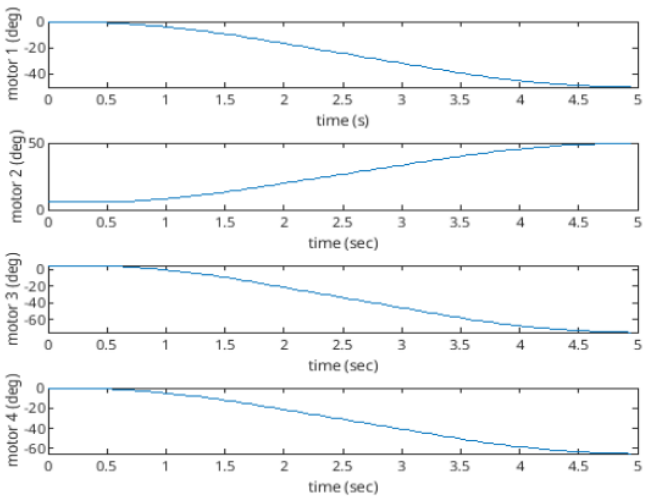

Fig. 2. Joint angles over time as the robot moves the motors to an angle.

When the motors are ordered to move to a position, their angular velocity is controlled by a trapezoidal function. The angular velocity increases as the motor accelerates. Once it reaches a fixed maximum angular velocity, it stops accelerating and maintains this constant speed. Lastly, the angular velocity decreases as the motor decelerates to zero. This is visible in Fig. 2, where the angular position curves at the beginning as the motor accelerates, followed by a perfectly uniform slope while the motor maintains a constant angular velocity, and a curve at the end of the motion as the motor decelerates.

IV. Forward Kinematics

The forward kinematics equations provide a method for computing the position of the end effector in space from the angular position of each of the robot joints. This is useful because it enables us to find the current position of the robot in space by reading the angular position values of each motor.

To test the forward kinematics algorithms, we programmed the robot to move to three different points in space and plotted both the angular position of the motors and the calculated position of the end-effector. Then we ensured that the results matched our predetermined positions and that the motion paths matched our expectations for what they should look like.

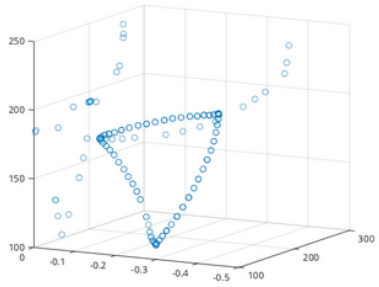

Fig. 3. Position of the end effector in space.

From Fig. 3, we see that the calculated positions of the end-effector location matched the actual position of the end-effector in space. The paths appear to curve slightly. This is because the angular positions of each motor are moving at a constant rate, which does not correspond to a straight trajectory of the end effector.

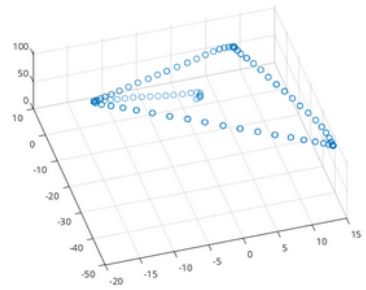

Fig. 4. The angular position of the servo motors, excluding the first revolute joint, facing upward at the base of the robot.

From Fig. 4, we see that the lines are very straight. This is caused by the trapezoidal control function being consistent across each of the motors. The bunching of points at the corners is caused by the acceleration and deceleration of the motors.

V. Inverse Kinematics

The inverse kinematics formulas provide a method of converting a position in space to a corresponding angle of each of the joints. This is useful because it enables the arm to find the corresponding joint position to reach any point within its workspace.

To formulate the Inverse Kinematics equations, a diagram of the arm's system was made at first, and using the geometric approach, it was concluded which axes were being affected by the change of angular position of each actuator (Please refer to Fig. 5 for more details on the process). Through these correlations, the formulas were concluded and can be observed at the rightmost side of Fig. 5.

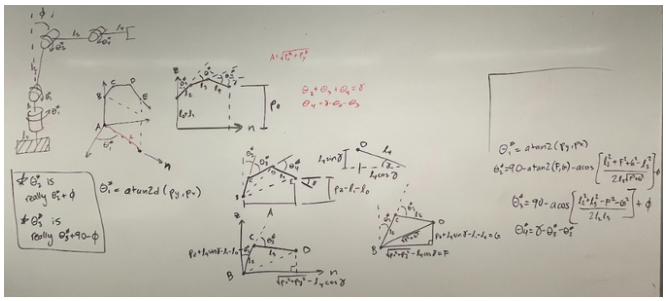

Fig. 5. Derivation of Inverse Kinematics formulas.

To test the effectiveness of our inverse kinematics formulas we programmed a function to move the arm in a straight line from the same points of the triangle as used earlier to test the forward kinematics. Assessing its motion we found that it was able to precisely move in the shape of the triangle. We also plotted the joint positions to see how they would differ from motion without inverse kinematics, shown in Fig. 6.

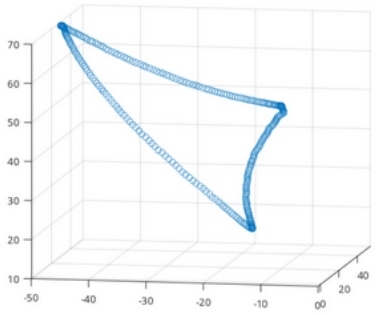

Fig. 6. The angular position of the servo motors, excluding the first revolute joint, facing upward at the base of the robot.

From Fig. 6, we see that the lines are now curved. This is because, in order to maintain a linear motion in task space, the velocity of each joint must vary independently.

VI. Trajectory Planning

To enable a smooth and linear motion of the end effector we implemented trajectory planning. To do this we used cubic and quintic equations of motion. Since the position of the end effector would be changing constantly, due to difference in initial and final position, functions were made with the following intention:

The Cubic function takes as inputs the initial time, initial velocity, final time, and final velocity. This function was provided with four equations, with the intention of calculating four unknown coefficients for the equations of motion.

The Quintic follows a similar scheme to the cubic function, but it takes in more variables and outputs more as well; this function takes in the inputs of initial time, initial velocity, initial acceleration, final time, final velocity, and final acceleration. This function was provided with six equations, with the intention of calculating six unknown coefficients for the equations of motion.

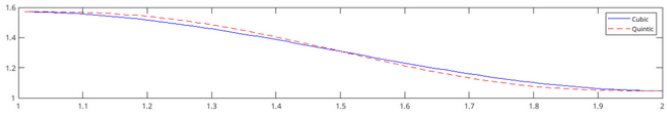

Fig. 7. Comparison for the Position Graph of the Arm's End Effector

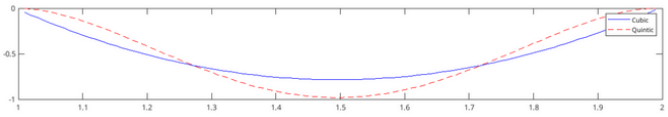

Fig. 8. Comparison for the Velocity Graph of the Arm's End Effector

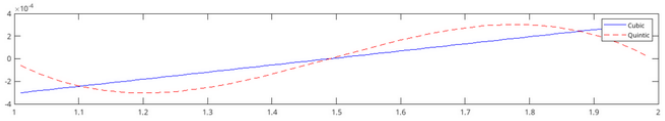

Fig. 9. Comparison for the Acceleration Graph of the Arm's End Effector

In Fig. 7-9, one can observe the difference between these approaches; a gentle but present difference. The reason why the use of these methods is vital is that they ensure a linear trajectory and a smooth transition from a dynamic state to a static state and vice versa, helping to prevent sudden jerking of the motors, allowing the system to work adequately for long periods of time, and extending the lifespan of the physical components of the arm.

Combining this with our forward kinematics and inverse kinematics equations, we were able to create a function that moves the end effector in a straight line from its current position to a new position specified within its workspace. It works by calculating the current position of the end effector using forward kinematics. Then it finds an array of points in a straight line from the current position to the target position with spacing determined by either the cubic or quintic function. Finally, it commands the end effector to move to each of these points using inverse kinematics.

VII. Singularities

Singularities are areas of the workspace in which the arm loses a degree of freedom of movement. This is harmful because when the robot passes through these points, the equations output an infinite velocity, which forces the motors to reach a high level of stress due to extremely sudden changes in motion. Determinants are used specifically by taking the determinant of the Jacobian matrix, which approaches zero as the system reaches a singularity.

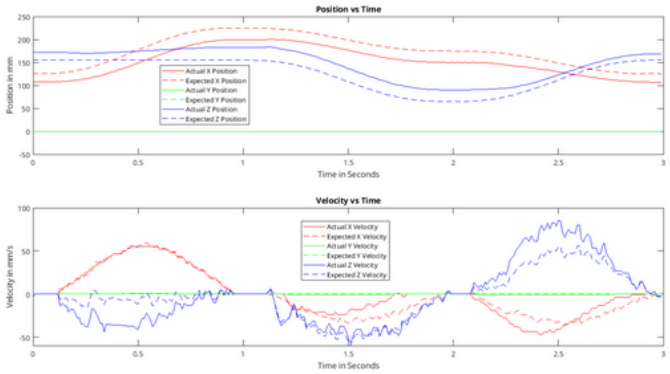

Fig. 10. Results of position and velocity using the Jacobian function

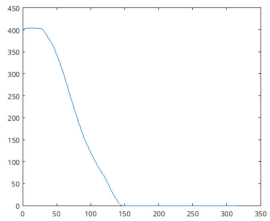

Fig. 11. Results of Determinant of Jacobian as approaching singularity with error stop.

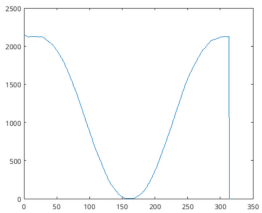

Fig. 12. Results of determinant of Jacobian as approaching singularity with NO error stop.

To prevent the robot from moving too close to singularities, we developed a function that uses the determinant of the Jacobian to detect when we are approaching a singularity. Fig. 11 shows how the determinant will drop to 0 as we approach a singularity. Our function detects when a determinant value of 0.1 or lower is reached, and an emergency stop is triggered, stopping the arm from reaching the Singularity's position.

VIII. Workspace Limits

The workspace of the arm is every point in space that the robot can reach, given its dimensions and joint limitations. We decided to limit the angle of the end effector between facing vertically down and vertically up. Using these constraints, we were able to determine a piecewise function that describes all of the reachable positions given a chosen angle of the end effector, shown in Fig. 13.

$$y = l_0 + l_2\cos g + l_3\sin g + \sqrt{l_1^2 - (x - l_3\cos g + l_2\sin g)^2}\ \{l_3\cos(g) - l_2\sin g - l_1 < x < -(l_1 + l_2)\sin g + l_3\cos g\}$$

$$y = l_0 + l_3\sin g + \sqrt{(l_1 + l_2)^2 - (x - l_3\cos g)^2}\ \{-(l_1 + l_2)\sin g + l_3\cos g < x < l_1 + l_2 + l_3\cos g\}$$

$$y = l_0 + l_3\sin g + \sqrt{l_2^2 - (x + l_1 - l_3\cos g)^2}\ \{l_3\cos(g) - l_2\sin g - l_1 < x < l_2 - l_1 + l_3\cos g\}$$

$$y = l_0 + l_3\sin g + \sqrt{(l_1 - l_2)^2 - (x - l_3\cos g)^2}\ \{l_2 - l_1 + l_3\cos g < x < (l_1 - l_2)\sin(-g) + l_3\cos g\}$$

$$y = l_0 - l_2\cos g - l_3\sin(-g) + \sqrt{l_1^2 - (x - l_2\sin g - l_3\cos g)^2}\ \{(l_1 - l_2)\sin(-g) + l_3\cos g < x < l_1 - l_2\sin(-g) + l_3\cos g\}$$

$$y = l_0 + l_3\sin g - \sqrt{l_2^2 - (x - l_1 - l_3\cos g)^2}\ \{l_1 - l_2\sin(-g) + l_3\cos g < x < l_1 + l_2 + l_3\cos g\}$$

Fig. 13. Equations governing the workspace of the arm where g is the angular position of the end effector.

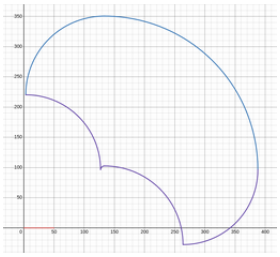

Fig. 14. Workspace when the end effector is pointed horizontally.

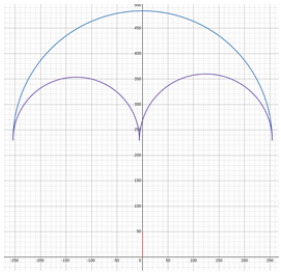

Fig. 15. Workspace when the end effector is pointing vertically upward.

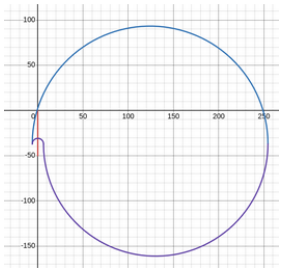

Fig. 16. Workspace when the end effector is pointing vertically downward.

We used the Equations in Fig. 13 to prevent the arm from attempting to reach positions outside of its workspace. This is important because the inverse kinematics functions are no longer valid for positions that the arm cannot reach. By adding this check, we prevent potential undefined behavior and can handle these errors in a much safer way.

IX. Image Processing

In the system, a camera is placed in front of the arm. The camera is used to detect the intersection points found within the checkerboard. As shown in Fig. 12, the total mean error was approximately 0.59 pixel(s). Meaning that the location of an object, within the camera's perception, is within a 59 percent unit length of error from what the camera detects in comparison to the actual value. The unit value of error increases the further away the object perceived is in the real world space, due to greater distortion at the edges.

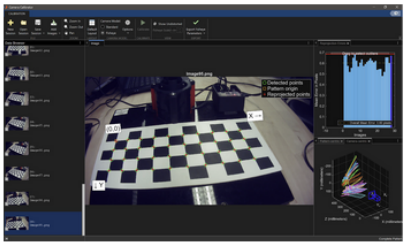

Fig. 17. Calibration of the camera.

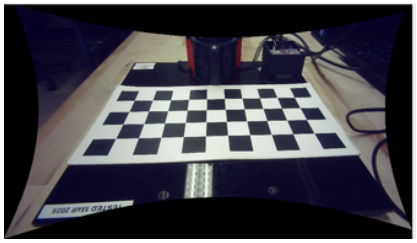

Fig. 18. Undistorted view of workspace.

A. Calibration

We start by calibrating the camera using a checkerboard pattern in our workspace. Matlab uses the image data from all the pictures to map pixel locations to a location in physical space. Due to the large amounts of picture data and the regular shape of the checkerboard pattern, the camera can account for the distortion caused by the lens. Using the known size of the square, the camera can scale the image to match its physical dimensions accurately. It creates an intrinsic camera matrix that, when multiplied by the pixel coordinate, transforms the coordinates to an undistorted version as shown in Fig. 18.

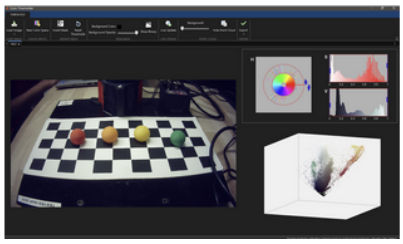

Fig. 19. Calibration of hue range for each color of object.

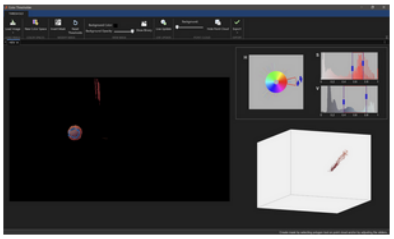

Fig. 20. Calibration for red objects.

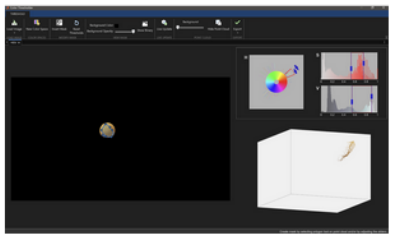

Fig. 21. Calibration for orange objects.

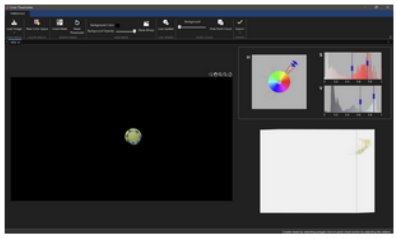

Fig. 22. Calibration for yellow objects.

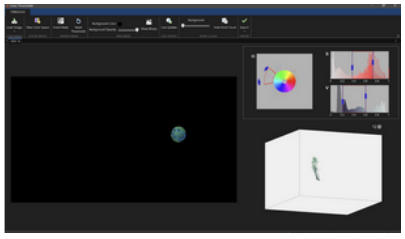

Fig. 23. Calibration for green objects.

B. Masking

A mask is used to filter out objects outside the workspace. The checkerboard square detection was used to find the corners of the camera's viewpoint in order to only process objects in the workspace. After this, different masks are applied to detect different colored objects. We used the HSV color space to filter the image for pixels within a range of hues, depending on the color of the object being processed shown in Fig. 19.

C. Position Computation

With these different masks, the location of the centers of mass of these objects can be determined in the camera's coordinate system (pixel coordinate system), and through mathematical equations, it is translated to the checkerboard's plane, and then translated to the arm's workspace plane. The process to determine the transformation matrix from the Arm's frame to the checkerboard's frame was the following:

$$Z_{rotation} = \begin{bmatrix} cos(90) & -sin(90) & 0 & 0 \\ sin(90) & cos(90) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$X_{rotation} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(180) & -sin(180) & 0 \\ 0 & sin(180) & cos(180) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Translation = \begin{bmatrix} 1 & 0 & 0 & 90 \\ 0 & 1 & 0 & -110 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = Translation * Z_{rotation} * X_{rotation}$$

$$T = \begin{bmatrix} 0 & 1 & 0 & 90 \\ 1 & 0 & 0 & -110 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The procedure to depict the location of a colored object from the camera to the checker board was lengthier: First data points were recollected from the points to world function which returned each intersection's coordinate, and later these points were found to the best of our ability by using the comment feature when using the imshow() function in Matlab, these values were recorded and can be found in seen in Fig. 24.

U:	X:	V:	Y:	U:	X:	V:	Y:
310	0.0678	228	-0.6213	530	125.0492	227	-0.4156
298	0.1301	253	25.1211	532	124.9818	253	25.0162
283	0.0636	283	50.1827	535	125.0906	282	50.2446
264	-0.1726	320	74.7021	537	125.0176	315	74.6663
356	25.0713	228	-0.1621	572	149.9861	226	-0.1586
345	25.1581	254	25.2	578	149.9361	252	25.3958
333	25.0441	283	50.0811	584	149.9597	280	50.1311
319	24.922	318	74.8485	589	150.0833	315	74.8545
400	50.0653	227	0.2369	615	175.0094	226	-0.28
392	50.0479	253	25.3561	624	174.9313	252	25.112
384	49.9944	282	50.1316	632	174.985	281	50.0261
374	49.9309	317	75.0286	642	175.0181	314	75.077
443	74.9847	227	-0.1227	658	199.9593	225	-0.2632
439	74.9704	253	25.3041	668	199.8859	251	25.2014
434	74.9593	283	50.1416	680	199.95	280	50.0206
429	74.9181	317	74.8936	695	200.0972	314	74.9591
487	99.952	227	-0.3202	700	224.9788	225	-0.0644
486	99.9696	253	25.1455	713	224.7381	251	24.985
485	99.9617	282	50.0524	729	224.8712	280	49.9743
484	99.9756	316	74.7355	747	225.0286	313	74.7572

Fig. 24. Correlating Pixel Coordinates to checkerboard coordinates.

U values are the x coordinates in the camera's frame, X values are the checkerboard's coordinate equivalent of these U values, V values are the y coordinates in the camera's frame, and Y values are the checkerboard's coordinate equivalent of the V values. With the recorded data, plots were made in Desmos, and represented them to the best of the program's ability using lines of best fit functions, as shown in Fig. 25.

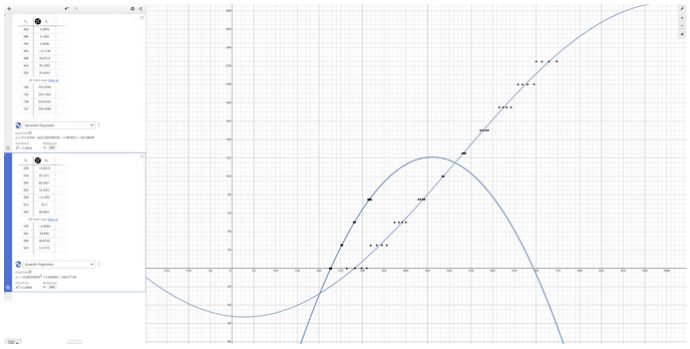

Fig. 25. Correlating pixel coordinates to checkerboard coordinates.

D. Centroid Correction

Converting pixel coordinates to checkerboard coordinates is very helpful. However, since the camera is converting from one 2D plane to another 2D plane, the position of the balls will be misinterpreted. Using trigonometry and the camera's transformation matrix, the true positions of the balls can be determined.

Before any math with the balls gets involved, the camera's position relative to the checkerboard must be determined first. Conveniently, the "camera" object provides a translation vector and a rotation matrix for the transformation matrix between the camera and the checkerboard.

Since the x-axes for the camera and checkerboard are similar, we can assume that the x-component of the translation vector is the camera's x-position in the checkerboard's space. Determining the y- and z-components is not as simple.

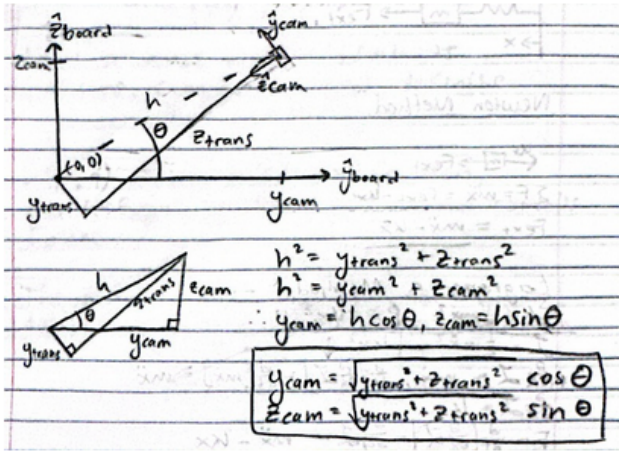

Fig. 26. The translation from the camera to the checkerboard

As pictured in Fig. 26, the assumption can be made that the camera's distance from the origin in the yz-plane is equivalent to both the hypotenuse of the y- and z-components of the translation vector, and the hypotenuse of the camera's y- and z-coordinates in the checkerboard's frame.

Using the rotation matrix, theta can be determined with relative ease since it is a rotation about the x-axis. Knowing the y- and z-components of the translation vector and theta, the y-and z-positions of the camera relative to the checkerboard can be determined.

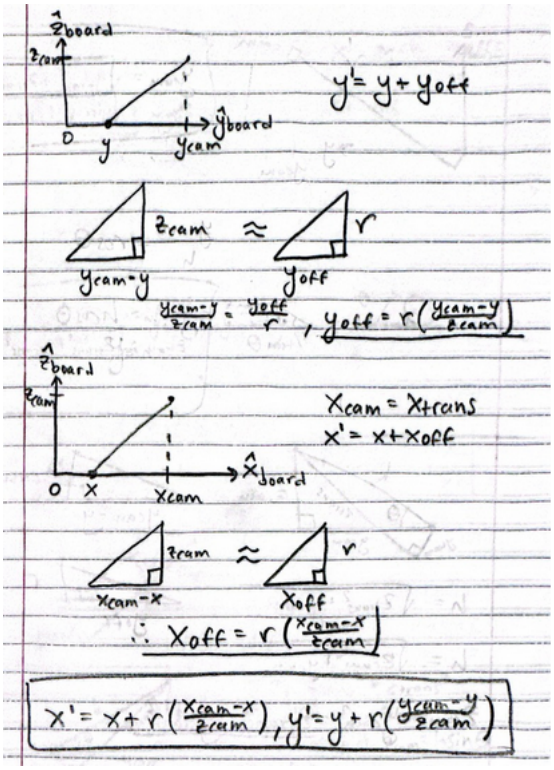

Fig. 27. Offset in the X-Y plane of the checkerboard from the camera's perspective.

Now that the position of the camera relative to the checkerboard is known, the offsets in the x- and y-directions can be found as well. Luckily, the right triangle created by the camera's position and the false centroid position is similar to the right triangle created by the radius of the ball and its offset, as pictured in Fig. 27. Using ratios, the offset in both dimensions can be found and applied to the true x- and y-positions. Now the camera can determine the true centroid of the ball regardless of its position on the checkerboard.

X. Conclusion

By combining all of our previous work together, we were able to create a single program that would collect and sort all objects colored green, red, yellow, and orange, into corresponding bins set at 4 locations specified in Table 1.

TABLE I
COORDINATES OF DROP OFF LOCATIONS

Location Color	**Coordinates (robot arm frame)**	
	X	***Y***
Green	75	-190
Red	75	190
Yellow	10	-190
Orange	10	190

The program starts by calibrating the camera and masking out all irrelevant areas of the camera view. Next, it iterates over the list of colors. For each color, it further masks the image to identify the relevant colored pixels and calculates the centroid of pixel groups that meet a specific size threshold, thereby eliminating noise. After these coordinates are found in the Camera space, they are translated to the Robot Arm space, for which the arithmetic was explained throughout the document. Finally, these coordinates are placed in a loop, in which the arm proceeds to approach, pick up, and sort the objects within the workspace, executing these steps with adequate precision and consistency, which is evident in the following video.

REFERENCES

[1] WPI RBE3001 A25 Team One_01, *RBE3001 GitHub Repository*, https://github.com/WPI-RBE-300n/RBE3001_A25_01, Accessed: Oct. 10, 2025.

[2] WPI RBE3001 A25 Team One_01, *RBE3001 Project Video*, https://drive.google.com/..., Accessed: Oct. 10, 2025.

TABLE II
CONTRIBUTIONS

Task	**Group Member Names**		
	Elwell, Gavin	***Behrens, Chase***	***Hage, Charbel Saeed***
Analysis of Motor Behavior	Programming	Programming Write Up	Programming Write Up
Forward Kinematics	Derivation Programming Write Up	Derivation Write Up	Derivation Write Up
Inverse Kinematics	Derivation Programming	Derivation Write Up	Derivation Write Up
Differential Kinematics	Programming	Derivation	Derivation
Singularity Detection	Programming Write Up	Derivation Programming	Derivation Write Up
Trajectory Generation	Programming Derivation	Derivation	Write Up
Workspace Limitations	Write Up	Derivation Programming Write Up	Write Up
Camera Calibration	Write Up	Programming Write Up	Programming Write Up
Masking Filters	Write Up	Programming Write Up	Programming Write Up
Object Centroid Detection	Programming	Programming Write Up	Programming Write Up
Centroid Correction	Derivation Programming Write Up	Write up	Write Up
Pixel to Checkerboard Conversion	Programming	Write Up	Derivation Programming Write Up
Checkerboard to Robot Conversion	Derivation	Write Up	Derivation Programming Write Up
Final Sequence	Programming Write Up	Programming Write Up	Programming Write Up
Final Composition	Write Up	Formatting Write Up	Write Up
Final Video	Filming	Link Creation	Script Production