RBE 3001 – Team 19 – Manipulator Demonstration Report

Theodore Winter, Leona Nhi Nguyen, and Owen Krause, *WPI RBE F22*

Abstract— **This final project demo will inherit all the implementation from the previous labs and incorporate computer vision to detect, sort, and do actions on real-world objects. This final project will include this report, a video for the sign-offs proof and code will be published on the Team's GitHub directory.**

I. Introduction with Background and Motivation

This final project is the result of all the team's implementations from the previous labs including the computer vision feature to enable the robotics arm to grab, detect and sort the objects on the checkerboard. The team's mission for the final project was to combine all the accomplishments from the previous labs including forward kinematics, inverse kinematics, and smooth trajectories including velocities and polynomials. Materials used are taken from the course RBE3001 of the WPI RBE department.

The programming language for this lab is the high-level language MATLAB, with the use of GitHub in the Linux Ubuntu 20.04 environment. The robot arm is provided by the lab. All code was written by the team.

II. Method

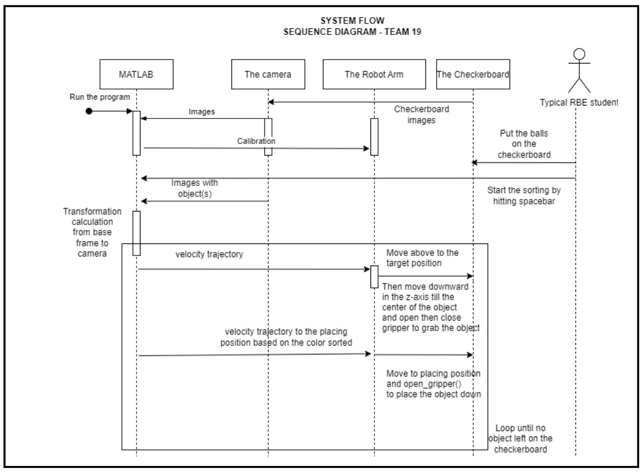

SYSTEM FLOW
SEQUENCE DIAGRAM - TEAM 19
MATLAB
The camera
The Robot Arm
The Checkerboard
Typical RBE student
Run the program
Images
Checkerboard images
Calibration
Put the balls on the checkerboard
Images with object(s)
Start the sorting by hitting spacebar
Transformation calculation from base frame to camera
velocity trajectory
Move above to the target position
Then move downward in the z-axis till the center of the object and open then close gripper to grab the object
velocity trajectory to the placing position based on the color sorted
Move to placing position and open_gripper() to place the object down
Loop until no object left on the checkerboard

Figure 1. System flow – sequence diagram

A. *Derivation of the robot's forward and inverse position Kinematics*

For the forward kinematics implementation, a DH convention table was generated to calculate the transformation matrices (*See Table 1 – DH convention Table*):

Frame transition	**ϴ**	***d***	***a***	***α***
T_0^1	0	L_0 = 55mm	0	0
T_1^2	θ_1	L_1 = 40mm	0	-90°
T_2^3	$\theta_2 - 90°$	0	L_2 = 100mm	0
T_3^4	$\theta_3 + 90°$	0	L_3 = 100mm	0

TABLE I. DH CONVENTION TABLE

To apply the calculation into the code base, the forward kinematics function is written in the *'Robot.m'* file in an object-orientated format:

- dh2mat(): takes in a 1x4 array corresponding to a row of the DH parameter table for a given link. It then generates the associated intermediate transformation and returns a corresponding symbolic 4x4 homogeneous transformation matrix.
- dh2fk(): takes in an nx4 array corresponding to the n rows of the full DH parameter table then generates a corresponding symbolic 4x4 homogeneous transformation matrix for the composite transformation.
- fk3001(): takes n joint configurations as inputs in the form of a nx1 then returns a 4x4 homogeneous transformation matrix representing the position and orientation of the tip frame with respect to the base frame.

For the inverse kinematics, the team used geometric approach – using ad-hoc geometric inspection of the kinematics chain to derive inverse relations (*See Figure 1 and 2 – inverse kinematics analysis in geometry approach*).

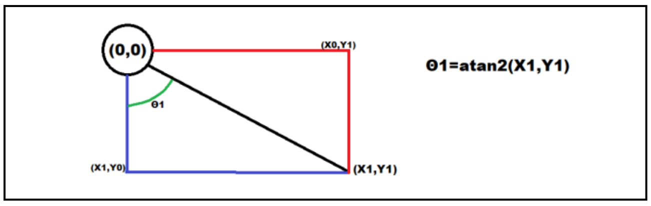


Figure 2. Two links system with theta 1

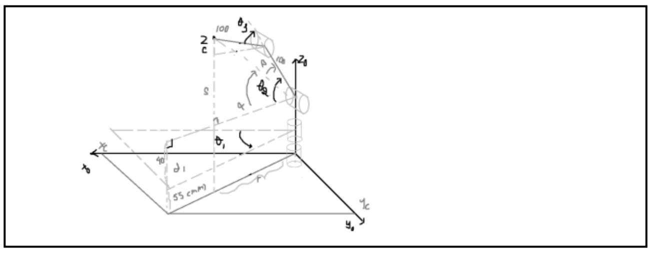
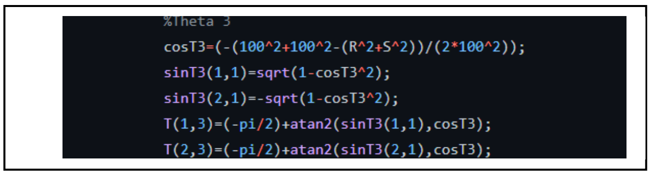


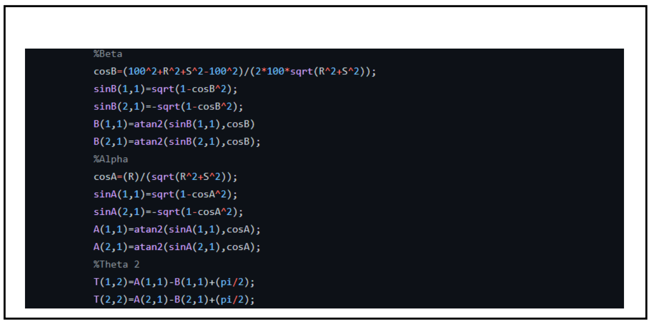
Figure 3. Robot arm system analysis in geometry appoarch

As inverse kinematics problems have multiple solutions, the team used atan2() function in MATLAB instead of atan() to collect all the possible values for the angle calculations. Using the robot arm configuration analysis, the team produced the calculations with the same method.



```
%Theta 3
cosT3=(-(100^2+100^2-(R^2+S^2))/(2*100^2));
sinT3(1,1)=sqrt(1-cosT3^2);
sinT3(2,1)=-sqrt(1-cosT3^2);
T(1,3)=(-pi/2)+atan2(sinT3(1,1),cosT3);
T(2,3)=(-pi/2)+atan2(sinT3(2,1),cosT3);
```

Figure 4. Theta 3 calculations



```
%Beta
cosB=(100^2+R^2+S^2-100^2)/(2*100*sqrt(R^2+S^2));
sinB(1,1)=sqrt(1-cosB^2);
sinB(2,1)=-sqrt(1-cosB^2);
B(1,1)=atan2(sinB(1,1),cosB)
B(2,1)=atan2(sinB(2,1),cosB);
%Alpha
cosA=(R)/(sqrt(R^2+S^2));
sinA(1,1)=sqrt(1-cosA^2);
sinA(2,1)=-sqrt(1-cosA^2);
A(1,1)=atan2(sinA(1,1),cosA);
A(2,1)=atan2(sinA(2,1),cosA);
%Theta 2
T(1,2)=A(1,1)-B(1,1)+(pi/2);
T(2,2)=A(2,1)-B(2,1)+(pi/2);
```

Figure 5. Theta 3 calculations

From all the observation and geometry approaches calculation above, the inverse kinematics function was implemented:

- Ik3001(): function takes a 1x3 matrix input as the position of the task space vector (robot's end-effector position) and returns a set of corresponding joint angles (q1, q2, q3) to make the robot move to that position.

All the inverse kinematics calculations are plugged into the code. As the team does not want to miss any valid solutions, multiple possibilities angles are included leading to a result of two possible theta1, four possible theta2 (since there are two alpha and two betas), and two possible theta3. However, not all combinations are valid inverse kinematic soluitions. To vet the outputs, they are passed into a checker function. The results are first put through an if-statement to check whether they are valid within the robot arm's joint limits. Afterwards solutions are validaded with forward kinematics before the joint angles are returned as a valid inverse kinematic solution

B. Derivation of the robot's forward and inverse velocity kinematics

To start the implementation of velocity kinematics the first step was to calculate the general Jacobian for the RRR manipulator used in this course. To do this, the team employed MATLAB's symbolic toolbox to easily derive the correct matrix. The symbolic Jacobian function takes in the DH table for the robot and using the time derivative method, solves a generalized Jacobian for the RBE3001 manipulator. To find the Jacobian for a given robot configuration, it is possible to simply substitute the joint angles into its generalized form.

Although the symbolic toolbox is excellent for initially deriving the Jacobian, symbolic MATLAB calculations are much slower than those using floating point arithmetic. The difference in speed is drastic, symbolic calculations are several thousand times slower than floating point ones. To preserve the advantages of calculating the Jacobian with the symbolic toolbox while also maintaining the speed of floating-point calculations, the team employed a MATLAB Function handle to convert the symbolically derived Jacobian into a floating-point equivalent. Upon starting the program, the Jacobian is evaluated symbolically once before being converted into a function. The function can be run much faster than the symbolic equivalent, but still allows the team to easily substitute in joint angles to find the Jacobian for a specific configuration.

Once the team had created a way to efficiently calculate the Jacobian for a given robot position, they proceeded to implement velocity kinematics for the RBE3001 manipulator. The robot's system for accomplishing this is quite simple. By multiplying the Jacobian by the current joint velocities, it is possible to derive the velocity of the end effector in the task space.

$$\dot{p} = J_p \dot{q}$$

Velocity-based motion control and inverse kinematics were also developed for the RBE3001 arm. For velocity-based motion control, the robot calculates the unit vector between its current position and its desired one. It then multiplies the unit vector by the desired velocity magnitude it wishes to move at during its trajectory, resulting in a velocity vector along the robot's desired motion path. Unlike before, end effector velocity is given, and the goal of the program was to find the joint space velocities. This is known as inverse velocity kinematics. To solve this, the robot takes the inverse of the Jacobian and multiplies it by the task space velocity vector. This operation gives the velocity each joint should rotate at to move the end effector closer to its goal. However, the RBE3001 arm's framework does not support controlling the motors based on velocity. To circumvent this issue, the team's velocity trajectory function is capable of dynamically measuring the time between each iteration of the loop and rapidly passing position commands that result in an equivalent joint velocity.

C. Image processing progress

The image processing pipeline is the robot's process used to detect and classify objects in the workspace. To start using computer vision in MATLAB, the team started by installing MATLAB's image processing toolbox. This addon provided the necessary framework to create an accurate computer vision system. The team started by calibrating the intrinsic parameters of the camera. The goal of calibration is to obtain a cameraIntrinsics object, that defines the internal parameters of the included webcam. To perform calibration, the team took 60 pictures of the workspace at differing angles and elevations. Using the Camera Calibrator App found in the Image Processing Toolbox, the team was able to quickly generate a script that automatically estimates the intrinsic parameters given the known square size of the checkerboard.

The next step is to convert the pixel coordinates of the image onto the physical workspace in the robot's frame of reference. The robot uses two transformation matrices to convert from image space coordinates to task space coordinates. The first transformation from the image frame to checkerboard $T_{Image}^{Checker}$ is calculated every time when the function getCameraPose() is used in the Camera.m class.

$$T_0^{Image} = T_0^{Checker} * T_{Checker}^{Image}$$

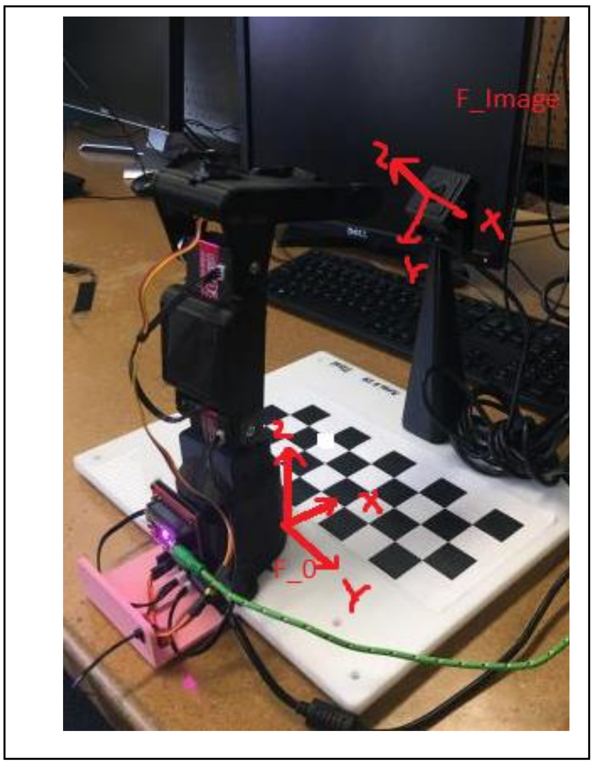


Figure 6. Frame coordinates

The 2nd transformation $T_{Checker}^{Robot}$ was calculated manually from the origin of the checkerboards coordinate system to the robot. To transform between image and task space coordinates, the robot first uses MATLAB's pointsToWorld() function to convert from image coordinates to checkerboard coordinates. This result is then translated by the matrix $T_{Checker}^{Robot}$ to convert to the robot's task space coordinates.

As the conversion from the camera's reference frame to the robot's reference frame is complete, the next procedure is to do object detection and classification. The first step in the image processing pipeline is the workspace mask. When the program is first initialized, it takes the known boundaries of the workspace and converts them to image coordinates. The program then uses these image coordinates to mask everything outside the workspace automatically. This is done to prevent false positives from foreign objects outside the workspace. Because the mask is dynamically created each calibration cycle, it masks the same space in world coordinates every time even if the camera is moved. Once all pixels outside the workspace are removed, the masked image is converted to the HSV color space. The robot then creates another mask to remove pixels that have either a very high or very low saturation value. This removes the checkerboard as the black and white checkerboard squares have saturation values close to either one or zero. Since the balls are the only objects present on the board that are not masked by this process, they are the only objects on the board after this operation. The program then uses MATLAB's imfindcircles() function to locate the centroid of all the balls present in the workspace. To determine the color, the team's algorithm samples the average hue of the pixels around the centroid of each identified object. The data regarding the color of each object is stored in an array, alongside the X and Y coordinates of the centroid in image coordinates and the radius of each detected object.

Figure 7. Workflow of the image processing pipeline chosen by the team

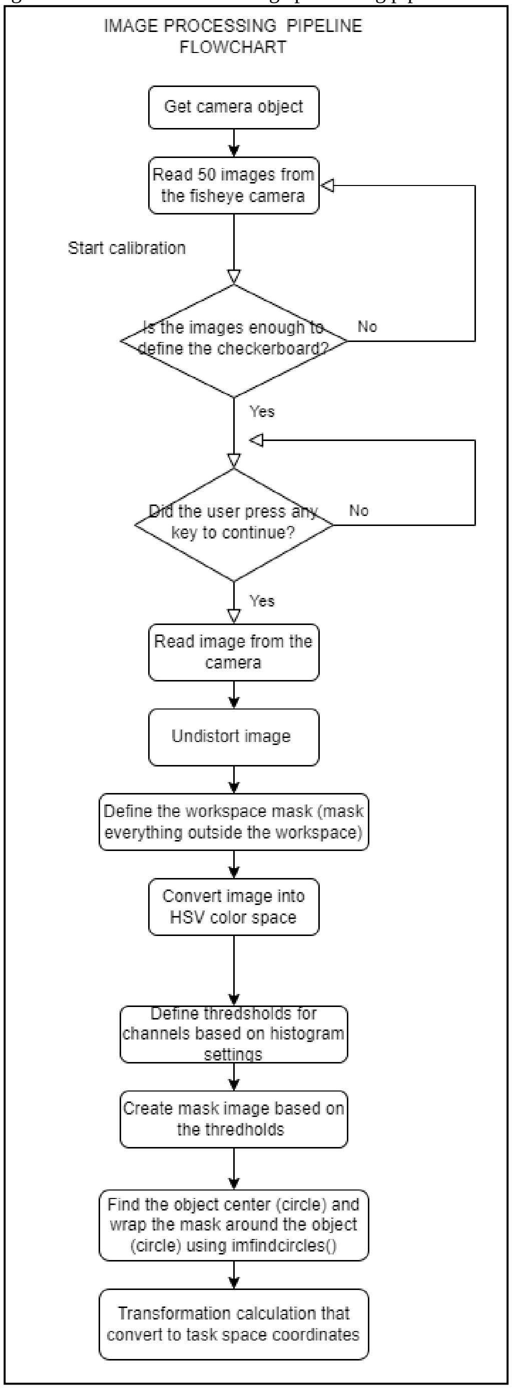


IMAGE PROCESSING PIPELINE FLOWCHART
Get camera object
Read 50 images from the fisheye camera
Start calibration
Is the images enough to define the checkerboard?
No
Yes
Did the user press any key to continue?
No
Yes
Read image from the camera
Undistort image
Define the workspace mask (mask everything outside the workspace)
Convert image into HSV color space
Define thredsholds for channels based on histogram settings
Create mask image based on the thredholds
Find the object center (circle) and wrap the mask around the object (circle) using imfindcircles()
Transformation calculation that convert to task space coordinates

Figure 8. Diagram showing approach for object detection

Following up, the team developed the object localization method by converting the centroid location of the balls into usable target positions for the robot to pick up. Since the balls are not flat circles lying on the grid, the team projected the ball's radius from the point that was obtained from the pointsToWorld() function and converted it into the real position of the ball's center in order to grab the ball in exact measurement.

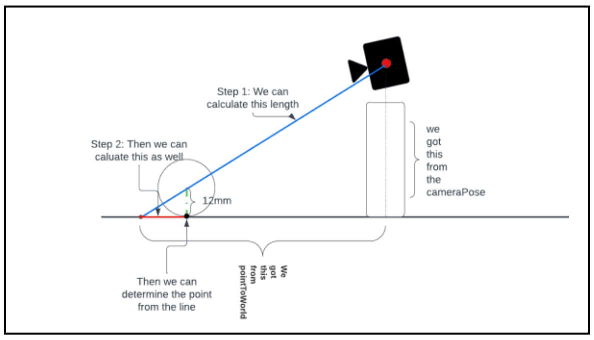
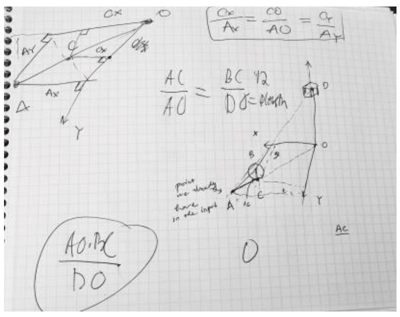

Step 1: We can calculate this length
Step 2: Then we can caluate this as well
12mm
we got this from the cameraPose
Then we can determine the point from the line
We got this from pointToWorld

Figure 9. Object localization appoarching idea



To find the centroid of the ball in 3D space, the team uses the principles of triangle similarity to find the coordinates of the centroid. The larger triangle's sides are defined by the raw coordinates returned by pointsToWorld() and the height of the camera post. This height is gotten from the inverse of the transformation matrix between the checkerboard and camera frame retrieved from the getCameraPose() function. Since the radius of the balls is known to be 12mm, it is possible to apply the principles of triangle similarity to find the XY coordinates of the ball's actual centroid.

The team also implemented dynamic tracking of objects within the workspace. To accomplish this, the robot first runs a streamlined version of the image processing pipeline outlined earlier. The version used for dynamic tracking has been slimmed down to increase its processing speed and does not track the color of the object. Once the object has been located, its position in the workspace is sent as the waypoint for an inverse velocity kinematic trajectory, which commands the arm to follow the location of the object.

Finally, the team used the Unified Robotic Descriptor File (URDF) XML format provided by ROS to create a 3D live model of the RBE3001 Arm. A basic URDF file consists of two basic components: links, and joints, which form a tree of rigid bodies. Links represent the fixed portions of the arm, while joints are the flexible connections between links. Properly defined these components form a basic stick model of the arm in 3D space. Next, the team used the STL files provided by the lab and changed the visual geometry of each link to the appropriate 3D STL file. To display the arm, the team used MATLAB's robotic systems toolbox, which has integrated support for URDF files. The team imported the URDF file into MATLAB, and while the robot is running, the virtual configuration of the robot is set to be equal to its real one, resulting in a 3D live model of the robot arm.

III. Result

To validate the camera performance, the team took a picture of the task space. The team then used ginput() to specify a position on the checkerboard in image coordinates. The team then compared the values returned by the pointsToWorld() function with the known values for the checkerboard. The team found that the coordinates returned by pointsToWorld() were within 1.5mm of the actual workspace coordinates which was satisfactory performance for this application.

The team begin to perform the procedure written in the mentioned methodology section to do the object detection. The results turn out to be just what the team expected after adjusting the threshold values and testing with the RGB values. There are also overlaying circles and titles drawn around the objects to make the detection clearer. Titles are also given in the image to specify the color sorting objects, including yellow, green, red, and orange. To not sort any other objects than just the ball on the checkerboard, the team also masked the workspace so that no unexpected result is given.

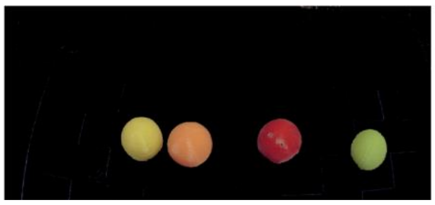


Figure 10. Object detection (raw with mask)

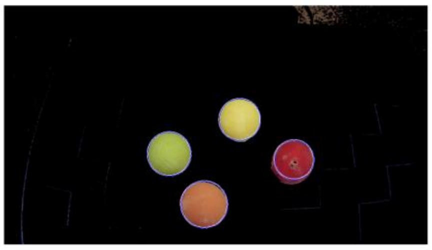


Figure 11. Object detection with circle outline

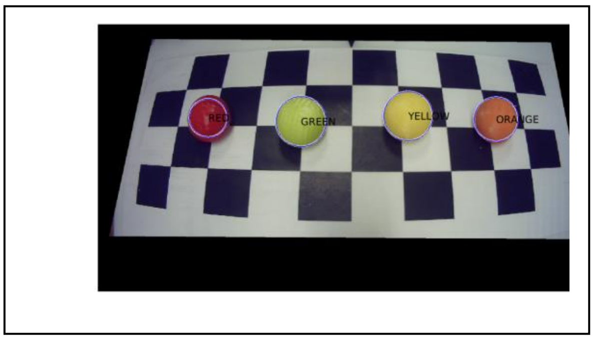


Figure 12. Complete object detection with workspace mask

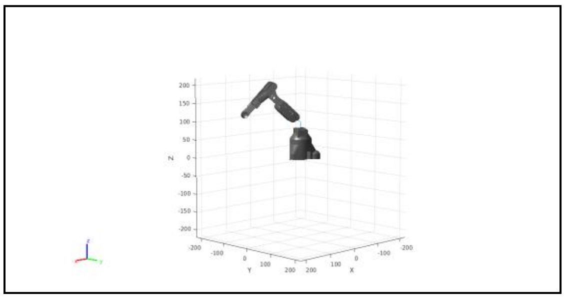


Figure 13. 3D robot arm made from stl files

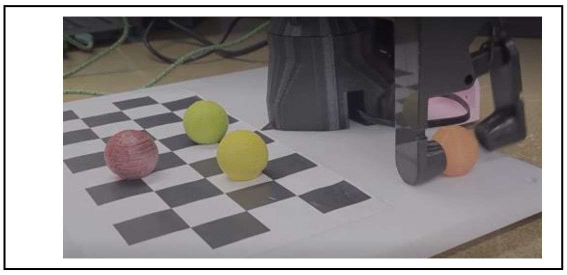


Figure 14. Robot arm grab and place object on checkerboard

Figure 15. Robot arm grab and place marker cap on checkerboard

IV. Discussion

The performance of the robot arm while sorting, grabbing, and placing the colorful objects on the checkerboard was successful, however, the team had to make some modifications to improve performance. For example, as seen in figure 10, although the robot does recognize the objects, it also recognized some material outside of the checkerboard (top right), which was light reflecting off the table. To rectify this, the team developed a function that masked the workspace so the robot could only detect the objects on the checkerboard (figure 11).

While grabbing the object, the gripper cannot grab the object exactly if moving to the target position immediately, so the team must move the gripper above the target position then move lower afterwards. This procedure increases the accuracy of the gripper grabbing the plastic ball on the checkerboard. Another issue that influences the performance of the robot arm is the lighting, if the robot is placed in a bad lighting condition, it will likely sort the color wrong for the objects like mistaking orange with red or worse, it will do the calibration wrong and cannot accurately detect the object center point.

Lastly, for extra credit 1 while doing the dynamic tracking, the overhead from the image processing caused issues. The team had to boost the speed by eliminating features like color detection and focus only on the dynamic tracking.

V. Conclusion

In conclusion, the team has completed all the lab requirements and is able to control the robot arm with the computer vision implementation to sort, grab and place four different plastic balls on the checkerboard and is able to gain the three extra credit signoffs for the lab. The robot arm has performed all the tasks successfully with the integration of images processing using the help of MATLAB add-in feature. The understanding of computer vision in MATLAB and transformation from camera to robot frame (and inverse) are highly achieved after the lab accomplishment.

Team programming collaboration is proved via Git commits throughout the lab progress. The GitHub release can be found in the RBE300X-Lab/RBE3001-Team19repository from WPI (Worcester Polytechnic Institute) RBE 300x Lab. This lab is the combination of all the previous lab work and the course target goal.

Appendix

Video:
https://drive.google.com/file/d/1XRnqRzaKZgl0WYH29EA2_pKb5834hK3b/view?usp=sharing
Code:
https://github.com/RBE300X-Lab/RBE3001_Team19/tree/FinalProject

Task	Participants
Implementation	Theodore(main), Leona
Experimentation	Theodore, Leona
Documentation	Theodore, Leona (main)
Video	Owen

Acknowledgment

The team thanks the RBE3001 lab staff for providing the robotic arm and materials to complete the lab.

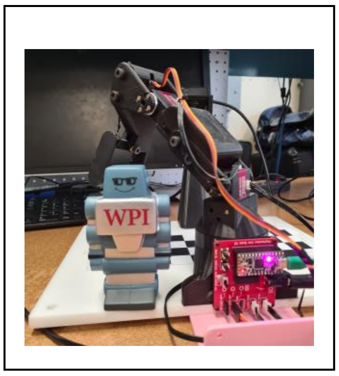


Figure 16. The team’s super awesome robot arm and its friend

REFERENCES

[1] github.com/RBE300X-Lab/RBE3001
[2] mathworks.com/help/vision/ref/worldtoimage.html