Hephaestus Arm Control – Sorting Balls

Akshay Jaitly, Cameron Earle, Ethan Chandler

***Abstract*— Utilizing the basic joint-space communication infrastructure implemented in lab one, and the position and velocity forward/inverse kinematics implemented thereafter, a pick-and-place routine was implemented via a computer vision system on the Hephaestus Arm serial linkage.**

I. INTRODUCTION

The Hephaestus Arm is a 3DOF Arm with a gripper built for manipulation. Three Hiwonder LX-224 Serial Bus Servos are used to control 3 Serial Links with a servo controlled Gripper.

This project aims to use robotic algorithms and control methods to control the arm and have it autonomously sort colored balls in it's work space. In order to do so, a checkerboard (to represent the workspace) and camera are added to the robot.

The robot's Forward Kinematics, Inverse Kinematics, and Velocity Kinematics have to be derived, as well as the camera properties, a vision pipeline, a way to control motion, a way to determine motion, and a state machine to make decisions.

II. METHOD

A. Kinematics

A transformation matrix T_n^m encodes information about the orientation and location of a frame m in reference to frame n. Frames are often assigned to objects in robotics applications, where their rotation and position relative to each other can be described by $T_{desiredframe}^{object}$.

The information of location and orientation are encoded as

$$T_n^m = \begin{bmatrix} R_n^m & t_n^m \\ 0 & 1 \end{bmatrix}$$

where R_n^m is a 3x3 special orthogonal matrix(SO(3)) representing frame m orientation and t_n^m is a vector representing the frame origin location of m in n.

1) Position Forward Kinematics: The Forward Kinematics of the robot presents the Homogeneous Transformation matrix that represents the position of the end effector as a function of joint positions. This can be given by finding each of the frames representing the links in reference to the last and multiplying them so $T_0^4 = T_0^1 T_1^2 T_2^3 T_3^4$. where T_n^m is a frame assigned to joint m on basis n.

The frames are found through the DH method (reference explaining DH method), where a transformation is represented by 4 parameters – d θ a α.

Frame T_n^m is represented as a translation along T_{nk} by d, and rotation about T_{nk} by θ – resulting in an intermediary frame $T_{n-intermediary}$ – followed by a translation along $T_{n-intermediary_i}$ by a, and rotation about $T_{n-intermediary_i}$ of α.

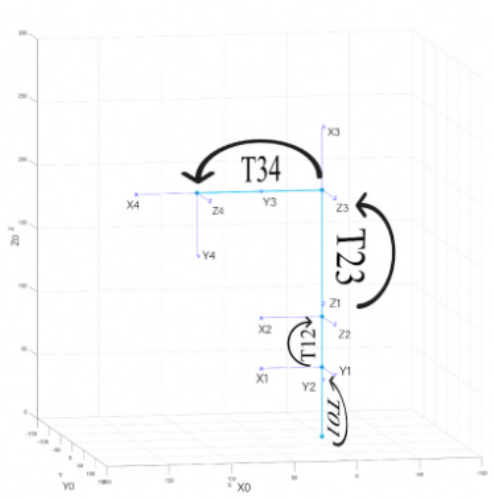

Fig. 1. the Frames representing the robot joint positions at Q = 0

Thus,

$$T_n^m = Rot_z(\theta) Trans_Z(d) Rot_x(\alpha) Trans_x(a)$$

for any frame defined by DH values.

If frames on a robot are defined so that T_Z is always along the rotation axis of a link (for a robot with rotary joints) then the end-effector position can be defined using only the θ values of the DH table as parameters.

The DH table for the Hephaestus Arm Robot is given by –

$$\begin{bmatrix} Frame & d & \theta & a & \alpha \\ DH_0^1 & 55 & 0 & 0 & 0 \\ DH_1^2 & 40 & q_1 & 0 & -\frac{\pi}{2} \\ DH_2^3 & 0 & q_2 - \frac{\pi}{2} & 100 & 0 \\ DH_3^4 & 0 & q_2 + \frac{\pi}{2} & 100 & 0 \end{bmatrix}$$

Which results in a table from which (given $\vec{Q} = [q_1, q_2, q_3]$) one can derive the frames describing each joint.

On performing $T_0^4 = T_0^1 T_1^2 T_2^3 T_3^4$, the position vector pointing to the end effector is given by

$$\begin{bmatrix} 100cos(q1)cos(q2-\frac{\pi}{2})+100cos(q1)cos(q2-\frac{\pi}{2}) \\ cos(q3+\frac{\pi}{2})-100cos(q1)sin(q2-\frac{\pi}{2})sin(q3+\frac{\pi}{2}) \\ \\ 100sin(q1)cos(q2-\frac{\pi}{2})+100sin(q1)cos(q2-\frac{\pi}{2}) \\ cos(q3+\frac{\pi}{2})-100sin(q1)sin(q2-\frac{\pi}{2})sin(q3+\frac{\pi}{2}) \\ \\ 40-100cos(q2-\frac{\pi}{2})sin(q3+\frac{\pi}{2})sin(q3+\frac{\pi}{2})- \\ 100cos(q3+\frac{\pi}{2})sin(q2-\frac{\pi}{2})-100sin(q2-\frac{\pi}{2}) \end{bmatrix}$$

This function, FK of Q, was used to solve the FK of the robot when the arm joint positions were polled and used (later) to estimate the target joint positions.

2) Velocity Kinematics: A solution to the linear velocity kinematics of the system are given by the equation which governs the relationship between joint space velocities and task space velocities. The solution lies in the form of a matrix which can transform task space velocities into joint space velocities, or the inverse. Inspiration for such a solution can be found in the Forward Kinematics Equation, which displays a linear conversion between joint-space and task-space positions. Such an equation is shown below: $\hat{p_0}(\hat{Q})$, which represents the change in position per change in joint n. Taking the partial derivative of each position with respect to the relative joint positions results in the following parametric equation:

$$\frac{\partial \hat{p}_0(\hat{Q})}{\partial Q_1}\dot{Q}_1+\frac{\partial \hat{p}_0(\hat{Q})}{\partial Q_2}\dot{Q}_2+\frac{\partial \hat{p}_0(\hat{Q})}{\partial Q_3}\dot{Q}_3$$

which can be rewritten in matrix form as:

$$\begin{bmatrix} \frac{\partial \hat{p}_0(\hat{Q})}{\partial Q_1} & \frac{\partial \hat{p}_0(\hat{Q})}{\partial Q_2} & \frac{\partial \hat{p}_0(\hat{Q})}{\partial Q_3} \end{bmatrix} \begin{bmatrix} \dot{Q}_1 \\ \dot{Q}_2 \\ \dot{Q}_3 \end{bmatrix}$$

As such, the linear velocity kinematics of the system are given by the Jacobian of the three rows in the last column of the Forward Kinematics equation with respect to the joint angles θ_1, θ_2, and θ_3, and is given by:

$$\begin{bmatrix} 100*sin(U1)*sin(U2-\frac{\pi}{2})* & -100*cos(U1)* & -100*cos(U1)*cos(U2-\frac{\pi}{2})* \\ sin(U3+\frac{\pi}{2})- & sin(U2-\frac{\pi}{2})- & sin(U3+\frac{\pi}{2})- \\ 100*cos(U2-\frac{\pi}{2})* & 100*cos(U1)*cos(U2-\frac{\pi}{2})* & 100*cos(U1)*cos(U3+\frac{\pi}{2})* \\ cos(U3+\frac{\pi}{2})*sin(U1)- & sin(U3+\frac{\pi}{2})- & sin(U2-\frac{\pi}{2}) \\ 100*cos(U2-\frac{\pi}{2})* & 100*cos(U1)*cos(U3+\frac{\pi}{2})* & \\ sin(U1) & sin(U2-\frac{\pi}{2}) & \\ \\ 100*cos(U1)* & -100*sin(U1)* & -100*cos(U2-\frac{\pi}{2})* \\ cos(U2-\frac{\pi}{2})+ & sin(U2-\frac{\pi}{2})- & sin(U1)*sin(U3+\frac{\pi}{2})- \\ 100*cos(U1)*cos(U2-\frac{\pi}{2})* & 100*cos(U2-\frac{\pi}{2})* & 100*cos(U3+\frac{\pi}{2})* \\ cos(U3+\frac{\pi}{2})- & sin(U1)*sin(U3+\frac{\pi}{2})- & sin(U1)*sin(U2-\frac{\pi}{2}) \\ 100*cos(U1)*sin(U2-\frac{\pi}{2})* & 100*cos(U3+\frac{\pi}{2})* & \\ sin(U3+\frac{\pi}{2}) & sin(U1)*sin(U2-\frac{\pi}{2}) & \\ \\ & 100*sin(U2-\frac{\pi}{2})* & 100*sin(U2-\frac{\pi}{2})* \\ & sin(U3+\frac{\pi}{2})- & sin(U3+\frac{\pi}{2})- \\ 0 & 100*cos(U2-\frac{\pi}{2})- & 100*sin(U2-\frac{\pi}{2})* \\ & 100*cos(U2-\frac{\pi}{2})* & 100*sin(U2-\frac{\pi}{2})* \\ & cos(U3+\frac{\pi}{2}) & sin(U3+\frac{\pi}{2})- \end{bmatrix}$$

3) Inverse Kinematics: Since the spatial positions cannot be controlled directly, but the actuator states can, finding a way to transfer the target spatial state to desired joint states will help control the robot. These are the Inverse Kinematics.

There are multiple approaches to solving the Inverse Kinematics of a robot arm like this one. One of which is the numerical method.

The Hephaestus Arm IK was solved geometrically, with the joints being solved from a series of triangles. On testing, it seemed faster to use a second method – Gradient Stepping.

Given a target task space position, X_{target}, $X_{target} = FK(Q_{target})$, or, $X_{target} - FK(Q_{target}) = 0$

Gradient stepping is an optimization algorithm that minimizes the error. $\epsilon = |X_{target} - FK(Q_{guess})|$ where, if the error is approx. 0, Q_{guess} is approximately Q_{target}.

Since ϵ is a vector valued function, the jacobian and inverse jacobian can be found. The gradient of the function Q_error at Q_guess is given by

$$inv(J(Q_{guess}))\frac{\epsilon}{|\epsilon|}$$

To step towards Q_error = 0, the algorithm checks new Qguesses. To get a better guess, Qis updated at each iteration as $Q_{guess} = Q_{lastguess} - inv(J(Q_{lastguess}))\frac{\epsilon}{|\epsilon|}h$ or $Q_{guess} = Q_{lastguess} - \dot{\hat{Q}}_{error}h$ where h is an arbitrary time step.

For the purposes of this project, the MATLAB optimization toolbox's fsolve function was used. The function uses $\epsilon(Q)$ and an initial guess for Q to run the algorithm. It was found that this worked faster and more accurately than the IK solver built. This could be because of rounding errors in the inverse trigonometric functions (which is used a lot) or the a significant amount of calculations. This mixed with MATLAB's intelligent step size determination algorithms makes the output through numerical methods more accurate.

A third proposed solution saw the robot stepping along the **task space** to reach it's target in a proportional control loop. Since the robot Jacobian is known, and the vector from current task space location to final (which is the heading of the desired task space velocity) is known, $\dot{\hat{Q}}(t) = inv(J(Q_t))(X_{target} - X_{current})$ and $Q_{t+1} = Q_t + \dot{\hat{Q}}(t)h$

This is very similar to the gradient stepping algorithm, and with $h = constant * norm(X_{target} - X_{current})$ is actually faster. It is also less accurate because of the stopping condition. The algorithms stops if $X_{target} - X_{current} < errorBound$, so the error is always approximately the error bound

B. Trajectory Planning

The trajectory of the robot is determined from a starting stateX_i, final stateX_f, and time to interpolate between them dt.

1) Cubic: If the trajectory is represented by a cubic polynomial, the function must satisfy the following conditions –

$$a_0 + a_1t + a_1t^2 + a_2t^3 = S(t), a_1 + 2a_2t + 3a_2t^2 = V(t)$$

$$X_{i1} = S(0), X_{i2} = V(0)$$

$$X_{f1} = S(dt), X_{f2} = V(dt)$$

These conditions allow the solving of $\vec{a} = [a_0, a_1, a_2, a_3]$ where $S(t) = \vec{a} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$

As $a_0 = X_{i1}, a_1 = X_{i2}$, and

$$[a_3, a_2]^T = \begin{bmatrix} 3dt^2 & 2dt \\ dt^3 & dt^2 \end{bmatrix}^{-1} \begin{bmatrix} X_{i2} - a1 \\ X_{i1} - a_1 - a_0 \end{bmatrix}$$

2) Quintic: If the trajectory is represented by a cubic polynomial, the function must satisfy the following conditions –

$$a_0+a_1t+a_1t^2+a_2t^3+a_3t^4+a_4t^5 = S(t), S'(t) = V(t), S''(t) = A$$

$$X_{i1} = S(0), X_{i2} = V(0), X_{i3} = A(0)$$

$$X_{f1} = S(dt), X_{f2} = V(dt), X_{f3} = A(0)$$

So, $\vec{a} =$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & dt & dt^2 & dt^3 & dt^4 & dt^5 \\ 0 & 1 & 2dt & 3dt^2 & 4dt^3 & 5dt^4 \\ 0 & 0 & 2 & 6dt & 12dt^2 & 20dt^3 \end{bmatrix}^{-1} \begin{bmatrix} \vec{X}_i \\ \vec{X}_f \end{bmatrix}$$

3) Class Setup: The MATLAB control of the Hephaestus arm relies on trajectory generation to generate smooth paths between two points.

In order to contain a trajectory a MATLAB class was generated where each instance represented a trajectory/move from the current robot point to a final robot point. The class was set up so the coefficients were calculated at initialization (which serves as t = 0) and can be polled at any time step for the current value. It was made to handle multiple DOF states.

In using this class to create robot trajectories, the trajectory built is always a quintic trajectory with initial acceleration = final acceleration = 0 in the joint space. This results in a very smooth trajectory because of the acceleration control and the faster calculation speeds.

If the trajectory was built in the task space, the IK function would need to run at every time step, which would extend the time between trajectory point polling. The robot would interpolate between less points in the same amount of time, making movement jerky.

C. Camera Intrinsic Calibration

The robotic system utilizes a 1080p wide angle webcam as the sensor to detect the presence and location of the colored balls in the work-space. Due to the extreme "fisheye" effect of the camera's lens, the camera must first be calibrated such that a camera intrisics matrix can be produced to undistort the image. To facilitate this calibration, MATLAB's built in camera calibration app was used.

To calibrate our camera, we removed it from its mount and set up the camera calibration app to take 60 images, with a new image being taken every two seconds. Between each image, we moved the camera to a new position and orientation. This technique helped to make our calibration more accurate by giving MATLAB a larger dataset to analyze. Figure 2 visualizes the camera poses each calibration image was captured at.

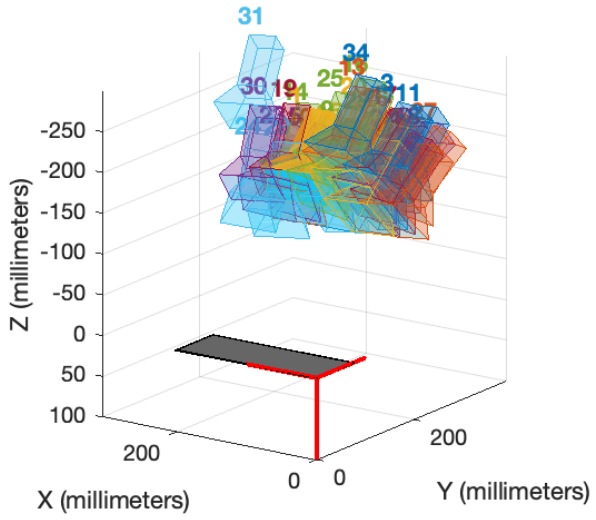

Fig. 2. Camera calibration poses relative to the workspace.

We utilized the histogram of reprojection errors to eliminate outliers (described in the results section) from the collection of images. We rejected several classes of images:

- Images where the X and Y axes projected on the checkerboard did not match what was expected
- Images where the detected points "drifted" across the checkerboard (points suddenly jumped to a different row or column, resulting in a warped coordinate system)
- Images where the checkerboard was partially obscured
- Images where not all of the expected checkerboard points were detected
- Any image with an unusually tall bar on the reprojection error histogram.

D. Extrinsic Calibration

In addition to intrinsic calbirations, the location of the camera with respect to the checkerboard coordinate frame must be identified. This is required to know the location of located balls with respect to the checkerboard. MATLAB provides the function `extrinsics` which uses the intrinsics and identified checkerboard coordinates to produce a translation vector and rotation matrix which represent the transformation from the camera's reference frame to the checkerboard's reference frame. This matrix and vector were found once and then saved to the workspace so that they

could be reused without recalibrating the camera each time we run the application.

E. Vision Pipeline

1) Color Thresholding: To locate colored balls in the workspace, we chose to divide the problem into logical steps. First, images captured by the webcam are converted to the HSV color space. We chose to operate in this color space since it was the easiest for us to use; the four balls have distinctly different hues, which makes it easy to dial in a tight hue range for each color. This means the saturation and value parameters are only responsible for handling lighting variations in the image.

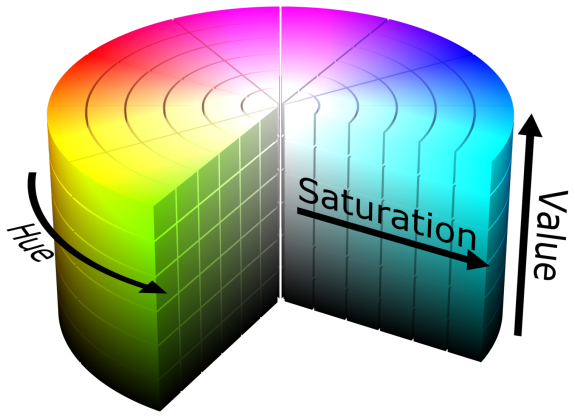

Fig. 3. Visualization of the Hue-Saturation-Value (HSV) color space

We utilized the built in color thresholder app within MATLAB to produce image masking functions. The app allows capturing of an image from the webcam, thresholding in the HSV color space, and exporting of a MATLAB function file which takes in the RGB image, converts it to HSV, and thresholds out the colors of interest, returning a binary mask image where pixels within the threshold are white (1), and pixels not within the threshold are black (0). Since we are identifying four colors, we produced four different thresholding functions. Each function filters out only the pixels of a specific color. Figure 4 shows the outputs of each thresholding function.

Fig. 4. Color thresholded images

Our color thresholding was vulnerable to a few issues:

- Orange and red appeared as similar colors in our thresholding, resulting in part of the orange ball appearing in our red thresholded image
- The yellow threshold included a lot of noise from the surrounding environment.
- Each threshold was extremely sensitive to the lighting conditions.

We solved the lighting issue by using a specific lamp for all calibration and testing. To solve the other issues, while we could have spent more time dialing in our color thresholding, we chose instead to implement a series of heuristics to filter out only the parts of the image that we care about. We implemented the `findBall` function in the `Vision` package in our codebase to implement these heuristics. Its functionality is described below.

MATLAB has an inbuilt function `regionprops` which detects contiguous regions of "1" pixels in binary images, and returns selected properties about each located region in a MATLAB table data structure. We chose to return the following properties about the regions:

- Area of the region. This is used to filter out objects that are too big or too small.
- Centroid of the region. This is used after filtering to locate the object in the image.
- Minor and major axis lengths. This is used to calculate the aspect ratio of the region, and filter out items that are not within a specified ratio.
- Circularity of the region. This is used to further filter objects that are not close to circular.
- Bounding box. This is used for debugging purposes, to draw the bounding box of the detected region on the image for visualization.

Once we had our table of regions, we started by removing rows whose centroid did not lie within a rectangle of interest. This filtering was meant to ignore elements that lied roughly outside of the checkerboard area, such as the noise present from the lab bench in the yellow thresholded image. Next, we applied filtering on area, aspect ratio, and circularity. Each of these parameters has a range of acceptable values, and regions that do not meet the criteria are removed from the table. Finally, the remaining regions are sorted by area in descending order. The region with the largest area in the sorted table is then returned. Toghether, all of these steps reliably detected the balls in the workspace while rejecting noise.

2) Ball Location Detection: With the identified camera intrinsics and extrinsics, it is now possible to locate balls in the workspace. The intrinsics matrix undistorts the image, and additionally facilitates the conversion between points in pixel coordinates to points in 3D camera coordinates, measured in millimeters. The MATLAB `pointsToWorld` function performs this conversion. Centroid coordinates, provided by our `findBall` function, are fed into `pointsToWorld`, which uses intrinsics, extrinsics, and the pixel coordinates of the ball centroid, to produce the 2D location of the

ball projected onto the checkerboard. Due to the alignment of our camera with respect to the workspace, there is a projection error between the located centroid in checkerboard coordinates and the actual 3D location of the centroid of the ball.

This measurement is only a projection of the real ball onto the checkerboard though, as the camera only determines the location in 2 dimensions (and not depth) (figure 5).

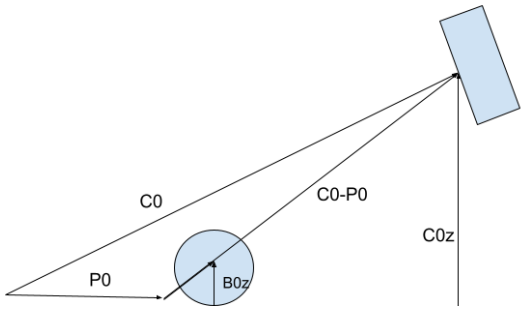

Fig. 5. The camera image projection correction

The camera only determines that the ball centroid is somewhere along the line C0-P0 where C0 is the camera location in the checkerboard frame, and P0 is the point of the centroid returned in checkerboard coordinates.

Since it is known that the centroid lays along the vector $C_0 - P_0$, and the radius of the ball is the height of the centroid, the ball centroid can be calculated as

$$B_0 = P_0 + offset = \vec{P_0} + \frac{B_{0z}}{C_{0z}}(\vec{C_0} - \vec{P_0})$$

where the offset is the vector (along C0-P0) with height equal to the ball.

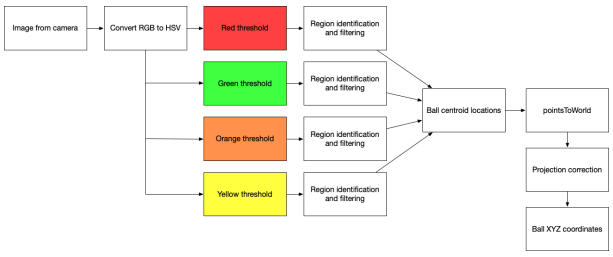

Fig. 6. Complete vision processing pipeline

F. Simulation

In order to maximize productivity, a Visualizer class was constructed, which acted as a real-time simulator for the robot. After creating the necessary STL files from scratch in Autodesk® Inventor®, a Unified Robotic Description Format (URDF) file was written so that the physical geometry of the robot was able to be viewed from within the MATLAB® environment. The primary uses of the visualizer class included the ability to plot the arm in real-time, animating its trajectory, and plotting the workspace. This class also allowed for the plotting of the checkerboard on which the robot rested, as well as the balls it sought to grab- all of which could be updated in real-time by feeding the position coordinates relative to the base frame obtained from the camera class.

G. System Architecture

It could be said that there were three main engineering challenges that had to be overcome in order to successfully complete this project. Between the kinematics of the serial linkage for physical motion and articulation, the modeling and simulation of abstract ideas such as inverse kinematics and linkage transformations, and the vision pipeline, the three structures that had the most time spent being refined served as invaluable tools to complete these tasks.

1) HParm: In order to centralize the development structure, an HParm class was constructed, which served to both instantiate and make use of exterior classes and functions like those found within the Visualizer and RobotController abstractions.

Originally, it was planned that HParm would house *all* necessary means of functionality for the robot, using MATLAB®'s varargin cell array argument list to call what was needed without bogging the user down in a list of argument types- but it was eventually decided that a more modular approach would be more beneficial to reducing development time as the project scaled.

HParm, along with the final demo file itself, served as the highest level of abstraction for our system, containing nearly everything needed to simulate and execute robotic motion along arbitrary points and trajectories.

2) Visualizer: Serving as a real-time simulator for a variety of robot attributes, the Visualizer system made itself a very useful tool for debugging. Beyond that, it served as a structure that held nearly all crucial graphical outputs of the various experiments, making it relatively simple to test functions and plot new data, both with and without the robot in play.

3) Vision: The Vision structure housed all interactions relating to the camera's vision pipeline, and allowed for the addition of the camera functionalities a relatively smooth integration with the rest of the system.

H. Designing a Demo

The demo requires the Robot to find, pick, and sort balls. In order to do this, a state machine was built using the camera and HParm class.

The initial state has the robot looking for the ball until one is found. If it is found, the robot makes sure that it is a valid ball reading by waiting a second before taking readings again (states 2 3)

If there is no ball seen in the third state, the robot returns to the initial state of looking for balls.

Otherwise, it is understood that the balls exist on the table and a target is stored. The robot sets a trajectory to position itself above the ball and opens the girpper, readying to pick.

The next state has the robot look for a stopping condition – the robot has reached it's target. If it has, the robot sets a

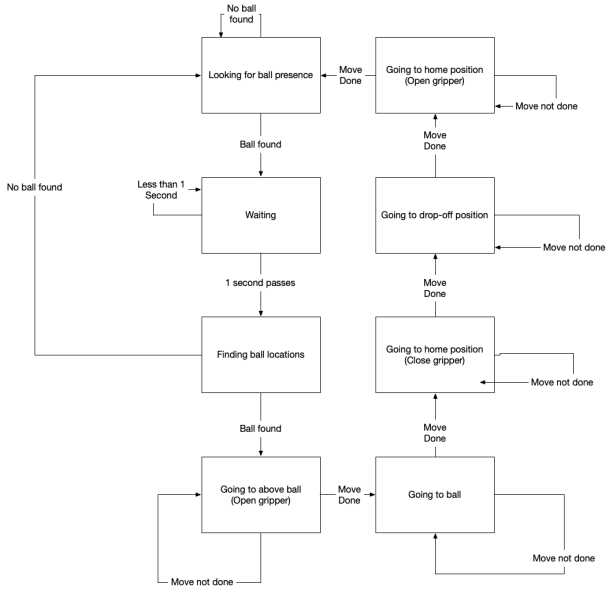

Fig. 7. the robot state diagram

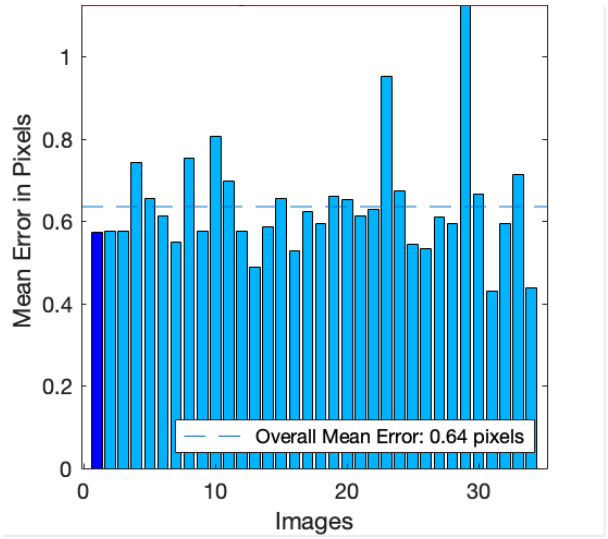

Fig. 8. Histogram of checkerboard reprojection errors for each calibration image

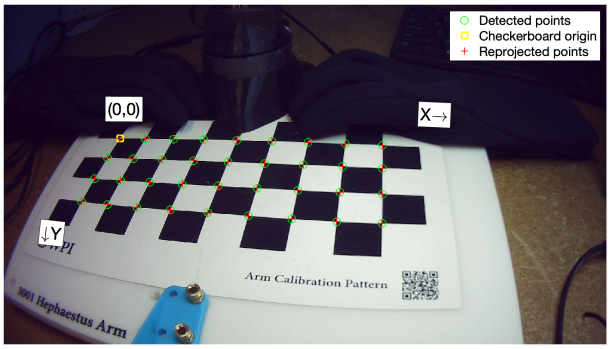

Fig. 9. Identified calibration checkerboard, showing reprojected points and checkerboard coordinate system

trajectory to surround the ball with the gripper and goes to the next state.

The next few states follow the same structure where they check to see if a move is done, and set a trajectory for the next target.

The robot goes to a ”home” position, then a drop-off position (which is dependent on the ball color) to drop it off by opening the gripper.F rom there the robot goes back to the home position of [100;0;140].

The robot now restarts the state machine, and runs ibject detection again.

III. RESULTS

A. Vision Calibration

Figure 8 shows the reprojection error for each frame. We utilized this plot to exclude certain outlier frames to minimize our reprojection error, which quantifies how many pixels of error perfect checkerboard corner points have from the detected checkerboard corners when reprojected back into image coordinates.

Figure 9 shows a well-detected checkerboard in the camera calibration app. The X and Y axes of the checkerboard match what is expected, the checkerboard is the size we expect, and the reprojection error appears low. The calibration utilizes the checkerboard pattern printed on the workspace, and the known width and squareness of each checkerboard square to correct for camera distortion. Figure 10 shows the same image undistorted using the identified camera intrinsics matrix. The undistortion compensates for the fisheye effect of the camera in the region of the checkerboard.

The characterized camera intrinsics are used to undistort an image, to compensate for the ”fisheye” effect of the camera lens. Figure 10 shows an undistorted image produced using our camera intrinsics.

The camera extrinsics translation, from the camera frame to the checker origin, was identified:

$$\begin{bmatrix} -118.5478 \\ -46.2863 \\ 359.3119 \end{bmatrix} mm$$

Additionally, the rotation matrix was identified:

$$\begin{bmatrix} 0.9999 & -0.0122 & -0.01170 \\ -0.0055 & 0.4181 & -0.9084 \\ 0.0160 & 0.9083 & 0.4180 \end{bmatrix}$$

Figure 11 shows the final result of this pipeline: four identified balls, with bounding box and centroid plotted. The color of the bounding box shows the detected color.

Figure 12 shows the results of the perspective correction and robot frame transformation: four colored balls located at the correct space in robot coordinates, shown in our custom 3D visualizer.

B. Trajectory Generation

To test the trajectory generator, we ran a sample trajectory and logged end effector position (Figure 13), joint positions

Fig. 10. Undistorted checkerboard, undistorted using the identified camera intrinsics

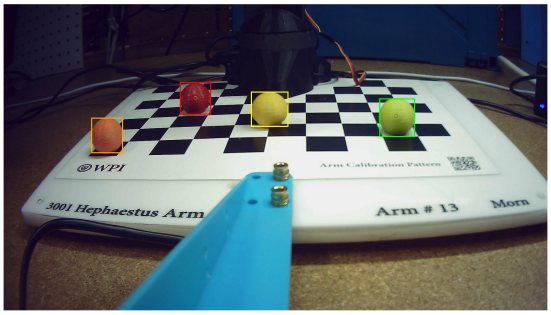

Fig. 11. Result of image processing pipeline

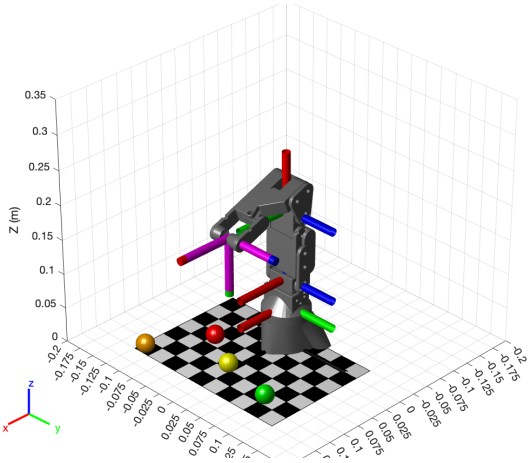

Fig. 12. Result of perspective correction and 3D transformation of ball locations

(Figure 14), and joint velocities (Figure 15). The quintic function is clearly visible in the joint positions, and we chose quintic trajectories over cubic since this gave us smoother motion.

+

IV. DISCUSSION

The robot ended up performing well, but faced a few issues.

- The servo would not release the balls on time when dropping them off on the right side of the board. We suspected this was a software issue, but it only occured on one side of the board. We determined that a loose wire was causing the servo to not receive commands until the robot moved out of a certain region.
- The camera would ignore the yellow or green balls

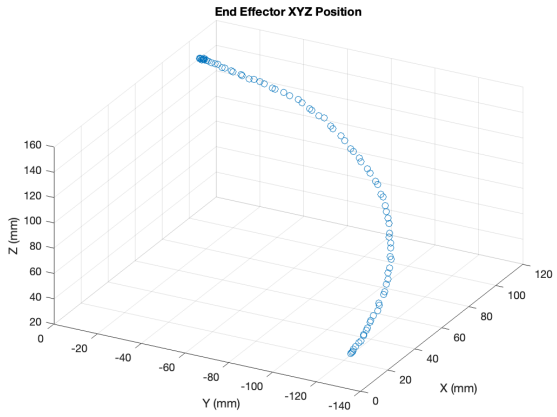

Fig. 13. Quintic trajectory end effector positions

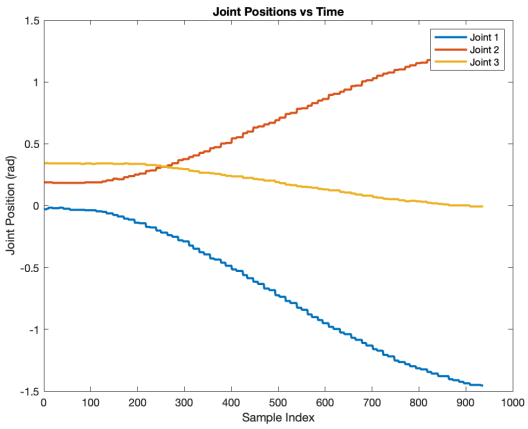

Fig. 14. Quintic trajectory joint positions vs time

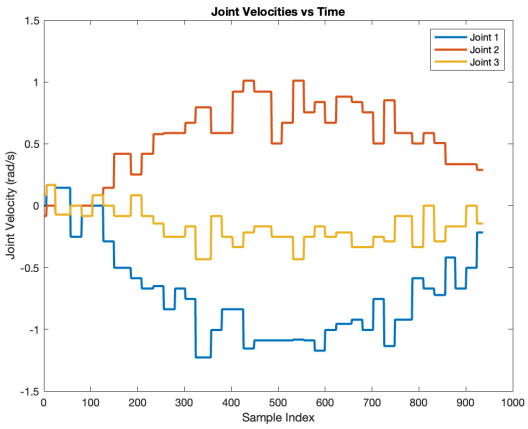

Fig. 15. Quintic trajectory joint velocities vs time

when moved to a new environment. This was due to inconsistent lighting.
- The Robot did not properly differentiate between orange and red in low lighting.
- The robot would have the same move go to different places during different runs
- The robot would not differentiate between blobs of red and orange, and neglect the "blob" for being too big if one ball was behind the other.

These issues were solved by

- Securing the servo wire, so that it would not come loose when pulled
- Re-calibrating camera thresholds when moving to new environments.
- Correcting for offsets in code whenever the robot was re-homed. The homing was inaccurate.

With the videos taken, a very high success rate of finding and placing the ball can be seen. The robot movements from any position works as expected, with a slow movement (high density in the scatter plot) at the ends of the move, and fast movement at the middle.

The Camera calibration and correction worked very well, as the plotted ball location was always correct in the visualizer. There was no visual difference between the ball location in the visualizer and in real life.

V. CONCLUSIONS

In this project, we combined forward kinematics, inverse kinematics, trajectory generation, and computer vision to deliver a complete sorting solution. We implemented supervisory logic in the MATLAB programming language, and explored data visualization and logging techniques to ensure our software was deterministic and debuggable. Our final implementation successfully detected, collected, and sorted the colored balls into known locations, and was robust to image noise and balls outside of the workspace.

Our final code is available at `https://github.com/RBE300X-Lab/RBE3001_Matlab13/releases/tag/final-release`. Our video presentation is available at `https://youtu.be/uCTJmeyEaGo`.

Our project contributions are as follows:

Planning	Cameron, Akshay, Ethan
Code	Cameron, Akshay, Ethan
Experimentation	Cameron, Akshay, Ethan
Results Analysis	Cameron, Akshay, Ethan
Write-Up	Cameron, Akshay, Ethan
Video	Cameron, Akshay, Ethan

REFERENCES

[1] Xue, Cuiping, et al. "An Advanced Broyden–Fletcher–Goldfarb–Shanno Algorithm for Prediction and Output-Related Fault Monitoring in Case of Outliers." Journal of Chemistry, Hindawi, 4 Feb. 2022, https://www.hindawi.com/journals/jchem/2022/7093835/.

[2] Hasan, Ali T., et al. "An Adaptive-Learning Algorithm to Solve the Inverse Kinematics Problem of a 6 D.O.F Serial Robot Manipulator." Advances in Engineering Software, Elsevier, 3 Feb. 2006, https://www.sciencedirect.com/science/article/pii/S0965997805001493.

[3] Robot Kinematics: Forward and Inverse Kinematics - Intechopen. https://cdn.intechopen.com/pdfs/379/InTech-Robot_kinematics_forward_and_inverse_kinematics.pdf.

[4] "A Simple and Systematic Approach to Assigning Denavit–Hartenberg Parameters." IEEE Xplore, https://ieeexplore.ieee.org/abstract/document/4252158.

[5] Robotics Toolbox for MATLAB. https://researchdata.edu.au/robotics-toolbox-matlab/www.petercorke.com/RTB/ARA95.pdf.

[6] "URDF Generator for Manipulator Robot." IEEE Xplore, https://ieeexplore.ieee.org/document/8675569.

[7] (PDF) Robot Dynamics with URDF amp; Casadi - Researchgate. https://www.researchgate.net/publication/338571184_Robot_Dynamics_with_URDF_Cas

[8] SORNet: Spatial Object-Centric Representations ... - Arxiv.org. https://arxiv.org/pdf/2109.03891v1.pdf.