RDKDC FINAL Report

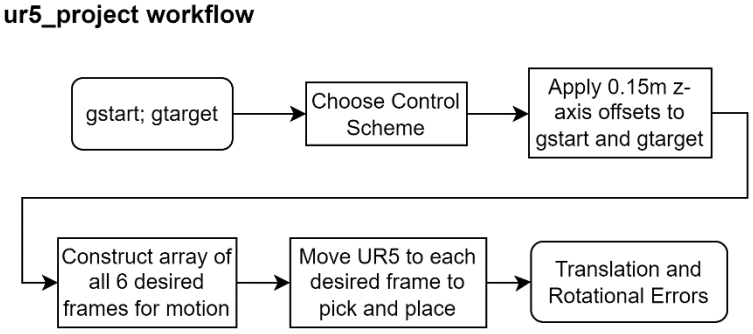
Team 3

12/20/2022

Team Members	Contributions
Chiadika Vincent	Resolved Rate Testing and Optimization, UR5 test cases setup
KB Ko	Main Script Function/Interfaces, Frame Offsets, Resolved Rate Testing
Nyeli Kratz	Resolved Rate Algorithm and Testing
Sam Ydenberg	Inverse Kinematics Algorithm, and EC Drawing Algorithm
Sohan Gadiraju	Inverse Kin & Transpose Jacobian Testing and Optimization

General Workflow of Control Algorithms

Our control schemes are called from the ur5_project script. In this script, the user records the start and target frames, from which the necessary offsets are applied to construct the pick and place frames. Each control scheme uses these frames as the target to complete the UR5 trajectory.



Position	Location	Frame Description
0	Home Configuration	ghome
1	Above Pickup	gstart + zoffset
2	At pickup	gstart
3	Above Pickup	gstart + zoffset
4	Above Place	gtarget + zoffset
5	At Place	gtarget
6	Above Place	gtarget + zoffset

Figure 1: A workflow summary of the ur5_project script. Note the table indicates how the 6 desired frames were setup for the pick and place task.

Although this was not implemented in the demo due to the missing gripper (in the RVIZ interface environment), our code has capability to apply a rigid frame transformation so that we can take in the gripper frame as an input and control the end-effector/tool frame position and orientation.

Inverse Kinematics Control

Algorithm:

There are 8 joint angle solutions provided by the inverse kinematics (InvKin) function. The multiple solutions arise from there being both negative and positive solutions for the 2nd, 3rd, and 5th joint angles. Thus, to select the joint angles to move the UR5, the algorithm includes three 'filters' to eliminate the joint angles considered to be "incorrect" (i.e., non-optimal). For simplicity, we define p_i to be the joint solution vectors that are the outputs of the InvKin function, while the final chosen joint solution is denoted as z. (Intermediate filtered solutions at stages 1 and 2 are denoted by x and y respectively.)

The first filter selects only solutions with a second joint angle greater than zero, described by equation 1 below.

$$p_i \in x \; if \; and \; only \; if \; \theta_{2,i} \geq 0 \qquad \text{Eq (1)}$$

This causes the robot to only choose "elbow up" configurations and avoids collisions with the table it is mounted on. The second filter sorts based on the first joint angle θ_1 in the remaining solutions, using equation 2.

$$y = \arg\min \left| \left| x_i(\theta_1) - z_{prev}(\theta_1) \right| \right| \qquad \text{Eq (2)}$$

It selects the joint angle θ_1 most similar to the previous configuration ($z_{prev}(\theta_1)$) of the robot, which decreases the need for extra movements. This criterion was especially important to use for filtering, as it causes all the other joints to move as well. The third and final filter chooses the solution most similar to the joint angles of the current robot position. This is described mathematically in equation 3.

$$z = \arg\min \left| \left| y_i - z_{prev} \right| \right| \qquad \text{Eq (3)}$$

By taking the absolute magnitude of the difference between the remaining joint angle vectors and the previous configuration, the robot can correctly select the best possible solution to move each time. The initial previous configuration is an important part of the second and third filters, so we chose it to be the home frame. This causes the robot to have another layer of protection for table avoidance, as it will choose the configuration that is most like the home frame, which tends to be in space and not colliding with the table.

We also have additional singularity checks in the algorithm. Should the Jacobian of the z joint vector indicate singularity, the algorithm searches for the next best joint vectors (not in singularity) to complete the motion.

Simulation Results:

To test the algorithm, simulations at various positions in RVIZ were done to guarantee correct behavior when working with the physical robot. The algorithm deals with the singularities and table collision cases gracefully. Figure 2 shows the resulting position of one of these RVIZ simulations.

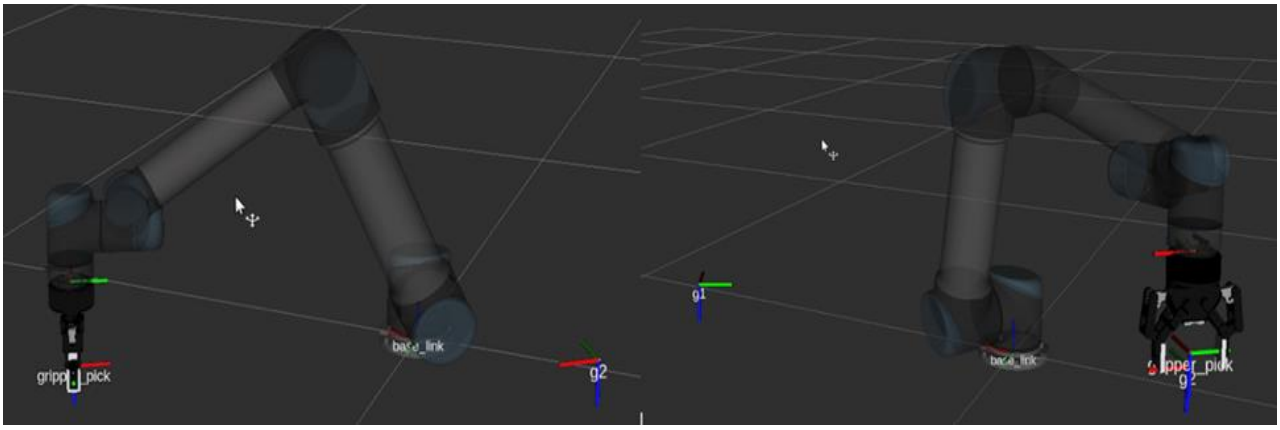


Figure 2: Inverse Kinematics Simulation Results

In this figure, there are 2 desired frames g1 and g2 for the robot to move to; these are the pick-up and place locations. The robot begins 15 cm above these locations, then moves down until it is in the correct position and orientation about the item to be picked up. With movements like this, there is error that occurs due to the calculation. These were measured using equations 4 and 5, where R_{ee} and p_{ee} are the rotation and position describing the end effector frame, and $R_{g_i}$and p_{g_i} are the rotation matrix and position vector of each desired frame in the pick and place sequence.

$$R_{error} = \sqrt{Tr\left(\left(R_{ee} - R_{g_i}\right) * \left(R_{ee} - R_{g_i}\right)^T\right)} \quad \text{Eq (4)}$$

$$perror = \left\|p_{ee} - p_{g_i}\right\| \quad \text{Eq (5)}$$

Throughout testing, these errors never increased above a marginal value of about -10^{-3}. This error is marginal and ultimately has little effect on the effectiveness of the overall inverse kinematics algorithm. For the simulation in Figure 2, the error was:

- Start Frame Error: $p_{err} = 0.01862\ cm$ and $R_{err} = 6.491\ \times 10^{-5}$
- End Frame Error: $p_{err} = 0.00185\ cm$ and $R_{err} = 8.340\ \times 10^{-5}$

These results were similar to our demo results. In general, the Inverse Kinematics Control Scheme performed the quickest out of our 3 control schemes, taking less than 2 minutes to complete the task.

Resolved Rate Control

Algorithm:

Our resolved rate control algorithm first defines our thresholds and time step and gets the initial error. We chose thresholds of $v = 0.4$ and $\omega = 12.5°$. We defined our timestep as constant $tstep = 0.1$ so that we could vary the gain to adjust the speed of the robot. We want the robot to take bigger steps when it is farther away from the goal frame so that it is fast and then take small steps when it is close to the target frame to avoid overshooting. We start with a gain $K = 1$ which we adjust dynamically based on the error.

We then calculate an initial error:

$$error = g_d^{-1} * g_{real}$$

We then have a getXi function which finds the twist of the error using the following formulas from MLS where R represents the rotation part of the error transformation and p represents the translation part of the error matrix:

$$\theta = \cos^{-1}\left(\frac{trace(R) - 1}{2}\right)$$

$$\omega = \frac{1}{2 \cdot \sin\theta}[R - R^T]^V$$

$$A = (I - R) \cdot skew(\omega) + \omega \cdot \omega^T \cdot \theta$$

$$v = inv(A) \cdot p$$

We then set up a while loop with our stopping condition being when v and ω_{diff} are both below their respective thresholds. Here, v_k represents the norm of the velocity of the error twist and w_k represents the angle between the real rotation and the target rotation calculated using the following equations where R_{ee} represents the rotation component of the end effector frame, R_d represents the rotation component of the desired frame, and R_{ed} represents the rotational transformation between the end effector and the desired frame:

$$R_{ed} = R_{ee}^T \cdot R_d$$

$$\omega_{diff} = \cos^{-1}\left(\frac{trace(R) - 1}{2}\right)$$

Inside of this while loop, we first find the current joint angles and calculate J_b, the body Jacobian of the current joint angles. We then use this information to calculate the next desired joint angles based on the equation given in assignment 3 where q2 is the next set of joint angles that we want to move the robot to:

$$q2 = q - \left[K \cdot tstep \cdot J_b^{-1} \cdot X_i\right]$$

We then check for a singularity and check to see if the robot will hit the table before moving the joints to the calculated q2. For singularity, we check whether the determinant of the body Jacobian of the q2 position is close to zero. If it is close to a singularity, we decrease the K by 33% recalculate q2, and then continue moving. This will allow the robot to move out of the singularity. To check if we are close to hitting the table, we see if the position of the ee_link frame is 15 cm above the table and that the joint angle θ_2 between $[-\pi, 0]$. If both conditions are true, then there is no configuration in which the robot could hit the table. If these conditions are not met, then we decrease the K by 50% and recalculate q2. All these conditions will once again be re-checked for the newly calculated q2, and if the conditions are once again not met, K will once again be decreased and so on. Once the conditions are met, the robot will continue moving to a new q2 position and K will be reset to its original value. This will allow the robot to move away from the table when a table hit is indicated on its next move and avoid approaching a singularity configuration.

We then move the robot to q2 and update the error and v and ω_{diff}. If v and ω_{diff} are below 0.6, indicating that we are getting close to the target position, we decrease K by 5% at each iteration within this threshold. As such, the robot slows down by taking smaller steps as it approaches its target to avoid overshooting. We repeated this until v and ω_{diff}were below their respective thresholds. After v and ω_{diff} are below their respective thresholds, we break out of the while loop and report the final error, using equations 4 and 5 defined previously.

Simulation Results:

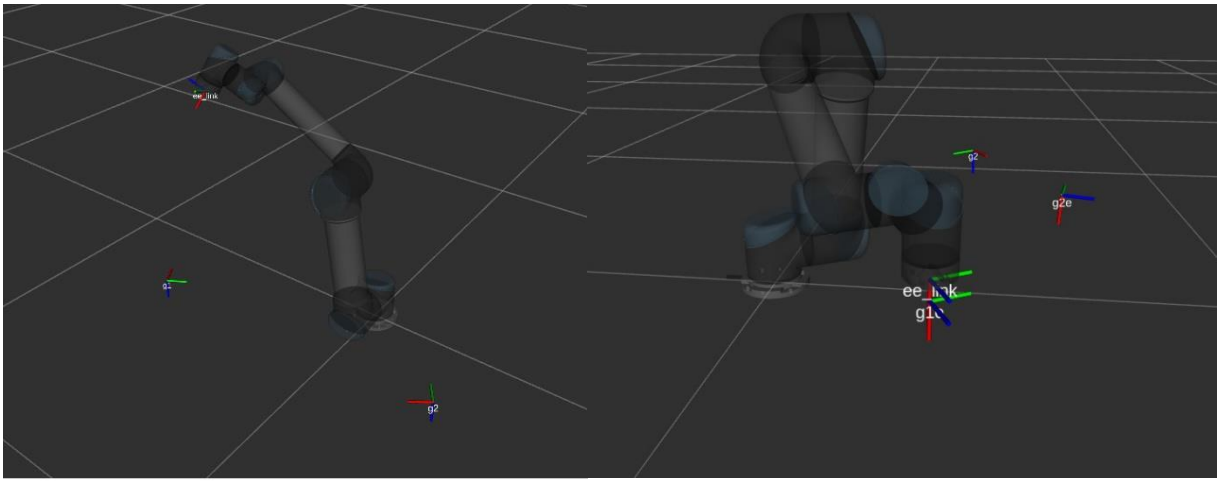


Figure 3: (Left) Resolved Rate Control testing starting position and (Right) Resolved Rate Control testing ending (pick) position.

We obtained the following error for the simulated pick-and-place task (Figure 3):

- Start Frame Error: $p_{err} = 2.98\ cm$ and $R_{err} = 0.0048$
- End Frame Error: $p_{err} = 2.86\ cm$ and $R_{err} = 5.668 \times 10^{-4}$

Our resolved rate control code took about 8 minutes to complete the full pick and place task depending on where we set the desired pick and place positions, with similar small errors. This is slower than we would have liked but, however, it was very accurate, especially for the rotational components. Note that the p_{err} stems from the vertical offset thresholding applied in the algorithm. We can increase this convergence threshold so that the ee_link frame aligns perfectly with the desired frames; however, this comes at the cost of increasing the task completion time.

Transpose-Jacobian Control

Algorithm:

Our algorithm approach for the Transpose Jacobian was very similar to the one used in Resolved Rate, however it consisted of different thresholds, different gains and time steps, and a different equation to find the next set of joint angles. The thresholds used for our demo were v = 0.5 and ω = 20° for faster convergence. Our initial gain (K) was set at 1 while the time step was always constant at 0.075. As done in Resolve Rate, we calculate initial error using $error = g_d^{-1} * g_{real}$ and then utilize the getXi() function to find the twist of the error, using the equations listed above in the Resolve Rate section.

Next, we iterate through a while loop until v and ω_{diff} are both below their respective thresholds. v_k and ω_k are continuously calculated throughout the loop to find the norm of the velocity of the error twist and the angle between the real rotation and the target rotation, respectively. The process of convergence consists of finding the current joint angles and calculating its associated

body Jacobian, J_b. We then calculate the next desired joint angles based, q2, using the transpose Jacobian equation, $q_2 = q\ - \left[K \cdot tstep \cdot J_b^T \cdot X_i\right]$. After calculating a q2, the algorithm checks if the new position will lead to a singularity or table hit. The singularity and table hit detection algorithms are identical to the Resolved Rate ones.

After the conditions are met, the robot moves to q2 and updates errors v and ω_{diff}. Once v and ω_{diff} are under 0.4, indicating we are close to the target position, we decrease the value of K by 50% with each iteration under these conditions. This is repeated until the v and ω_{diff} are below their respective thresholds and the robot has converged on the target position. Then we end the loop and report the final error, using R_{error} and p_{error} defined in Equations 4 and 5.

Simulation Results:

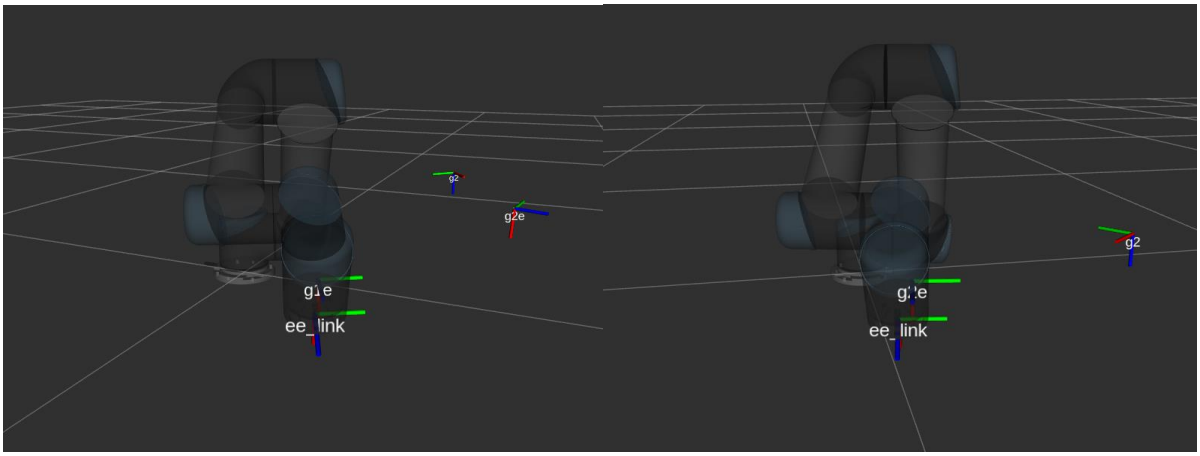


Figure 4: Transpose Jacobian Pick-And-Place Task

When running the simulations with higher thresholds of $v = 0.5$ and $\omega = 30°$, the error to reach our target positions was larger than normal. This is since the robot converges faster than usual, taking 6 minutes to get to all 6 locations. Figure 4 depicts a pick and place task moving from two test positions, g1e (start) to g2e (target) using the pick and place methodology. The errors for this task were as follows:

- Start Frame Error: $p_{err} = 5.8\ cm$ and $R_{err} = 0.235$
- End Frame Error: $p_{err} = 4.973\ cm$ and $R_{err} = 0.271$

As shown in Figure 4, the rotation of the robot is a little off from the desired position while also having a noticeable translational disparity. In our demo, the robot behaved erratically, undergoing many oscillations (overshooting and undershooting), while converging. This is attributed to a higher initial gain (K-value) of 1. Lowering the gain, or the tstep, will help reduce these oscillations; however, the cost of smoother operation is the robot converging at a slower rate to each target position. In hindsight, we should have demoed with smoother operation, however, due to the unclear specifications and the 30-minute demo time limit, we decided that speed was more important than accuracy.

Note that when simulated with lower thresholds, of $v = 0.4$ and $\omega = 15°$, the robot would take around 15 minutes to go through the full motion but resulted in much lower error. The error for these conditions is approximately 0.13 for rotational error and 2.6 cm for translational error, which is considerably lower than the ones for the higher thresholds.

Extra Credit Task - Drawing

Algorithm:

The extra credit task is creating an algorithm for drawing an imported image. The algorithm begins by importing a JPEG file and then converting that file to grayscale. Next, the image is resized so that it can fit on a piece of paper. Since the algorithm converts each pixel in the image to a mm-sized coordinate on the paper, resizing helps fit the drawing into the robot's workspace. Once this task is completed, a canny edge filter is applied to find the outlines of the drawing. Finally, the pixel row and column indices (where the filter has found an edge) are compiled into a list, which is sorted to draw more efficiently. An example of the edge filter being applied to an JPEG is shown in Figure 4 below.

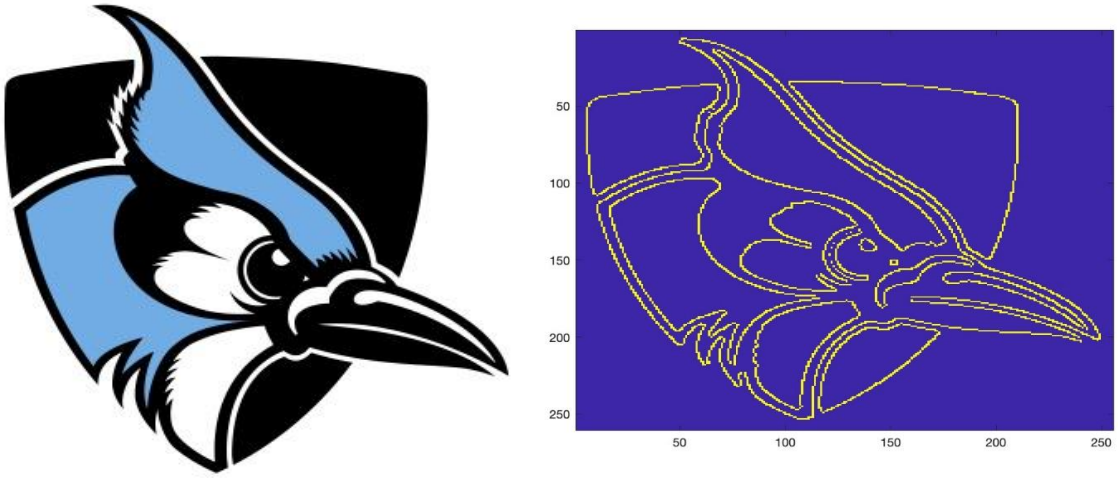


Figure 5: Hopkins Blue Jay Outline

To draw efficiently, the robot should move as little as possible. Therefore, the image points are sorted to minimize distances between points. Essentially, a point in the list is selected, and the algorithm determines the closest point indices by calculating the norm of all the differences between this initial point (pixel) index and the others. This closest point is then appended sequentially in a new list. By doing this, the robot will draw more like a human and not have to pick up the pen as often.

All points were then converted into frames with the following definition:

$$g_k = \begin{pmatrix} 1 & 0 & 0 & i_{pixel} \\ 0 & 1 & 0 & j_{pixel} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where i_{pixel} and j_{pixel} denotes the row and column indices of the pixels. Now that all the pixel coordinates were sorted to continuous lines, we added additional offset frames at the end of the continuous lines to tell the robot to pick up the pen. To do this, in the sorting algorithm, points where distances were greater than 3 pixels were recorded. At these positions, the following frames were added

$$g_k = \begin{pmatrix} 1 & 0 & 0 & i_{pixel} \\ 0 & 1 & 0 & j_{pixel} \\ 0 & 0 & 1 & 0.01 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and } g_{k+1} = \begin{pmatrix} 1 & 0 & 0 & (i+1)_{pixel} \\ 0 & 1 & 0 & (j+1)_{pixel} \\ 0 & 0 & 1 & 0.01 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The final part of the algorithm is then to take the frames of all the pixels and translate them to the robot so it can draw the image. First the robot should be positioned at the corner of the paper. This frame is then recorded and used as the base frame. Then all the pixel frames were transformed to this base frame. Finally, we used the inverse kinematics control scheme developed in the pick and place task to create the UR5 trajectory to make the drawing.

Results:

Using the algorithm described above, the robot could draw any image provided the correct resizing and edge-filter thresholds were applied. We duct-taped a pen to the gripper fingers of the UR5 and aligned the pen perpendicularly to the table. We also taped down the page corners to the table and positioned the robot to one corner. We initially drew a simple smiley face to test the algorithm and calibrate the starting positions. This took several iterations to perfect, the drawings of which are shown in Figure 6.

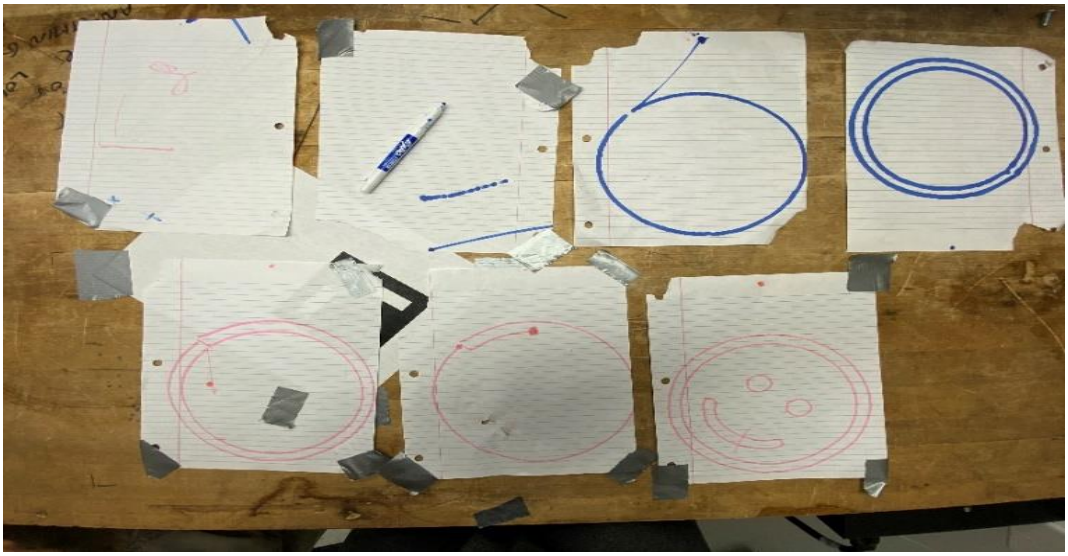


Figure 6: Drawing Algorithm Smiley Face Tests

After drawing the face correctly, we wanted to assess the robustness of our algorithm. We wanted to determine the drawing accuracies; thus, to test this, the Johns Hopkins Blue Jay logo was drawn, and this is displayed in Figure 7 below.

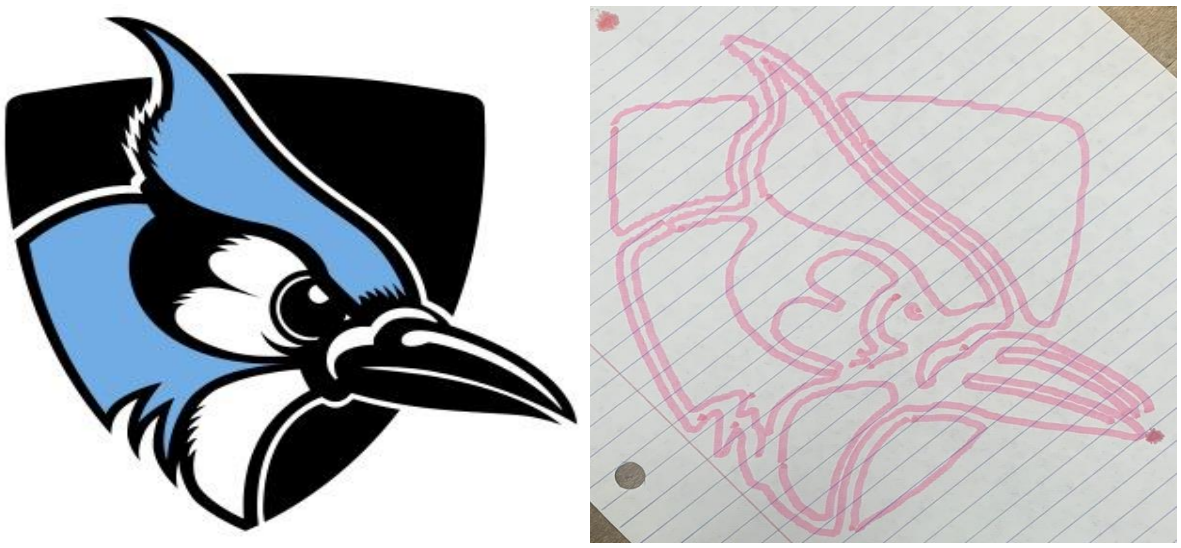


Figure 7: Johns Hopkins Logo Drawing

As the image shows, our robot can correctly draw this logo in one color. It loses some precision due to the size of the marker we used on the robot and the fact that it is the edge outline of the image, but overall, correctly creates the image desired by the user. This provided enough evidence for us to demonstrate that our algorithm was successfully in controlling the robot to draw an image.

Video Link for Extra Credit Task: BlueJay_DemoVideo.MOV