Final Project Report

Members: Zachary Frey, Tushar Singh, Tarun Prasad Senthilvel, Seyi

Introduction:

The final project for RDKDC centers on precisely controlling the UR5 robot to execute a "place-and-draw" task. Our team enabled the UR5 to identify start and target locations, subsequently automating the task with various control schemes. We implemented and showcased three distinct control trajectories for the UR5 in both simulated and real-world settings. The control schemes are:

1. Inverse Kinematics (IK)
2. Resolved-Rate Control Using Differential Kinematics (RR)
3. Jacobian Transpose

Inverse Kinematics:

Overview of Function

The Inverse Kinematics algorithm was tasked with positioning a start and end pose of the robot, and then drawing three line segments to connect them: two segments in the same direction, and one in a perpendicular direction to the other two. The function also takes a percentage value between zero and one, allowing for one of the two parallel line segments to be longer or shorter than the other, allowing for control as to how the shape is drawn. Finally, the function takes the number of requested steps to take per line, and the time required to move the robot to these steps.

Algorithm for Calculating Next Pose

The algorithm to determine the next pose first determines the transformation from the start to the end frame. This transformation is then broken apart into rotation and position components, with three position vectors determined based on the X and Y coordinates of the transformation matrix. The first vector is simply the X coordinate multiplied by the percentage term, the second vector is the Y coordinate, and the third vector is the X coordinate multiplied by one minus the percentage term. This ensures the position term is properly conserved, but will be broken up into three distinct movements. The rotation matrix component is fed into the EULERXYZINV function to extract three angles. These three angles are then divided by three

times the step variable since there are three total paths required, thus a total of three times the number of specified steps.

Three for loops are now defined, one for each line drawn. Each for loop iterates from one to the requested number of steps. Each subsequent rotation matrix is calculated by multiplying the divided angles by the iteration over the three times the steps variable. This multiplication will increase to the steps variable plus the iteration over three times the steps variable, to two times the step variable plus the iteration over 3 times the steps variable. This ensures that each subsequent rotation is preserved through the for loops. In the first for loop, the rotation of the transform is determined by multiplying the starting frame's rotation by the EULERXYZ rotation matrix of the step-divided angles with the proper step input calculation. Then, the position is determined through dividing the correct position vector by the iteration divided by the steps variable. The transform is then defined as the rotation from the step-adjusted angles, and the step-adjusted position vector.

Once this transform is defined, the transform is fed into ur5InvKin to extract the eight possible joint angle solutions of the UR5. Once these solutions are determined, they are then fed into a new function called determineangles. This function takes the ur5 interface variable and the joint solution matrix, and determines the best choice of angles. This determination is done by taking the current joint angles of the UR5, and iterating through each possible solution in a for loop. Within the for loop, each joint solution is extracted, and then a sum variable is calculated through adding up each difference between the current angle and the calculated angle of a particular joint. Once this is done, the term with the smallest sum is considered the best solution due to minimizing the total movement, and the function returns the column index of the joint solution. This is then used within the for loop to move the UR5 via ur5.move_joints function, using the joint solutions variable indexed at the correct column index, and given the timestep variable as defined by the user. Each for loop runs until completion, and by the end the robot is within a few millimeter tolerance of the end effector, and moves quickly to these positions.

Photos of Inverse Kinematics Functionality

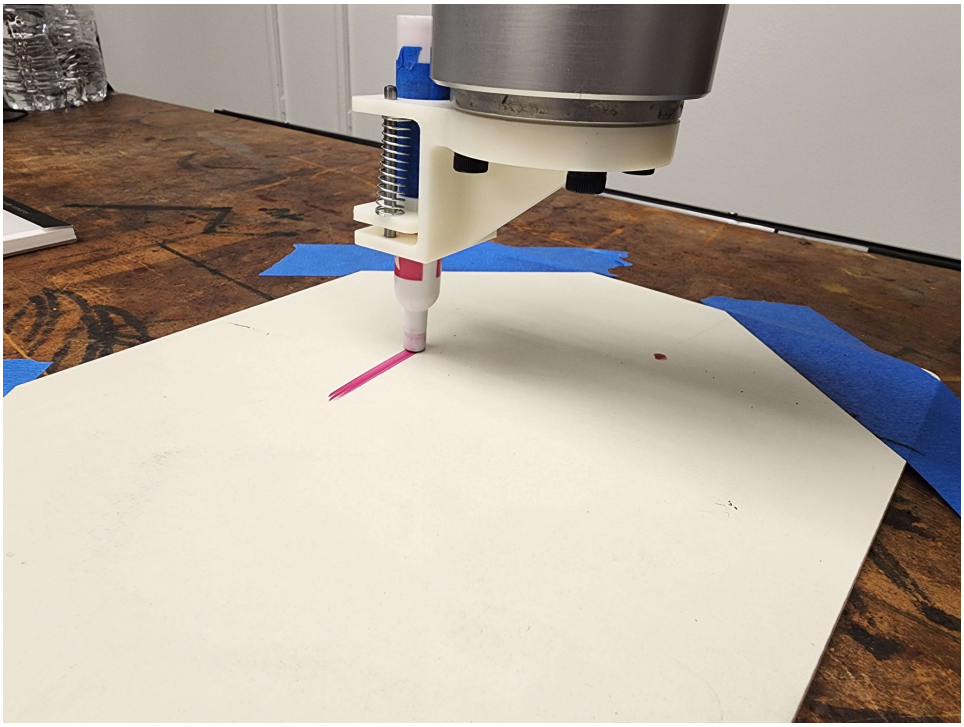


Photo 1: Robot is beginning first straight line drawing, starting where the marker streak begins. The end point is shown as a small dot.

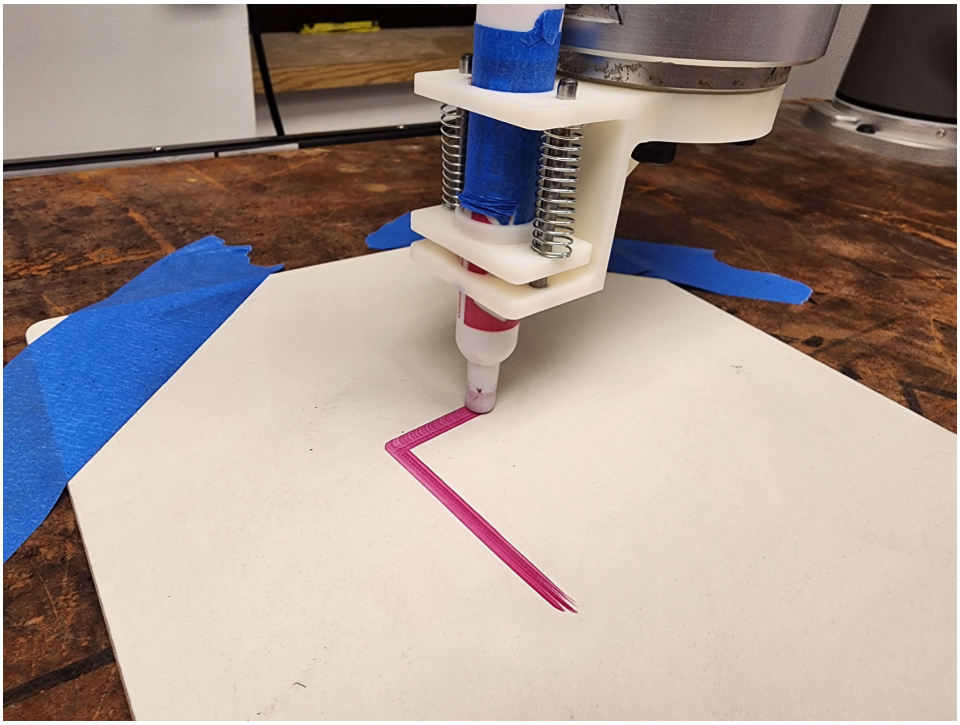


Photo 2: Robot is now beginning the second line segment, which is perpendicular to the first.

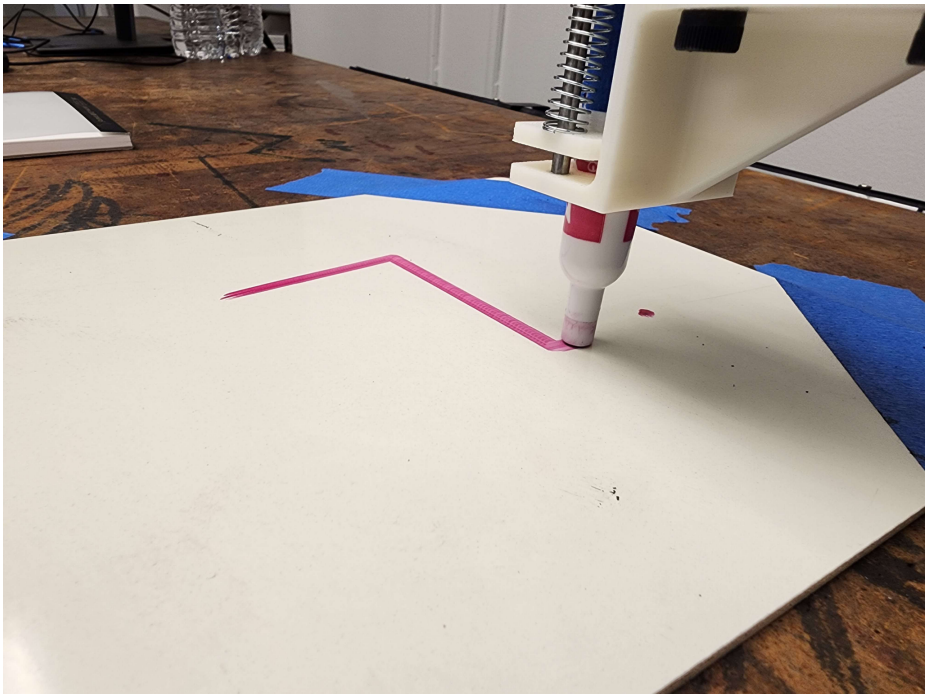


Photo 3: Robot is now beginning the third line segment drawing, and is approaching the end position.

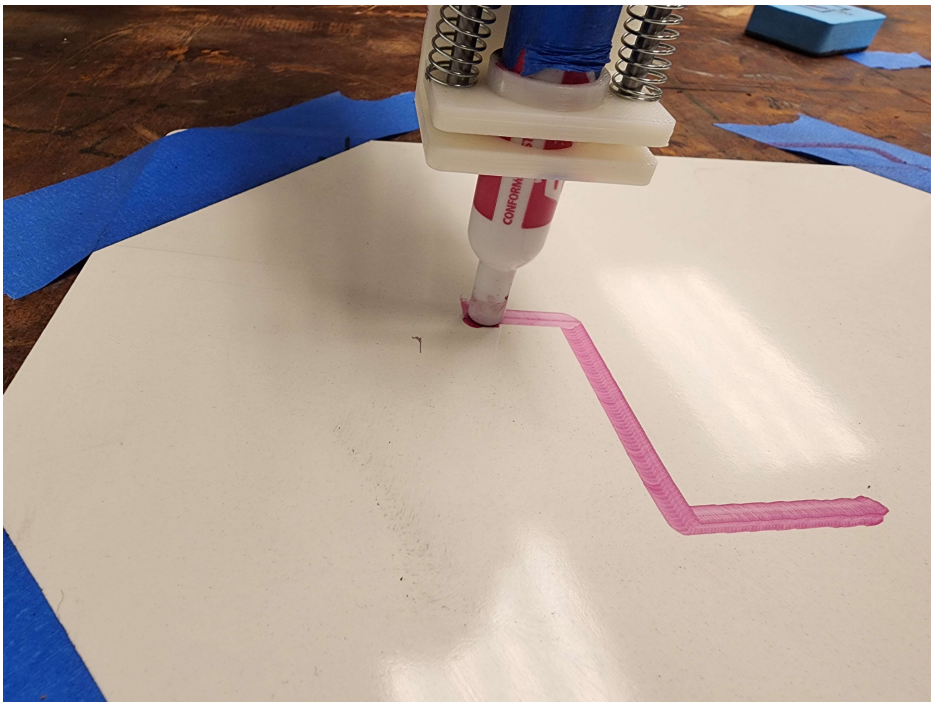


Photo 4: The robot has arrived at the end position. The offset from the expected to the actual is three to five millimeters.

Overall Error

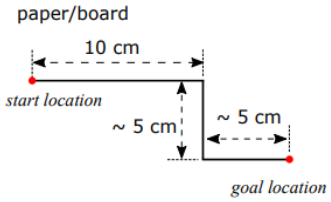
After running the inverse control several times in both simulation and on the real robot, an average error calculated from the rotation matrix (dS03) was 0.001, and the positional error dR3 average around 0.008. This seems to be strong accuracy, with the error being accounted for in floating point error over each step due to the large number of steps, or the robot's encoders being slightly off at each step.

Resolved-Rate Control:

The resolved rate control function uses the forward kinematics function, and body Jacobian function, getXi function from the Lab3 homework to implement the control of the robot's motion from a start pose to a goal pose.

The control function starts by moving the joints from the current start pose to the target pose. This is done by incrementing the joint values with small steps accounting for the error to the target pose. The error is computed as the product of the inverse of the goal pose with the current pose. The twist of this is obtained by the getXi function from Lab3 homework. This error is translated to joint value error by pre-multiplying it with the inverse of the body jacobian for the current joint values. We move the joint in small steps with a negative gain K multiplied with the joint error sequentially till our error converges to below a minimum threshold.

This function also aims to move the robot through 3 consecutive perpendicular lines from the start pose to the goal pose as shown in the below figure:



Source: FinalProject.pdf

Similar to the approach in Inverse Kinematics, we initialize the robot with a start pose and a target pose. The robot tries to move 2/3rds of the distance along the major axis first using the resolve rate control function. Then it moves the full distance along the minor axis second using the resolve rate control function. Then it completes its motion to the the target pose through the use of the resolve rate control function again.

To account for the differences in orientation between the start and end frame the ur5 robot is driven through intermediate orientation poses between the start pose orientation and the end pose

orientation. This is facilitated through the use of the EULERXYZINV function which helps in discretizing the intermediate vertice pose orientations. This helps us to reduce the error in converging to the end pose because of errors in the orientation of the robot.

Photos of Resolved-Rate Control Functionality

We were able to achieve an accurate path with a minor error of 3mm in position from the target pose. The following images are evidence of it:

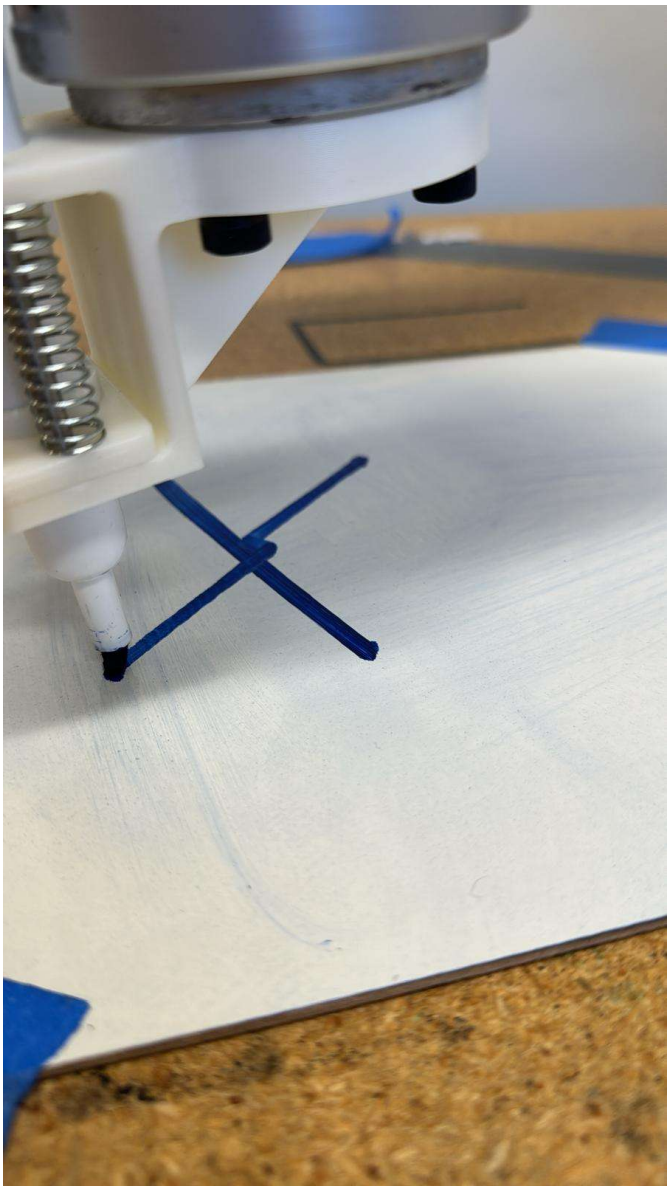


Photo 5: Photo of Resolved Rate control at the end position of motion.

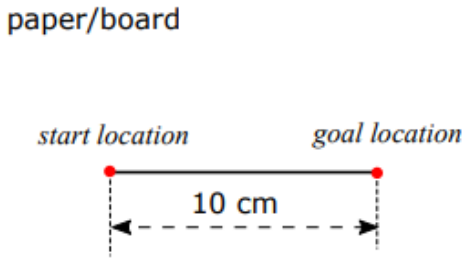
Overall Error

The positional error term for the Resolved Rate control was 0.001 on average, and the rotational error was 0.156 on average.

Jacobian Transpose

The Jacobian transpose control function uses the forward kinematics function, and the body Jacobian function, getXi function from the Lab3 homework to implement the control of the robot's motion from a start pose to a goal pose. The control function is implemented identically to that of the resolved rate control function except using the transpose of the body jacobian in place of the inverse of the body jacobian.

This function also aims to move the robot through 1 straight line from the start pose to the goal pose as shown in the below figure:



Source: FinalProject.pdf

Photos of Resolved-Rate Control Functionality

This is not an optimal control law but is computationally more efficient than the resolved rate control function as it uses a transpose operation instead of the computationally expensive inverse operation. However, as a compromise, we lose out on an optimal path relative to the performance of the resolved rate control function along with a position error of 7mm which is larger than the error from the resolved rate control. The path followed is also not the shortest path from the start pose to the target pose. This can be seen in the images below:

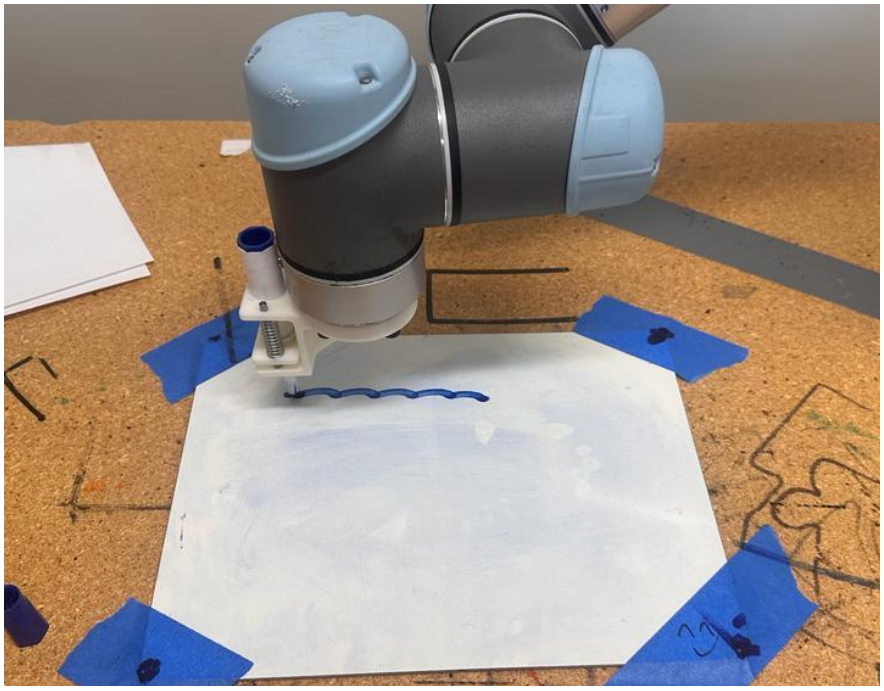


Photo 5: Jacobian Transpose Control Function drawing a straight line

Overall Error

The positional error term for the Jacobian Transpose control was 0.002 on average, and the rotational error was 0.214 on average.

Extra Task: The Batman Logo

For our extra credit, the team ambitiously chose to craft the intricate and iconic Batman logo. This endeavor involved several key steps: sourcing specific equations to define the shape of the logo, developing algorithms for discretizing these equations and eliminating duplicate points, and finally, manually hardcoding a select number of points to accurately generate the symbol. This part of the project was executed using three scripts, which encompassed a total of four functions, effectively combining mathematical precision and programming skills to bring the Batman logo to life.

Batman_equations.m

The bat_equations.m script utilizes six intricate mathematical equations sourced from a MATLAB forum. Each equation corresponds to a distinct segment of an iconic and complex symbol. The script is designed to calculate specific points that collectively render the logo. Central to this process is the ‘calculate_points’ function. This function iteratively finds the roots (solutions) of each equation within specified x-ranges. It employs MATLAB's fzero function to

locate y-values that satisfy the equation for given x-values, thereby effectively plotting the curve described by each equation. To ensure a comprehensive representation of each logo segment, the script searches for roots in both the positive and negative y-ranges.

Different x-ranges are defined for various parts of the logo. The ‘calculate_points’ function is invoked with these ranges and their corresponding equations. This methodical approach is crucial for ensuring that each segment of the Batman symbol is accurately depicted according to its unique mathematical characterization. After calculating all the points, the script consolidates them into a single array, removing any duplicates to streamline the dataset. This dataset is then plotted to exhibit all the unique points derived from the equations, offering a visual overview of the symbol's composition.

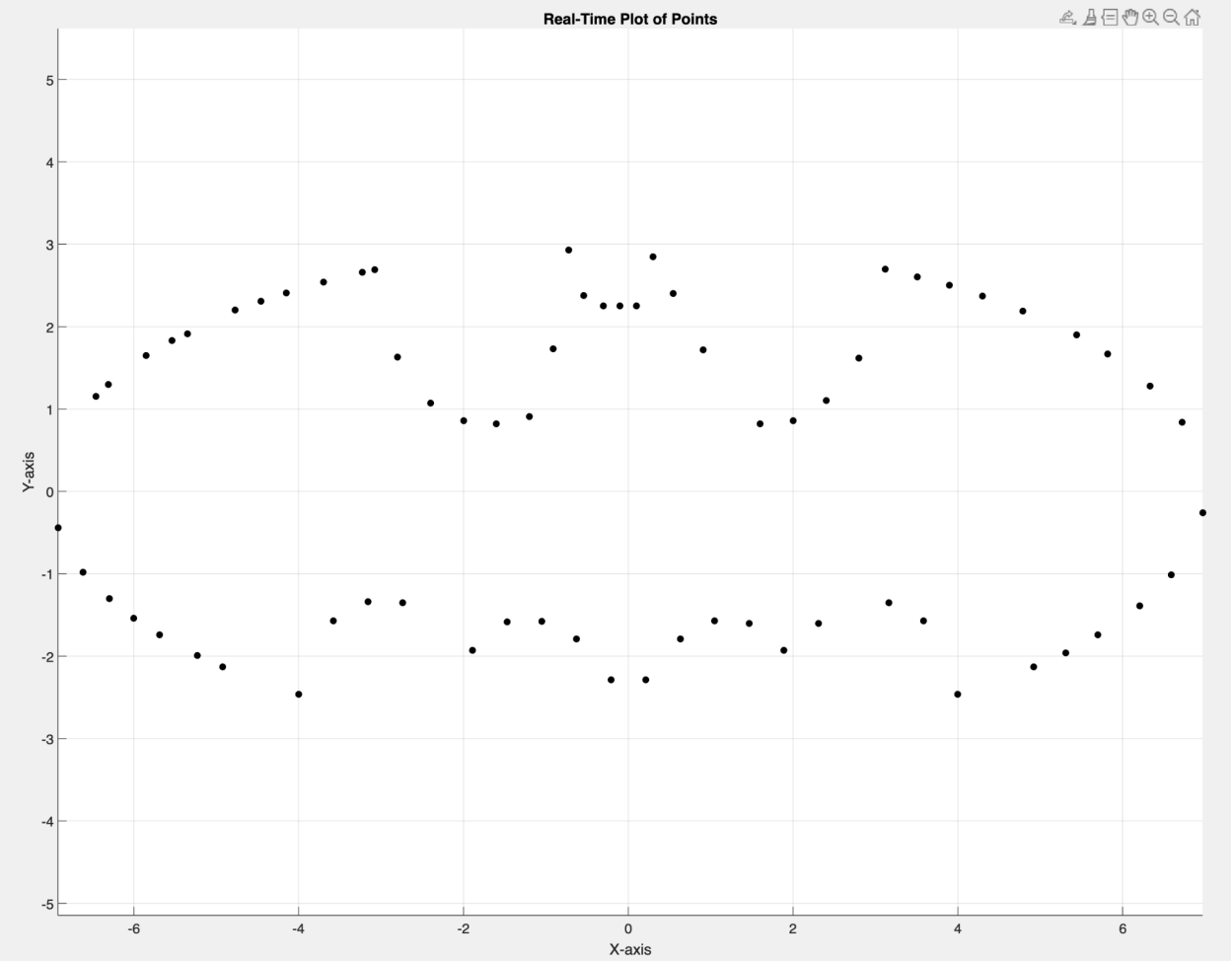
batman_points.m

The batman_points.m is designed to generate a set of coordinates that form a specific pattern of the Batman symbol. This function represents an application of discretized coordinate geometry, where path planning is achieved through hardcoding of the ‘points_array’ variable obtained from ‘bat. It starts by defining arrays: ‘x’ and ‘y’, which represents the x and y coordinates of a series of points, respectively. These coordinates are hardcoded into the function. The crucial part of the function is where it combines these x and y coordinates into a single two-dimensional array. This is achieved by first forming a 2-row matrix with x and y values and then transposing it to create a 2-column matrix where each row corresponds to a point in the Cartesian plane. This function ultimately returns this matrix, ‘final’ which holds the (x, y) coordinates. When these points are plotted, they are expected to form an outline of the batman logo.

The final array comprises 70 hard coded points. This quantity proves adequate for creating a visually appealing Batman logo, striking a balance between detail and simplicity. While introducing additional discrete points along the gradients could enhance the sharpness and definition of the logo, we found that 70 points achieved a satisfying level of detail. This choice not only served our presentation well but also ensured quick and efficient execution of the function.

batman_real_time_plot.m

Before implementing our algorithm on the UR5 robotic arm, we conducted a preliminary test using a simulation algorithm. This test was aimed at verifying whether the hardcoded points accurately traced the intended path of the Batman logo. For this purpose, we utilized the real_time_plot function in MATLAB. This function systematically plots the coordinates obtained from the batman_points function, rendering each point as a black dot and sequentially updating the visual display. This method creates a dynamic, animation-like effect that mimics the live drawing of the logo. The logo is shown below.



ur5InverseDraw Algorithm for Shape Drawing

The ur5InverseDraw function was given additional functionality to connect points that are given to it through an array of X and Y points. The algorithm only requires a start point, and begins to draw the resulting points or shape from this starting position. First, the function determines the number of points in the array, and also chooses the start point as the very first point in the array. A for loop then runs that uses the size of the array as the end term. The subsequent points are then called through the index, and subtracted from the starting point to get a difference in position. This difference is then multiplied by a correcting term which uses the inverse of the start position's rotation matrix multiplied by a rotation matrix in the Z-axis by ninety degrees. This correction is done so the shame is always drawn with respect to the end effector's orientation, and not from the base frame. Further, this correction leads to better outcomes in the created drawings, as the space the drawing would take up could be more accurately predicted due to the change in orientation. After this, a new transformation is created with the start frame's rotation matrix, and the start frame's position vector added with the point

transform variable. This point transform variable is also scaled with an extra variable so the size of the drawing can be adjusted without the points themselves needing correcting. The loop runs through every point, moving the robot at each intermittent frame. The very first frame of this loop is saved for the very end. As the robot completes the drawing, the final point still needs connecting, and thus a small section of code at the very end uses the first transformation to move the robot back to the start point.

Photos of Inverse Kinematics Functionality

We were able to effectively draw the Batman symbol on the paper on the table which can be seen from the image below:

Photo 6: Shape Draw Function drawing the Batman symbol.

Team Contribution

Team Member	Contribution
Tushar Singh	Wrote Jacobian Transpose control code Wrote Resolved Rate and Jacobian transpose sections in report Helped debug Inverse Control code Helped debug Batman equations function
Tarun Prasad	Wrote Resolved rate control code Helped debug Inverse Control and Jacobian Transpose Code Helped with debugging of the batman points discretization
Olufwaseyi Afolayan	Formatted and latex edited the final report Wrote batman equation functions Wrote batman point discretization function and real time plot. Drafted outline of report Wrote the Batman Point section of the report
Zachary Frey	Wrote Inverse Kinematics control code and Batman drawing code Wrote Forward Kinematics function Wrote Inverse Kinematics and Batman Point drawing portion of report Wrote Main script to select control method Wrote short safety check and error calculation functions