Bulb Equipped Artistic Manipulator (BEAM)

A 4-DOF Robotic Manipulator for Light Painting

Project Report for UCLA MAE C263A: Kinematics of Robotic Systems

Agathiya Tharun, Benjamin Forbes, Che Jin Goh, Debajyoti Chakrabarti, Eric Wei, Kevin Lee, Premkumar Sivakumar

Instructed by Dr. Dennis Hong

Department of Mechanical and Aerospace Engineering
University of California, Los Angeles

December 13, 2024

Abstract

This report explains the design, implementation, and testing of a 4-DOF robotic manipulator called the Bulb Equipped Artistic Manipulator (BEAM). The system uses four rotary-joint arm with an LED end-effector to trace user-defined paths in three-dimensional space. These paths are captured as glowing text or patterns through long-exposure photography. The manipulator's hardware was designed after performing static force analysis and simulations to ensure stability and accuracy. On the software side, the contour extraction algorithm converts user inputs (in terms of text or image) into desired positions and orientations. Further, based on Forward and Inverse Kinematics the joint trajectories are computed for the corresponding goal positions and orientations. Singularities were handled with joint range limits and Damped Least Squares (DLS) methods to ensure smooth motion. MATLAB simulations validated the system at every step, from image processing to final trajectory execution. The manipulator, equipped with Dynamixel MX-28AR motors and tuned PD controllers, successfully reproduced patterns like text and images with clean lines and accurate orientations.

Contents

1 Introduction

The Bulb Equipped Artistic Manipulator (BEAM) is a 4-DOF robotic manipulator created to produce long-exposure light paintings of text and patterns. BEAM was designed with an artistic touch to paint with light without leveraging a simple gantry system, but rather a complex robotic arm, highlighting the multifaceted dynamics between art and technology. This posed many challenges. The 4-DOF manipulator constrained the workspace to a spherical surface with limited orientation control. The strength and precision of the motors added additional constraints for the design to navigate. The design difficulties of torque limitations, structural integrity, and a limited workspace were considered through an iterative process to continuously optimize the design of BEAM. Static force analyses, FEA simulations, and inverse kinematics simulations guided design. Singularities were mitigated with joint range limits and damping methods to maintain stability.

2 Hardware

The Bulb Equipped Artistic Manipulator (BEAM) is a 4-DOF robotic arm designed to create long-exposure light paintings by moving an LED along planned paths. The manipulator's hardware was developed to provide stability and precision while handling gravitational loads and avoiding excessive torque. Static force analyses ensured the motors could handle the chosen configuration, and Finite Element Analysis (FEA) verified the links' structural integrity. The hardware design included refinements to link lengths and joint limits to achieve stability and sufficient workspace coverage for creating patterns.

2.1 Manipulator Configuration

The manipulator features a 4 rotary-joint configuration (R-R-R-R), with joint angles denoted as θ_1, θ_2, θ_3, and θ_4. Each joint is driven by a Dynamixel MX-28AR servo motor. The length of first three links were chosen to ensure an adequate workspace for drawing text. For the fourth link, the length was chosen to be the smallest, and the same as the motor's length, as the precision in orientation is not of paramount interest in this application.

$$l_1 = 0.15\,\text{m}, \quad l_2 = 0.15\,\text{m}, \quad l_3 = 0.15\,\text{m}, \quad l_4 = 0.04\,\text{m}.$$

2.2 Design Overview

The mechanical design ensured the motors could handle the weight of the links and LED assembly within their torque limits. Link lengths and joint angle ranges were optimized to provide a workspace large enough for legible patterns while maintaining precision and stability. The CAD model was designed in Solidworks, and the links were 3D printed. The physical realization of the BEAM manipulator is shown in Figure 1.

Figure 1: CAD Model (left), The BEAM manipulator (right)

2.3 Static Force Analysis

Our approach to selecting the link lengths was to have them long enough to maximize the available workspace while staying within the motors' torque limit and resolution bounds. Hence, static force analysis was crucial to change and select the link lengths iteratively. To analyze the static force on the manipulator in any given configuration, the forces and moments were balanced for each link about the corresponding joint axis. Secondly, the torque needed at each joint was calculated based on the following equations:

$$T_4 = I_4\alpha_4 + \frac{1}{2}m_4 g l_4 \cos(\theta_4 - \theta_3 + \theta_2)$$

$$T_3 = I_3\alpha_3 + T_4 + \frac{1}{2}m_3 g l_3 \cos(\theta_3 - \theta_2) + (M_4 + m_4) g l_3 \cos(\theta_3 - \theta_2)$$

$$T_2 = I_2\alpha_2 + T_3 + \frac{1}{2}m_2 g l_2 \cos(\theta_2) + (M_3 + m_3 + M_4 + m_4) g l_2 \cos(\theta_2)$$

Where, T_i is the torque at each joint, I_i is the moment of inertia, α_i is angular acceleration, m_i is the mass of the i-th link, M_i is the mass of the i-th motor, and g is the acceleration due to gravity. The calculated torque at each joint was less than the motor's torque capacity of 1.2 N-m. To validate the calculation, Finite Element Analysis was performed on the link experiencing the highest load—link 2. The results from static structural FEA gave a maximum deformation of 0.4 mm, maximum von-Mises stress of 2.283×10^7 Pa, and a minimum factor of safety of 2.6291 in exaggerated scale. The following figures show FEA results.

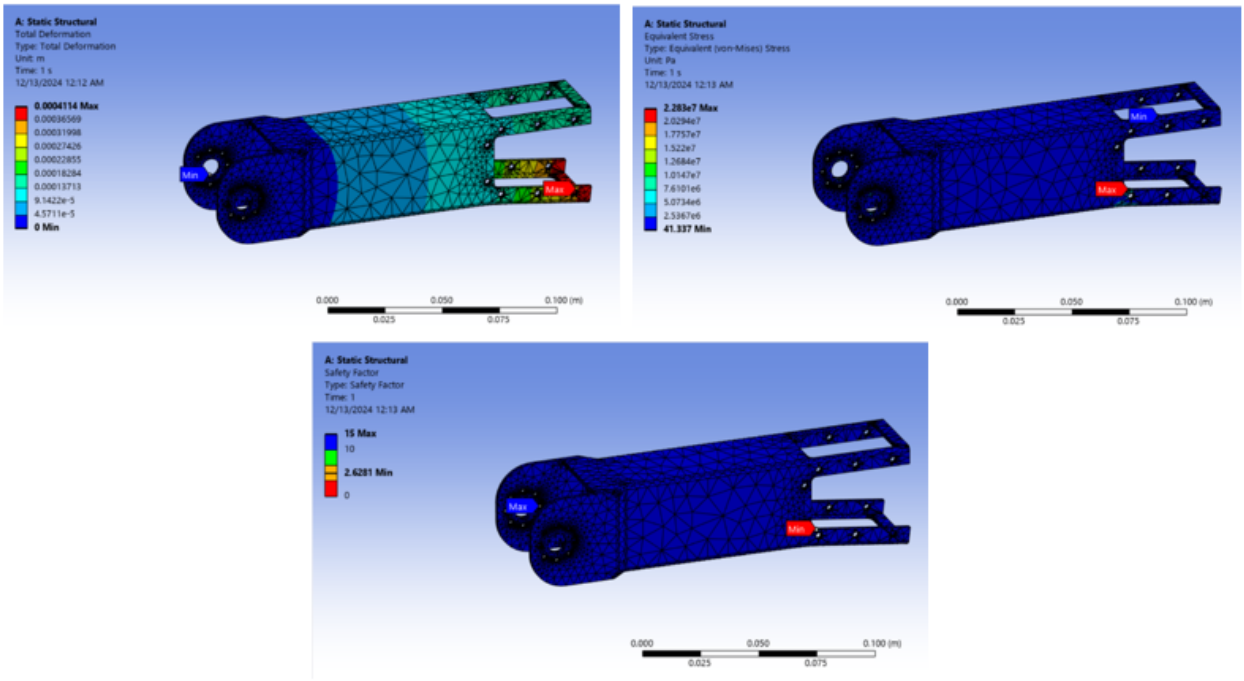

Figure 2: Results of static structural FEA: total deformation (top left), equivalent stress (top right), safety factor (bottom)

3 Software and Methodology

3.1 Trajectory Generation

The trajectory generation process converts input text into 3D coordinates that the robotic manipulator can follow. The process begins by rendering the text into a 2D image with font specifications like type, size, and style. The image is then binarized to separate the text from the background, followed by noise removal to clean the representation and an optional "skeletonization". Contours of the text are extracted to define an optimized tracing path for the robot's end-effector, minimizing unwanted retracing and leveraging pen-up moments. The contours are converted to Cartesian coordinates and scaled to fit the robot's workspace and maximize a non-distorted plane on a spherical surface. The final matrix was also manipulated to maximize smooth and natural movements for the manipulator to follow. Fig. 3 shows the output of trajectory generation script in MATLAB for the given input text "UCLA" as a *bubble letter* and as a *lower-case letter*, respectively.

Figure 3: Contour from input text

3.2 Forward Kinematics

The forward kinematics analysis calculates the position and orientation of the end-effector relative to the base frame using joint angles and link geometry. It is implemented using the Denavit-Hartenberg (DH) parameters, which define the geometric relationship between each joint and link of the robotic arm. The joint matrices are multiplied iteratively to compute the cumulative transformation matrix that represents the end-effector's position and orientation. The final transformation matrix from the base to the end-effector is expressed in terms of the DH parameters and joint angles, with the '0' frame representing the base frame and the '5' frame representing the frame attached to the end-effector tip.

$$f(\theta) = {}^{0}T_{5} = {}^{0}T_{1} \cdot {}^{1}T_{2} \cdot {}^{2}T_{3} \cdot {}^{3}T_{4} \cdot {}^{4}T_{5}$$

The DH parameters for the robotic manipulator are given in Table 1.

Table 1: DH Parameters for the Robotic Manipulator

$i-1$	i	a_i (m)	α_i (rad)	d_i (m)	Θ_i (rad)
0	1	0	0	0.1	θ_1
1	2	0	$\pi/2$	0	θ_2
2	3	0.15	0	0	θ_3
3	4	0.15	0	0	θ_4
4	5	0.04	0	0	0

The transformation matrix for each joint is defined as:

$$T_i = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The final transformation matrix from the base to the end-effector is:

$${}^{0}T_{5} = \begin{bmatrix} C_{234}C_1 & -S_{234}C_1 & S_1 & C_1(a_3C_{23} + a_2C_2 + a_4C_{234}) \\ C_{234}S_1 & -S_{234}S_1 & -C_1 & S_1(a_3C_{23} + a_2C_2 + a_4C_{234}) \\ S_{234} & C_{234} & 0 & a_1 + a_3S_{23} + a_2S_2 + a_4S_{234} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where $C_x = \cos(\theta_x)$, $S_x = \sin(\theta_x)$, and x represents the joint angles.

3.3 Inverse Kinematics and Solution Selection

The inverse kinematics (IK) process converts a desired end-effector position (P_x, P_y, P_z) into a set of joint angles $(\theta_1, \theta_2, \theta_3, \theta_4)$. The MATLAB implementation uses algebraic and trigonometric principles to calculate these angles and includes criteria for selecting the best

solution to ensure smooth movement. Let the desired homogenous transformation matrix be,

$$ {}^{0}_{5}\mathbf{T}_{\text{desired}} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Where,

$$ {}^{0}_{5}\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} $$

is the rotation matrix from the base frame to frame 5, and

$$ \mathbf{T}_{5\text{ORIG}} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} $$

is the location of frame 5's origin with respect to the base frame.

To solve this inverse kinematics problem, the forward kinematics is equated to the desired homogenous transformation matrix. The solutions are given by:

$$ \theta_1 = \text{Atan2}(r_{13}, r_{23}) \quad [\textbf{1 Solution}] $$

$$ \theta_{234} = \theta_2 + \theta_3 + \theta_4 = \text{Atan2}(r_{31}, r_{32}) $$

If $\cos\theta_1 \neq 0$, define,

$$ \zeta = \frac{r_{14}}{C_1} $$

Else,

$$ \zeta = \frac{r_{24}}{S_1} $$

Define,

$$ \zeta - a_4 C_{234} = a_3 C_{23} + a_2 C_2 = \alpha $$

and

$$ r_{34} - a_1 - a_4 S_{234} = a_3 S_{23} + a_2 S_2 = \beta $$

Squaring and adding α, β,

$$ C_3 = \frac{(\alpha^2 + \beta^2) - (a_3^2 + a_2^2)}{2a_2 a_3}, \quad S_3 = \pm\sqrt{1 - C_3^2} $$

$$ \theta_3 = \text{Atan2}(S_3, C_3) \quad [\textbf{2 Solutions}] $$

Now,

$$ \alpha = a_3 C_{23} + a_2 C_2, \quad \beta = a_3 S_{23} + a_2 S_2 $$

Which implies,

$$ (a_3 C_3 + a_2) S_2 + (a_3 S_3) C_2 = \beta $$

$$ (a_3 C_3 + a_2) C_2 - (a_3 S_3) S_2 = \alpha $$

The above equations can be re-written as,

$$\begin{bmatrix} -a_3S_3 & (a_3C_3 + a_2) \\ (a_3C_3 + a_2) & a_3S_3 \end{bmatrix} \begin{bmatrix} S_2 \\ C_2 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

If $\sin\theta_3 \neq 0$,

$$C_2 = \frac{\alpha(a_3C_3 + a_2) + \beta a_3S_3}{(a_3C_3 + a_2)^2 + (a_3S_3)^2}, \quad S_2 = \frac{\beta(a_3C_3 + a_2) - \alpha a_3S_3}{(a_3C_3 + a_2)^2 + (a_3S_3)^2}$$

$$\theta_2 = \text{Atan2}(S_2, C_2) \quad [\textbf{2 Solutions}] \quad \text{(each corresponds to unique } \theta_3\text{)}$$

Else,

$$\theta_2 = \text{Atan2}(\beta, \alpha) \quad [\textbf{1 Solution}]$$

Finally,

$$\theta_4 = \theta_{234} - (\theta_2 + \theta_3) \quad [\textbf{2 Solutions}] \quad \text{(each corresponds to unique } \theta_3 \text{ and } \theta_2\text{)}$$

3.4 Singularities

Singularities create challenges in robotic manipulators by causing instability and loss of control. To address this, a singularity check was added to the inverse kinematics function to ensure stable operation of the 4-DOF robotic arm. Geometric singularities, such as the alignment of manipulator links, were avoided by limiting the joint ranges. These constraints ensured the manipulator stayed within a well-conditioned workspace, reducing the risk of instability during operation.

Although the Damped Least Squares (DLS) method is not strictly necessary, it was implemented to handle near-singular configurations for added robustness. The DLS method modifies the Jacobian matrix by introducing a damping factor, preventing instability near singularities. The modified Jacobian is calculated as:

$$J^{\dagger}_{\text{damped}} = J^T(JJ^T + \lambda^2 I)^{-1},$$

where J is the Jacobian matrix, λ^2 is the damping factor, and I is the identity matrix. This adjustment allows the manipulator to move smoothly and avoids abrupt changes in joint angles, even in complex scenarios. While simpler techniques may suffice, implementing DLS provided an extra layer of stability for the system.

3.5 Workspace Analysis

Workspace refers to all locations the end-effector of the 4-DOF robotic arm can reach. In this project, the workspace was analyzed using MATLAB by evaluating every point within the range of each joint angle. A mapping was created to associate each joint variable with its range:

$$\theta_1 : [-\frac{\pi}{2}, \frac{\pi}{2}], \quad \theta_2 : [-\frac{\pi}{2}, \frac{\pi}{2}], \quad \theta_3 : [-\frac{\pi}{2}, \frac{\pi}{2}], \quad \theta_4 : [-\frac{\pi}{2}, \frac{\pi}{2}]$$

The "plot3dworkspace" function was used to compute and visualize all possible positions by iterating through all joint angle combinations. The resulting workspace is shown in Fig 4.

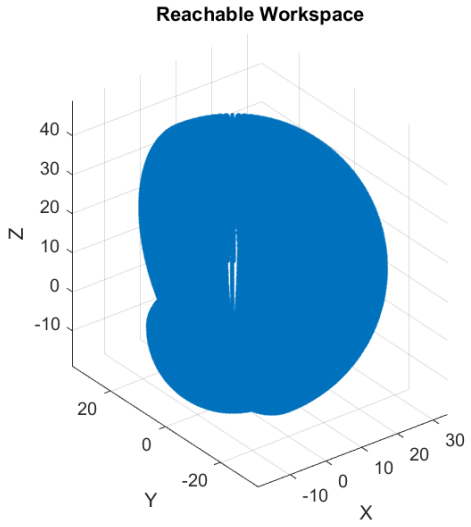

Figure 4: Visualization of workspace

3.6 Simulation and Visualization

The simulation and visualization environment in MATLAB was essential to confirm the manipulator's ability to produce patterns like text or circular trajectories. Before commanding the hardware, the software replicated the entire motion planning process virtually, ensuring each trajectory segment was kinematically feasible and visually coherent.

Workflow Integration: The simulation began by extracting contours from an input, generating waypoints (x, y, z). These waypoints, derived from image processing and coordinate transformations, accounted for pen-up and pen-down states, scaling, and axis adjustments to fit within the manipulator's workspace. Based on computed orientations for a position and the position itself, the inverse kinematics solver determined joint angles for each waypoint, ensuring smooth transitions by selecting solutions closest to the current configuration. Joint angle constraints and singularity handling strategies prevented erratic motions and kept configurations stable within the workspace.

Software Design for Path Verification: The software stepped through each computed joint configuration and applied forward kinematics to determine the end-effector's simulated position. This closed-loop verification cross-checked input waypoints, joint angle solutions, and forward kinematics outputs to ensure patterns were physically realizable. If the simulation revealed misalignments, unexpected drift, or awkward orientations, adjustments were made to parameters, contour extraction steps, or joint constraints before testing on the actual robot.

Visualization of the Simulated Trajectory: Figure 5 shows a simulation snapshot where the manipulator's end-effector traces "UCLA" in three-dimensional space. The blue line represents the robotic arm's configuration, while the red path corresponds to the text trajectory. This visualization confirmed that the scaled and transformed coordinates produced legible patterns and orientation adjustments worked as intended.

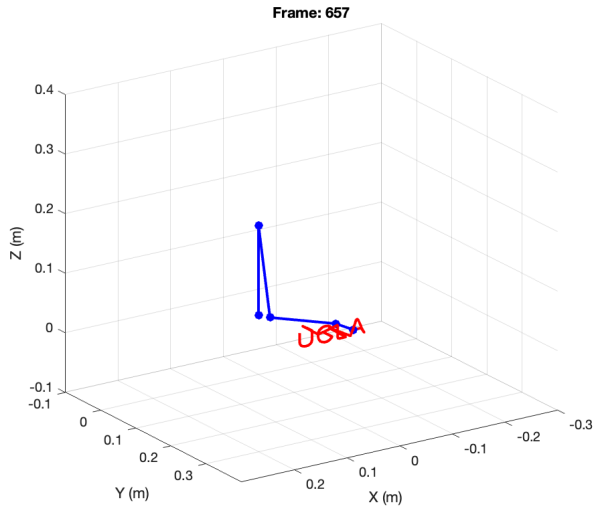

Figure 5: Simulation snapshot of the BEAM manipulator (blue) virtually tracing "UCLA"

Orientation and Trajectory Consistency: The simulation addressed orientation issues by applying axis flips and coordinate adjustments after contour extraction. These transformations ensured that rendered patterns matched the intended alignment and directionality. Testing these adjustments during simulation guaranteed that text like "UCLA" appeared correctly oriented rather than mirrored or rotated.

Outcome and Confidence Building: The simulation verified the entire motion, from text contour extraction to workspace fitting, IK computation, and forward kinematics validation. This process ensured stable commands, smooth transitions between waypoints, and visually coherent patterns, reducing reliance on trial-and-error testing with hardware. It provided confidence that the manipulator would perform as expected in physical demonstrations.

3.7 Control Scheme

All motors were daisy-chained together to minimize cabling complexity. This configuration allows multiple servos to be addressed individually via their unique IDs over a single communication line.

The Dynamixel motors were put into position control mode so that the desired angles could be fed from the .mat files to each motor individually. Proportional and derivative control was utilized as seen below with gains of $K_p = 800$ and $K_d = 100$. These gains allowed

for accurate position control without many oscillations. This controller was implemented in Dynamixel Wizard 2.0 by selecting "position control" and setting the gains in the firmware.

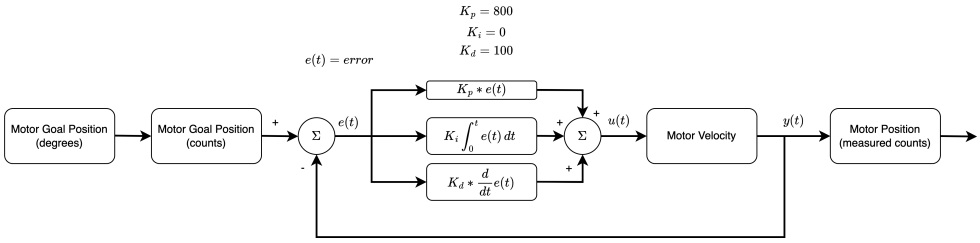

Figure 6: PID control scheme block diagram

The motors were controlled via U2D2 in Python. The code initializes the motors and their ID's and reads in the .mat file that was outputted by the inverse kinematics script by way of the text scraper code, which creates the workspace and projects the text onto a plane the robot can "print" on. The code (Appendix 1) allows each motor to reach the desired position at a single time. The code moves on to the next point after the position is reached within a certain specified tolerance.

4 Discussion

The BEAM manipulator demonstrated its capabilities through successful hardware, kinematic, and control integration, as verified by demonstrations referenced in Appendix C. However, BEAM has plenty of room for improvement as it represents a budget-friendly alternative to existing similar models on the market. Current light-painting robots leverage a KUKA arm or other existing robotic arms. BEAM, being built from scratch for its hardware and software, is thus a more unique and special alternative to current solutions as it leverages more affordable hardware and a more artistic and individualistic approach to light painting by capitalizing on a spherical, non-planar, surface. BEAM, being multifaceted and multidimensional in its characteristics, thus cannot be simply compared to commercial alternatives.

The closest commercial alternatives leverage advanced robotic arms, giving them an advantage over BEAM. These areas for improvement for BEAM include: enhanced workspace, smoother control and joint dynamics, reduced edge-warping from a spherical workspace, automating systems to reduce human intervention, improved tracing algorithm, and the ability to paint with more colors.

Figure 7: Long-exposure photograph of the manipulator writing "UCLA" in light

5 Conclusion

The BEAM project showed how a 4-DOF robotic arm can create "light paintings" from user inputs such as text and simple images. By applying mechanical design principles, kinematic analysis, and control strategies, the system transformed patterns into luminous trajectories suspended in space, captured through long-exposure photography. Each stage of the project, from hardware design using static force and finite element analyses to forward and inverse kinematics implementation, contributed to the manipulator's reliable and stable performance.

The project's strengths included its integrated approach. Steps such as text contour extraction, scaling, workspace transformations, and singularity handling ensured all parts of the process worked effectively. The manipulator accurately rendered patterns like "UCLA" with smooth motion, aided by a control system tuned for stability. While the system met its goals, further improvements like smoother control, usage automation, user interfaces, or sensor feedback could enhance its reliability and broaden its applications. The BEAM system demonstrated how thoughtful engineering can turn abstract ideas into functional and creative robotic outputs.

6 Acknowledgments

We thank Dr. Dennis Hong for his guidance and support throughout this project. His feedback helped us improve our understanding of robotics and refine our design approach. We also thank our teaching assistant, Arturo F. Alvarez, for his assistance and practical advice, which were valuable during the development process. We thank our team for their dedication and commitment to the project and the MAE department for their funding and support.

7 References

1. https://www.cameronchaney.com/light-drawing-robot

2. ASM Material Data Sheet, (n.d.). [Online]. Available: https://asm.matweb.com/search/SpecificMaterial.asp?bassnum=mq316q

3. J. J. Craig, *Introduction to Robotics: Mechanics and Control*, Pearson, 3rd ed., 2005.

4. V. Kurtz *et al.*, "Passivity-Based Control Barrier Functions for Robotic Manipulators Near Singularities," *arXiv preprint arXiv:2109.13349*, 2021. [Online]. Available: https://www.arxiv.org/abs/2109.13349

5. Dynamixel Documentation and Tutorials. [Online]. Available: https://emanual.robotis.com/

Appendix

Appendix A: Code Files

The following sections include the code implementations developed throughout the project. These listings cover trajectory generation, forward and inverse kinematics, workspace plotting, and simulation of the manipulator's motion.

Static Force Analysis

This MATLAB script computes torques at each joint under various gravitational loading conditions and checks if any configuration exceeds the motor's torque capabilities. By iterating over a specified range of joint angles, the code ensures that the chosen link dimensions and motor selections remain within safe torque limits.

```
clc;
clear;

% Assuming all the links are 10x3x3 cm in dimension
% Mass of the links are calculated based off PLA density and assumed
% dimensions for the links

l = [0.1 0.1 0.1 0.1]; % length of the links (m)
w = [0.03 0.03 0.03 0.03]; % width of the links (m)
h = [0.03 0.03 0.03 0.03]; % thickness of the links (m)
alpha = [8 8 8 8]; % angular acceleration (deg/s^2)
m = [0.11 0.11 0.11 0.11]; % mass of links (kg)
M = [0.077 0.077 0.077 0.077]; % mass of motors (kg)
g = 9.81; % acceleration due to gravity (m/s^2)
Ic = (1/12)*(l.^2).*(w.^2).*M; % moment of interia through the
   centroid of the links
I = Ic + (w/2).^2.*M; % moment of interia about an axis through the
   joint
flag = 0;
maxT4 = 0;
maxT3 = 0;
maxT2 = 0;
Output=[];

for theta2 = 0:90
    for theta3 = 0:90
        for theta4 = 0:60

            T4 = I(4)*alpha(4) + m(4)*g*(l(4) /2)*cos(theta4 -
               theta3 + theta2);
            T3 = I(3)* alpha(3) + T4 + m(3)*g*(l(3)/2)*cos(theta3 -
               theta2) + (M(4) + m(4))*g*l(3)*cos(theta3 - theta2);
```

```
            T2 = I(2)*alpha(2) + T3 + m(2)*g*(l(2)/2)*cos(theta2) +
               (M(3) + m(3) + M(4) + m(4))*g*l(2)*cos(theta2);
            if abs(T2) > 1.2 || abs(T3) > 1.2 || abs(T4) > 1.2
                flag = 1;
                break;
            end
            %Output = [Output; T4 T3 T2 theta4 theta3 theta2 flag];
            if T4>maxT4
                maxT4 = T4;
            end
            if T3>maxT3
                maxT3 = T3;
            end
            if T2>maxT2
                maxT2 = T2;
            end
        end
    end
end

fprintf('Maximum needed torque at joint 2: %f N-m\n', maxT2)
fprintf('Maximum needed torque at joint 3: %f N-m\n', maxT3)
fprintf('Maximum needed torque at joint 4: %f N-m\n', maxT4)

if flag == 0
    fprintf('\nThe moment at each joint is within the motor torque
       capacity')
else
    fprintf('\nThe motor torque is insufficient')
end

%plot(Output(1), Output(7))



%theta4 = 0:60;
%theta3 = 0:90;
%theta2 = 0:90;
%[THETA2, THETA3, THETA4] = meshgrid(theta4);
%T4 = I(4)*alpha(4) + m(4)*g*(l(4) /2)*cos(THETA4 - THETA3 + THETA2)
   ;
%T3 = I(3)* alpha(3) + T4 + m(3)*g*(l(3)/2)*cos(THETA3 - THETA2) + (
   M(4) + m(4))*g*l(3)*cos(THETA3 - THETA2);
%T2 = I(2)*alpha(2) + T3 + m(2)*g*(l(2)/2)*cos(THETA2) + (M(3) + m
   (3) + M(4) + m(4))*g*l(2)*cos(THETA2);

%volshow('CData', T4); % Visualize the volume data
```

```
%view(3); % Set a 3D view

% Add labels and title
%xlabel('X-axis');
%ylabel('Y-axis');
%zlabel('Z-axis');
%title('Volume Visualization of f(x, y, z)');
%colorbar; % Add a color bar
```

Listing 1: `StaticForceAnalysis.m` - Computes torques and checks motor capacity under gravitational loading

Trajectory Generation from Input

Below is the MATLAB implementation of the trajectory generation algorithm, which converts input text or images into a set of 3D end-effector coordinates for the manipulator, reachable within its workspace.

```
function coordinatesMatrix = motion(inputText, fontSize, fontName,
    fontStyle, thinOption)
    % Parameters
    thinning = thinOption;  % Reduce to skeleton; turn off for
        images
    xOffset = 19; % [16 to 19] cm
    scaleFactor = 1;  % Initial scale factor; untuned
    pauseTime = 0.001;  % Time delay between each movement (seconds)
    debug = false;  % Hide animations?
    spherical = true; % Print fisheyed or planar (false)?
    jerk = 3; % Magnitude of jerkiness; reduces fidelity to
        compensate for position control and jerky movements

    % Create image from text if not uploading an image
    if ~inputText == 0 % Enter 0 for using an image and not text
        % Styles: normal, bold
        % Font size (ideally 40+) increases number of coordinates
            and text resolution (dpi).
            % Reduce font size if text is getting truncated
        createTextImage(inputText, fontSize, 10, 10, 'image.png',
            fontName, fontStyle);
    end

    % Load and process the image
    img = imread('image.png');          % Load image
    grayImg = rgb2gray(img);            % Convert to grayscale
    binaryImg = imbinarize(grayImg);    % Binarize the image
    binaryImg = ~binaryImg;             % Invert (black text on white
        )

```

```
    % Remove edge artifacts and small noise
    binaryImg = imclearborder(binaryImg);  % Clear objects connected
        to the image border
    binaryImg = bwareaopen(binaryImg, 10); % Remove small objects (
       noise)

    % Identify connected components (each letter or shape)
    cc = bwconncomp(binaryImg);  % Find connected components (
       letters)
    coordinatesMatrix = [];  % Initialize coordinates matrix
    maxY = size(binaryImg, 1) * scaleFactor;  % Flip Y-coordinates
       based on image height

    % Set up the animation
    figure;
    hold on;
    axis equal;
    axis off;
    xlim([0, size(binaryImg, 2) * scaleFactor]);
    ylim([0, size(binaryImg, 1) * scaleFactor]);
    set(gcf, 'Color', 'w');

    % Loop through each connected component
    for i = 1:cc.NumObjects
        % Create a mask for the current connected component
        componentMask = false(size(binaryImg));
        componentMask(cc.PixelIdxList{i}) = true;

        % Extract boundaries for this component (outer and inner
           boundaries)
        componentBoundaries = bwboundaries(componentMask, 'holes');
            % Include inner holes

        % Optional thinning
        if thinning
            thinnedImg = bwmorph(binaryImg, 'thin', Inf); %
               Skeletonize the binary image

            % Retain components with area between 1 and 10 pixels (
               adjust thresholds as needed)
            dotMask = bwareaopen(thinnedImg, 1) & ~bwareaopen(
               thinnedImg, 10);

            % Prune small spurs (short branches)
            prunedImg = bwmorph(thinnedImg, 'spur', 5); % Removes
               branches shorter than 5 pixels

```

```
62              % Combine the pruned image with preserved dots
63              thinnedImg = prunedImg | dotMask;
64
65              % Extract contours (without holes, due to thickness)
66              componentBoundaries = bwboundaries(thinnedImg, 'noholes'
                    ); % Get only outer boundaries
67          end
68
69          % Draw all boundaries for this letter/component
70          for k = 1:length(componentBoundaries)
71              boundary = componentBoundaries{k};
72              scaledBoundary = boundary * scaleFactor;
73
74              scaledBoundary = removeUnnecessaryRetraces(
                    scaledBoundary);
75
76              % "Lift pen" before moving to the start of a new
                    boundary
77              if k > 1
78                  if thinning
79                      disp('Next component')
80                      coordinatesMatrix = [coordinatesMatrix; NaN NaN
                            NaN];
81                  else
82                      disp('Inner boundary')
83                      coordinatesMatrix = [coordinatesMatrix; NaN NaN
                            NaN];
84                  end
85                  xPenUp = scaledBoundary(1, 2);
86                  yPenUp = maxY - scaledBoundary(1, 1);
87                  coordinatesMatrix = [coordinatesMatrix; xPenUp,
                        yPenUp, 0];
88              end
89
90              % Draw and store the current boundary points
91              for pointIndex = 1:size(scaledBoundary, 1) - 1
92                  x1 = scaledBoundary(pointIndex, 2);
93                  y1 = maxY - scaledBoundary(pointIndex, 1);
94                  x2 = scaledBoundary(pointIndex + 1, 2);
95                  y2 = maxY - scaledBoundary(pointIndex + 1, 1);
96
97                  % Append the current point to the coordinates matrix
98                  % disp('Light on')
99                  if isnan(x1)
100                     coordinatesMatrix = [coordinatesMatrix; x1, y1,
                            NaN];
101                 else
```

```
102                     coordinatesMatrix = [coordinatesMatrix; x1, y1,
                            0];
103                 end
104
105                 % Animate the drawing; uncomment to see trace cursor
106                 if ~debug
107                     plot([x1, x2], [y1, y2], 'k-', 'LineWidth', 2);
108                     %currentPoint = plot(x1, y1, 'ro', 'MarkerSize',
                            6, 'MarkerFaceColor', 'r'); % Cursor point
109                     drawnow;
110                     pause(pauseTime);
111                     %delete(currentPoint); % Remove the cursor
112                 end
113             end
114         end
115
116         disp('Next component / End')
117         coordinatesMatrix = [coordinatesMatrix; NaN NaN NaN];
118
119         if thinning
120             break % Force exit if there are no "components"
121         end
122     end
123
124     % Reorganize to print on yz plane
125     temp = coordinatesMatrix(:, 1:2);
126     coordinatesMatrix(:, 1) = coordinatesMatrix(:, 3);
127     coordinatesMatrix(:, 2:3) = temp;
128
129     % Offset and rescale coordinates
130     max_Y_len = max(coordinatesMatrix(:, 2)) - min(coordinatesMatrix
            (:, 2))
131     max_Z_len = max(coordinatesMatrix(:, 3)) - min(coordinatesMatrix
            (:, 3))
132     constrainingDim = max(max_Y_len, max_Z_len);
133
134     if spherical
135         lim = 1;
136     else
137         lim = 0.8; % 80% downsize to reduce edge warping
138     end
139
140     % if max_Y_len > max_Z_len
141     %     scaleFactor = scaleFactor * (xOffset * 2 * lim) /
            constrainingDim
142     % else
143     %     scaleFactor = scaleFactor * (xOffset - 1) * lim /
```

```
        constrainingDim % -1 to account for extra 1cm offset in z (15
            to 16)
144     % end
145
146     scaleFactor = scaleFactor * (xOffset - 1) * lim /
            constrainingDim
147
148     coordinatesMatrix = scaleFactor * coordinatesMatrix;
149
150     noNaN = rmmissing(coordinatesMatrix);
151
152     mids = min(noNaN) + (max(noNaN) - min(noNaN)) / 2;
153
154     yOffset = -mids(2);
155     zOffset = -mids(3);
156
157     coordinatesMatrix(:, 1) = coordinatesMatrix(:, 1) + xOffset;
158     coordinatesMatrix(:, 2) = coordinatesMatrix(:, 2) + yOffset;
159     coordinatesMatrix(:, 3) = coordinatesMatrix(:, 3) + zOffset +
            max(coordinatesMatrix(:, 3) + zOffset);
160
161     % Reduction
162     if jerk > 0
163         coordinatesMatrix = pointReduction(coordinatesMatrix, jerk);
164     end
165
166     if debug
167         plot(coordinatesMatrix(:, 2), coordinatesMatrix(:, 3), '.-',
                'LineWidth', 2); % Plotting on yz graph
168         xlim([min(coordinatesMatrix(:, 2)) max(coordinatesMatrix(:,
                2))]);
169         ylim([min(coordinatesMatrix(:, 3)) max(coordinatesMatrix(:,
                3))]); % y axis plots physical z coordinates
170         disp(['ylim: ', num2str([min(coordinatesMatrix(:, 2)) max(
                coordinatesMatrix(:, 2))])])
171         disp(['zlim: ', num2str([min(coordinatesMatrix(:, 3)) max(
                coordinatesMatrix(:, 3))])])
172     end
173
174     % Add z-offset (link 1 length)
175     zOffset = 16;
176     coordinatesMatrix(:, 3) = coordinatesMatrix(:, 3) + zOffset;
177
178     % coordinatesMatrix contains all the final coordinate output
            values in sequential order relative to the base frame origin
179 end
180
```

```
function filtered = pointReduction(arr, jerk)
    % Logical array to identify rows with [NaN NaN NaN]
    is_nan_row = all(isnan(arr), 2);

    % % Logical array for alternating rows
    % alternating_rows = false(size(arr, 1), 1);
    % alternating_rows(1:1+jerk:end) = true; % Mark every other row
    %
    % % Combine conditions: keep rows that are either NaN or not part of alternating rows
    % rows_to_keep = is_nan_row | ~alternating_rows;

    % Create a logical mask for rows to keep
    rows_to_keep = false(size(arr, 1), 1);
    counter = 1;

    % Iterate through rows and decide which to keep
    while counter <= size(arr, 1)
        % Keep the current row
        rows_to_keep(counter) = true;

        % Skip the specified number of rows to remove
        counter = counter + jerk + 1;
    end

    % Combine conditions: keep rows with [NaN NaN NaN] or rows marked to keep
    rows_to_keep = is_nan_row | rows_to_keep;

    % Filter the matrix
    filtered = arr(rows_to_keep, :);
end

function reduced = removeUnnecessaryRetraces(arr)
    % Initialize the reduced array
    reduced = [];

    % Keep track of previously encountered sets
    uniqueSets = [];

    % Flag to track if a discontinuity has been inserted
    discontinuityInserted = false;

    % Iterate through the rows of arr
    i = 1;
    while i <= size(arr, 1)
        % Look ahead to find the end of the current set
```

```
        j = i;
        while j <= size(arr, 1) && isequal(arr(j, :), arr(i, :))
            j = j + 1;
        end

        % Extract the current set
        currentSet = arr(i:j-1, :);

        % Check if this set is unique
        if isempty(uniqueSets) || ~ismember(currentSet, uniqueSets,
           'rows')
            % Append the current set to the reduced array
            reduced = [reduced; currentSet];

            % Mark this set as unique
            uniqueSets = [uniqueSets; currentSet];

            % Reset the discontinuity flag
            discontinuityInserted = false;
        else
            % Insert [NaN NaN] only once per removed chunk
            if ~discontinuityInserted && ~isempty(reduced) && ~
               isequal(reduced(end, :), [NaN NaN])
                reduced = [reduced; NaN NaN]; % Note: PENUP will not
                    displayed but coordinatesMatrix will reflect it
                % disp('Next Component')
                discontinuityInserted = true;
            end
        end

        % Move to the next set
        i = j;
    end
end

function createTextImage(inputText, fontSize, imageWidth,
   imageHeight, outputFileName, fontName, fontStyle)
    % fontName       : Name of the font (e.g., 'Arial', 'Times New
       Roman')
    % fontStyle      : Font style ('normal', 'bold')

    % Create a blank canvas
    figure('Visible', 'off');
    imshow(ones(imageHeight, imageWidth), 'InitialMagnification', '
       fit'); % White canvas
    hold on;
    axis off;
```

```
267
268     % Render the text with specified font and style
269     text(imageWidth / 2, imageHeight / 2, inputText, ...
270         'HorizontalAlignment', 'center', 'VerticalAlignment', '
            middle', ...
271         'FontSize', fontSize, 'FontName', fontName, 'FontWeight',
            fontStyle, ...
272         'Color', 'black');
273
274     % Capture the figure as an image
275     frame = getframe(gcf);          % Capture the figure content
276     textImg = frame.cdata;          % Extract image data
277     close;                          % Close the figure
278
279     % Save the generated image to a file
280     imwrite(textImg, outputFileName); % Save as PNG or any format
281
282     % Inform the user
283     disp(['Image␣saved␣as␣' outputFileName]);
284 end
```

Listing 2: `motion.m` - Generating a 3D trajectory from input text

Rotation Matrix Utility

This function computes the rotation matrix achievable for a given position within the workspace. This allows the inverse kinematics to be solved.

```
1  function rotation_matrix = findRotMat(arr)
2
3      debug = false;
4
5      if debug
6          % Prompt the user for a point on the planar surface
7          disp('Enter␣a␣point␣on␣the␣plane␣(x␣   ␣[16,␣19],␣y␣   ␣[-x,
               ␣x],␣z␣   ␣[0,␣x]):');
8          x_input = input('x-coordinate:␣');
9          y_input = input('y-coordinate:␣');
10         z_input = input('z-coordinate:␣');
11
12         % Validate the input point
13         if ~((x_input <= 19 && x_input >= 16) && (y_input <= x_input
               && y_input >= -x_input) && (z_input <= x_input &&
              z_input >= 0))
14             error('Point␣is␣not␣on␣the␣planar␣surface.␣Please␣try␣
                   again.');
15         end
16     else
```

```
        y_input = arr(2);
        z_input = arr(3);
        x_input = arr(1);
    end

    % Define the vertices of the planar surface
    vertices = [-x_input, x_input, x_input;
                 x_input, x_input, x_input;
                -x_input,  0, x_input;
                 x_input,  0, x_input];

    % Define the point and vector from the origin
    point_on_plane = [x_input, y_input, z_input];
    vector = point_on_plane - [0, 0, 0];
    unit_vector = vector / norm(vector); % Normalize the vector

    % Compute the rotation matrix to align the new X-axis with the
       vector
    % Step 1: The new X-axis is the direction of the vector
    new_x = unit_vector;

    % Step 2: Determine a perpendicular vector for the new Y-axis
    % Arbitrary cross-product with a reference vector to find a
       perpendicular vector
    if abs(new_x(1)) < abs(new_x(2)) && abs(new_x(1)) < abs(new_x(3)
       )
        ref_vector = [1, 0, 0];
    else
        ref_vector = [0, 0, 1];
    end
    new_y = cross(new_x, ref_vector);
    new_y = new_y / norm(new_y); % Normalize the new Y-axis

    % Step 3: Determine the new Z-axis as perpendicular to both new
       X and Y
    new_z = cross(new_x, new_y);

    % Step 4: Construct the rotation matrix with manual 90 degree
       rotation about x'
    rotation_matrix = [new_x; new_y; new_z]' * [1 0 0; 0 0 -1; 0 1
       0] * [1 0 0; 0 -1 0; 0 0 -1];

    if debug
        % Extract angles for reference
        theta_x = atan2(rotation_matrix(3, 2), rotation_matrix(3, 3)
           );
        theta_y = atan2(-rotation_matrix(3, 1), sqrt(rotation_matrix
```

```
    (3, 2)^2 + rotation_matrix(3, 3)^2));
theta_z = atan2(rotation_matrix(2, 1), rotation_matrix(1, 1)
    );

% Display the resulting rotation matrix
disp('Rotation Matrix to align X-axis with vector:');
disp(rotation_matrix);

% Display angles for rotations
disp(['Rotation about X-axis: ', num2str(theta_x * (180/pi))
    , ' degrees']);
disp(['Rotation about Y-axis: ', num2str(theta_y * (180/pi))
    , ' degrees']);
disp(['Rotation about Z-axis: ', num2str(theta_z * (180/pi))
    , ' degrees']);

% 3D Plot Visualization
figure;
hold on;
grid on;
axis equal;
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
title('3D Rotation and Vector Visualization');

% Plot the planar surface
fill3(vertices([1, 2, 4, 3], 3), vertices([1, 2, 4, 3], 1),
    vertices([1, 2, 4, 3], 2), ... % Swap Y and Z
        'cyan', 'FaceAlpha', 0.3);

% Plot the vector from origin to the input point
quiver3(0, 0, 0, vector(1), vector(2), vector(3), 0, 'r', '
    LineWidth', 2, 'MaxHeadSize', 2);
text(vector(1), vector(2), vector(3), 'Input Vector', '
    FontSize', 12, 'Color', 'r');

% Plot the original coordinate axes
quiver3(0, 0, 0, 10, 0, 0, 0, 'k', 'LineWidth', 1); % X-axis
text(10, 0, 0, 'X', 'FontSize', 12);
quiver3(0, 0, 0, 0, 10, 0, 0, 'k', 'LineWidth', 1); % Y-axis
     (was Z)
text(0, 10, 0, 'Y', 'FontSize', 12);
quiver3(0, 0, 0, 0, 0, 10, 0, 'k', 'LineWidth', 1); % Z-axis
     (was Y)
text(0, 0, 10, 'Z', 'FontSize', 12);

```

```
        % Plot the rotated coordinate axes at the tip of the vector
        tip = vector; % Translate axes to the vector's endpoint
        rotated_x = rotation_matrix * [10; 0; 0];
        rotated_y = rotation_matrix * [0; 10; 0];
        rotated_z = rotation_matrix * [0; 0; 10];

        quiver3(tip(1), tip(2), tip(3), ...
                rotated_x(1), rotated_x(2), rotated_x(3), 0, 'b', '
                   LineWidth', 1); % Rotated X-axis
        text(tip(1) + rotated_x(1), tip(2) + rotated_x(2), tip(3) +
           rotated_x(3), ...
             'X''', 'FontSize', 12, 'Color', 'b');

        quiver3(tip(1), tip(2), tip(3), ...
                rotated_y(1), rotated_y(2), rotated_y(3), 0, 'g', '
                   LineWidth', 1); % Rotated Y-axis
        text(tip(1) + rotated_y(1), tip(2) + rotated_y(2), tip(3) +
           rotated_y(3), ...
             'Y''', 'FontSize', 12, 'Color', 'g');

        quiver3(tip(1), tip(2), tip(3), ...
                rotated_z(1), rotated_z(2), rotated_z(3), 0, 'm', '
                   LineWidth', 1); % Rotated Z-axis
        text(tip(1) + rotated_z(1), tip(2) + rotated_z(2), tip(3) +
           rotated_z(3), ...
             'Z''', 'FontSize', 12, 'Color', 'm');

        % Enable 3D rotation
        rotate3d on;

        hold off;
    end
end
```

Listing 3: `findRotMat.m` - Returns the appropriate rotation matrix for a position

DH Parameter and Workspace Utilities

The following functions handle DH parameter transformations, input validation, and calculating the reachable workspace.

```
% Generic link transform function that generates a homogeneous
   transform
% matrix from the DH parameters.
function T = get_DH_matrix(a,alpha,d,theta)
T = [cos(theta) -sin(theta) 0 a
     sin(theta)*cos(alpha) cos(theta)*cos(alpha) -sin(alpha) -sin(
        alpha)*d
```

```
      sin(theta)*sin(alpha) cos(theta)*sin(alpha)  cos(alpha)  cos(
         alpha)*d
      0 0 0 1];
end
```

Listing 4: `get_DH_matrix.m` - Computes a homogeneous transform from DH parameters

```
function validate_inputs(dh_parameters, parameter_ranges)
    % validate_inputs Validates the inputs to the plotting functions
        .
    %
    % This function checks if the given DH parameters are a symbolic
        matrix and if the parameter ranges are
    % a valid containers.Map object with each value as a numeric
       array. Also checks for symbol consistency
    % between dh_parameters and parameter_ranges.
    %
    % Parameters:
    %   dh_parameters : A symbolic matrix that represents the DH
       parameters of the robot.
    %
    %   parameter_ranges : A containers.Map that represents the
       range of each DH parameter.
    %


    % Check DH parameters type
    if ~isa(dh_parameters, 'sym') || ~ismatrix(dh_parameters)
        error('dh_parameters must be a symbolic matrix')
    end

    % Validate the size of DH parameters
    [~, cols] = size(dh_parameters);
    if cols ~= 4
        error('dh_parameters must be a n-by-4 matrix for n links
           each with 4 DH parameters.')
    end

    % Validate that parameter_ranges is a containers.Map
    if ~isa(parameter_ranges, 'containers.Map')
        error('parameter_ranges must be a containers.Map')
    end

    % Validate that each value in b is a triple of numbers
    values_in_params = parameter_ranges.values();
    for i = 1:length(values_in_params)
        if ~isa(values_in_params{i}, 'double')
            error('Each parameter range must be an array of numbers'
```

```
                )
        end
    end

    % Find all unique symbols in dh_parameters
    symbols_in_dh = arrayfun(@char, symvar(dh_parameters), '
       UniformOutput', false);

    % Get all keys in the map ranges
    keys_in_ranges = parameter_ranges.keys();
    extra_dh = setdiff(symbols_in_dh, keys_in_ranges);
    extra_ranges = setdiff(keys_in_ranges, symbols_in_dh);

    % Check if the symbols are the same
    if ~isempty(extra_dh) || ~isempty(extra_ranges)
        error(['The␣symbols␣in␣dh_parameters␣and␣keys␣in␣
           parameter_ranges␣must␣be␣the␣same.␣' ...
            'Excess␣in␣dh_parameters:␣%s.␣Excess␣in␣parameter_ranges
              :␣%s.'], strjoin(extra_dh, ',␣'), strjoin(
              extra_ranges, ',␣'))')
    end
end
```

Listing 5: `validate_inputs.m` - Validates DH parameters and parameter ranges

```
function [x, y, z] = get_xyz_expressions(dh_parameters,
   dh_transform_fn, verbose)
%GET_XYZ_EXPRESSIONS Calculates the x, y, and z positions from
   Denavit-Hartenberg parameters.
%
% Syntax:
%   [x, y, z] = get_xyz_expressions(dh_parameters, dh_transform_fn,
   verbose)
%
% Inputs:
%   dh_parameters: A matrix containing the Denavit-Hartenberg
   parameters.
%                  Each row represents one link, and the columns are
%                  [a,alpha,d,theta] for each link.
%
%   dh_transform_fn: Function handle to a function that computes the
%                    Denavit-Hartenberg transformation matrix. This
%                    function should accept four arguments
   corresponding
%                    to the four Denavit-Hartenberg parameters (a,
   alpha,d,theta).
%
%   verbose: A boolean flag that, when true, triggers the function
```

```
    to
%            display the final transformation matrix and the
    expressions
%            for position.
%
% Outputs:
%   x, y, z: The symbolic expressions for the x, y, and z
    coordinates of
%            the end-effector (i.e., the position of the end-
    effector in
%            the base frame).
%
    % Calculate transformation matrix from base to end-effector
    T0_ee = eye(4);
    % Apply each transformation
    for i = 1:size(dh_parameters, 1)
        T{i} = dh_transform_fn(dh_parameters(i,1), ...
            dh_parameters(i,2), ...
            dh_parameters(i,3), ...
            dh_parameters(i,4));
        T0_ee = T0_ee*T{i};
    end

    x = T0_ee(1,4);
    y = T0_ee(2,4);
    z = T0_ee(3,4);

    if verbose
        disp('final␣transformation␣matrix␣from␣base␣to␣end-effector:
            ')
        disp(T0_ee)
        disp('expressions␣for␣position:')
        fprintf('x␣=␣%s\n', char(x))
        fprintf('y␣=␣%s\n', char(y))
        fprintf('z␣=␣%s\n', char(z))
    end

end
```

Listing 6: `get_xyz_expressions.m` - Obtains symbolic expressions for end-effector position

```
function plot3dworkspace(dh_parameters, parameter_ranges,
    dh_transform_fn, verbose)
%PLOT3DWORKSPACE Plots a 3D representation of the reachable
    workspace of a robot.
%
% Syntax:
%   plot3dworkspace(dh_parameters, parameter_ranges, dh_transform_fn
```

```
    , verbose)
%
% Inputs:
%   dh_parameters: A matrix containing the Denavit-Hartenberg parameters.
%                  Each row represents one link, and the columns are
%                  [a,alpha,d,theta] for each link.
%
%   parameter_ranges: A containers.Map object where the keys are the names
%                     of the parameters (as strings), and the values are arrays
%                     of the values that each parameter can take.
%
%   dh_transform_fn: (Optional) Function handle to a function that computes the
%                    Denavit-Hartenberg transformation matrix. This
%                    function should accept four arguments corresponding
%                    to the four Denavit-Hartenberg parameters (a, alpha,d,theta).
%                    Default: @get_DH_matrix.
%
%   verbose: (Optional) A boolean flag that, when true, triggers the function to
%            display various calculation details like the transformation matrix
%            and more. Default: false.
%
% Outputs:
%   No outputs. The function directly plots the reachable workspace.
%

    arguments
        dh_parameters
        parameter_ranges
        dh_transform_fn = @get_DH_matrix
        verbose = false % Whether to log various calculation details like the transformation matrix and more.
    end

    validate_inputs(dh_parameters, parameter_ranges);
    [x,y,z] = get_xyz_expressions(dh_parameters, dh_transform_fn, verbose);


    % Get parameter grid
```

```
    if verbose
        ranges = values(parameter_ranges);
        lengths = cellfun(@length, ranges);
        product_of_lengths = prod(lengths);
        fprintf('allocating␣%d␣possible␣DH␣parameter␣combinations
            ...\n', product_of_lengths);
    end

    keys = parameter_ranges.keys;
    param_values = parameter_ranges.values(keys);
    [param_grid{1:numel(param_values)}] = ndgrid(param_values{:});

    % Reshape into value arrays in map
    param_value_map = containers.Map;
    for i = 1:numel(param_grid)
        param_value_map(keys{i}) = reshape(param_grid{i}, 1, []);
        param_grid{i} = 0;
    end


    if verbose
        disp('calculating␣positions␣from␣expressions...');
    end
    % Calculate x,y,z positions
    positions = cell(3,1);
    expressions = {x,y,z};
    for i = 1:3
        % Get variables from expression
        expr = expressions{i};
        sym_cell = symvar(expr);
        keys_cell = arrayfun(@char, sym_cell, 'UniformOutput', false
            );
        % Convert from symbolic to numerical functions for faster
            evaluation
        pos_func = matlabFunction(expr, 'Vars', sym_cell);
        func_params = values(param_value_map, keys_cell);
        positions{i} = pos_func(func_params{:});
    end

    % Plot points
    figure(1)
    plot3(positions{:},'.');
    title('Reachable␣Workspace')
    xlabel('X')
    ylabel('Y')
    zlabel('Z')
    axis equal
```

```
    grid on
end
```

Listing 7: `plot3dworkspace.m` - Plots the reachable workspace of the manipulator

```
clc; clear all;

syms theta1 theta2 theta3 theta4 real
% syms d1 d2 real
d1=4;
a2=15;
a3=15;
a4=15;
% DH parameters
test_dh = [0  0 d1 theta1; ...
           0 pi/2 0 theta2; ...
           a2 0 0 theta3; ...
           a3 0 0 theta4; ...
           a4 0 0 0; ...
           ]
% Parameter ranges
points=40;
theta1_range = linspace(0,2*pi, points);
 theta2_range = linspace(0,pi/2, points);
 theta3_range = linspace(0,pi/2, points);
 theta4_range = linspace(0,pi/3, points);
% d1_range = linspace(6,7, 180);
% d2_range = linspace(4,5,180);
test_map = containers.Map({'theta1','theta2','theta3','theta4'},{
   theta1_range,theta2_range,theta3_range,theta4_range});

% Workspace plotting function
plot3dworkspace(test_dh, test_map, @get_alternative_dh_transform)


function out = arr2Rad(A)
    out = arrayfun(@(angle) deg2rad(angle), A);
end

function T = get_alternative_dh_transform(a,alpha,d,theta)
T = [cos(theta) -cos(alpha)*sin(theta) sin(alpha)*sin(theta) a*cos(
   theta)
     sin(theta) cos(alpha)*cos(theta) -sin(alpha)*sin(theta) a*sin(
        theta)
     0 sin(alpha) cos(alpha) d
     0 0 0 1];
end
```

Listing 8: `workspaceProj.m` - Example script demonstrating workspace plotting

```
% simulation.m
function simulation(anglesMat, link_lengths)
    % simulation.m
    % Animates the robotic arm based on joint angles matrix and exports the animation to an MP4 file.
    %
    % Inputs:
    %   anglesMat - [Nx4] Matrix containing joint angles [theta1, theta2, theta3, theta4] in degrees
    %   link_lengths - [1x4] Link lengths [a1, a2, a3, a4] in meters

    % Convert angles from degrees to radians
    jointAngles_rad = deg2rad(anglesMat);

    % Time settings for animation
    numFrames = size(jointAngles_rad, 1);
    % pauseTime = 0.05; % Adjust based on desired animation speed (optional)

    % Initialize painting variables
    isPainting = false; % Whether the pen is down

    % Create the figure
    fig = figure('Name', 'Robotic Arm Animation', 'NumberTitle', 'off');
    hold on;
    grid on;
    xlabel('X (m)');
    ylabel('Y (m)');
    zlabel('Z (m)');
    title('Robotic Arm Animation');
    axis equal;
    xlim([-0.3, 0.3]); % Adjust based on workspace (meters)
    ylim([-0.1, 0.4]);
    zlim([-0.1, 0.4]);
    view(45, 30);

    % Initialize arm plot (lines and joint markers)
    armPlot = plot3(NaN, NaN, NaN, 'b-', 'LineWidth', 2); % Line plot for the arm links
    jointMarkers = plot3(NaN, NaN, NaN, 'bo', 'MarkerSize', 6, 'MarkerFaceColor', 'b'); % Markers for joints

    % Initialize path plot using animatedline
    pathPlot = animatedline('Color', 'r', 'LineWidth', 2);

    % Set up video writer
```

```
    videoFileName = 'robotic_arm_animation.mp4';
    videoWriter = VideoWriter(videoFileName, 'MPEG-4');
    videoWriter.FrameRate = 20; % Adjust the frame rate as needed
    open(videoWriter);

    % Loop through each frame
    for i = 1:numFrames
        % Check for NaN values to determine if the pen should lift
        if any(isnan(jointAngles_rad(i, :)))
            isPainting = false; % Lift the pen
            continue; % Skip this frame without moving the arm
        else
            isPainting = true; % Pen is down
        end

        % Get the joint angles for this frame
        theta1 = jointAngles_rad(i, 1);
        theta2 = jointAngles_rad(i, 2);
        theta3 = jointAngles_rad(i, 3);
        theta4 = jointAngles_rad(i, 4);

        % Calculate the positions of each joint using FK
        [fx, fy, fz] = forwardKinematics([theta1; theta2; theta3;
           theta4], link_lengths);

        % Update the arm plot data (links)
        set(armPlot, 'XData', fx, 'YData', fy, 'ZData', fz);

        % Update the joint markers
        set(jointMarkers, 'XData', fx, 'YData', fy, 'ZData', fz);

        % If painting, add the end-effector position to the path
        if isPainting
            addpoints(pathPlot, fx(end), fy(end), fz(end));
        end

        % Update the title to show the current frame number
        title(['Frame: ', num2str(i)]);

        % Render the updates
        drawnow;

        % Capture the current frame
        frame = getframe(fig);
        writeVideo(videoWriter, frame);

        % Pause for animation (optional)
```

```
88          % pause(pauseTime);
89      end
90
91      % Close the video writer
92      close(videoWriter);
93
94      disp('Animation complete!');
95      disp(['Video saved as ', videoFileName]);
96  end
```

Listing 9: `simulation.m` - Animates and visualizes the manipulator's motion

Inverse Kinematics Implementation

This code computes the joint angles required for the manipulator to reach a specified end-effector position, ensuring it respects joint limits and avoids configurations below the z=0 plane.

```
1   function [angles] = inverseKinematics(posMat, link_params,
        current_config)
2       % Inputs:
3           % Desired transformation matrix (HTM) of end effector wrt.
                the base
4           % Link parameters (link lengths in this case)
5           % Current_config: Current joint configuration of the arm [
                theta1, theta2, theta3, theta4] in degrees
6
7       % Output:
8           % Theta angles in degrees
9
10      % Notes:
11          % Design link lengths such that (a3 * cos(theta3) + a2) != 0
12
13      %% Input Processing
14      % Position vector
15      P = [posMat(1) posMat(2) posMat(3)]; % [x y z]
16
17      % Rotation matrix
18      r = findRotMat(P);
19
20      % Link parameters; no change by default (escape character: -1)
21      try link_params == -1;
22          link_params = [15 15 15 4]; % [a1 a2 a3 a4]
23      end
24
25      % Assign link parameters
26      a1 = link_params(1);
27      a2 = link_params(2);
```

```
    a3 = link_params(3);
    a4 = link_params(4);

    % HTM
    A = [r(1, 1) r(1, 2) r(1, 3) P(1);
         r(2, 1) r(2, 2) r(2, 3) P(2);
         r(3, 1) r(3, 2) r(3, 3) P(3);
         0       0       0       1];

    a1 = link_params(1);
    a2 = link_params(2);
    a3 = link_params(3);
    a4 = link_params(4);

    current_config = current_config * pi / 180;

    %% Singularity Check
    singularity_tolerance = 1e-6;
    if abs(a3*cos(0) + a2) < singularity_tolerance
        error('Singularity detected: a3*cos(theta3) + a2 is near
           zero. Adjust link parameters or avoid this configuration.
           ');
    end

    %% Solving for Angles
    % Solve for theta1
    t1 = atan2(A(1,3), -A(2,3));

    % Solve for theta2+theta3+theta4
    t234 = atan2(A(3,1), A(3,2));
    cost_234 = A(3,2);
    sint_234 = A(3,1);

    % Solve for theta3
    store0 = cos(t1);
    if(abs(store0) <= singularity_tolerance)
        term1 = A(2,4)/sin(t1);
    else
        term1 = A(1,4)/cos(t1);
    end

    alpha = term1 - a4*cost_234;
    beta = A(3,4) - a1 - a4*sint_234;

    cost_3 = ((alpha^2+beta^2)-(a3^2+a2^2))/(2*a2*a3);
    sint_3 = [sqrt(1-(cost_3)^2);...
             -sqrt(1-(cost_3)^2)];
```

```
73
74      store1 = sint_3(1,1);
75
76      if(store1 ~= 0)
77          t3 = atan2(sint_3,cost_3); % column vector
78      else
79          t3 = atan2(store1,cost_3); % scalar
80      end
81
82      % Solve for theta2
83      t2 = [];
84      if(store1 ~= 0)
85          M = a3 * cost_3 + a2;
86          for i = 1:2  % i corresponds to each theta3
87              N(i,1) = a3*sint_3(i,1); % NOTE: M is a scalar but N is
                    a vector
88              matr = [M N(i,1);-N(i,1) M];
89              vec =[beta;alpha];
90              soln = inv(matr)*vec;
91              sin_term = soln(1,1); cos_term= soln(2,1);
92              t2_out = atan2(sin_term,cos_term);
93              t2 = [t2; t2_out]; % t2 is a vector, each row corr.
                    particular theta3
94          end
95      else
96          t2 = atan2(beta,alpha); % t2 is a scalar
97      end
98
99      % Solve for theta 4
100     t4 = t234-(t3+t2);
101     % roll off t4 between 0 deg and +/-180 deg
102     t4 = mod(t4 + pi, 2*pi) - pi;
103
104     %% Cost Minimization; Solution Selection
105     paths = [];
106     for i = 1:length(t2)
107         paths = [paths; [t1, t2(i), t3(i), t4(i)]];
108     end
109
110     costs = zeros(size(paths, 1), 1);
111     for i = 1:size(paths, 1)
112         costs(i) = norm(paths(i, :) - current_config); % Euclidean
                distance
113         % disp("Optimizing...")
114     end
115     [~, best_path_idx] = min(costs);
116     best_path = paths(best_path_idx, :);
```

```
117
118     %% Output Processing
119     angles = best_path * 180 / pi; % Return the best path in degrees
120 end
121
122 % T0_tool=
123 % [cos(t2 + t3 + t4)*cos(t1), -sin(t2 + t3 + t4)*cos(t1),  sin(t1),
      cos(t1)*(a3*cos(t2 + t3) + a2*cos(t2) + a4*cos(t2 + t3 + t4))]
124 % [cos(t2 + t3 + t4)*sin(t1), -sin(t2 + t3 + t4)*sin(t1), -cos(t1),
      sin(t1)*(a3*cos(t2 + t3) + a2*cos(t2) + a4*cos(t2 + t3 + t4))]
125 % [         sin(t2 + t3 + t4),           cos(t2 + t3 + t4),         0,
           a1 + a3*sin(t2 + t3) + a2*sin(t2) + a4*sin(t2 + t3 + t4)]
126 % [                         0,                           0,         0,
                                                                    1]
```

Listing 10: `inverseKinematics.m` - Computes joint angles for a desired position

Forward Kinematics Implementation

This function calculates the positions of each joint and the end-effector given a set of joint angles and link lengths.

```
1  % forwardKinematics.m
2  function [fx, fy, fz] = forwardKinematics(theta, link_lengths)
3      % forwardKinematics.m
4      % Computes the positions of each joint and the end-effector
          based on joint angles and link lengths.
5      %
6      % Inputs:
7      %   theta - [4x1] Joint angles [theta1; theta2; theta3; theta4]
          in radians
8      %   link_lengths - [1x4] Link lengths [a1, a2, a3, a4] in meters
9      %
10     % Outputs:
11     %   fx, fy, fz - [1x5] Positions of base, joint1, joint2, joint3
          , end-effector
12
13     % Extract link lengths
14     a1 = link_lengths(1);
15     a2 = link_lengths(2);
16     a3 = link_lengths(3);
17     a4 = link_lengths(4);
18
19     % Extract joint angles
20     theta1 = theta(1);
21     theta2 = theta(2);
22     theta3 = theta(3);
23     theta4 = theta(4);
```

```
24
25      % Compute cumulative angles
26      theta12 = theta1 + theta2;
27      theta123 = theta12 + theta3;
28      theta1234 = theta123 + theta4;
29
30      % Base position
31      fx(1) = 0;
32      fy(1) = 0;
33      fz(1) = 0;
34
35      % Joint 1 position
36      fx(2) = 0;
37      fy(2) = 0;
38      fz(2) = a1;
39
40      % Joint 2 position
41      fx(3) = fx(2) + cos(theta1) * a2 * cos(theta2);
42      fy(3) = fy(2) + sin(theta1) * a2 * cos(theta2);
43      fz(3) = fz(2) + a2 * sin(theta2);
44
45      % Joint 3 position
46      fx(4) = fx(3) + cos(theta1) * a3 * cos(theta2 + theta3);
47      fy(4) = fy(3) + sin(theta1) * a3 * cos(theta2 + theta3);
48      fz(4) = fz(3) + a3 * sin(theta2 + theta3);
49
50      % End-effector position
51      fx(5) = fx(4) + cos(theta1) * a4 * cos(theta2 + theta3 + theta4)
            ;
52      fy(5) = fy(4) + sin(theta1) * a4 * cos(theta2 + theta3 + theta4)
            ;
53      fz(5) = fz(4) + a4 * sin(theta2 + theta3 + theta4);
54  end
```

Listing 11: `forwardKinematics.m` - Determines joint and end-effector positions

Primary Execution File

This primary script, (`main.m`), loads governs the overall procedure for accepting user input, deriving positions, and creating appropriate joint angles.

```
1  clc
2  clear all
3  close all
4
5  coordinatesMatrix = motion(0, 60, 'Arial', 'normal', true)
6
7  currentConfig = [0 0 0 0]; % Initial position
```

```
8
9  for i = [1:height(coordinatesMatrix)]
10     if isnan(coordinatesMatrix(i))
11         anglesMat(i, :) = [NaN NaN NaN NaN];
12     else
13         %anglesMat(i, :) = inverseKinematics_bounded(
               coordinatesMatrix(i, :), -1, currentConfig);
14         anglesMat(i, :) = inverseKinematics(coordinatesMatrix(i, :),
               -1, currentConfig);
15         currentConfig = anglesMat(i, :);
16     end
17 end
18
19 %simulation_new(anglesMat, [15 15 15 4])
```

Listing 12: `main.m` - Creates joint angles based on user input

Appendix B: Python Scripts

Robot Control Code

The Python script integrates the Dynamixel servo motor communication and control logic with the trajectory planning output produced in MATLAB. Its primary function is to interface between the high-level joint angle commands generated by the kinematic analyses and the low-level servo instructions executed on the physical manipulator. This script ensures that the robotic arm moves along the prescribed path to create the intended light-painted patterns.

```
import os
import time
import numpy as np
from dynamixel_sdk import *  # Uses Dynamixel SDK library
import scipy.io as sio

# Handle platform-specific imports for getch functionality
if os.name == 'nt':
    import msvcrt

    def getch():
        return msvcrt.getch().decode()
else:
    import sys
    import tty
    import termios

    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)

    def getch():
        try:
            tty.setraw(sys.stdin.fileno())
            ch = sys.stdin.read(1)
        finally:
            termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
        return ch

# Constants and Configuration
MY_DXL = 'MX_SERIES'
ADDR_TORQUE_ENABLE = 64
ADDR_GOAL_POSITION = 116
ADDR_PRESENT_POSITION = 132
DXL_MINIMUM_POSITION_VALUE = 1050
DXL_MAXIMUM_POSITION_VALUE = 3050
BAUDRATE = 57600
PROTOCOL_VERSION = 2.0
```

```
DXL_ID = np.array((1, 2, 3, 4))
DEVICENAME = '/dev/tty.usbserial-FT3FSNI8'
TORQUE_ENABLE = 1
TORQUE_DISABLE = 0
DXL_MOVING_STATUS_THRESHOLD = 100
DEGREE_TO_COUNT = 4096 / 360
ANGLE_OFFSET = 180

# Load MATLAB Data
mat_contents = sio.loadmat('/Users/benjaminforbes/Desktop/heart.mat')
angles_deg = np.array(mat_contents['anglesMat'])
mapped_angles_deg = angles_deg + ANGLE_OFFSET
goal_positions = np.round(mapped_angles_deg * DEGREE_TO_COUNT).astype(int)

print(goal_positions)

# Initialize PortHandler and PacketHandler
portHandler = PortHandler(DEVICENAME)
packetHandler = PacketHandler(PROTOCOL_VERSION)

# Open Port
if not portHandler.openPort():
    print("Failed to open the port")
    getch()
    quit()
print("Succeeded to open the port")

# Set Baudrate
if not portHandler.setBaudRate(BAUDRATE):
    print("Failed to change the baudrate")
    getch()
    quit()
print("Succeeded to change the baudrate")

# Enable Torque for each motor
for i in DXL_ID:
    dxl_comm_result, dxl_error = packetHandler.write1ByteTxRx(
        portHandler, i, ADDR_TORQUE_ENABLE, TORQUE_ENABLE)
    if dxl_comm_result != COMM_SUCCESS:
        print(packetHandler.getTxRxResult(dxl_comm_result))
    elif dxl_error != 0:
        print(packetHandler.getRxPacketError(dxl_error))
    else:
        print(f"Dynamixel ID {i} has been successfully connected")

```

```
# Main Loop
while True:
    print("Press any key to continue! (or press ESC to quit!)")
    if getch() == chr(0x1b):  # ESC key
        break

    for i in range(np.size(goal_positions, 0)):
        flag = True
        print("New goal!")
        dxl_goal_position = goal_positions[i, :]

        # Validate and adjust goal positions
        for j in range(len(DXL_ID)):
            if goal_positions[i, j] <= 0:
                print(f"Skipping row {i} due to NaN values")
                flag = False
                break

            if goal_positions[i, j] >= 3400:
                goal_positions[i, j] = 3400
                print(f"Changing row {i} due to large values")

            if goal_positions[i, j] <= 1023:
                goal_positions[i, j] = 1023
                print(f"Changing row {i} due to small values")

            if j == 1:  # Special handling for second motor
                dxl_goal_position[1] = 3600 - (dxl_goal_position[1]
                   - 1024)

        if not flag:
            time.sleep(5.0)
            continue

        # Write goal positions
        for j, dxl_id in enumerate(DXL_ID):
            dxl_comm_result, dxl_error = packetHandler.
               write4ByteTxRx(portHandler, dxl_id,
               ADDR_GOAL_POSITION, dxl_goal_position[j])
            if dxl_comm_result != COMM_SUCCESS:
                print(packetHandler.getTxRxResult(dxl_comm_result))
            elif dxl_error != 0:
                print(packetHandler.getRxPacketError(dxl_error))

        # Monitor positions
        while True:
            completed = True
```

```
126                 for j, dxl_id in enumerate(DXL_ID):
127                     dxl_present_position, dxl_comm_result, dxl_error =
                           packetHandler.read4ByteTxRx(portHandler, dxl_id,
                           ADDR_PRESENT_POSITION)
128                     if dxl_comm_result != COMM_SUCCESS:
129                         print(packetHandler.getTxRxResult(
                               dxl_comm_result))
130                     elif dxl_error != 0:
131                         print(packetHandler.getRxPacketError(dxl_error))
132                     print(f"[ID:{dxl_id:03d}] GoalPos:{dxl_goal_position
                           [j]} PresPos:{dxl_present_position}")
133
134                     if abs(dxl_goal_position[j] - dxl_present_position)
                           > DXL_MOVING_STATUS_THRESHOLD:
135                         completed = False
136                 if completed:
137                     break
138
139 # Disable Torque and Close Port
140 for i in DXL_ID:
141     dxl_comm_result, dxl_error = packetHandler.write1ByteTxRx(
           portHandler, i, ADDR_TORQUE_ENABLE, TORQUE_DISABLE)
142     if dxl_comm_result != COMM_SUCCESS:
143         print(packetHandler.getTxRxResult(dxl_comm_result))
144     elif dxl_error != 0:
145         print(packetHandler.getRxPacketError(dxl_error))
146
147 portHandler.closePort()
```

Listing 13: `light_robot.py` - Python script for robot control using Dynamixel SDK

Appendix C: Demonstration

The following images showcase the results of the BEAM manipulator writing the names of the project team members using long-exposure photography. Each image captures the luminous path traced by the robotic arm's LED end-effector as it follows the planned trajectories to render the characters in three-dimensional space. These demonstrations highlight the system's ability to produce visually compelling and accurate patterns.

Figure 8: Long-exposure image of "Aggy" written by the BEAM manipulator.

Figure 9: Long-exposure image of "CJ" written by the BEAM manipulator.

Figure 10: Long-exposure image of "Deb" written by the BEAM manipulator.

Figure 11: Long-exposure image of "Eric" written by the BEAM manipulator.

Figure 12: Long-exposure image of "Kevin" written by the BEAM manipulator.

Figure 13: Long-exposure image of "PK" written by the BEAM manipulator.

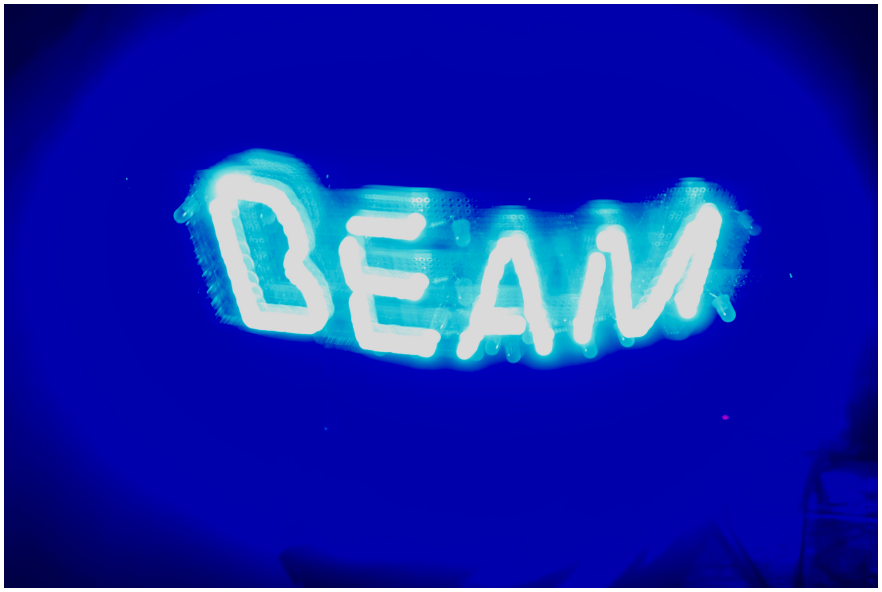

Figure 14: Long-exposure image of "BEAM" demonstrating the manipulator's capability to render the project name.