Maze Solving with Mycobot Pro 600

Devika Shaj Kumar Nair[1], Darsh Patel[2], Aman Milind Dhale[3]
[1]Ira Fulton School of Engineering, Arizona State University, Arizona, United States of America
[2]Ira Fulton School of Engineering, Arizona State University, Arizona, United States of America
[3]Ira Fulton School of Engineering, Arizona State University, Arizona, United States of America

Article Info

Article history:

Received December 14, 2024

Keywords:

ArUco Markers
Inverse Kinematics
TCP Socket Connection
MyCobot 600 Pro
Breadth First search Algorithm
Homogeneous Transformation
Digital Twin

ABSTRACT

This project focuses on developing a vision-guided robotic maze-solving system using the MyCobot Pro 600 robotic arm. The process begins with capturing an image of a 4x4 maze printed on a plastic board using the AI Kit's camera. The maze-solving algorithm employs Breadth-First Search (BFS) to identify the maze's entrance, exit, and solution path composed of straight-line segments. These segments are used as input for the MATLAB inverse kinematics solver, accessed from Python using the MATLAB Engine API, to compute the joint angles required for the robot's end effector to navigate through the maze. The planned path is validated in a digital twin simulation of the robot to ensure accuracy before real-world execution. The robotic arm autonomously replicates the motion, successfully navigating the maze from start to finish. This project demonstrates the integration of computer vision, kinematics, and algorithmic planning to achieve precise and autonomous robotic navigation.

Corresponding Author:

Devika Shaj Kumar Nair
Ira Fulton School of Engineering, Arizona State University
Tempe, Arizona, United States of America
Email: dshajkum@asu.edu

1. INTRODUCTION

Mycobot 600 Pro is a 6 axis collaborative robot that was designed for both commercial and educational purposes. It has an operating radius of 600mm, with an effective load capability of 2kg. The robot weighs around 8.8kg and it utilizes a raspberry pi microprocessor embedded with RoboFlow visual programming software. It has 6 joints with the following joint angle limits[1].

Table 1: Joint angle range for Mycobot 600 Pro

Joint	Angle Range
J1	+/- 180°
J2	-270~90°
J3	+/- 150°
J4	-260~80°
J5	+/- 168°
J6	+/- 174°

This project explores the integration of robotics, computer vision, and path planning, demonstrated using the MyCobot Pro 600 robotic arm. The primary goal was to design a system capable of detecting visual inputs through a camera, mapping them to real-world coordinates, and enabling the robotic arm to navigate its environment based

on these inputs. The project began with the development of a digital twin of the robotic arm, modeled as a URDF file using SolidWorks, to simulate its kinematic behavior accurately. An experimental setup was established with an AI Kit camera overlooking a defined arena, which featured a calibration plate with ArUco markers for coordinate mapping.

The system utilized the ArUco markers to establish a linear mapping between pixel coordinates from the camera and physical coordinates in the real world, enabling precise positioning of the robot's end effector. To further demonstrate the capabilities of this setup, a 4 x 4 maze was introduced into the workspace, and the shortest path through it was determined using the BFS[7] algorithm. The maze's turning points, identified in pixel coordinates, were converted into physical coordinates using the established mapping function. These coordinates were then used as target positions for the robot's end effector. MATLAB's Robotics Toolbox was employed to compute the joint angles corresponding to these target positions using its generalized inverse kinematics capabilities. The joint angles were transmitted to the robotic arm via a TCP[9] connection, allowing the robot to autonomously follow the computed path and solve the maze. At each step outlined above, the results are stored in an excel sheet and read from it whenever required. Reading from and writing to the Excel sheet ensures the entire process is automated, providing a streamlined workflow for the user and simplifying navigation between each step. This approach makes it easier to manage the data and reduces the chances of manual errors, enhancing the user experience.

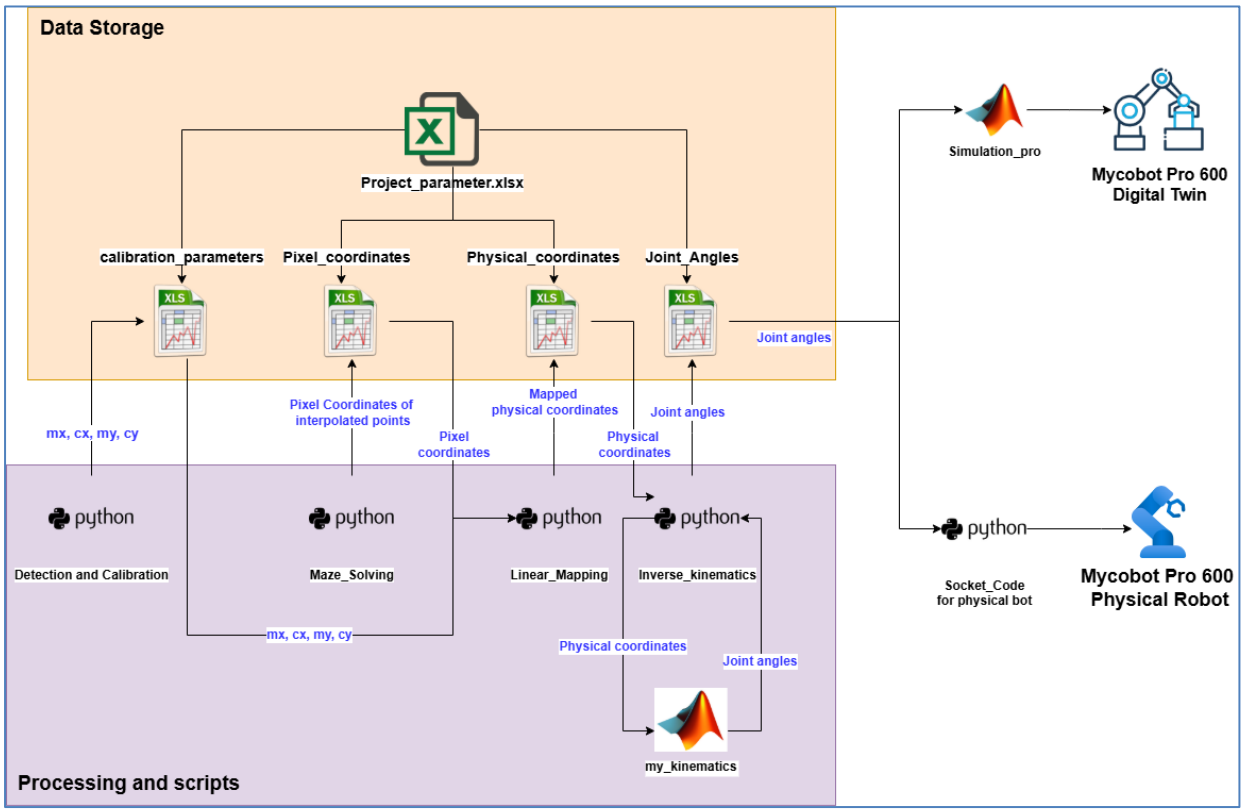


Figure 1: Automated code system walkthrough

The following sections of this report provide a detailed explanation of the methodology, including the kinematic modeling, coordinate mapping process, path planning, and the implementation of the robotic system for maze traversal.

2. KINEMATIC MODEL AND HOMOGENEOUS TRANSFORMATIONS

Homogeneous transformation matrices are derived from the configuration diagram of the robot model. Here, the homogeneous transformation matrices of the MyCobot 600 pro robot is generated manually from the configuration diagram shown in figure 2: and the corresponding transformation matrices obtained are as follows:

```
H01 = [cos(t1)      0    sin(t1)    0;
       sin(t1)      0   -cos(t1)    0;
       0            1      0        210;
       0            0      0        1];

H12 = [-cos(t2)  -sin(t2)    0      -250*cos(t2);
       -sin(t2)   cos(t2)    0      -250*sin(t2);
        0           0       -1      76.2;
        0           0        0      1];

H23 = [-cos(t3) -sin(t3) 0       250*cos(t3);
       -sin(t3)  cos(t3) 0       250*sin(t3);
       0         0       -1      76.2;
       0         0        0      1];
```

```
H34 = [-cos(t4)   0  sin(t4)    0;
       -sin(t4)   0  -cos(t4)   0;
       0         -1        0    76.2;
       0          0        0    1];

H45 = [cos(t5)   0   sin(t5)    0;
       sin(t5)   0  -cos(t5)    0;
       0         1        0     107;
       0         0        0     1];

H56 = [-cos(t6)   sin(t6) 0     0;
       -sin(t6)  -cos(t6) 0     0;
       0         0        1     109.5;
       0         0        0     1];
```

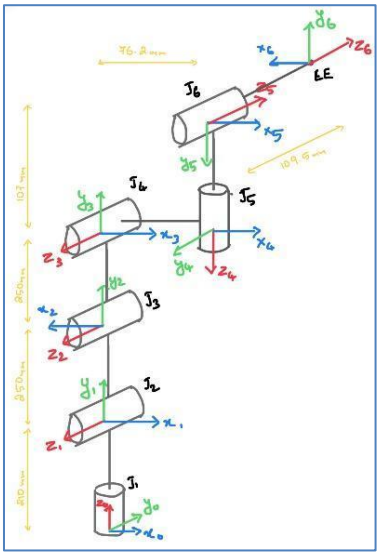


Figure 2: Configuration Diagram for Mycobot 600 Pro

The above homogeneous transformations yielded the following coordinates in matlab.

```
>> Homogeneous_Transformation
H06 =
   -1.0000         0         0         0
         0         0   -1.0000 -185.7000
         0   -1.0000         0  817.0000
         0         0         0    1.0000

X =
     0

Y =
  185.7000

Z =
   817
```

```
>> Homogeneous_Transformation
H06 =
         0         0    1.0000  185.7000
   -0.2588    0.9659         0 -521.6122
   -0.9659   -0.2588         0  479.1751
         0         0         0    1.0000

X =
 -185.7000

Y =
  521.6122

Z =
  479.1751
```

```
>> Homogeneous_Transformation
H06 =
    1.0000         0         0 -500.0000
         0         0   -1.0000 -185.7000
         0    1.0000         0  103.0000
         0         0         0    1.0000

X =
   500

Y =
  185.7000

Z =
   103
```

Figure 3: Homogeneous transformation results for Joint angles [0, -90, 0, -90, 0, 0], [90, -45, 30, -90, 0, 0] and [0,0,0,0,0,0] respectively

3. DIGITAL TWIN DESIGN

The simulation of the MyCobot Pro 600 started with downloading the CAD model from the Elephant Robotics website. This model was imported into SolidWorks for further processing and the following steps were followed.

- Each joint in the assembly file was separated into a separate part and saved as a STEP file to allow independent movement of each joint during the simulation.
- After separating the joints, the parts were reassembled in SolidWorks. Joints were stacked in sequence from the base to the end effector, with appropriate mating of surfaces to ensure unhindered rotational motion
- Local coordinate systems were assigned to each joint, defining the rotational axes necessary for tracking motion in the simulation.
- The assembled model was then exported as a URDF file using the Simscape Multibody plugin within SolidWorks. This file format is required for importing the model into MATLAB for simulation.

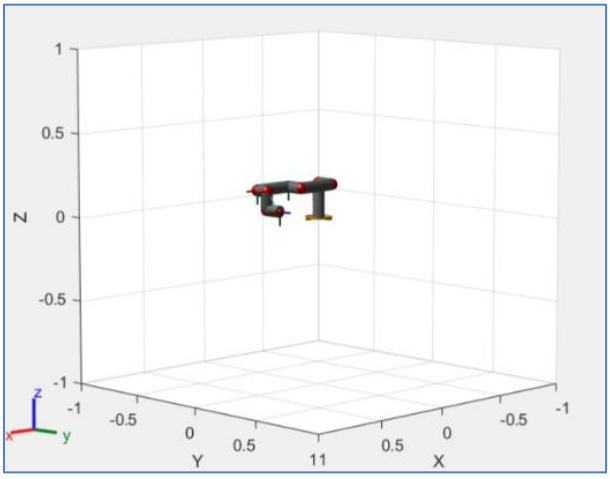


Figure 4: Exported URDF visualised in MATLAB

4. DETECTION AND CALIBRATION OF THE ARENA

The detection and calibration of the arena formed the foundation for precise navigation and manipulation tasks in this project. The primary objective was to transform pixel coordinates from the AI Kit camera feed into accurate physical coordinates, ensuring reliable robot control within the workspace.

4.1. Arena Setup and ArUco Marker Placement

The calibration process began with the placement of a plastic board in the camera's field of view. This board featured two ArUco markers (Marker 1 and Marker 2) on its surface. These markers served as reference points for mapping pixel coordinates captured by the camera to their corresponding physical coordinates in the real world.

4.2. Initial Camera Configuration and Zoom

As soon as the live feed from the camera became available, the user was prompted to draw a rectangular area on the feed encompassing both ArUco markers. This step allowed the system to zoom in on the selected region of interest, ensuring that subsequent pixel coordinate captures would be more accurate and localized. The same zoom settings were maintained throughout the process to preserve consistency. After the calibration markers were captured, an image of the maze within the same zoomed area was also saved for later processing.

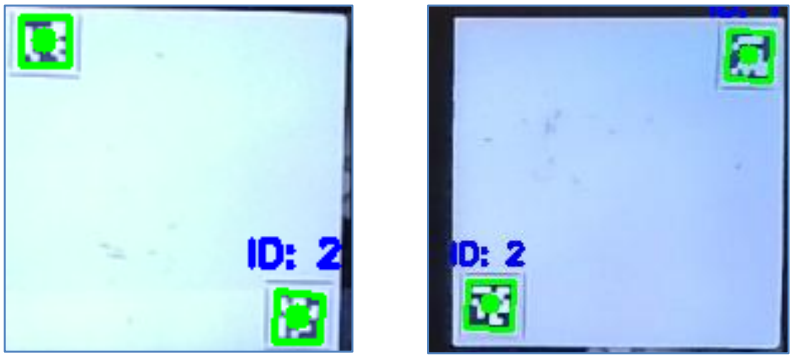


Figure 5: Detected aruco markers for calibration (Initial and rotated scenario)

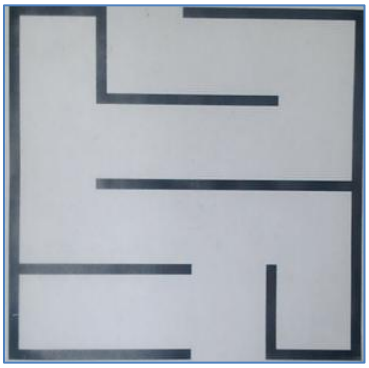


Figure 6: Maze image captured and saved as png file

4.3. Pixel and Physical Coordinate Capture

Following the zoom configuration, the camera detected the ArUco markers, and their pixel coordinates were recorded. To obtain the corresponding physical coordinates, the robotic arm's end effector was manually maneuvered to each marker's position, and its physical location was documented.

Next, the board was rotated to introduce a new set of reference points, with two additional ArUco markers (Marker 3 and Marker 4). The detection and manual calibration process were repeated for this second pair of markers to increase the robustness of the calibration.

4.4. Deriving the Linear Mapping Function

To convert pixel coordinates into physical coordinates, a linear mapping function based on the equation of a line y=mx+c was implemented. The parameters for the mapping function, including the slopes (mx,my) and intercepts (cx,cy), were calculated for each pair of markers using the following formulas:

$$m_x = \frac{x_2 - x_1}{u_2 - u_1}, \quad c_x = x_1 - m_x \cdot u_1$$

$$m_y = \frac{y_2 - y_1}{v_2 - v_1}, \quad c_y = y_1 - m_y \cdot v_1$$

Here:

- (u1,v1) and (u2,v2) are the pixel coordinates of the two markers.
- (x1,y1) and (x2,y2) are their corresponding physical coordinates.

The same calculations were performed for the second pair of markers. To ensure accuracy, the average of the slope and intercept values from both pairs was computed, resulting in final calibration parameters (mx,my,cx,cy).

5. SOLVING THE MAZE

To solve the maze, a systematic approach was taken, starting with image preprocessing and proceeding through entrance/exit detection, pathfinding, simplification, alignment, and instruction generation. The **Breadth-First Search (BFS)** algorithm was specifically chosen to compute the shortest path through the maze due to its efficiency and guaranteed shortest-path results. BFS was chosen because it guarantees the shortest path without the need for heuristics, making it optimal for environments where all paths are equally weighted. It operates in O(N), where N is the number of cells in the maze, making it computationally feasible for real-time applications. Its systematic approach and computational efficiency ensured reliable and rapid pathfinding, laying the groundwork for effective robotic navigation.

5.1. Preprocessing the Maze Image

Before detecting the entrance and exit points, the captured maze image underwent preprocessing to ensure reliable and accurate detection.

5.1.1. Thresholding:

- The binary maze image was generated by applying an optimal threshold value to the grayscale version of the maze.
- This step ensured that the maze's black obstacles (lines) and white paths were clearly separated.

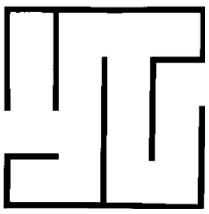


Figure 7: Thresholded binary image of the maze

5.1.2. Dilation:

- The binary image was dilated to thicken the black obstacles and eliminate any white borders around the maze.
- This preprocessing step ensured that the maze's boundaries were well-defined and prevented false positives during entrance and exit detection.

Figure 8: Maze image with dilation

5.2. Entrance and Exit Detection

The entrance and exit points of the maze were detected by scanning the borders of the binary image. The following steps outline the detection logic used.

- The code scanned the top or left border to locate the entrance by identifying the first white pixel opening.
- Similarly, the bottom or right border was scanned to detect the exit point.
- The preprocessing steps ensured that the white paths were cleanly defined, making detection robust and reliable.
- Once the entrance and exit of the maze was identified, the pixel coordinates were recorded as the follows:

Entrance: (168, 0), Exit: (98, 297)

5.3. Pathfinding with BFS

Once the entrance and exit points were determined, BFS was used to find the shortest path between them. BFS starts at the entrance and explores neighboring cells in four cardinal directions (up, down, left, right). It only traverses cells corresponding to white paths (value 255) and avoids revisiting cells by maintaining a set of visited nodes.

5.3.1. Execution Steps

- **Queue Initialization**: The entrance point was added to a queue to start the exploration.
- **Neighbor Exploration**: For each node, its neighbors were checked for validity (within bounds, unvisited, and traversable).
- **Path Reconstruction**: A `prev` dictionary was used to track the predecessor of each node, enabling path reconstruction upon reaching the exit.

Code logic used:

```
queue = [start]
visited = set()
prev = {start: None}

found_end = False

while queue:
    current = queue.pop(0)
    if current in visited:
        continue
    visited.add(current)

    if current == end:
        found_end = True
        break

    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        neighbor = (current[0] + dx, current[1] + dy)
        if (
            0 <= neighbor[0] < maze_map.shape[0]
            and 0 <= neighbor[1] < maze_map.shape[1]
            and maze_map[neighbor] == 1
            and neighbor not in visited
        ):
            queue.append(neighbor)
            prev[neighbor] = current
```

5.3.2. Path Simplification

After BFS provided the shortest path, simplification was performed to reduce unnecessary movements. The intermediate points that did not result in significant changes in direction were removed. The angles between consecutive path segments were calculated, and points were retained only if the angular deviation exceeded a predefined threshold.

5.3.3. Cardinal Path Alignment

To align the robot's movement to cardinal directions (horizontal and vertical), the path was refined further. Intermediate points were added between path segments to create smooth transitions. This step ensured that the robot moved only along horizontal or vertical lines, avoiding diagonal movements.

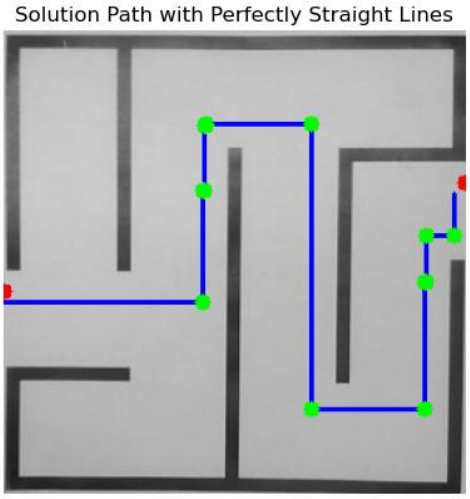


Figure 9: Shortest path solution for the given maze

5.3.4. Path Instructions Generation

The final path was converted into robot-executable instructions and the instructions were adjusted for the robot's dimensions, ensuring accurate movement within the maze. The instructions generated were saved in a text file as shown in figure. The common instruction types were as follows :

- Go Straight X Steps: For continuous movement in a single direction.
- Turn Left/Right: At points where the path direction changed.

```
1  Go Straight 1 Block
2  Turn Left
3  Go Straight 1 Block
4  Turn Right
5  Go Straight 1 Block
6  Turn Left
7  Go Straight 1 Block
8  Turn Right
9  Go Straight 1 Block
10 Turn Left
11 Go Straight 1 Block
12 Turn Right
13 Go Straight 1 Block
14 Turn Right
```

```
14 Turn Right
15 Go Straight 1 Block
16 Turn Left
17 Go Straight 1 Block
18 Turn Left
19 Go Straight 1 Block
20 Turn Right
21 Go Straight 1 Block
22 Turn Left
23 Go Straight 1 Block
24 Turn Right
25 Go Straight 1 Block
26 Turn Left
27 Go Straight 1 Block
```

Figure 10: Instructions captured for the given maze

5.3.5. Pixel coordinates of turning points

The pixel coordinates of the interpolated points of the solution path is obtained and written into the "**Pixel_coordinates**" sheet in the excel file "**Project_parameters_file.xlsx**".

```
Please enter the name of the image file: captured_frame_1734031703
Entrance: (168, 0), Exit: (98, 297)
x coordinate: 175, y coordniate: 128
x coordinate: 103, y coordniate: 128
x coordinate: 103, y coordniate: 129
x coordinate: 61, y coordniate: 129
x coordinate: 61, y coordniate: 130
x coordinate: 60, y coordniate: 130
x coordinate: 60, y coordniate: 198
x coordinate: 244, y coordniate: 198
x coordinate: 244, y coordniate: 271
x coordinate: 162, y coordniate: 271
x coordinate: 162, y coordniate: 272
x coordinate: 132, y coordniate: 272
x coordinate: 132, y coordniate: 290
The 'Pixel_coordinates' sheet has been updated in E:/RASLAB/PROJECT_RUN_THROUGH/Project_parameters_file.xlsx.
```

Figure 11: Output showing pixel coordinates obtained for the solution path interpolated points

6. LINEAR MAPPING OF THE POINTS

This section outlines the process for transforming camera pixel coordinates into the robot's physical coordinates using calibration parameters stored in an Excel file. The process involves retrieving necessary parameters, performing the transformations, and managing data in a structured manner.
The equation used for transformation is:

$$y = mx+c$$

Here, y represents the physical coordinate, x is the pixel coordinate retrieved from the Excel sheet, m is the scaling factor, and c is the offset. The scaling factors (mx,my) and offsets (cx,cy) were calculated in the detection and calibration step and are saved in an excel file named "Project_parameters_file.xlsx" under a sheet called "calibration_parameters".

- The first step involves retrieving the calibration parameters. The read_calibration_parameters function is responsible for this. It extracts the values for mx,cx,my,cy from the calibration sheet. These parameters form the foundation for mapping the pixel coordinates into physical ones.

- Next, the pixel coordinates are read using the read_pixel_coordinates function. This function accesses the "Pixel_coordinates" sheet in the same Excel file and retrieves a list of 2D coordinates. Each coordinate pair corresponds to specific points identified in the camera's frame.

- Once the pixel coordinates and calibration parameters are available, the getRobotCoordinates function performs the transformation. For each pair of pixel coordinates, the function applies the linear equations for both x and y directions, yielding the corresponding physical coordinates.

- The transformed data is then organized and stored. The add_pixel_coordinates function is used to save or update a sheet named "Pixel_coordinates" with the 2D pixel coordinates. Similarly, the add_physical_coordinates function saves the computed physical coordinates into a new sheet named "Physical_coordinates". This sheet includes an additional fixed Z -value of 0.07m, ensuring the coordinates are compatible with 3D robotic applications. Both functions are designed to overwrite existing sheets with the same name, ensuring consistency in data management.

- Finally, the process is orchestrated in the main workflow. The calibration parameters and pixel coordinates are read, and the physical coordinates are computed for each pixel point. These are then converted into the desired units (e.g., millimeters to meters) before being saved.

```
Physical coordinates of the points:
Point 1: Camera Coordinates (168, 0) -> Physical Coordinates (-0.230008238568, -0.3288951356)
Point 2: Camera Coordinates (175, 128) -> Physical Coordinates (-0.22670742095, -0.386287017712)
Point 3: Camera Coordinates (103, 128) -> Physical Coordinates (-0.260658687878, -0.386287017712)
Point 4: Camera Coordinates (103, 129) -> Physical Coordinates (-0.260658687878, -0.38673539179099997)
Point 5: Camera Coordinates (61, 129) -> Physical Coordinates (-0.280463593586, -0.38673539179099997)
Point 6: Camera Coordinates (61, 130) -> Physical Coordinates (-0.280463593586, -0.38718376587000003)
Point 7: Camera Coordinates (60, 130) -> Physical Coordinates (-0.28093513896, -0.38718376587000003)
Point 8: Camera Coordinates (60, 198) -> Physical Coordinates (-0.28093513896, -0.417673203242)
Point 9: Camera Coordinates (244, 198) -> Physical Coordinates (-0.194170790144, -0.417673203242)
Point 10: Camera Coordinates (244, 271) -> Physical Coordinates (-0.194170790144, -0.450404511009)
Point 11: Camera Coordinates (162, 271) -> Physical Coordinates (-0.23283751081200002, -0.450404511009)
Point 12: Camera Coordinates (162, 272) -> Physical Coordinates (-0.23283751081200002, -0.450852885088)
Point 13: Camera Coordinates (132, 272) -> Physical Coordinates (-0.246983872032, -0.450852885088)
Point 14: Camera Coordinates (132, 290) -> Physical Coordinates (-0.246983872032, -0.45892361851)
Point 15: Camera Coordinates (98, 297) -> Physical Coordinates (-0.26301641474799997, -0.462062237063)
The 'Physical_coordinates' sheet has been updated in E:/RASLAB/PROJECT_RUN_THROUGH/Project_parameters_file.xlsx.
```

Figure 12: Real world coordinates obtained after linear mapping for the solution obtained for the given maze

7. INVERSE KINEMATICS AND JOINT ANGLE CALCULATION

The joint angle computation and inverse kinematics (IK) process ensure the robotic arm can navigate through the waypoints of the maze. This involves translating Cartesian coordinates of waypoints into joint angles required to position the robotic arm's end effector precisely. The IK solver computes these joint angles using the robot's kinematic model, ensuring smooth and accurate movement.

7.1. MATLAB Implementation of Inverse Kinematics

The core computation is implemented in a MATLAB file named **my_inversekinematics.m**, which was specifically designed for this project. This file contains a function called ik, which handles the entire IK process, including setup, target pose definition, and joint angle computation.

- **Setting Up the Robotic Arm Model**

  To begin, the digital twin of the robotic arm is imported using its URDF file. This model outlines the kinematic structure, joint limits, and degrees of freedom (DoF) of the robot. MATLAB's importrobot function loads the model, and the Generalized Inverse Kinematics (GIK)[10] solver is initialized to compute joint configurations. The solver's maximum iterations are set to ensure convergence during computation.

- **Defining the Target Pose**

  Each waypoint represents a target position for the robotic arm's end effector. The pose includes the waypoint's physical coordinates (position) and a fixed orientation specified in Euler angles, such as [178, 0, 0] degrees. The Euler angles are converted into a rotation matrix and combined with the position data to form a homogeneous transformation matrix, which defines the complete target pose.

- **Solving the Inverse Kinematics Problem**

  For each waypoint, the GIK solver calculates the joint angles needed to position the robot's end effector at the desired pose. The pose constraint is set for the final link (link6), and the transformation matrix is used as the target. The solver iterates from an initial guess configuration, refining the solution to ensure smooth motion.

- **Processing Multiple Waypoints**

  The process is repeated iteratively for all waypoints. The joint angles for one waypoint serve as the initial guess for the next, optimizing performance and ensuring smooth transitions. A matrix is created where each row corresponds to a waypoint and each column represents a joint.

- **Validation and Execution**

  Before executing the joint angles, they are validated against the robot's joint limits. While the GIK solver inherently handles these constraints, additional safety checks can be implemented if required.

7.2. Python Integration with MATLAB Engine

The MATLAB function ik is invoked from Python to integrate the IK computation into a larger workflow. This involves reading waypoint data from an Excel file, passing it to MATLAB for computation, and saving the results back to Excel.

- Python reads the "Physical_coordinates" sheet from "Project_parameters_file.xlsx", converting the data into a format suitable for MATLAB.

- Using the MATLAB Engine API [8], the ik function is invoked, passing the coordinates as input. The function computes the joint angles for each waypoint.

- The calculated joint angles are received in Python, processed into a list format, and written back to the "Joint_Angles" sheet in the same Excel file.

```
Successfully read 3D coordinates from 'Physical_coordinates' in E:/RASLAB/PROJECT_RUN_THROUGH/Project_parameters_file.xlsx.
        Joint1     Joint2      Joint3      Joint4     Joint5     Joint6
0  -141.106852 -41.849200  110.588472 -159.995321 -88.443488 -51.089789
1  -134.806336 -36.874435   96.877244 -151.422081 -88.590719 -44.788880
2  -137.850422 -34.636068   91.124521 -147.830890 -88.517345 -47.833051
3  -137.808193 -34.587318   91.004015 -147.760227 -88.518334 -47.790820
4  -139.449584 -33.072283   87.246110 -145.474366 -88.480462 -49.432337
5  -139.407436 -33.021856   87.124354 -145.404155 -88.481419 -49.390184
6  -139.445365 -32.983748   87.031037 -145.347941 -88.480557 -49.428117
7  -136.715780 -29.279021   78.277587 -140.370097 -88.544216 -46.698354
8  -128.899834 -35.278316   92.903333 -149.181756 -88.744233 -38.882773
9  -126.410412 -31.191768   82.941445 -143.359480 -88.813027 -36.393735
10 -130.006843 -28.667874   76.961261 -139.825580 -88.714395 -39.989651
11 -129.973344 -28.603764   76.813130 -139.742311 -88.715291 -39.956156
12 -131.213064 -27.519991   74.296114 -138.280918 -88.682429 -41.195761
13 -130.611606 -26.308193   71.537188 -136.747537 -88.698296 -40.594355
14 -131.729795 -24.418879   67.257874 -134.331848 -88.668913 -41.712453
The DataFrame has been saved to the 'Joint Angles' sheet in E:/RASLAB/PROJECT RUN THROUGH/Project parameters file.xlsx.
```

Figure 13: Joint angles calculated for the given set of physical coordinates

The joint angles and inverse kinematics process, implemented across MATLAB and Python, translates Cartesian maze waypoints into precise robotic arm movements. This combined approach ensures robust and accurate IK calculations while maintaining a streamlined workflow between Python and MATLAB.

To connect the physical robot to a laptop, an Ethernet connection is used. The robot is connected to the laptop's Ethernet port, and the network settings are configured by assigning the IP address of the robot's server to the laptop. Once this connection is established, a Python script is executed to send joint angles to the robot, facilitating movement control.

8. VERIFICATION OF RESULTS IN DIGITAL TWIN

This section outlines the The approach involved in animating the robotic arm's movement using MATLAB, with joint angles provided in the Excel file, and visualizing the resulting end-effector path. The primary goal is to simulate and verify the robot's motion virtually, ensuring correctness before implementing it physically.

- **Reading Data from Excel File**

  The joint angles are read from the “Project_parameters_file_simulation.xlsx”, specifically from the sheet “Joint_Angles”. The angles are initially in degrees and are converted to radians to be compatible with robot kinematics calculations.

- **Loading the Robot Model**

  The robot model is defined in a URDF file that describes the robot's structure and kinematics. Using MATLAB's robotics toolbox, the model is loaded and displayed in its initial home configuration to establish a baseline visualization.

- **Setting Up the Visualization Environment**

  The animation environment is customized with features like lighting, camera angles, zoom, and axis limits to ensure that the robot's movements are clearly visible during the simulation process.

- **Interpolating Joint Angles**

  To ensure smooth transitions between consecutive sets of joint angles, linear interpolation is applied. The interpolation is carried out using a parameter (alpha), which defines the intermediate steps, and the process executes over a specified number of steps (nSteps).

- **Animating Robot Movements**

  The robot's joint configurations are updated iteratively for each interpolated step. As these updates occur, the robot's motion is animated in real time, clearly showing the dynamic movement of its links and joints.

- **Tracking the End-Effector Path**

  At every configuration step, the position of the end effector is calculated using the function getEndEffectorPosition. These positions are stored in an array called path, which represents the complete trajectory of the end effector during the motion sequence.

- **Plotting the Path**

  The end-effector's trajectory is dynamically plotted during the animation. This visual feedback allows continuous monitoring of the path traced by the robot's end effector throughout the motion.

- **Final Visualization**

  Once the animation concludes, the robot model is removed from the visualization, leaving only the traced path of the end effector on the plot. This provides a clear and isolated view of the trajectory for better analysis and documentation.

- **Saving the Path Visualization**

  The final trajectory of the end effector is saved as an image file (robot_path_only.png) to document the results or for further review.

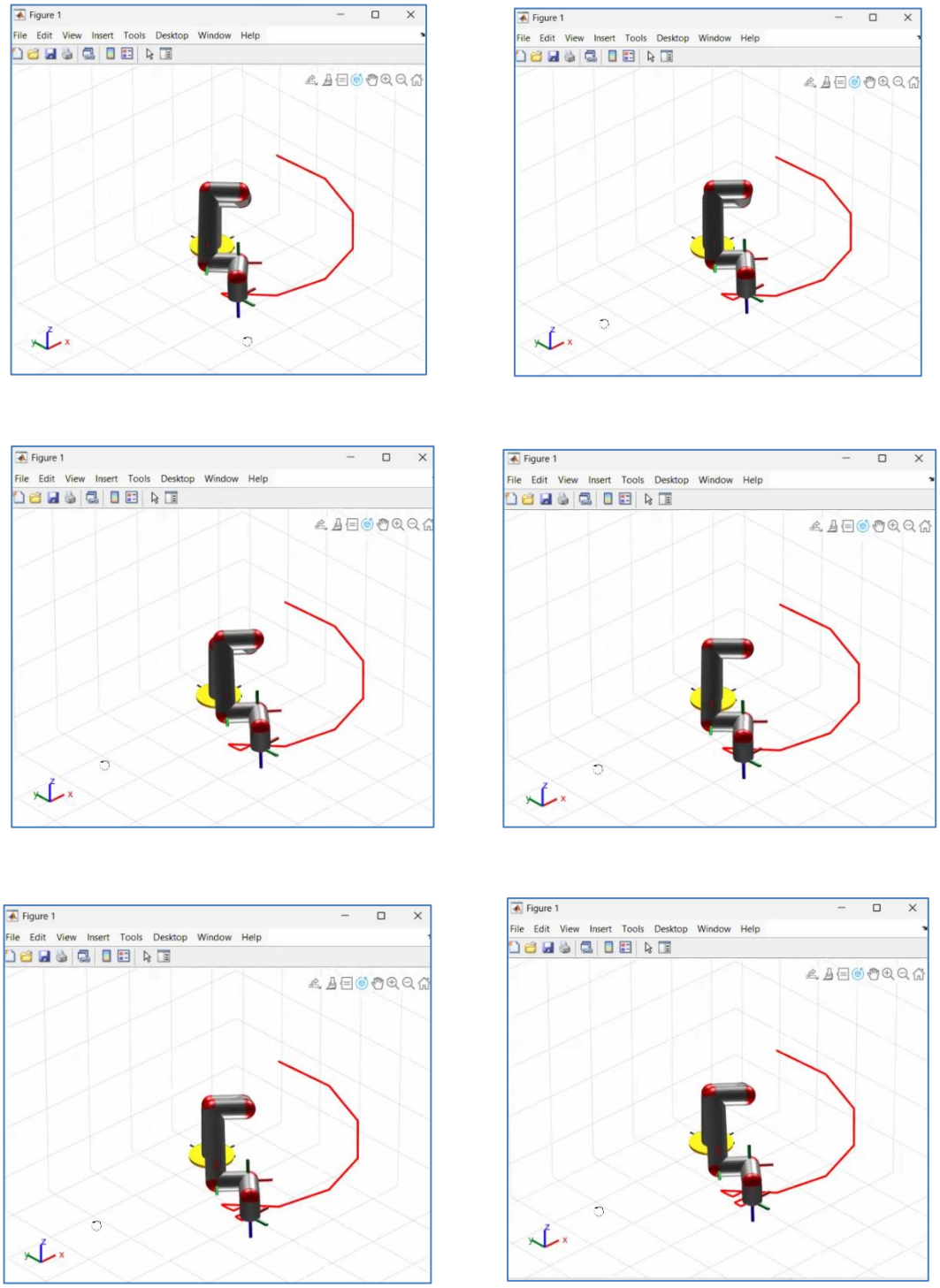


Figure 14: MATLAB simulation results of the digital twin traversing through the interpolated solution points

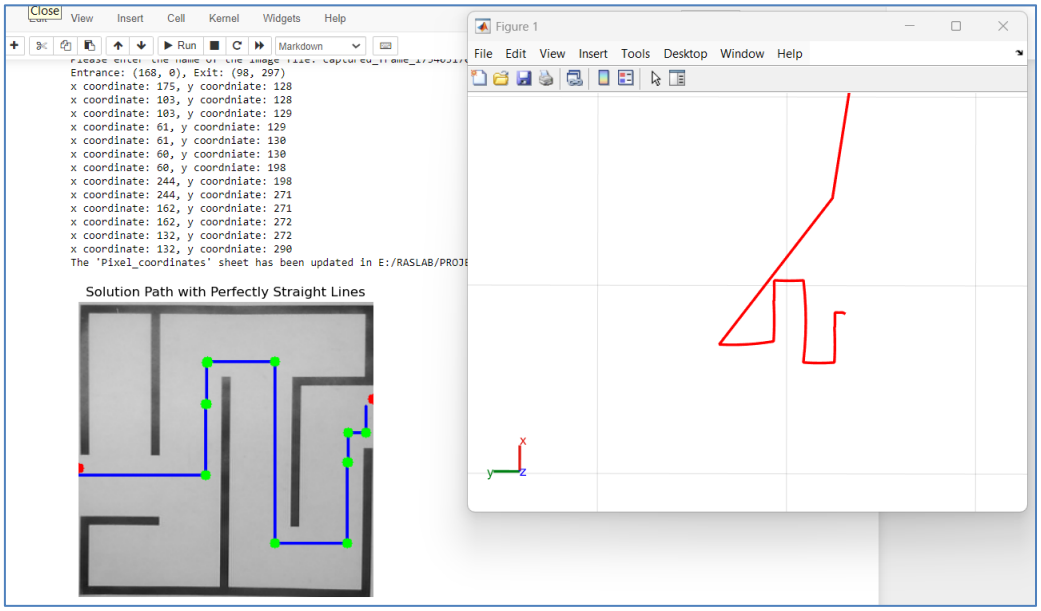


Figure 15: The path traced by the digital twin matching the maze solution obtained initially

9. RESULTS IN PHYSICAL ROBOT

9.1. CONNECTING TO PHYSICAL ROBOT

To connect the physical robot to a laptop, an Ethernet connection is used. The robot is connected to the laptop's Ethernet port, and the network settings are configured by assigning the IP address of the robot's server to the laptop. Once this connection is established, a Python script is executed to send joint angles to the robot, facilitating movement control. The python script outlines the following steps in order to implement the movement control:

- The read_excel_to_list function loads joint angle configurations stored in the "**Joint_Angles"** sheet of the Excel file. The file is read into a Pandas DataFrame. Data is converted into a list of lists, where each inner list corresponds to a set of joint angles for one position.

- The send_tcp_packet function establishes communication with the robot's server. It takes the server's IP address and port as inputs. A command message (joint angles) is sent as a UTF-8 encoded string. Optionally, the robot's response is printed to confirm successful communication.

- The main script loops through the list of joint angle configurations and formats them into the required command message format. A predefined speed (SPEED) is appended to each configuration. The command is formatted as a string starting with set_angles, followed by joint angles and the speed. The command is transmitted to the robot using the send_tcp_packet function. A delay (DELAY) between commands ensures smooth transitions, avoiding abrupt movements. The execution status of each command is logged for monitoring.

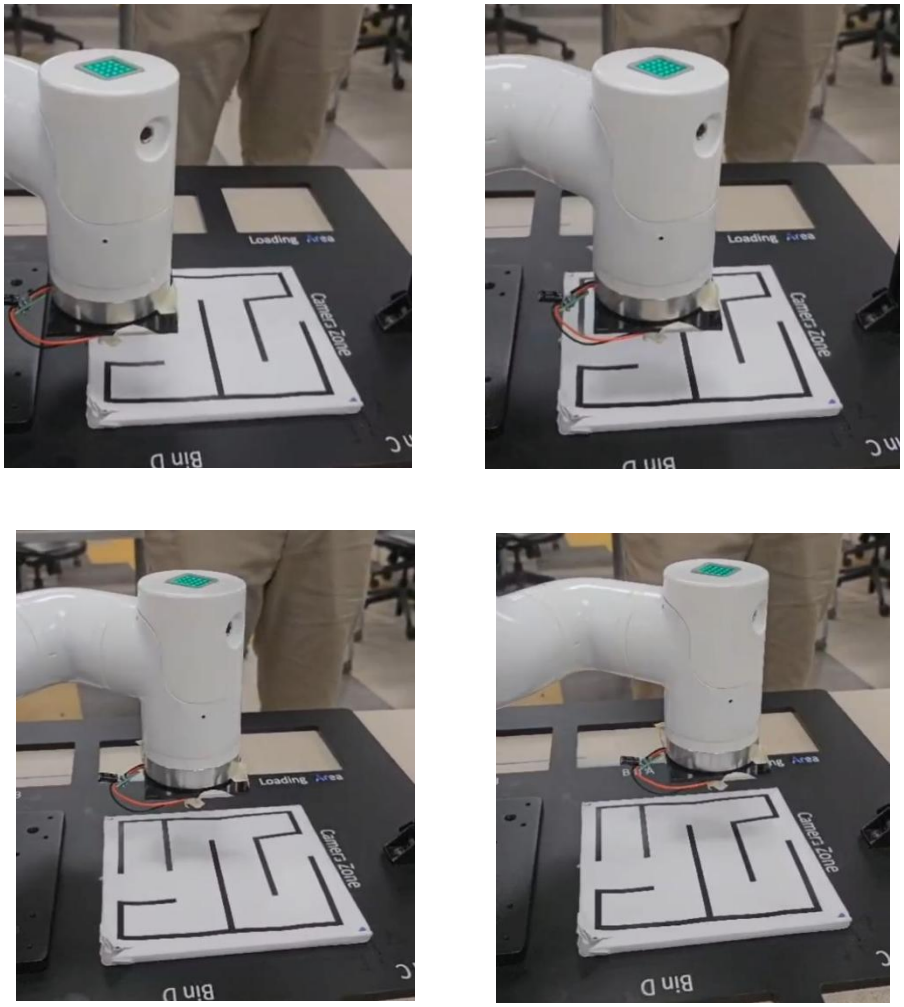
```
Data successfully read from sheet 'Joint_Angles' in E:/RASLAB/PROJECT_RUN_THROUGH/Project_parameters_file.xlsx

 MOVING TO -
Point 1
Point 2
Point 3
Point 4
Point 5
Point 6
Point 7
Point 8
Point 9
Point 10
Point 11
Point 12
Point 13
Point 14
Point 15

 Motion completed!
```

Figure 16: Console output on executing the python script for motion control

9.2. PHYSICAL ROBOT EXECUTION



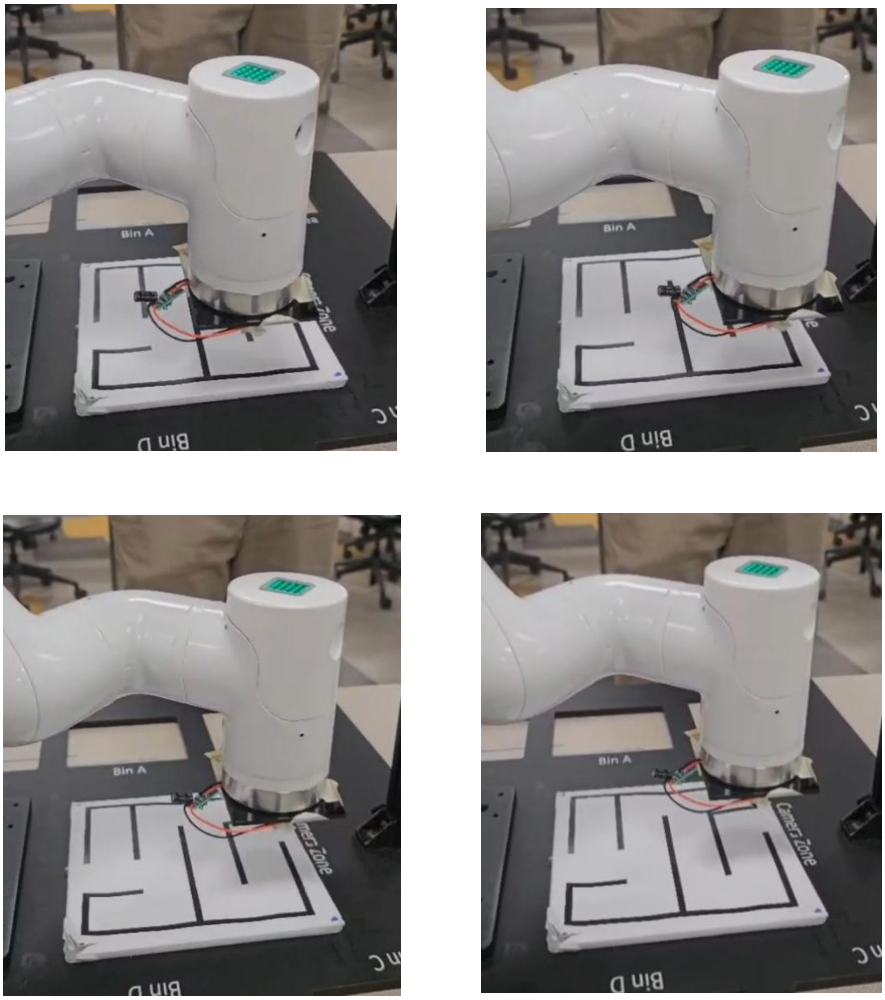


Figure 17: Physical robot tracing the maze solution by traversing through the interpolated points

10. CONCLUSION

This project effectively integrated robotics, computer vision, and path planning to achieve accurate and reliable control of the MyCobot Pro 600 robotic arm in both a digital twin simulation and a physical environment. The approach began with the detection of ArUco markers using an AI Kit camera, which allowed for the mapping of pixel coordinates to real-world physical coordinates. This was achieved through a well-calibrated system, where calibration parameters were stored in an Excel sheet. The maze-solving task was made possible by using the BFS algorithm, and the maze's turning points were identified in pixel coordinates, which were then converted to physical coordinates using the established linear mapping function.

The key innovation of the project was the seamless integration between Python and MATLAB, where joint angles corresponding to the target physical positions were computed using MATLAB's inverse kinematics capabilities. The joint angles were then transmitted to the robot via a TCP/IP connection. The entire process, from calibration to joint angle calculation and robot control, was automated through reading and writing data to the Excel sheets. This not only ensured ease of use but also streamlined the process, making it easy for users to manage the robot's motion and adjustments.

The robot's movements were controlled in both the digital twin and physical robot scenarios. In both cases, the robot successfully followed the computed path to navigate the maze without any errors, demonstrating the

precision and reliability of the system. The Excel-based automation of the entire process—ranging from the initial calibration of the camera to the final execution of the robot's movements—ensured that each step was systematically and accurately executed, without requiring manual intervention.

This result underscores the precision of the methodology employed, validating the robustness and accuracy of the system for both simulation and real-world robotic applications. By successfully bridging computer vision, path planning, and robotic control, this project establishes a foundation for further advancements in autonomous robot navigation and system integration, making the process more user-friendly and efficient. The seamless interaction between the digital twin and physical robot, coupled with the accurate execution of the maze-solving task, proves the effectiveness of the integrated system in achieving reliable and precise robot control.

REFERENCES

[1] Elephant Robotics, "MyCobot Pro 600 Documentation," Available: https://docs.elephantrobotics.com/docs/gitbook-en/2-serialproduct/2.3-myCobot_Pro_600/2.3-myCobot_Pro_600.html. Accessed: Sep. 14, 2024.

[2] Elephant Robotics, "Environment Building for ROS1," Available: https://docs.elephantrobotics.com/docs/pro600-en/12-ApplicationBaseROS/12.1-ROS1/12.1.2-EnvironmentBuilding.html. Accessed: Oct. 12, 2024.

[3] OpenCV, "ArUco: Table of Contents," Available: https://docs.opencv.org/3.4/d9/d6d/tutorial_table_of_content_aruco.html. Accessed: Nov. 23, 2024.

[4] OpenCV, "OpenCV Tutorials," Available: https://docs.opencv.org/4.x/d9/df8/tutorial_root.html. Accessed: Nov. 23, 2024.

[5] MathWorks, "Inverse Kinematics Algorithms," Available: https://www.mathworks.com/help/robotics/ug/inverse-kinematics-algorithms.html. Accessed: Nov. 23, 2024.

[6] Elephant Robotics, "Socket API Interface Description for myCobot Pro 600," Available: https://docs.elephantrobotics.com/docs/gitbook-en/2-serialproduct/2.3-myCobot_Pro_600/2.3.5%20socket%20API%20interface%20description.html. Accessed: Nov. 23, 2024.

[7] GeeksforGeeks, "Breadth-first search or BFS for a graph," Available: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. Accessed: Dec. 13, 2024.

[8] MathWorks, "MATLAB Engine for Python," Available: https://www.mathworks.com/help/matlab/matlab-engine-for-python.html. Accessed: Dec. 13, 2024.

[9] MathWorks, "TCP/IP Communication with MATLAB Engine," Available: https://www.mathworks.com/help/matlab/matlab-engine/tcp-ip-communication.html. Accessed:

[10] MathWorks, "GeneralizedInverseKinematics System Object," Available: https://www.mathworks.com/help/robotics/ref/generalizedinversekinematics-system-object.html. Accessed: Dec. 13, 2024.