Solving a 4x4 Maze with MyCobot Pro 600

Manasi Rajan Variar[1]
[1]School of Computing and Augmented Intelligence, Ira A. Fulton Schools of Engineering, Arizona State University

Corresponding Author:

Manasi Rajan Variar
School of Computing and Augmented Intelligence, Ira A. Fulton Schools of Engineering, Arizona State University
Tempe, Arizona, USA
Email: mrajanva@asu.edu

1. ABSTRACT

This project presents a robotics solution for navigating a 4x4 rectangular maze using the MyCobot Pro 600 robot. The maze is represented as a printed board within the visual range of the robot's AI kit camera. The project integrates computer vision, path-finding algorithms, robotic kinematics, and simulation to achieve precise end-effector movements through the maze. By utilizing Python for maze detection and solving and MATLAB for path planning and kinematics, the project demonstrates an interdisciplinary approach to robotics challenges. The results validate the effectiveness of this method, showcasing accurate maze navigation and robotic path execution.

2. INTRODUCTION

Maze navigation has long been a benchmark problem in robotics, combining challenges in perception, planning, and execution. This project aims to enable the MyCobot Pro 600 robot to navigate a predefined maze autonomously, leveraging its advanced features and computational tools. The project focuses on detecting the maze layout using computer vision, solving it with efficient algorithms, and translating the solution into executable robotic motions. This report details the methodologies employed, the results achieved, and the insights gained from integrating Python and MATLAB in a robotics context.

3. DETECTION OF ARUCO MARKERS

A critical task in this lab was the identification and localization of ArUco markers using the AI Kit camera. This facilitated real-time detection of the target positions for the MyCobot Pro 600. For this purpose, we utilized the Python OpenCV library, which provides a comprehensive set of tools for ArUco marker detection. A continuous video stream was initialized, enabling the cobot to monitor its workspace in real time.

The detection process was initiated using the *detector.detectMarkers(frame)*. This function extracted the corner coordinates of the detected ArUco markers, along with their unique IDs. The corner coordinates were then processed to compute the center of each marker using their IDs, which served as the target point for the robot's end effector.

A mapping function was developed to ensure seamless integration between the marker's camera coordinates and the cobot's coordinate system. This mapping function transformed the marker's detected position into the cobot's operational workspace, accounting for the differences in coordinate systems and ensuring precise movement to the desired target positions.

4. MAPPING FUNCTION

Since the camera and cobot were positioned differently and operated along distinct axes, it was essential to establish a transformation to convert the pixel-based camera coordinates into the cobot's Cartesian workspace coordinates. In a Cartesian system, such transformations typically involve:

1. An additive factor to account for translation.
2. A multiplicative factor to account for scaling.

We applied this principle to determine the cobot's coordinates. The transformation for the cobot's *$robot_X$-coordinate* was formulated as:

$robot_x = a_x * camera_x + b_x$

Similarly, for the *$robot_y$*-coordinate:

$robot_y = a_y * camera_y + b_y$

Here, a_x, b_x, and a_y, b_y are unknown coefficients. We required two known data points for each axis to solve for these coefficients. Using observed values, the following system of equations was constructed:

1. $-182.001 = a_x * 438 + b_x$
2. $-349.074 = a_y * 361 + b_y$
3. $-284.584 = a_x * 198 + b_x$
4. $-222.725 = a_y * 98 + b_y$

By solving equations (1) and (3) for a_x and b_x, and equations (2) and (4) for a_y and b_y, we obtained the following coefficients:

$a_x = 0.409636, b_x = -361.017; a_y = -0.480382, b_y = -175.516$

Using these coefficients, the final transformation equations are:

1. $robot_x = 0.409636 * camera_x - 361.017$
2. $robot_y = -0.480382 * camera_y - 175.516$

These equations allow us to calculate the corresponding *$robot_x$* and *$robot_y$* coordinates for any given *$camera_x$* and *$camera_y$* coordinates, ensuring precise mapping between the camera and cobot coordinate systems. The calculated *$robot_x$* and *$robot_y$* coordinates are then fed into the Inverse Kinematics block as input for the corresponding Px and Py values.

5. CALCULATING INVERSE KINEMATICS

We utilized the same Model-Based Design (MBD) framework developed in Lab 3 for this step. To simulate the final end effector position, we added a small solid at the end effector, as the system lacks an active end effector. Additionally, we incorporated another square solid in the workspace to align with the location of the ArUco board, ensuring the model accurately reflected the experimental setup.

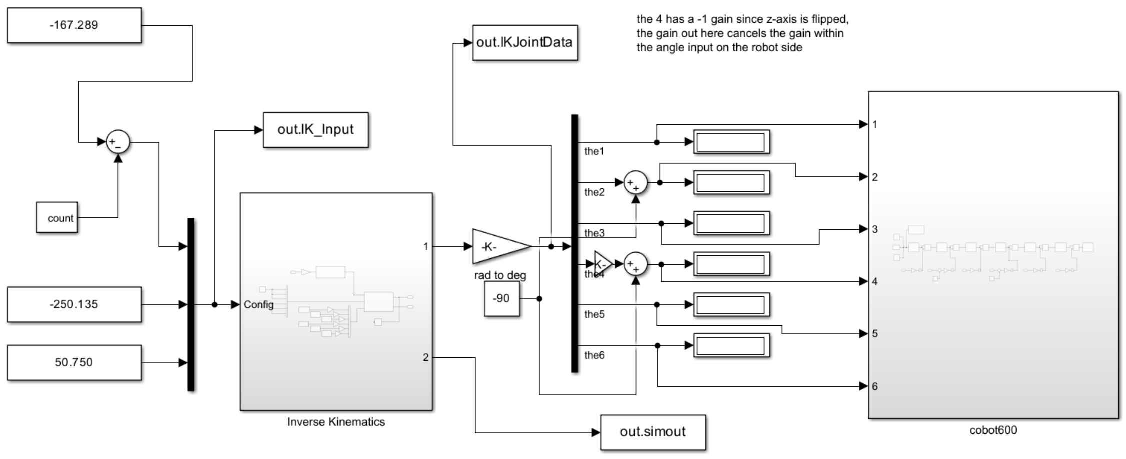


Fig 2: myCobot Pro 600 MBD diagram

Forward kinematics blocks from the original model were retained to simulate the robot's motion. To ensure accurate angle conversion, input gains of $\pi/180$ were applied to transform degrees into radians. Inverse kinematics (IK) blocks were introduced to enhance the system's adaptability, replacing the user-defined inputs previously used for the cobot Simulink subsystem. The IK block's input consisted of the end effector's x, y, and z positions, which were converted into a homogeneous transformation matrix to complete the kinematics. The IK block output provided the joint angles necessary to achieve the desired end effector position.

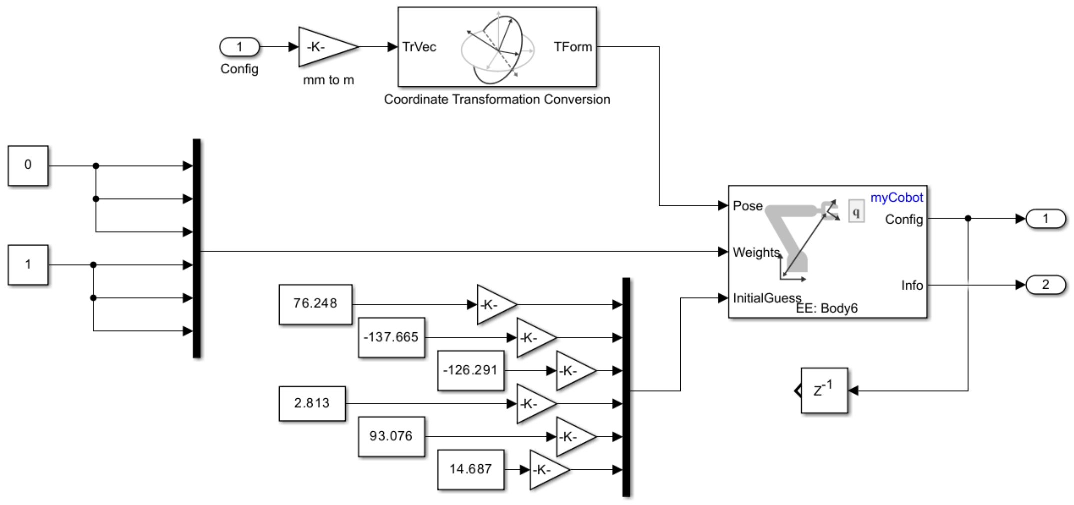


Fig 3: Inverse Kinematics sub-system

However, the displayed angles were incompatible with the physical robot. To reconcile this, the output was modified to account for adjustments necessary for the forward kinematics, resulting in angles that could be sent to the physical cobot for precise motion control. Digital displays were also added for real-time data visualization.

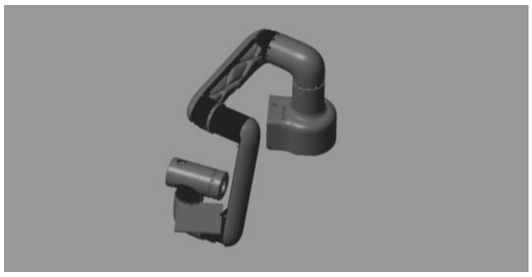


Fig 4.1: Before adding constraints

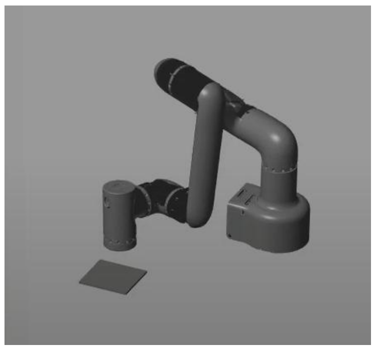


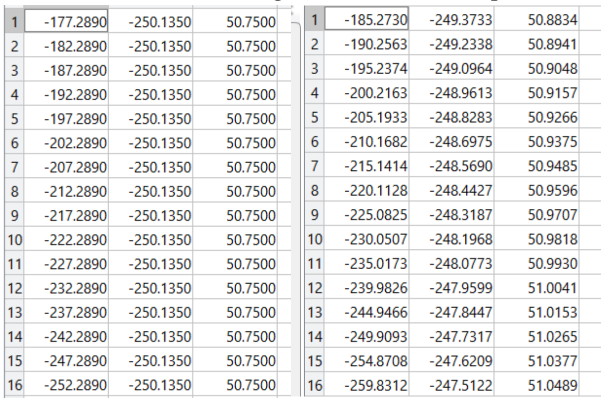
Fig 4.2: After adding constraints

After these changes, the cobot accurately reached the target position in the virtual twin. However, the default Simulink IK blocks prioritized major joints for movement, leading to awkward configurations that risked collision conditions. To address this, joint limits were tightened within the model, particularly for joints (4, 5, and 6). These limits were based on positional data collected in earlier labs. With these constraints in place, the system produced realistic and collision-free joint configurations, significantly improving the model's performance.

6. VALIDATING THE IK OUTPUT

We selected specific points within the workspace for path planning to create a linear motion trajectory. A counter was implemented to adjust the X-coordinate by up to 100 mm incrementally, simulating a linear movement along the X-axis. The resulting positional data was saved to an output variable in the MATLAB workspace for further processing.

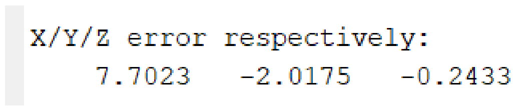
We wrote a MATLAB script to facilitate data transfer and analysis to reshape the data into a format compatible with the physical robot. Another script was developed to send the joint angle data (calculated by the Simulink IK block) to my MATLAB code, allowing me to track all the positions in the trajectory.



1	-177.2890	-250.1350	50.7500
2	-182.2890	-250.1350	50.7500
3	-187.2890	-250.1350	50.7500
4	-192.2890	-250.1350	50.7500
5	-197.2890	-250.1350	50.7500
6	-202.2890	-250.1350	50.7500
7	-207.2890	-250.1350	50.7500
8	-212.2890	-250.1350	50.7500
9	-217.2890	-250.1350	50.7500
10	-222.2890	-250.1350	50.7500
11	-227.2890	-250.1350	50.7500
12	-232.2890	-250.1350	50.7500
13	-237.2890	-250.1350	50.7500
14	-242.2890	-250.1350	50.7500
15	-247.2890	-250.1350	50.7500
16	-252.2890	-250.1350	50.7500

1	-185.2730	-249.3733	50.8834
2	-190.2563	-249.2338	50.8941
3	-195.2374	-249.0964	50.9048
4	-200.2163	-248.9613	50.9157
5	-205.1933	-248.8283	50.9266
6	-210.1682	-248.6975	50.9375
7	-215.1414	-248.5690	50.9485
8	-220.1128	-248.4427	50.9596
9	-225.0825	-248.3187	50.9707
10	-230.0507	-248.1968	50.9818
11	-235.0173	-248.0773	50.9930
12	-239.9826	-247.9599	51.0041
13	-244.9466	-247.8447	51.0153
14	-249.9093	-247.7317	51.0265
15	-254.8708	-247.6209	51.0377
16	-259.8312	-247.5122	51.0489

Fig 5: Comparison of Simulink(left) and MATLAB(right)



```
X/Y/Z error respectively:
    7.7023    -2.0175    -0.2433
```

Fig 5: Average across all 22 points

The average error across 22 points was calculated to evaluate the accuracy, as shown in Figure 5. These results demonstrate that the linear motion trajectory is highly accurate, with only minor deviations that are well within acceptable tolerances for this application. This validation confirms that the model and path planning process are robust and can effectively apply to the physical cobot.

7. CONNECTING TO MYCOBOT PRO 600 VIA TCP TO SEND ANGLES

The joint angles obtained from the inverse kinematics (IK) calculations were sent to the MyCobot Pro 600 via a TCP connection. To achieve this, we first connected our laptop to the cobot using an Ethernet cable and enabled the communication port. Configuring the laptop to match the cobot's IP address was crucial, ensuring a secure and stable connection.

Once the connection was established, socket messaging was used to transmit commands to the cobot. These commands included the joint angle data generated by the Simulink IK block. Upon executing the commands, the cobot moved to the desired positions, validating the success of the inverse kinematics implementation for motion control.

This step demonstrated the effective integration of software-based motion planning and hardware execution, marking a significant milestone in achieving precise cobot operation based on computational outputs.

8. CONCLUSION:

This lab successfully demonstrated the application of homogeneous transformation matrices and inverse kinematics for controlling the MyCobot Pro 600 robot. The MATLAB-generated transformation matrix accurately represented the cobot’s end-effector position, while the virtual twin, though less precise, visually depicted the robot’s position changes.

By detecting and processing ArUco marker coordinates using the AI Kit camera, we derived the necessary transformation to map camera coordinates to the robot's workspace. The calculated coordinates were input to the Inverse Kinematics block in MATLAB, which generated joint angles for the cobot. When sent to the robot via TCP, these angles facilitated accurate motion control, demonstrating reliable marker-driven navigation.

Overall, this lab successfully integrated computer vision, kinematic modeling, and robotic control, achieving the goal of precise movement based on ArUco markers. These results lay the foundation for further enhancements in subsequent projects.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Prof. Sangram Redkar, my lab supervisor, and teaching assistants Rajdeep Adak and Kunal Palasdeokar for their invaluable guidance, encouragement, and support throughout this experiment.

REFERENCES

[1] "1 Introduction to Robot Parameters · GitBook," *Elephantrobotics.com*, 2017. https://docs.elephantrobotics.com/docs/pro600-en/2-serialproduct/2.3-myCobot_Pro_600/2.3.1%20Introduction%20to%20product%20parameters.html (accessed Oct. 13, 2024).

[2] "myCobot - Elephant Robotics," Dec. 02, 2020. https://www.elephantrobotics.com/en/mycobot-en/

[3] "How to design Robots using MATLAB 2021 | SimScape Toolbox | Robotics System Toolbox," *www.youtube.com*. https://www.youtube.com/watch?v=_8YCc3pJDPI

[4] https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html

[5] https://docs.opencv.org/3.4/d9/d6d/tutorial_table_of_content_aruco.html

[6] https://docs.elephantrobotics.com/docs/gitbook-en/2-serialproduct/2.3-myCobot_Pro_600/2.3.5% 20socket% 20API% 20interface %20description.html